tsp_report_final

December 30, 2024

Course	Combinatorial Algorithms
Semester	**2024 Winter**
Assignment	**01**
Group	**03**
Member 01	**999014681 Mingshan, LI**
Member 02	**999014772 Shunxi, XIAO**
Member 03	**999022064 Weizhi, LU**

1 Q2. TSP: Branch and Bound and Approximation

1. Implement in Python the branch and bound and the approximation algorithms for the traveling salesman problem.
2. Provide test cases to ensure the correctness of your programs.
3. Report on the comparison of the running times of the backtracking with bounding, the branch and bound, and the approximation implementations.

1.1 Part 01: Test Cases Generator

Thanks to the course materials, we already have the implementation of TSP bounding algorithm. We can therefore make the use of `tsp_bounding` to generate the correct solutions of randomly generated test cases.

[1]:
```
from math import inf, isinf
from tsp_bounding import tsp_bounding
import random

def tsp_generate_test_cases(fname: str, tests_num: int, init: int, step: int,
 ↪maxDist: int, conRate: float) -> None:
    '''
    Generate valid test cases for TSP problem that must guarentee the
 ↪triangular inequality.

    Arguments:
        - fname:     the name of file that stores all the test cases
        - tests_num: the number of test cases to generate
        - init:      the initial value of nodes number
```

```
      - step:      the step nodes number is increased each loop
      - maxDist:   the maximun distance between nodes
      - conRate:   the posibility rate of connection between two nodes, range␣
↪from 0 to 1
    '''

    assert 0 < conRate and conRate < 1 and f"Error: conRate = {conRate} is␣
↪invalid, conRate is in (0, 1)."

    file = open(fname, 'w')

    count = 1
    nodes_num = init

    while count <= tests_num:

        '''
        Step 1: Generate an empty graph
        '''

        graph = valid_graph(nodes_num, maxDist, conRate)

        if not is_connected(graph):

            continue

        '''
        Step 4: Generate a TSP solution to the graph
        '''

        sol = tsp_bounding(graph)

        if len(sol) != nodes_num:

            continue

        # for row in graph: print(row)
        # print(f'sol: {sol}')
        sol_ = ' '.join(map(str, sol))

        '''
        Step 5: Print graph and sol into file {fname}
        '''

        test_case = ''

        for i in range(nodes_num):
```

```
            row = ' '.join(map(str, graph[i]))
            test_case += row + '#'

        test_case += sol_

        print(test_case)

        if count != tests_num:
            test_case += '\n'

        file.write(test_case)

        count += 1
        nodes_num += step

    file.close()


def valid_graph(nodes_num: int, maxDist: int, conRate: float) -> list[list]:
    '''
    Generate a valid graph that satisfies triangle inequality.
    Arguments:
        - nodes_num: number of nodes
        - maxDist:   maximum value of edge distance
        - conRate:   possibility rate of the connection between two nodes
    '''

    graph = []

    for r in range(nodes_num):
        row = [0] * nodes_num
        row[r] = inf
        graph.append( row )

    def set_edges(currNode, vistedNodes) -> None:

        nonlocal graph

        if currNode == nodes_num - 1:

            return

        nodesLeft = [ node for node in range(currNode+1, nodes_num) ]

        for newNode in nodesLeft:
```

```
            randInt = random.randint(1, 100)
            connInt = int(conRate * 100)

            newEdge = inf # new edge between currNode and newNode

            # 1. Decide whether newNode and currNode is connnected
            if randInt <= connInt:

                # 2. Set the distance of newEdge according to triangle␣
↪inequelity
                if len(vistedNodes) != 0:

                    # Found the bound of newEdge

                    maxBound = maxDist
                    minBound = 1

                    for vNode in vistedNodes:

                        edge1 = graph[vNode][newNode]
                        edge2 = graph[vNode][currNode]

                        if not isinf(edge1) and not isinf(edge2):

                            newMaxBound = edge1 + edge2
                            newMinBound = abs(edge1 - edge2)

                            if newMaxBound < maxBound: maxBound = newMaxBound
                            if newMinBound > minBound: minBound = newMinBound

                    # set new edge between currNode and newNode

                    if minBound < maxBound:

                        newEdge = random.randint(minBound, min(maxDist,␣
↪maxBound))

                    elif minBound == maxBound:

                        newEdge = min(maxDist, maxBound)

                    # if minBound > maxBound, then currNode and newNode cannot␣
↪be connected.

                else:

                    newEdge = random.randint(1, maxDist)
```

```
            graph[currNode][newNode] = newEdge
            graph[newNode][currNode] = newEdge

        set_edges( currNode + 1, vistedNodes + [currNode] )

    set_edges(0, [])

    return graph


def is_connected(graph) -> bool:
    """
    Check whether the graph is connected.
    """
    n = len(graph)
    visited = [False] * n
    # print(visited)

    def is_connected_recursive(node):

        visited[node] = True

        # print(visited)

        for neighbor in range(n):

            if graph[node][neighbor] != inf and not visited[neighbor]:

                is_connected_recursive(neighbor)

    is_connected_recursive(0)

    return all(visited)


# g = [
#         [inf, 3, inf, 4, inf, inf],
#         [3, inf, 6, 5, inf],
#         [inf, 6, inf, inf, inf],
#         [4, 5, inf, inf, 4],
#         [inf, inf, inf, 4, inf]
#     ]

# flag = is_connected(g)
# print(flag)
```

We can then run the following code snippet to generate test cases.

[2]:
```
fname = 'tsp_test_cases'
num = 8
init = 4
step = 1
maxDist = 20
conRate = 0.83
print('All generated tests:')
tsp_generate_test_cases(fname, num, init, step, maxDist, conRate)
```

```
All generated tests:
inf 4 inf 9#4 inf 2 8#inf 2 inf 8#9 8 8 inf#0 1 2 3
inf 13 8 2 16#13 inf 20 13 5#8 20 inf 8 17#2 13 8 inf inf#16 5 17 inf inf#0 1 4
2 3
inf 19 19 inf 5 inf#19 inf inf 14 inf inf#19 inf inf 13 16 17#inf 14 13 inf 18
13#5 inf 16 18 inf 20#inf inf 17 13 20 inf#0 1 3 5 2 4
inf 15 20 1 11 10 14#15 inf inf 16 15 5 13#20 inf inf 20 15 10 8#1 16 20 inf inf
inf 15#11 15 15 inf inf 10 14#10 5 10 inf 10 inf inf#14 13 8 15 14 inf inf#0 3 1
5 2 6 4
inf inf 2 20 inf 18 inf 12#inf inf 14 8 inf inf 16 13#2 14 inf 20 1 20 7 11#20 8
20 inf 19 3 18 10#inf inf 1 19 inf 19 inf 12#18 inf 20 3 19 inf 19 10#inf 16 7
18 inf 19 inf 10#12 13 11 10 12 10 10 inf#0 2 4 7 6 1 3 5
inf 14 2 5 9 3 17 11 11#14 inf 13 inf 20 17 7 12 10#2 13 inf 3 inf 4 19 10 12#5
inf 3 inf 13 inf 16 7 15#9 20 inf 13 inf 10 15 inf 18#3 17 4 inf 10 inf 17 11
10#17 7 19 16 15 17 inf 16 17#11 12 10 7 inf 11 16 inf 15#11 10 12 15 18 10 17
15 inf#0 2 3 7 8 1 6 4 5
inf inf 15 1 2 2 inf 2 inf 8#inf inf 3 6 inf 14 8 inf 9 9#15 3 inf inf 13 17 inf
16 6 7#1 6 inf inf 3 inf 10 1 12 9#2 inf 13 3 inf 4 11 4 inf 8#2 14 17 inf 4 inf
8 2 14 10#inf 8 inf 10 11 8 inf 9 8 8#2 inf 16 1 4 2 9 inf 12 10#inf 9 6 12 inf
14 8 12 inf 6#8 9 7 9 8 10 8 10 6 inf#0 3 7 5 6 1 2 8 9 4
inf 9 7 inf 18 1 6 inf 18 18 8#9 inf 12 13 16 8 6 11 15 inf 15#7 12 inf 2 18 8
11 8 15 16 inf#inf 13 2 inf 16 6 inf 10 14 15 10#18 16 18 16 inf 17 16 inf 18 11
15#1 8 8 6 17 inf 7 11 inf 18 8#6 6 11 inf 16 7 inf 16 15 16 10#inf 11 8 10 inf
11 16 inf 16 15 12#18 15 15 14 18 inf 15 16 inf 20 18#18 inf 16 15 11 18 16 15
20 inf 10#8 15 inf 10 15 8 10 12 18 10 inf#0 5 3 2 7 10 9 4 8 1 6
```

1.2 Part 02: Branch and Bound

First, we implement the brach and bound strategy of TSP problem.

Starting with the distance function which calculates the total distance of a given path in the graph.

[3]:
```
def distance(path, graph):
    """
    The distance of {path} in {graph}
    """

    size = len(graph)
    length = len(path)
```

```
    result = 0 if length < size else graph[path[-1]][0]

    for i in range(length-1):
        result += graph[path[i]][path[i+1]]

    return result
```

The function iscycle checks if the given path forms a Hamiltonian cycle in the provided graph. And a Hamiltonian cycle is a cycle that visits each node exactly once and returns to the starting node.

[4]:
```
def iscycle(path, graph):
    """
    Is {path} a Hamiltonian cycle in {graph}?
    """

    size = len(graph)
    result = set(path) == set(range(size)) # is it a permutation?

    if result:

        result = path[0] == 0 # does it start at 0?

        for i in range(size):

            if not result:
                break

            result = not isinf(graph[path[i]][path[(i+1)%size]]) # is there an␣
↪edge?

    return result
```

The functio nmincost calculates the estimated minimum cost of completing a given path in the graph.It can be used as cost bound used in the implement of branch and bound algorithm.

[5]:
```
def mincost(path, graph):
    """
    The MinCostBound function
    """
    size = len(graph)
    result = distance(path, graph)

    if len(path) != size:

        for i in (set(range(size))-set(path[:-1])):
            result += min(graph[i]);
```

```
    return result
```

The function sort_tries sorts a list of possible next nodes 'tries' based on their estimated cost of extending the current 'path' in the given 'graph'. The sorting uses function 'mincost' to prioritize nodes that lead to lower overall cost paths.

[6]:
```
def sort_tries(graph, tries, path, bound=mincost):
    """
    Sort possible next nodes based on estimated cost.
    """
    return sorted(tries, key=lambda x: bound(path + [x], graph))
```

The following is the completement of the branch an bound algorithm which solves the Traveling Salesman Problem (TSP) using a backtracking algorithm with branch and bound.### Function Introduction:

Key Features: 1. **Parameters**: - graph: An adjacency matrix representing the graph where edge weights indicate distances. - path: A list of visited nodes forming the current partial path (default starts at node 0). - shortest: The current shortest cycle cost (initialized to infinity). - bound: A bound function (mincost) that estimates the lower bound cost of completing a path. 2. **Logic**: - Checks if the current path forms a valid cycle (by iscycle function). If valid, the path is returned. - Feasible next nodes 'tries are determined by checking if adding a node keeps the estimated cost below the current shortest cost. - The feasible nodes are sorted using a function 'sort_tries' to prioritize promising paths. - Each candidate path is explored recursively, updating the shortest path and cost if a better solution is found. 3. **Output**: - Returns the shortest cycle found during the search.

[7]:
```
def tsp_branchAndBound(graph: list, path=[0], shortest=inf, bound=mincost) ->␣
↪list:
    """
    Solve TSP using backtracking with branch and bound.
    """
    size = len(graph)
    result = path if iscycle(path, graph) else []  # If path is a valid cycle,␣
↪return it

    tries = []  # List to store possible next nodes

    # Add all feasible targets to tries
    for target in (set(range(size)) - set(path)):

        if bound(path + [target], graph) < shortest:
            tries.append(target)

    sort_tries(graph, tries, path)  # Sort tries (no return value used here)

    # Explore each try recursively
    for i in range(len(tries)):
```

```
        tour, cost = [], inf

        if bound(path + [tries[i]], graph) < shortest:
            tour = tsp_branchAndBound(graph, path + [tries[i]], shortest)
            cost = distance(tour, graph)

            if tour and cost < shortest:  # Update shortest tour if a better␣
↪one is found
                shortest = cost
                result = tour

    return result
```

1.3 Part 03: Approximation Algorithm

Next, we implement the approximation strategy of TSP problem.

The function check_undirected checks if a given graph (represented as an adjacency matrix) is undirected. An undirected graph has symmetric adjacency, meaning the value at $graph[i][j]$ must be equal to the value at $graph[j][i]$ for all pairs of indices i and j.

[8]:
```
def check_undirected(graph: list):
    """
    Checks if a given graph is undirected.
    """
    for i in range(len(graph)):
        for j in range(len(graph)):
            if graph[i][j] != graph[j][i]:
                return False
    return True
```

The function satisfies_trangle_inequality verifies whether a graph (represented as an adjacency matrix) satisfies the triangle inequality. The triangle inequality states that for any three distinct nodes i, j, and k in the graph, the direct path between i and k should not be longer than the sum of the paths through an intermediate node j

i.e.

$$graph[i][k] \leq graph[i][j] + graph[j][k]$$

The function skips unreachable paths (represented by infinite values).

[9]:
```
def satisfies_triangle_inequality(graph):
    """
    Checks if the given graph satisfies the triangle inequality.
    """
    n = len(graph)

    for i in range(n):
```

```python
        for j in range(n):
            for k in range(n):
                if i != j and j != k and i != k:  # Ensure three distinct nodes
                    if graph[i][j] == inf or graph[j][k] == inf or graph[i][k] == inf:
                        continue  # Skip unreachable paths
                    if graph[i][k] > graph[i][j] + graph[j][k]:  # Check inequality
                        return False
    return True
```

The function min_spanning_tree constructs a Minimum Spanning Tree (MST) for the input undirected graph, represented as an adjacency matrix, using a greedy algorithm (similar to Prim's algorithm) in the following steps: 1. **Input Validation**: It first checks if the graph is undirected using the check_undirected function. If not, a 'ValueError' will be raised. 2. **Initialization**: The MST is initialized as an empty adjacency matrix (result), and the algorithm starts with the first node in the MST set T. 3. **Edge Selection**: The algorithm repeatedly identifies the minimum-weight edge that connects a node in the MST set T to a node not yet in the set. 4. **Output**: The resulting adjacency matrix (result) represents the MST of the input graph.

This function ensures that all nodes are connected with the minimum total edge weight while maintaining the properties of a spanning tree.

def $TSP\_mst(G : N^{n\times n})$**:**

    require: G is directed.

    optP $\leftarrow [[0] \times n]$

    choice $\leftarrow [0]$

    while $len(choice) \neq n$ **do**

        $i,\ j \leftarrow 0, 0$

        for $s \in choice$ **do**

            for $t \in (\vec{n} - choice)$ **do**

                if $G[s][t] < G[i][j]$ **then**

                    $i,j \leftarrow s, t$

        $opt[i][j],\ opt[j][i] \leftarrow G[i][j], G[j][i]$

        $choice \leftarrow choice + [j]$

        return $optP$

[10]:
```python
def min_spanning_tree(graph: list) -> list:
    """
    Constructs a minimum spanning tree (MST) of the given graph using a greedy algorithm.
    """
    if not check_undirected(graph):
        raise ValueError("The input graph is not undirected!")

    n = len(graph)
```

```
    result = [[inf for _ in range(n)] for _ in range(n)]  # Initialize MST␣
↪matrix
    T = [0]  # Start from the first node (arbitrarily chosen)

    # Iterate until all nodes are included in the MST
    while set(T) != set(range(n)):
        i, j = 0, 0
        # Find the minimum weight edge connecting the MST set (T) to the␣
↪remaining nodes
        for s in T:
            for t in (set(range(n)) - set(T)):
                if graph[s][t] < graph[i][j]:
                    i, j = s, t
        # Add the edge to the MST
        result[i][j], result[j][i] = graph[i][j], graph[j][i]
        T.append(j)  # Add the new node to the MST set

    return result
```

The function depth_first_search performs a depth-first traversal of a tree represented as an adjacency matrix. It starts at a specified node (default is the first node, 0) and explores as far as possible along each branch before backtracking.

Key features: 1. **Recursive Traversal**: The function uses recursion to explore all unvisited neighbors of the current node. 2. **Parameters**: - tree: An adjacency matrix where edges are represented by weights, and 'float('inf')' indicates no direct connection. - path: A list that tracks the traversal path, initialized with the starting node (0 by default). 3. **Traversal Logic**: For each neighbor of the current node, if the node is unvisited and an edge exists, the function recurses into that neighbor. 4. **Output**: Returns a list result containing the nodes visited in depth-first order.

This function is useful for exploring tree-like structures systematically.

def DFS($\mathrm{T} : \mathbb{N}^{n\times n}, \sigma$: seq of $\vec{n}$)**:**
  require: T is a tree
  $\pi \leftarrow \sigma$;
  $s \leftarrow \sigma[-1]$;
  for $t \in \vec{n} \setminus \sigma$ **do**
    if $\mathrm{T}[s][t] \neq \infty$ **then**
      $\pi \leftarrow \pi +\!\!+ [t]$;
      $\pi \leftarrow \mathrm{DFS}(\mathrm{T}, \pi)$;
  return π;

[11]:
```
def depth_first_search(tree: list, path=[0]):
    """
    Performs a depth-first traversal on the given tree.
```

```python
    """
    n = len(tree)
    result = path
    s = path[-1]  # Current node

    # Explore all unvisited neighbors of the current node
    for t in (set(range(n)) - set(path)):
        if t not in path and tree[s][t] != float('inf'):  # Check if edge exists
            result.append(t)
            result = depth_first_search(tree, result)

    return result
```

The function remove_duplicate_nodes removes duplicate nodes from a given traversal path while preserving the original order of the nodes.

Logic: - Iterates through the 'path' and maintains a list 'seen' of nodes that have already been added to the result. - If a node is not in 'seen', it is appended to the result 'tsp_path' and marked as seen.

Output: Returns a list 'tsp_path' containing each node from the input path exactly once, in the order of their first appearance.

```python
[12]: def remove_duplicate_nodes(path):
    """
    Removes duplicate nodes from a given path while maintaining the order.
    """
    tsp_path, seen = [], []  # Initialize the resulting path and a list to␣
 ↪track seen nodes

    for node in path:
        # If the node hasn't been seen before, add it to the result and mark it␣
 ↪as seen
        if node not in seen:
            tsp_path.append(node)
            seen.append(node)

    return tsp_path
```

The following is the completement of the approximation algorithm who provides an approximate solution to the Traveling Salesman Problem (TSP) for a graph represented as an adjacency matrix. It utilizes the Minimum Spanning Tree (MST)-based approach to generate a near-optimal TSP path.

Key steps: 1. **Input Validation**: The function ensures the graph satisfies the triangle inequality. If not, a 'ValueError' is raised. 2. **Construct MST**: It computes the Minimum Spanning Tree (MST) of the graph using the 'min_spanning_tree' function. 3. **Depth-First Traversal**: Performs a depth-first traversal of the MST using the 'depth_first_search' function to generate a traversal path. 4. **TSP Path Construction**: Removes duplicate nodes from the traversal path to produce a valid TSP route, typically ensuring all nodes are visited once. 5. **Output**: Returns a list 'tsp_path' representing the approximate solution to the TSP. This approach is efficient and provides a reasonable solution, especially when the triangle inequality is satisfied.

[13]:
```
def tsp_approximation(graph: list):
    """
    Approximates a solution to the Traveling Salesman Problem (TSP) using a␣
↪minimum spanning tree (MST).
    """
    if not satisfies_triangle_inequality(graph):
        raise ValueError("The input matrix does not satisfy the triangle␣
↪inequality!")

    # 1. Construct the minimum spanning tree
    tree = min_spanning_tree(graph)

    # 2. Perform a depth-first traversal
    traversal_path = depth_first_search(tree)

    # 3. Remove duplicate nodes to create a valid TSP path
    tsp_path = remove_duplicate_nodes(traversal_path)

    return tsp_path
```

1.4 Part 04: Comparison of Running Time

Finally, we make a comparison of running times of three algorithms implemented above. - general - bounding - branch and bound - approximation

To begin with, it is necessary for us to implement a test case builder.

[14]:
```
def build_tests(fname: list) -> list:
    '''
    Return a list consisting of all test cases in file {fname}.
    '''

    file  = open(fname, "r")
    lines = file.read().split("\n")
    file.close()

    tests = []

    for line in lines:

        test = line.split("#")

        nodes_num = len(test) - 1

        graph = []
```

```
        sol = list(map(int, test[nodes_num].split()))


        for i in range(nodes_num):

            row = []
            currRow = list(test[i].split(' '))

            for i in range(nodes_num):

                currEdge = currRow[i]

                if currEdge == 'inf': row.append(inf)
                else: row.append( int(currEdge) )

            graph.append( row )

        tests += [(graph,sol)]

    return tests
```

After that, We implement the following function to make a comparison of all variants of TSP backtracking algorithms.

Moreover, to better visualize the comparision, we can use matplot to draw a graph that shows the running times of these algorithms.

[15]:
```
import time
import matplotlib.pyplot as plt

def compare_tsp_algos( fname: str, algos: list, names: list, if_plt: bool) ->␣
↪None:

    count = 0
    tests = build_tests( fname )
    nodes_numbers = []
    running_times = []

    for test in tests:

        graph, sol = test
        nodes_num = len(graph)

        print('------------------ ' + f'Test No.{count+1}' + '␣
↪------------------\n')
        print(f'graph: ')
        for row in graph:
            print(f'       {row}')
```

```
        print(f'\nsolution:')
        print(f'       {sol}\n\n')

        nodes_numbers.append( nodes_num )
        case_running_times = []

        for i in range( len(algos) ):

            startT   = time.process_time()
            sol_algo = algos[i]( graph )
            endT     = time.process_time()
            elapT    = endT - startT

            minDist      = distance(sol, graph)
            minDist_algo = distance(sol_algo, graph)

            if  names[i] == 'TSP-Approximation': # TSP Algorithms

                print(
                    f'algorithm:    {names[i]}',
                    f'runningTime:  {elapT:.10f}',
                    f'distance:     {distance(sol_algo, graph)}',
                    f'ratio:        {minDist / minDist_algo:.2f}',
                    sep = '\n',
                    end = '\n\n'
                )

            else: # TSP Approximation

                correctness  = True if minDist == minDist_algo else False
                case_running_times.append( elapT )

                print(
                    f'algorithm:    {names[i]}',
                    f'correctness:  {correctness}',
                    f'runningTime:  {elapT:.10f}',
                    f'distance:     {distance(sol_algo, graph)}',
                    sep = '\n',
                    end = '\n\n'
                )

        running_times.append( case_running_times )
        count += 1


    if if_plt:
```

```
        running_times = list(zip(*running_times))  # Transpose for easier␣
↪plotting
        plt.figure(figsize=(10, 6))
        for i in range(len(algos) - 1):
            plt.plot(nodes_numbers, running_times[i], label=names[i],␣
↪marker='o')

        plt.xlabel('Nodes Number')
        plt.ylabel('Running Time (seconds)')
        plt.title('Running Time Comparison of TSP Algorithms')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
```

Finally, we excecute the following code snippet to finish.

[16]:
```
from tsp_bounding import tsp_bounding
from tsp_general import tsp_general

algos = [
    tsp_general,
    tsp_bounding,
    tsp_branchAndBound,
    tsp_approximation
]

algoNames = [
    'TSP-General',
    'TSP-Bounding',
    'TSP-Brand-and-Bound',
    'TSP-Approximation'
]

testFile = 'tsp_test_cases'

compare_tsp_algos(testFile, algos, algoNames, True)
```

```
------------------ Test No.1 ------------------

graph:
       [inf, 4, inf, 9]
       [4, inf, 2, 8]
       [inf, 2, inf, 8]
       [9, 8, 8, inf]

solution:
       [0, 1, 2, 3]
```

```
algorithm:    TSP-General
correctness:  True
runningTime:  0.0000280000
distance:     23

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0000230000
distance:     23

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0000720000
distance:     23

algorithm:    TSP-Approximation
runningTime:  0.0000290000
distance:     23
ratio:        1.00

------------------ Test No.2 ------------------

graph:
       [inf, 13, 8, 2, 16]
       [13, inf, 20, 13, 5]
       [8, 20, inf, 8, 17]
       [2, 13, 8, inf, inf]
       [16, 5, 17, inf, inf]

solution:
       [0, 1, 4, 2, 3]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0000820000
distance:     45

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0000670000
distance:     45

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0001830000
```

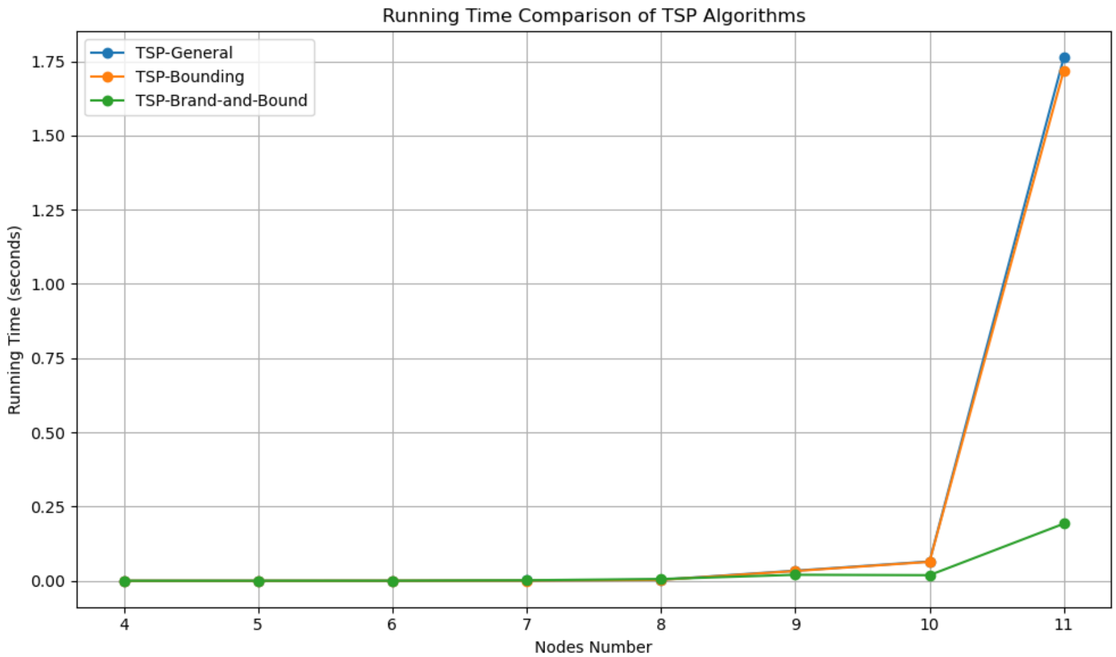
```
distance:     45

algorithm:    TSP-Approximation
runningTime:  0.0000360000
distance:     41
ratio:        1.10

------------------ Test No.3 ------------------

graph:
       [inf, 19, 19, inf, 5, inf]
       [19, inf, inf, 14, inf, inf]
       [19, inf, inf, 13, 16, 17]
       [inf, 14, 13, inf, 18, 13]
       [5, inf, 16, 18, inf, 20]
       [inf, inf, 17, 13, 20, inf]

solution:
       [0, 1, 3, 5, 2, 4]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0001550000
distance:     84

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0003020000
distance:     84

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0003270000
distance:     84

algorithm:    TSP-Approximation
runningTime:  0.0000540000
distance:     inf
ratio:        0.00

------------------ Test No.4 ------------------

graph:
       [inf, 15, 20, 1, 11, 10, 14]
       [15, inf, inf, 16, 15, 5, 13]
       [20, inf, inf, 20, 15, 10, 8]
       [1, 16, 20, inf, inf, inf, 15]
```

```
        [11, 15, 15, inf, inf, 10, 14]
        [10, 5, 10, inf, 10, inf, inf]
        [14, 13, 8, 15, 14, inf, inf]

solution:
        [0, 3, 1, 5, 2, 6, 4]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0009520000
distance:     65

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0007930000
distance:     65

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0018480000
distance:     65

algorithm:    TSP-Approximation
runningTime:  0.0000770000
distance:     inf
ratio:        0.00

------------------ Test No.5 ------------------

graph:
        [inf, inf, 2, 20, inf, 18, inf, 12]
        [inf, inf, 14, 8, inf, inf, 16, 13]
        [2, 14, inf, 20, 1, 20, 7, 11]
        [20, 8, 20, inf, 19, 3, 18, 10]
        [inf, inf, 1, 19, inf, 19, inf, 12]
        [18, inf, 20, 3, 19, inf, 19, 10]
        [inf, 16, 7, 18, inf, 19, inf, 10]
        [12, 13, 11, 10, 12, 10, 10, inf]

solution:
        [0, 2, 4, 7, 6, 1, 3, 5]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0035850000
distance:     70
```

```
algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0031290000
distance:     70

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0059780000
distance:     70

algorithm:    TSP-Approximation
runningTime:  0.0001100000
distance:     inf
ratio:        0.00

------------------ Test No.6 ------------------

graph:
       [inf, 14, 2, 5, 9, 3, 17, 11, 11]
       [14, inf, 13, inf, 20, 17, 7, 12, 10]
       [2, 13, inf, 3, inf, 4, 19, 10, 12]
       [5, inf, 3, inf, 13, inf, 16, 7, 15]
       [9, 20, inf, 13, inf, 10, 15, inf, 18]
       [3, 17, 4, inf, 10, inf, 17, 11, 10]
       [17, 7, 19, 16, 15, 17, inf, 16, 17]
       [11, 12, 10, 7, inf, 11, 16, inf, 15]
       [11, 10, 12, 15, 18, 10, 17, 15, inf]

solution:
       [0, 2, 3, 7, 8, 1, 6, 4, 5]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0340870000
distance:     72

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0330040000
distance:     72

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0202640000
distance:     72
```

```
algorithm:    TSP-Approximation
runningTime:  0.0001670000
distance:     inf
ratio:        0.00

------------------ Test No.7 ------------------

graph:
       [inf, inf, 15, 1, 2, 2, inf, 2, inf, 8]
       [inf, inf, 3, 6, inf, 14, 8, inf, 9, 9]
       [15, 3, inf, inf, 13, 17, inf, 16, 6, 7]
       [1, 6, inf, inf, 3, inf, 10, 1, 12, 9]
       [2, inf, 13, 3, inf, 4, 11, 4, inf, 8]
       [2, 14, 17, inf, 4, inf, 8, 2, 14, 10]
       [inf, 8, inf, 10, 11, 8, inf, 9, 8, 8]
       [2, inf, 16, 1, 4, 2, 9, inf, 12, 10]
       [inf, 9, 6, 12, inf, 14, 8, 12, inf, 6]
       [8, 9, 7, 9, 8, 10, 8, 10, 6, inf]

solution:
       [0, 3, 7, 5, 6, 1, 2, 8, 9, 4]


algorithm:    TSP-General
correctness:  True
runningTime:  0.0650860000
distance:     45

algorithm:    TSP-Bounding
correctness:  True
runningTime:  0.0638790000
distance:     45

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.0189270000
distance:     45

algorithm:    TSP-Approximation
runningTime:  0.0002090000
distance:     inf
ratio:        0.00

------------------ Test No.8 ------------------

graph:
       [inf, 9, 7, inf, 18, 1, 6, inf, 18, 18, 8]
       [9, inf, 12, 13, 16, 8, 6, 11, 15, inf, 15]
```

```
        [7, 12, inf, 2, 18, 8, 11, 8, 15, 16, inf]
        [inf, 13, 2, inf, 16, 6, inf, 10, 14, 15, 10]
        [18, 16, 18, 16, inf, 17, 16, inf, 18, 11, 15]
        [1, 8, 8, 6, 17, inf, 7, 11, inf, 18, 8]
        [6, 6, 11, inf, 16, 7, inf, 16, 15, 16, 10]
        [inf, 11, 8, 10, inf, 11, 16, inf, 16, 15, 12]
        [18, 15, 15, 14, 18, inf, 15, 16, inf, 20, 18]
        [18, inf, 16, 15, 11, 18, 16, 15, 20, inf, 10]
        [8, 15, inf, 10, 15, 8, 10, 12, 18, 10, inf]

solution:
        [0, 5, 3, 2, 7, 10, 9, 4, 8, 1, 6]


algorithm:    TSP-General
correctness:  True
runningTime:  1.7621630000
distance:     95

algorithm:    TSP-Bounding
correctness:  True
runningTime:  1.7192330000
distance:     95

algorithm:    TSP-Brand-and-Bound
correctness:  True
runningTime:  0.1922970000
distance:     95

algorithm:    TSP-Approximation
runningTime:  0.0002940000
distance:     67
ratio:        1.42
```


Running Time Comparison of TSP Algorithms
TSP-General
TSP-Bounding
TSP-Brand-and-Bound
Running Time (seconds)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
4
5
6
7
8
9
10
11
Nodes Number