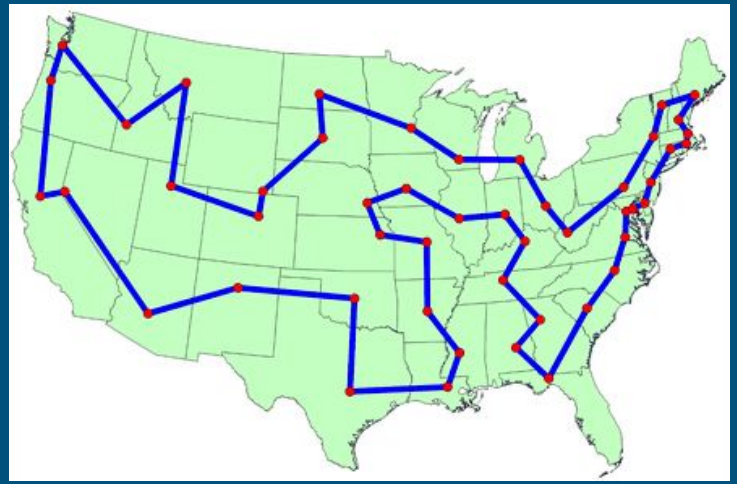
Traveling Salesman Problem

Alex Karapetkov

What is the Traveling Salesman Problem (TSP)?

- Asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”
- NP-hard problem
- Challenge lies in finding optimal route among the exponentially many possibilities as more cities are added while minimizing total distance traveled



Why is the TSP important?

- Logistics and Transportation
  - Optimizing delivery routes to minimize fuel consumption, time, and distance traveled
- Telecommunications
  - Designing efficient network layouts for data routing, fiber optic cable installation, etc to minimize latency or cost
- Chip Manufacturing
  - Planning most efficient path for laser to follow when etching integrated circuits to minimize manufacturing time and cost
- Vehicle Routing
  - Optimizing routes for ride-sharing services and public transportation to reduce overall travel time and efficiently serve passengers

TSP Decision and Optimization Formulations

- Decision problem:
  - “Given a weighted graph G and an integer D, is there a Hamiltonian cycle (a tour that visits each city exactly once) with a total weight less than or equal to D?”
  - Helps in understanding whether a solution to the optimization problem exists with a given constraint (distance threshold D)
  - NP - Complete

- Optimization Problem:
  - “Given a weighted graph G, find a Hamiltonian cycle with the minimum total weight.”
  - Seeks to find best solution among all possible tours
  - NP - Hard

Certifier Process

- Given instance of decision problem which includes graph and distance threshold D, certifier process verifies whether a proposed solution satisfies two conditions:
  - Hamiltonian cycle: proposed solution must form a Hamiltonian cycle
  - Total distance: total distance of the tour must be less than or equal to the given threshold D

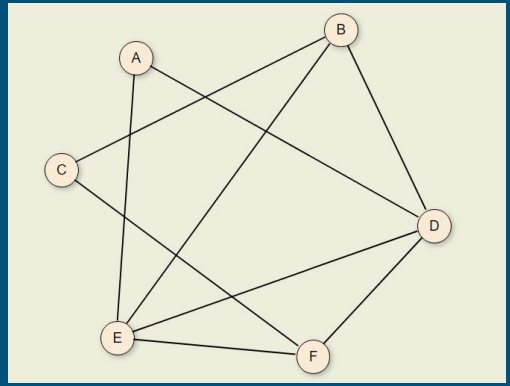
- Certifier process verifies these conditions in polynomial time
  - Hamiltonian cycle verification: Must traverse edges of tour to determine if the tour forms hamiltonian cycle; requires at most $O(n)$ operations with n being number of cities
  - Total distance calculation: must sum distances of edges in tour to get total distance; since number of edges is at most n (n is number of cities), can be done in $O(n)$ operations

- Both conditions can be verified in polynomial time so certifier process of TSP decision version is polynomial

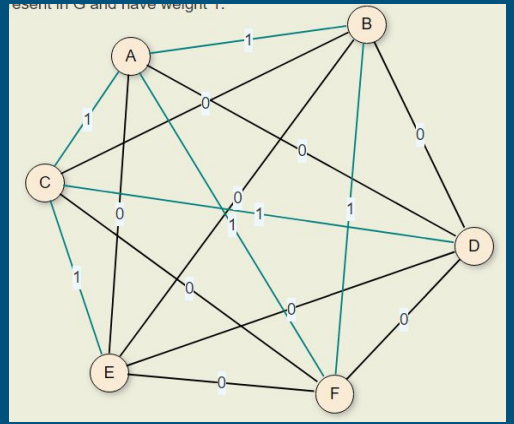
Reduction from known NP-hard problem

- The Hamiltonian Cycle problem (HSP) can be reduced to an instance of TSP in polynomial time
  - For a given graph *G = (V, E)*, the HSP is to find whether *G* contains a Hamiltonian cycle
- To reduce HSP to TSP, complete the graph *G* by adding edges between all pairs of vertices that were not connected in *G*; let this new graph by *G'*
  - For edges in *G'* that were also present in *G*, assign a weight 0; assign weight 1 to all other edges

G



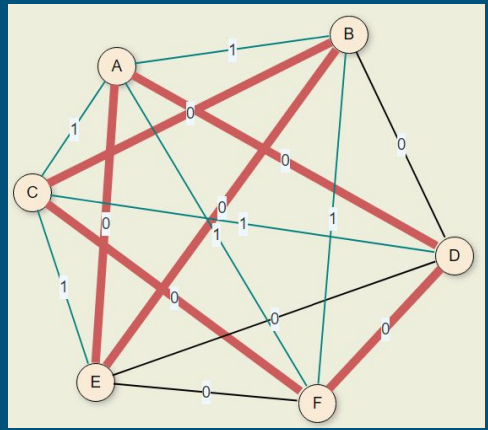
G'

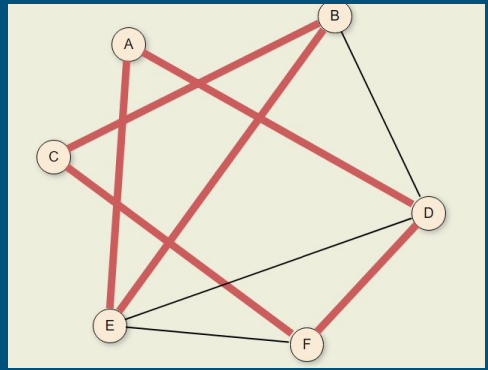


Reference:
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/hamiltonianCycle_to_TSP.html

Reduction from known NP-hard problem

- If there is a cycle that passes through all vertices exactly once, and has length <= 0 in graph *G'*, the cycle contains only edges that were originally present in graph *G*; therefore there is a hamiltonian cycle in *G*
  - New edges in *G'* have weight 1 and cannot be part of cycle with length <= 0
- If there is a hamiltonian cycle in *G*, it forms a cycle in *G'* with length = 0, since all edges have weight 0; therefore there exists a solution for TSP in *G'* with length <= 0





Reference:
https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/hamiltonianCycle_to_TSP.html

Approximation Solution Pseudocode

- Key idea: utilize the minimum spanning tree to find the shortest path
- Section with Prim's algorithm is greedy; solution is overall deterministic

APPROX-TSP-TOUR(*G, c*)

1) select a vertex $r \in G.V$ to be a "root" vertex
2) compute a minimum spanning tree *T* for *G* from root *r*

   using *MST - PRIM(G, c, r)*

3) let *H* be a list of vertices, ordered according to when they are first visited

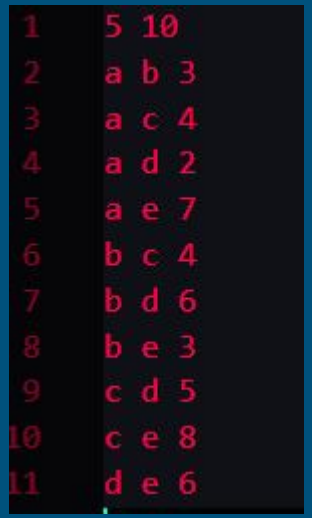
   in a preorder tree walk of *T*

4) return the Hamiltonian cycle *H*

Referenced from approximation algorithms lecture notes

When Approximation is not Optimal

Input:



Approximation:

```
Running cs412_tsp_approx.py on test_cases/testcase2
Input file contents (test_cases/testcase2_input.txt):
Program output:
13
a d b e c a
13
Execution time: .025762770 seconds
```

Optimal:

```
Running cs412_tsp_optimal.py on test_cases/testcase2
Input file contents (test_cases/testcase2_input.txt):
Program output:
19
b c a d e b
19
Execution time: .025762770 seconds
```

Exact Solution Pseudocode

TWO - OPT(distance_matrix)

- Initialize best route as a list containing indices of cities in order they are visited
- Calculate the total distance of the initial tour
- Repeat until no more improvements can be made:
    - For each pair of cities(i, k):
        - Generate a new route by performing a **2-opt swap** on the current best route
        - Calculate the total distance of the new route
        - If the new route distance is less than the current best:
            - Update the current best route
            - Update the current best distance
- Return the best route and its corresponding total distance

TOUR - DISTANCE (tour, distance_matrix)

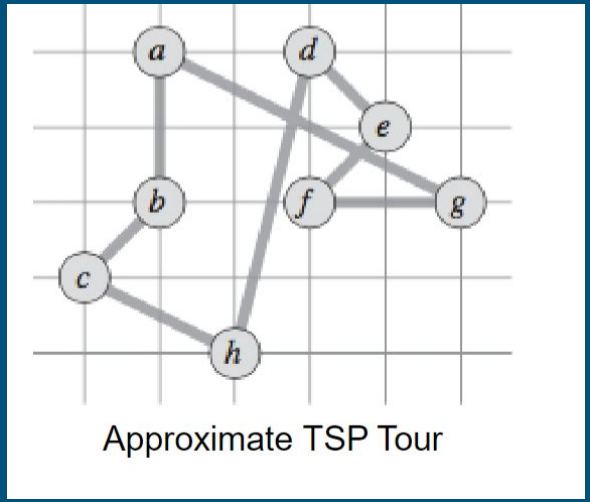
- Calculate the total distance of the tour by summing distances between consecutive cities
- Add distance from last city back to starting city to complete tour
- Return the total distance

TWO - OPT - SWAP (route, i, k)

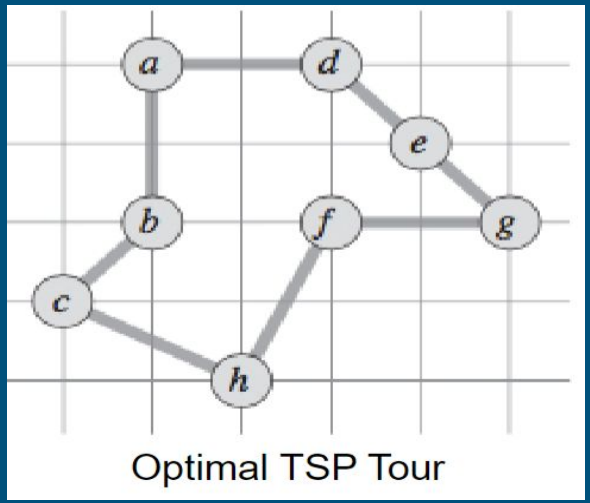
- Copy the current route
- Reverse the order of nodes between indices i and k
- Return the new route

Performance Analysis

- C: tour cost of approximate TSP tour
- C*: tour cost of optimal TSP tour
- C ≤ 2C*: 2-approximation algorithm



Approximate TSP Tour



Optimal TSP Tour

Reference: approximation algorithms lecture notes

Example Inputs and Outputs

Test case #2

Input:

5 10
a b 3
a c 4
a d 2
a e 7
b c 4
b d 6
b e 3
c d 5
c e 8
d e 6

Test case #3

Input:

4 6
a b 10
b c 35
a c 15
b d 25
c d 30
a d 20

Output:

Test case #2:

Approx:

13
a d b e c a

Exact:

19
b c a d e b

Test case #3:

Approx:

50
a b c d a

Exact:

80
b a c d b

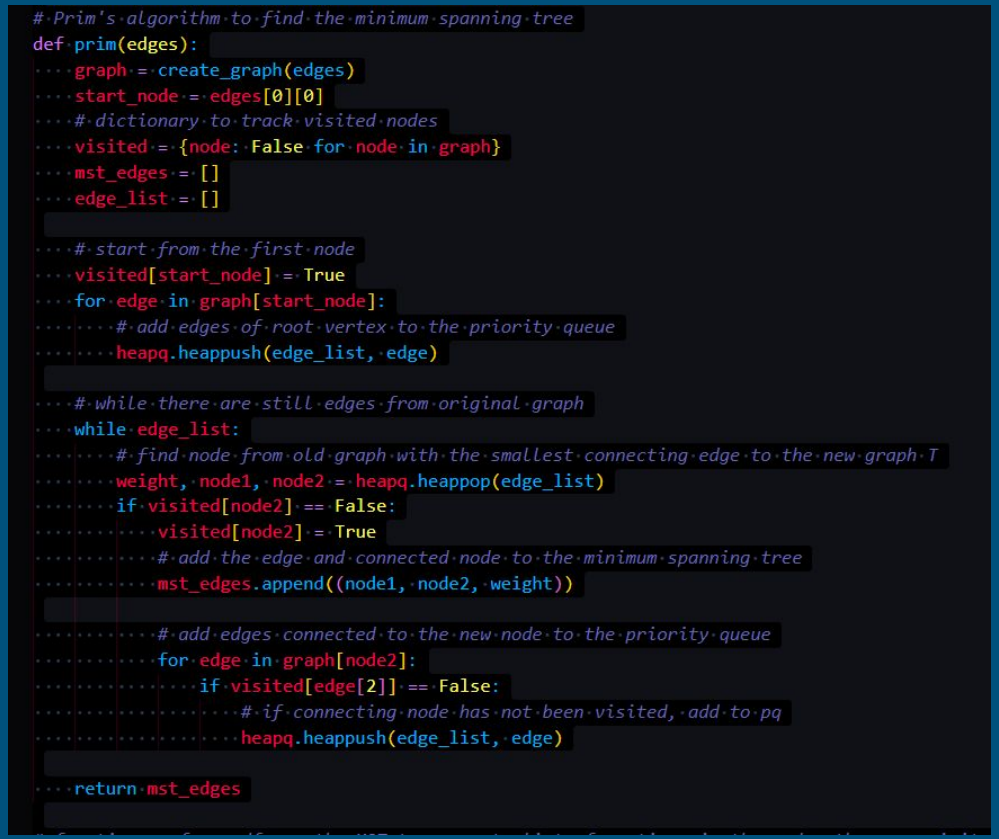
Analytical Runtime Analysis

Approximation solution:

- Create_graph function iterates over each edge once; has time complexity *O(m)*, where *m* is the number of edges
- Prim function iterates over all edges in graph in the worst case (*O(m)* time complexity); also uses a priority queue which can perform insertion and removal in *O(log n)* time where *n* is number of elements in queue
  - Overall complexity is *O(m log m)*
- Preorder_walk function performs DFS of MST; in worst case, each node and edge may be visited once, resulting in *O(n + m)* runtime
- Approx_tsp_tour function iterates over each node in walk and each connected edge; has time complexity of *O(n)*

Analytical Runtime Analysis

Section that drives highest order term is Prim's algorithm: *O(m log m)* runtime



```python
# Prim's algorithm to find the minimum spanning tree
def prim(edges):
    graph = create_graph(edges)
    start_node = edges[0][0]
    # dictionary to track visited nodes
    visited = {node: False for node in graph}
    mst_edges = []
    edge_list = []

    # start from the first node
    visited[start_node] = True
    for edge in graph[start_node]:
        # add edges of root vertex to the priority queue
        heapq.heappush(edge_list, edge)

    # while there are still edges from original graph
    while edge_list:
        # find node from old graph with the smallest connecting edge to the new graph T
        weight, node1, node2 = heapq.heappop(edge_list)
        if visited[node2] == False:
            visited[node2] = True
            # add the edge and connected node to the minimum spanning tree
            mst_edges.append((node1, node2, weight))

            # add edges connected to the new node to the priority queue
            for edge in graph[node2]:
                if visited[edge[2]] == False:
                    # if connecting node has not been visited, add to pq
                    heapq.heappush(edge_list, edge)

    return mst_edges
```

Analytical Runtime Analysis

Optimal Solution:

- Constructing the distance matrix involves nested loops and results in a time complexity of $O(n^2)$ where n is the number of vertices
- 2-Opt Algorithm:
  - Outer loops runs until no new improvements can be made which can potentially run for a large number of iterations depending on the input
  - Nested loops over all pairs of cities have combined time complexity of $O(n^2)$
  - Call to two_opt_swap has time complexity of $O(n)$
  - Overall, the 2-opt algorithm has a time complexity of $O(n^3)$ where n is the number of cities
- Tour_distance function calculates total distance of a tour by summing distances between consecutive cities; has time complexity of $O(n)$ where n is the number of cities

Analytical Runtime Analysis

Section that drives highest order term is the 2-opt algorithm with a time complexity of $O(n^3)$

```python
# 2-opt algorithm
# distance matrix as input; represents the distance between each pair of cities
def two_opt(dist_matrix):
    num_cities = len(dist_matrix)
    # initialize best_route as list containing indices of cities in order they are visited
    best_route = list(range(num_cities))
    # calculate total distance of initial tour with tour_distance function
    best_distance = tour_distance(best_route, dist_matrix)

    # stay in improvement loop until no more improvements can be made in any iteration
    improvement = True
    while improvement:
        improvement = False
        # iterate over all pairs of cities where i is index of starting city and k is index of ending city
        for i in range(num_cities):
            for k in range(i+1, num_cities):
                # for each pair of cities, generate new route by performing 2-opt swap on current best route
                new_route = two_opt_swap(best_route, i, k)
                # calculate total distance of new route
                new_distance = tour_distance(new_route, dist_matrix)

                # if new route distance is less than current best route distance, update current best route
                if new_distance < best_distance:
                    best_route = new_route
                    best_distance = new_distance
                    # continue iterating
                    improvement = True

    return best_route, best_distance
```