Traveling Salesman

Bryden Mollenauer
Alex Milanese

The Traveling Salesman Problem

The TSP is finding the shortest hamiltonian cycle in a complete undirected graph starting at the given first node and ending at the same node.

Inputs/Outputs

For input the first line is the number of vertices and then the number of edges and then every line after that is an edge going from the first letter to the second letter and then the number is the weight of the edge. The inputs will make an undirected complete graph. For test case generation i chose values from 1-100 and randomized them for each edge weight while running double for loop to add the value to each edge combination.

```
3 3
a b 3
b c 4
a c 5
```

```
12

a b c a
```

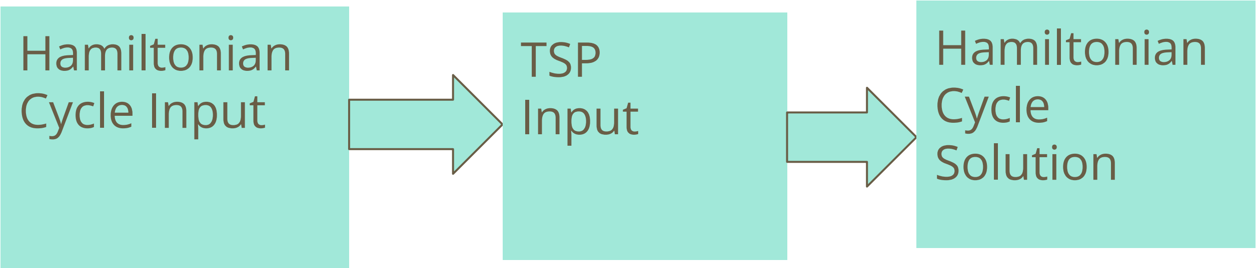
Certifier Process

We know that the certifier process is polynomial because if someone handed me a graph and said this is the shortest path i can prove if it is or isn't with the tsp algorithm and if i increase the sample size by one the run time will double.

Why is the problem important

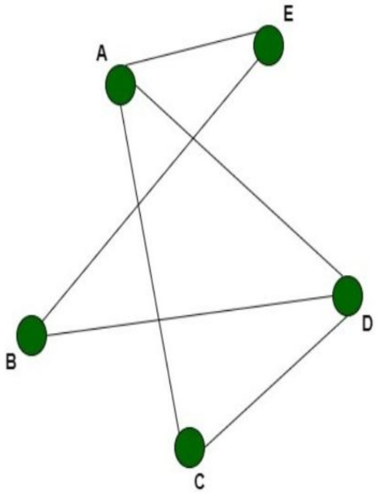
This problem is important so delivery companies can find the shortest path to take on their routes. If amazon can make it so their trucks spend less gas and take a shorter route they can deliver more product a day for less money while making them more money.

Np Hard Reduction

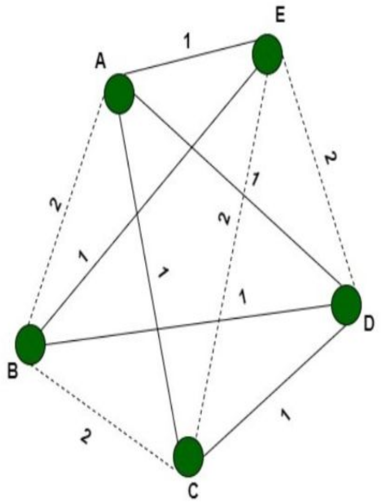
- Our Reduction is taking the hamiltonian cycle problem and reduce it to the Traveling Salesman Problem.
- 

NP hard Reduction

- A known NP hard problem is finding a Hamiltonian Cycle
- For every HC there is a graph that can be converted to a TSP problem just by my making it complete
- This can be done in polynomial time just by adding edges and weights



G =
Hamiltonian cycle {EACDBE}



G' =
TSP {EACDBE}
Cost = 5 (=n)

Code explanation

For reading the input i am creating a Cost matrix which looks like this

```
5 10
a b 1
b c 2
c d 3
d e 4
e a 5
e b 10
a d 11
a c 13
e c 14
d b 15
```

- Input for explanation
- 5 vertices
- 10 edges(v(v-1))/2

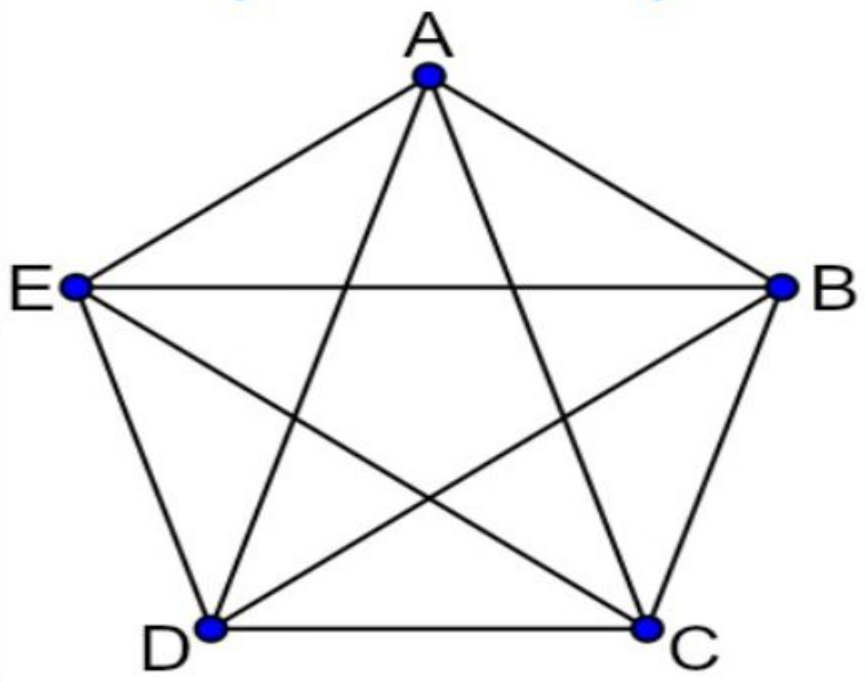
```
Cost Matrix:
[0, 1, 13, 11, 5]
[1, 0, 2, 15, 10]
[13, 2, 0, 3, 14]
[11, 15, 3, 0, 4]
[5, 10, 14, 4, 0]
```

- Each row and column represent the weight of the edge to to get the vertice row and the vertice column

Solution Visualized

- Solution will go around in a circle because that is the shortest path
- The code will check every single subset

```
5 10
a b 1
b c 2
c d 3
d e 4
e a 5
e b 10
a d 11
a c 13
e c 14
d b 15
```



Code for matrix

```
lines = input_str.strip().split("\n")
num_vertices, num_edges = map(int, lines[0].split())

vertex_to_index = {}
vertex_index = 0
cost_matrix = [[float('inf')] * num_vertices for _ in range(num_vertices)]

for line in lines[1:]:
    u, v, cost = line.split()
    if u not in vertex_to_index:
        vertex_to_index[u] = vertex_index
        vertex_index += 1
    if v not in vertex_to_index:
        vertex_to_index[v] = vertex_index
        vertex_index += 1
    i, j = vertex_to_index[u], vertex_to_index[v]
    cost_matrix[i][j] = cost_matrix[j][i] = int(cost)

for i in range(num_vertices):
    cost_matrix[i][i] = 0

return cost_matrix, vertex_to_index
```

- Creates a dictionary of all the vertices and their index
- Create cost matrix size
- Then fill the cost matrix using the vertices index and the weight to the edge between the vertices

Held_Karp

*makes 2 2d arrays

*goes through the mask

* check individual every possible Subset of vertice paths and records the cost of each path in the dp 2d array

* Then calculates the fast path and backtracks through the predecessor 2d array to return the final path

```
def held_karp_tsp_with_path_and_reconstruction(cost_matrix):
    n = len(cost_matrix)
    # Initialize dynamic programming and predecessor matrices.
    dp = np.full((1<<n, n), float('inf'))
    dp[1][0] = 0
    predecessor = np.full((1<<n, n), -1)

    # Iterate through subsets of vertices and calculate minimum costs.
    for mask in range(1, 1<<n):
        for u in range(n):
            if not (mask & (1 << u)):
                continue
            for v in range(n):
                if mask & (1 << v) and u != v: #checks if v is in the subset and u and v d
                    if dp[mask][u] > dp[mask ^ (1 << u)][v] + cost_matrix[v][u]: # checks
                        dp[mask][u] = dp[mask ^ (1 << u)][v] + cost_matrix[v][u]  # update
                        predecessor[mask][u] = v
    # Find the minimum cost to complete the tour.
    mask = (1<<n) - 1
    min_cost = float('inf')
    last_vertex = 0
    for i in range(1, n):
        cost = dp[-1][i] + cost_matrix[i][0] #dp[-1][i] is the fastest cost for the cycle
        if cost < min_cost:
            min_cost = cost
            last_vertex = i

    # Reconstruct the path taken for the minimum cost tour.
    path = [0]  # Start from vertex 0
    while mask != 1:
        path.append(last_vertex)
        mask &= ~(1 << last_vertex)
        last_vertex = predecessor[mask | (1 << last_vertex)][last_vertex]
    path.append(0)  # Complete the cycle by returning to vertex 0


    return min_cost, path
```

The mask

The bit mask is a string of ones and zeros and the max length is the number of vertices in the graph. Each iteration of the loop increase the length of the mask by one.

0b1
1000
1001
1010
1011
1100
1101

1110
1111
10000

- Starts at 0 slowly adds length and records the total cost of traversal of each possible vertice it is currently looking at

The dp tables

- There are 2^n rows in each table
- Each row represents each mask iteration
- To get the shortest path you can Visually look the last row and pick the lowest number and add the cost of the subset path and the weight of the edge to the starting vertice from the ending vertice so it creates a cycle.

```
Iteration for mask (binary): 0b1
  Checking vertex u: 0
  Current DP matrix:
[[inf inf inf inf inf]
 [ 0. inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]
 [inf inf inf inf inf]]
```

```
[[inf inf inf inf inf]
 [ 0. inf inf inf inf]
 [inf inf inf inf inf]
 [inf  1. inf inf inf]
 [inf inf inf inf inf]
 [inf inf 13. inf inf]
 [inf inf inf inf inf]
 [inf 15.  3. inf inf]
 [inf inf inf inf inf]
 [inf inf inf 11. inf]
 [inf inf inf inf inf]
 [inf 26. inf 16. inf]
 [inf inf inf inf inf]
 [inf inf 14. 16. inf]
 [inf inf inf inf inf]
 [inf 16. 19.  6. inf]
 [inf inf inf inf inf]
 [inf inf inf inf  5.]
 [inf inf inf inf inf]
 [inf 15. inf inf 11.]
 [inf inf inf inf inf]
 [inf inf 19. inf 27.]
 [inf inf inf inf inf]
 [inf 21. 17. inf 17.]
 [inf inf inf inf inf]
 [inf inf inf  9. 15.]
 [inf inf inf inf inf]
 [inf 24. inf 15. 20.]
 [inf inf inf inf inf]
 [inf inf 12. 22. 20.]
 [inf inf inf inf inf]
 [inf 14. 18. 20. 10.]]
```

Big O analysis

- The dominant term in my code is the Held Karp algorithm implementation
- The outer loop goes over every possible subset of vertices which is 2^n long
- Then the inner loop goes over every pair of vertices
- So overall the Big O is (n^2 * 2^n) runtime

Wall Clock analysis

With smaller inputs the program runs almost instantaneously but once we get to 15+ vertices the program will start to double in time.

10	0.00603
11	0.00652
12	0.00671
13	0.00868
14	0.01419
15	0.02545
16	0.05117
17	0.10998
18	0.23751
19	0.51356
20	1.51025
21	4.7451
22	9.2581
23	19.1334