Traveling Salesman Problem (TSP) Exact Solution

By: Samuel Snyder & Andrew Bailey

Optimization/ Decision problem

Optimization:

The optimization problem that is being solved is given a graph of nodes and distances between them, what is the shortest possible cycle you can create by visiting every node exactly once and returning to the original node.

Decision:

The decision problem is given a value k and a proposed route, is the total length of the proposed route less then or equal that value k. The answer is yes if that's true and false otherwise.

Why is TSP important

The Traveling Salesman Problem is an important algorithm due to the range of applications it can be applied to. Some examples are:

- Scheduling Problems
- Vehicle Routing
- Computer Wiring
- Drilling of Circuit Boards

This list is not exhaustive but does sum up the main idea in that all these problems need to calculate the shortest distance in a cycle to save the most time.

Polynomial Certification

Steps to show how to check TSP in polynomial time:

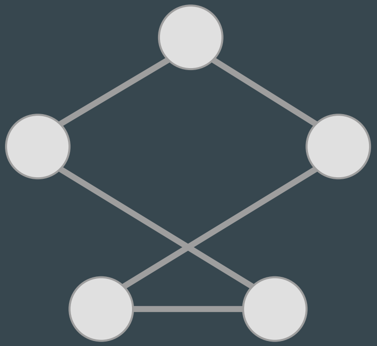
We are given 2 things, first the maximum length of a acceptable route called k. Secondly we get the proposed route to be checked against k.

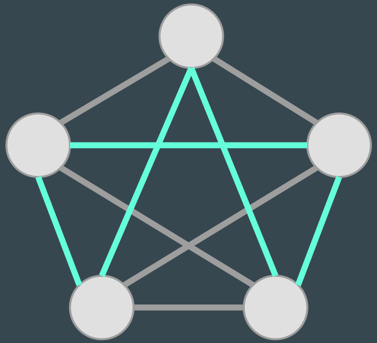
Next we can calculate the length of the proposed solution by looping over the path once and adding each of the edges to the final result. This is done is O(n).

If this cycle's final results length equals or is less than k, we can then say the cycle is correct, otherwise classify it as incorrect

Reduction of Hamiltonian Cycle to TSP

Given a Hamiltonian Cycle we will reduce it to a TSP problem and show how TSP is in NP-Hard

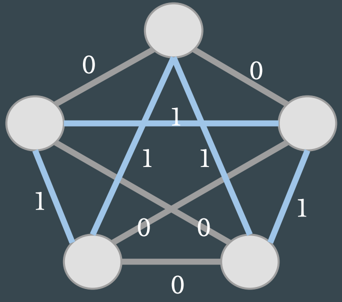

Hamiltonian Cycle Problem Input


Traveling Salesman Problem Input

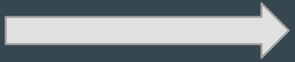
To reduce the problem to traveling salesman you first add any edges that were not present to the original hamiltonian cycle problem input (Continued on next page)

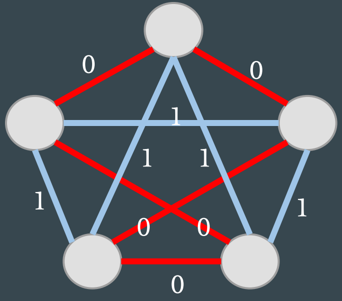
Reduction of Hamiltonian Cycle to TSP Continued

The next step is to give weights to all the edges of the TSP graph. The givens weights will be 1 for every blue line (non-original edge) and 0 for every gray line (original edge). This is so that when running the traveling salesman problem you will know if there is a hamiltonian cycle only if the length of the cycle is 0 because that means TSP only used the original lines to calculate its route.



Traveling Salesman Problem
Input with weights





Traveling Salesman Problem
cycle Solution (red)

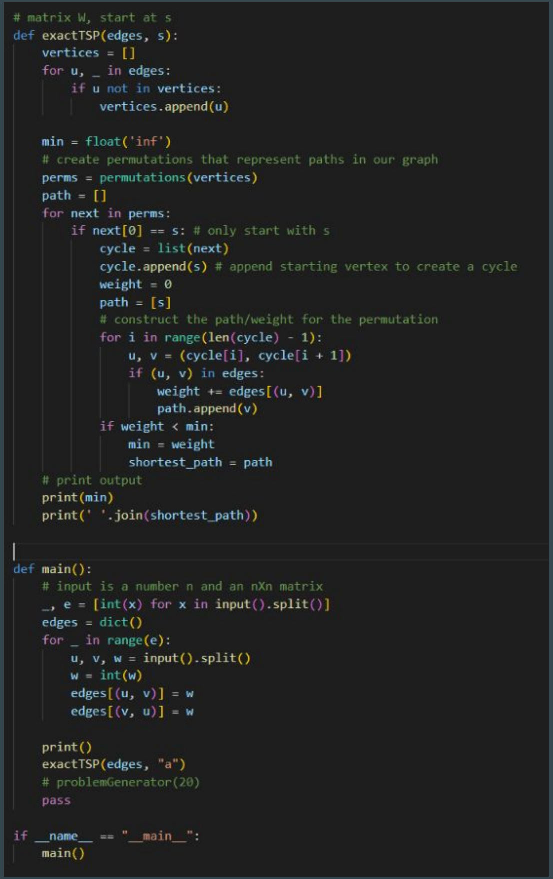
In this case the red lines show the solution to running the traveling salesman problem on the graph and because the length of the path is 0 we know that there is a hamiltonian cycle

Our Exact Solution for TSP

Our solution to the traveling salesman problem comes in two main parts, creating permutations for every vertex and then looping over the edges of the permutation to get the length.

We then compare the length of that permutation against our previous best and if it is shorter we keep that length and store the path.

We decided to store the input of the problem as a dictionary so then we could have the keys be the connecting vertices to the edge.



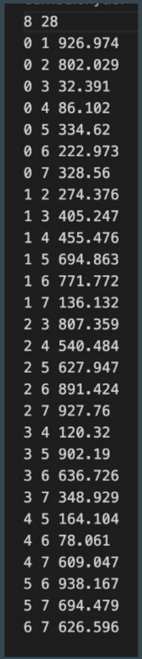
```python
# matrix W, start at s
def exactTSP(edges, s):
    vertices = []
    for u, _ in edges:
        if u not in vertices:
            vertices.append(u)

    min = float('inf')
    # create permutations that represent paths in our graph
    perms = permutations(vertices)
    path = []
    for next in perms:
        if next[0] == s: # only start with s
            cycle = list(next)
            cycle.append(s) # append starting vertex to create a cycle
            weight = 0
            path = [s]
            # construct the path/weight for the permutation
            for i in range(len(cycle) - 1):
                u, v = (cycle[i], cycle[i + 1])
                if (u, v) in edges:
                    weight += edges[(u, v)]
                    path.append(v)
            if weight < min:
                min = weight
                shortest_path = path
    # print output
    print(min)
    print(' '.join(shortest_path))


def main():
    # input is a number n and an nXn matrix
    _, e = [int(x) for x in input().split()]
    edges = dict()
    for _ in range(e):
        u, v, w = input().split()
        w = int(w)
        edges[(u, v)] = w
        edges[(v, u)] = w

    print()
    exactTSP(edges, "a")
    # problemGenerator(20)
    pass

if __name__ == "__main__":
    main()
```

Input and Output

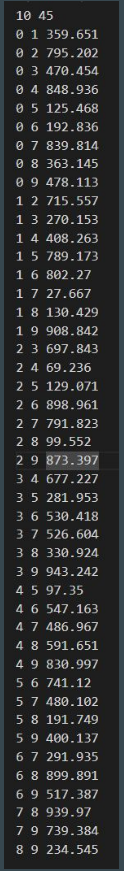
Some examples of what our input and output looked like:

The input has a structure of n x m for the first line which tells how many vertices and edges there are respectively. the rest describes the vertices that share a edge and the distance between them

The output shows what the distance of the exact cycle was and then follows it with the path that it followed. Under that we record the time and then move onto our next test case

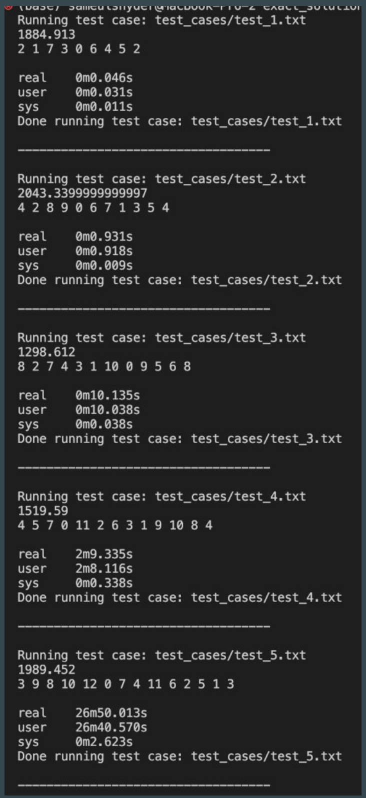


```
8 28
0 1 926.974
0 2 802.029
0 3 32.391
0 4 86.102
0 5 334.62
0 6 222.973
0 7 328.56
1 2 274.376
1 3 405.247
1 4 455.476
1 5 694.863
1 6 771.772
1 7 136.132
2 3 807.359
2 4 540.484
2 5 627.947
2 6 891.424
2 7 927.76
3 4 120.32
3 5 902.19
3 6 636.726
3 7 348.929
4 5 164.104
4 6 78.061
4 7 609.047
5 6 938.167
5 7 694.479
6 7 626.596
```



```
10 45
0 1 359.651
0 2 795.202
0 3 470.454
0 4 848.936
0 5 125.468
0 6 192.836
0 7 839.814
0 8 363.145
0 9 478.113
1 2 715.557
1 3 270.153
1 4 408.263
1 5 789.173
1 6 802.27
1 7 27.667
1 8 130.429
1 9 908.842
2 3 697.843
2 4 69.236
2 5 129.071
2 6 898.961
2 7 791.823
2 8 99.552
2 9 873.397
3 4 677.227
3 5 281.953
3 6 530.418
3 7 526.604
3 8 330.924
3 9 943.242
4 5 97.35
4 6 547.163
4 7 486.967
4 8 591.651
4 9 830.997
5 6 741.12
5 7 480.102
5 8 191.749
5 9 400.137
6 7 291.935
6 8 899.891
6 9 517.387
7 8 939.97
7 9 739.384
8 9 234.545
```

Input for size 8 and 10



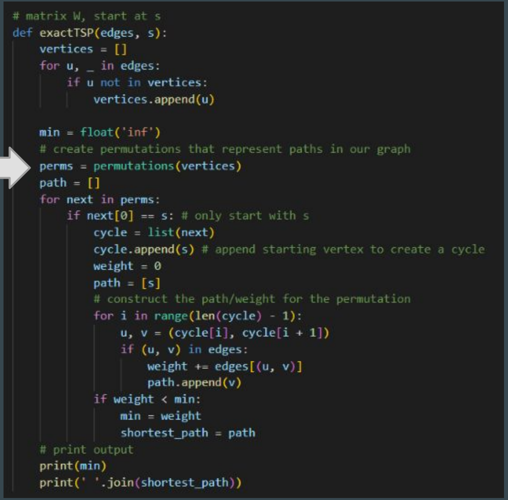
```
Running test case: test_cases/test_1.txt
1884.913
2 1 7 3 0 6 4 5 2

real	0m0.046s
user	0m0.031s
sys	0m0.011s
Done running test case: test_cases/test_1.txt

-----------------------------------

Running test case: test_cases/test_2.txt
2043.3399999999997
4 2 8 9 0 6 7 1 3 5 4

real	0m0.931s
user	0m0.918s
sys	0m0.009s
Done running test case: test_cases/test_2.txt

-----------------------------------

Running test case: test_cases/test_3.txt
1298.612
8 2 7 4 3 1 10 0 9 5 6 8

real	0m10.135s
user	0m10.038s
sys	0m0.038s
Done running test case: test_cases/test_3.txt

-----------------------------------

Running test case: test_cases/test_4.txt
1519.59
4 5 7 0 11 2 6 3 1 9 10 8 4

real	2m9.335s
user	2m8.116s
sys	0m0.338s
Done running test case: test_cases/test_4.txt

-----------------------------------

Running test case: test_cases/test_5.txt
1989.452
3 9 8 10 12 0 7 4 11 6 2 5 1 3

real	26m50.013s
user	26m40.570s
sys	0m2.623s
Done running test case: test_cases/test_5.txt

-----------------------------------
```

Output for all test cases minus the 1000 input one

Big O Runtime Analysis

The big O Runtime of this program is O(n!). This is because when creating our exact solution we decided to do a brute force method where we calculated all the permutations of the vertices in order to see all the paths. By creating all the permutations that will represent the paths in our graph, we created a runtime of O(n!)

The culprit

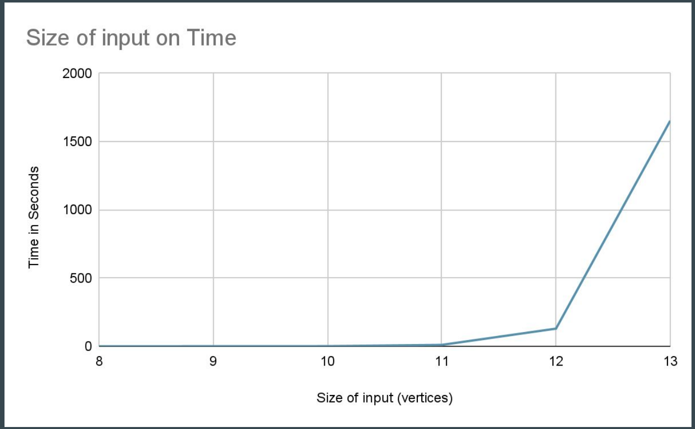


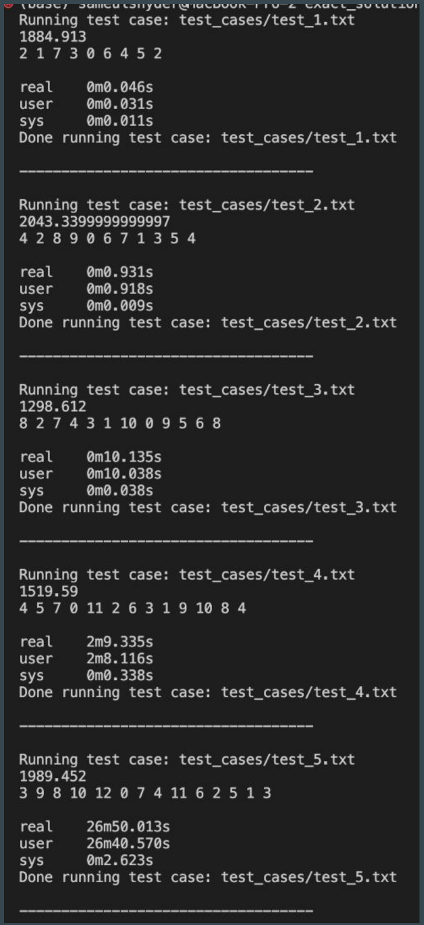
```python
# matrix W, start at s
def exactTSP(edges, s):
    vertices = []
    for u, _ in edges:
        if u not in vertices:
            vertices.append(u)

    min = float('inf')
    # create permutations that represent paths in our graph
    perms = permutations(vertices)
    path = []
    for next in perms:
        if next[0] == s: # only start with s
            cycle = list(next)
            cycle.append(s) # append starting vertex to create a cycle
            weight = 0
            path = [s]
            # construct the path/weight for the permutation
            for i in range(len(cycle) - 1):
                u, v = (cycle[i], cycle[i + 1])
                if (u, v) in edges:
                    weight += edges[(u, v)]
                    path.append(v)
            if weight < min:
                min = weight
                shortest_path = path
    # print output
    print(min)
    print(' '.join(shortest_path))
```

Wall Clock Runtime Analysis

Our graph shows as the input increased marginally, the amount of time it took to complete the runtime was factorial.

We can see this in the beginning stages (8-10 inputs), as it only took a few seconds or less, but once the input increased just by 3 to 13, the time cost went to up to almost 27 minutes





```
Running test case: test_cases/test_1.txt
1884.913
2 1 7 3 0 6 4 5 2

real	0m0.046s
user	0m0.031s
sys	0m0.011s
Done running test case: test_cases/test_1.txt

-----------------------------------

Running test case: test_cases/test_2.txt
2043.3399999999997
4 2 8 9 0 6 7 1 3 5 4

real	0m0.931s
user	0m0.918s
sys	0m0.009s
Done running test case: test_cases/test_2.txt

-----------------------------------

Running test case: test_cases/test_3.txt
1298.612
8 2 7 4 3 1 10 0 9 5 6 8

real	0m10.135s
user	0m10.038s
sys	0m0.038s
Done running test case: test_cases/test_3.txt

-----------------------------------

Running test case: test_cases/test_4.txt
1519.59
4 5 7 0 11 2 6 3 1 9 10 8 4

real	2m9.335s
user	2m8.116s
sys	0m0.338s
Done running test case: test_cases/test_4.txt

-----------------------------------

Running test case: test_cases/test_5.txt
1989.452
3 9 8 10 12 0 7 4 11 6 2 5 1 3

real	26m50.013s
user	26m40.570s
sys	0m2.623s
Done running test case: test_cases/test_5.txt

-----------------------------------
```