Project 4

Traveling Salesperson Problem:
A Dynamic Programming Solution Implementation

Due Date: Posted on the BlackBoard web site for the course.

Project Name and What to Submit:

- Project Name MUST be **CS320P04LastName** according to your section number.
- Zipped file name must be named: CS320P04*LastName*.zip.
- Zipped file must contain **the entire** project folder containing your Visual Studio 2019 C++ x32/Debug solution-project for Project 4. Also, any test cases (input and corresponding output) must be included in that zip file. Tests must include **more** than the samples given to you. Include an example with a graph that is not complete. Also include a graph that does not contain a Hamiltonian circuit.
- Your solution must be based upon the dynamic programming design. Include a test file that has 10 or more cities.
- Provide a word document, CS320P04Analysis*LastName* describing the time analysis ***and*** space analysis of your algorithm. Express in the appropriate asymptotic ($O(f(n))$, $\Theta(f(n))$, or $\Omega(f(n))$ notation as a function of n, which is the number of cities (or vertices). Use of those three notation; use the one that you can show the most information about the complexity of your implementation. In your presentation, show ALL your assumptions, what each function and variable represents, and give a detailed derivation of your analyses.
- If your program does not work, please include a Word or notepad file explaining what works and what doesn't work.
- The zip file (including full project, test results, and Word document) should be submitted to the assessment icon on Blackboard. Do not attach file outside of the zip file.

Given: A finite set of "cities" $C = \{c_0, c_1, \ldots, c_{m-1}\}$ and a cost function $d: C \times C \rightarrow$ Unsigned-Number $\cup \{\infty\}$, which represents a cost for traveling between c_i and c_j each city for all $i, j \in \{0 \ldots m-1\}$ and if there is no direct way to travel between two cities c_i and c_j, $i \neq j$, let $d(c_i,c_j) = \infty$ and assume $d(c_i,c_i) = 0$ for all $i \in \{0,\ldots,m-1\}$. (Note: Number can equal unsigned integer, or real number, such as float or double in many programming languages. The symbol, ∞, can be represented as the max value representation of the respective data type.)

Find least cost tour of all cities. Of all the simple paths involving all cities in C and with the starting and ending city the same (a.k.a., a Hamiltonian cycle), find the least cost Hamiltonian cycle. This is called the ***optimal*** tour.
Assumption: we will not consider graphs with self-loops; that is there are no edges (c_i, c_i) for all $i \in \{0, ..., m-1\}$. For this project we will consider only weighted, directed graphs. However, some of the discussion below includes representation possibilities for weighted, undirected graphs.

Another formulation: Let π be a permutation function over $\{0, \ldots, m-1\}$. The cost of each Hamiltonian cycle is based upon a permutation π (with the first member of π appended on the end of π) can be expressed as follows.

$$\left(\sum_{i=0}^{m-2} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(m-1)}, c_{\pi(0)})\}$$

Find the minimum cost over all permutations.

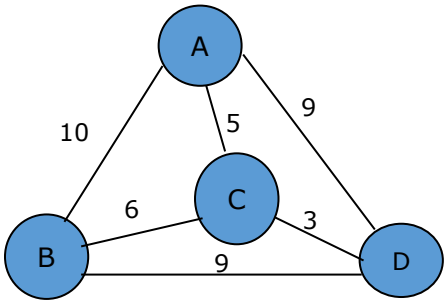
Representing the TSP Instances: One can model the TSP with directed or undirected edge-weighted graphs. The nodes or vertices represent the cities and the edges (undirected, {ci,cj}) or (directed (ci, cj)) represent the connections between cities and the edges are mapped to a non-negative number or ∞ to represent the cost or no connection, respectively. That is, if there is no connection between two cities in one direction, the label the directed edge with ∞, or if no connection for the undirected graph, map the edge to ∞.

If the cost to travel between two cities in either direction has the same cost and this is true for all pairs of vertices that do have connections, then one can model this with an undirected graph. That is for all cities c_i, c_j in the set of cities, $d(c_i, c_j) = d(c_j, c_i) < \infty$, then one can model both connections with one undirected edge.

Undirected or Directed graphs can be represented with adjacency matrices or adjacency lists. In this project we will use adjacency matrices since most of the challenging TSP instances have respective dense graphs. The entries in these graphs represent the weights (costs) of the edges. If there is a connection (a directed edge) from city c_i to city c_j there is a cost entry in row i and column j. For the undirected case, there is the same value in row i and column j and row j and column i.

Thus, for adjacency matrix M, $M[i][i] = 0$, $M[i][j] \geq 0$ if $i \neq j$, and $M[i][j] = \infty$ if there is no edge between i an j. For matrices representing undirected graphs, a triangular matrix is more space efficient.

Example 1
TSP Instance: Complete, Undirected, Edge-Weighted Graph
|Cities| = |Vertices|= n = 4 |Edges| = n*(n-1)/2 = 6. M is the adjacency matrix.
Solution: Optimal Tour Cost = [27, <A,B,D,C,A>]



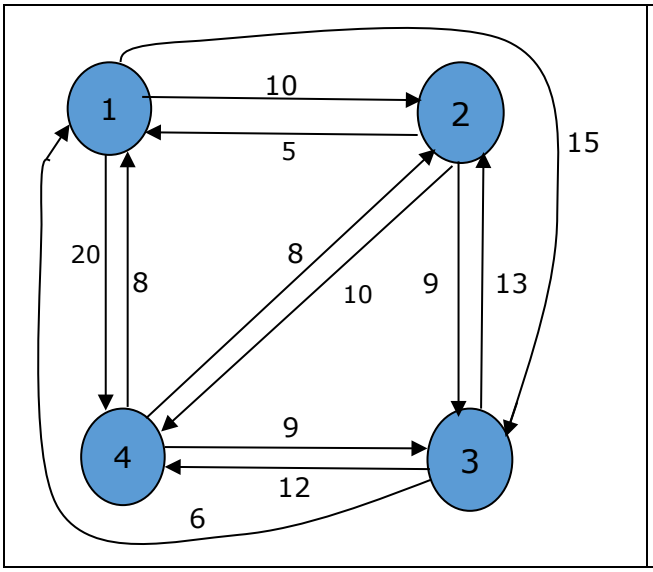
M =

	A	B	C	D
A	0	10	5	9
B	10	0	6	9
C	5	6	0	3
D	9	9	3	0

Example 2
TSP Instance: Directed Complete Graph TSP Instance. This instance has 4 cities and directed edges. It is also complete. If there were a missing edge, there entry would have the cost value of ∞.
This is represented by Graph G = (V, E). V = {1, 2, 3, 4} and E = {(1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)} (set of ordered pairs) and the cost (or weight) function:
d((1,2)) = 10, d((1,3)) = 15, d((1,4)) = 20
d((2,1)) = 5, d((2,3)) = 9, d((2,4)) = 10
d((3, 1)) = 6, d((3,2)) = 13, d((3,4)) = 12
d((4, 1)) = 8, d((4,2)) = 8, d((4,3)) = 9

|Cities| = |Vertices|= n =4 |Directed Edges| = n*(n-1) = 12 M is the adjacency matrix.
Solution: Optimal Tour Cost = [35, <1,2,4,3>]

The graph depiction and the corresponding adjacency matrix are shown below.



M =

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Algorithm Design using Dynamic Programming. See Section 3.6 in your textbook for more examples of problems that can be solved using dynamic programming. The idea is to solve smaller problems from the bottom-up, save the results, and use these results to solve larger subsets of the problem until the full problem instance is reached. Following a short presentation of how dynamic programming can work for the TSP, we will use the second TSP problem instance to illustrate how dynamic programming is concretely applied.

Suppose that G = < V, E> is a directed graph with V = {1, 2, ..., m} and E and the cost function defined and represented by an adjacency matrix M as described earlier: M[i][i] = 0, M[i][j] ≥ 0 if i ≠ j, and M[i][j] = ∞ if there is no edge between vertices i and j.

Without loss of generality assume that the tour starts vertex 1 and ends with vertex 1. Hence, that tour must go through edge (1, i) for at least one vertex i≠1 then followed by a path from i to 1 exactly once through each vertex in V – {1, i}. (The minus is the set difference operator.) If the tour is optimal (sum of

weights is minimum of all such tours), then so is the path from i to 1. Thus, the *principle of optimality holds.* The *Principle of Optimality* states that in a sequence of decision points, each subsequence (sub-problem) must also be optimal.

Suppose the following are given: a set of vertices $S \subseteq V - \{1\}$ and a vertex $i \in V - S$ with $i = 1$ only if $S = V - \{1\}$. Suppose that function g(i, S) is defined as the length of the shortest path from vertex i to vertex 1 and that passes through each vertex in S one and only one time. Given this definition of g, one can represent the length of the entire optimal Hamiltonian cycle as $g(1, V - \{1\})$.

So, by the principle of optimality, we have

$$g(1, V-\{1\}) = \min_{2 \le j \le m} (M[1][j]+g(j, V-\{1,j\})) \qquad (1)$$

To generalize, consider any vertex $i \neq 1$ and $S \neq \emptyset$, and $S \neq V - \{i\}$ and $i \notin S$., and define

$$g(i, S)= \min_{j \in S} (M[i][j]+g(j, S-\{j\})) \qquad (2)$$

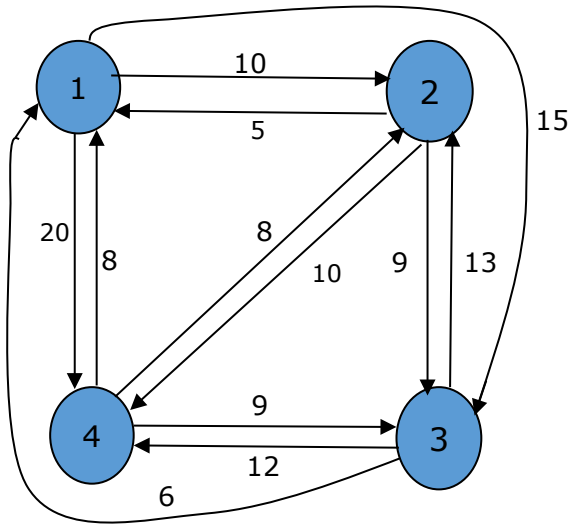
and

$$g(i,\emptyset)=M[i][1], \quad \text{for } i=2, 3,\ldots, m$$

Given that we know the values of g(i,S) when $S = \emptyset$, we can then calculate g(i,S) when the size of S is 1 ($|S| = 1$) and with vertex $1 \notin S$. Then, we can calculate g when $|S| = 2$ with vertex $1 \notin S$. We continue on until we know all the values for $g(j, V - \{1\})$ for all j, such $j \neq 1$, we can finally use eq. (1) to arrive at the solution.

Example: TSP solution using Dynamic Programming.

The graph depiction and the corresponding adjacency matrix are shown below.



M =

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

- Initialize: g(2,∅) = 5, g(3,∅) = 6, g(4,∅) = 8

Using eq. (2) for two nodes

- g(2,{3}) = M[2][3] + g(3,∅) = 9 + 6 = 15
- g(2,{4}) = M[2][4] + g(4,∅) = 10 + 8 = 18

Similarly, we get:

- g(3,{2}) = 18 g(3,{4}) = 20
- g(4,{2}) = 13 g(4,{3}) = 15

- g(2,{3,4}) = min(M[2][3]+g(3,{4}), M[2][4]+g(4,{3}))
  = min(29,25) = 25
- g(3,{2,4}) = min(M[3][2]+g(2,{4}),M[3][4]+g(4,{2}))
  = min(31,25) = 25
- g(4,{2,3}) = min(M[4][2]+g(2,{3}), M[4][3]+g(3,{2}))
  = min(23,27) = 23
- g(1,{2,3,4}) = min(M[1][2]+g(2,{3,4}), M[1][3]+g(3,{2,4}), M[1][4]+g(4,{2,3}))

  = min(35,40,43) = 35

To keep track of the optimal path and eventually report the optimal cycle, one can introduce a function Path(i,S) which is defined as the value of j used to minimize function g when eq. 1 or eq. 2 are applied.

For this problem instance we have
Path(2, {3,4}) = 4
Path(3,{2,4}) = 4;
Path(4,{2,3}) = 2;
Path(1, {2,3,4}) = 2;

The optimal cycle is
1 → Path(1,{2,3,4}) = 2
→ Path(2,{3,4}) = 4
→ Path(4,{3}) = 3
→ 1

INPUT: (1) The directed-edge-weighted-graph file in an ASCII text file. The user should be prompted for this file name via standard input.
(2) Format of graph file: (Designed to make the adjacency list or matrix easy to build.)

Each vertex appears on a separate line.
|V| = n = number of lines. Values are all non-negative integers. Space-separated values.
The second value of each line is to be read as a string, even if it is an integer. These will be the "names" of the vertices so that more meaningful names (such as city names) can be given to each vertex.

<$vertex_0$-id> <vertex-string-$name_0$> <out-edge-01-vertex-id> <out-edge-01-value> <edge-02-vertex_id> <out-edge-02-value>... <edge-0m_o-vertex-id> <out-edge-0m_0-value>
...
<$vertex_i$-id> <vertex-string-$name_i$> <out-edge-i1-vertex-id> <out-edge-i1-value> <edge-i2-vertex_id> <out-edge-i2-value>... <edge-im_i-vertex-id> <out-edge-im_i-value>
...
<$vertex_{(n-1)}$-id> <vertex-string-$name_{(n-1)}$> <out-edge-(n-1)1-vertex-id> <out-edge-(n-1)1-value> <edge(n-1)2-vertex_id> <out-edge(n-1)2-value>... <edge-(n-1)$m_{(n-1)}$-vertex-id> <out-edge-(n-1)$m_{(n-1)}$-value>
Note edges can be missing. In your program you should assume those corresponding missing edges must have edge value cost of ∞

OUPUT: [min-cost, '<'min-cost Hamiltonian Cycle'>'] The tour must be printed with the string value of vertex i given in the input file. Note that the cycle will be of length |V|+1. The '<' and '>' are quoted since these are usually meta-symbols for denoting a sequence and ARE to be shown in the output. So, both the square brackets and angle brackets must be shown.

Example I/O format:

The input for TSP Example shown above must have the following format:
Input:

1	1	1 0	2 10	3 15	4 20
2	2	1 5	2 0	3 9	4 10
3	3	1 6	2 13	3 0	4 12
4	4	1 8	2 8	3 9	4 0

Output: Optimal Tour Cost = [35, <1,2,4,3>]

REQUIREMENTS:

- Implementation Language: C++ using VS 2019 x32/Debug Solution/Project.
- Algorithm: For full credit use dynamic programming as described above and related tables where appropriate.
- Problem Instances: Use edge-weighted directed graphs (not necessarily complete).
- Constraints: Implement each edge-weighted directed graph using an adjacency matrix, not an adjacency list.
- Tests: Besides complete, directed, and weighted graphs, include a weighted, directed graph that is NOT complete. Also include a weighted, directed graph that does not contain a Hamiltonian circuit. Add to the sample tests shown in this document and supplementary files.
- Do the time and space complexity analyses as a function of the number of cities (i.e., number of vertices), n, and as described above.