Algorithms

Lecture 5
Graph Algorithms (2)

A. S. M. Sanwar Hosen

Email: sanwar@wsu.ac.kr

Date: 6 April, 2023

Graph Traversal Algorithm

- **Shortest Path Algorithm:** It finds the shortest paths between nodes. It is also knows as Greedy Algorithm developed by Dijkstra.

- ✓ **Basics of the Shortest Path**

  Step 1: Dijkstra algorithm starts from a source node to find the shortest paths from that node to all the other nodes in a graph.

  Step 2: It keeps track of the current known shortest distance from source node to each node and update the values once it finds the shortest path.

  Step 3: Once it finds the shortest path from a source node to another node, the node is marked as 'visited', and added to the path.

  Step 4: The process continues until all the nodes are visited and added to the path.

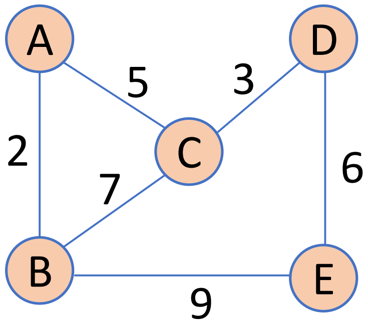
- ✓ **Requirement of the Shortest Path**

  This algorithm can only work with graphs that have positive weights.

Graph Traversal Algorithm

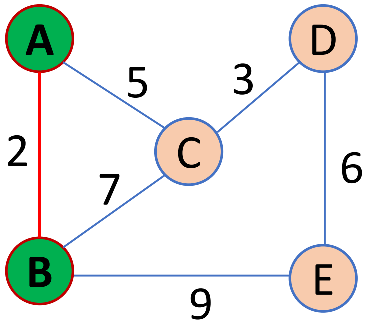
❑ Shortest Path Algorithm

Example: Consider the following weighted graph to perform the Dijkstra's algorithm. We will have the shortest path from a vertex to all other vertices in the graph.



Step 1:

- Select a vertex **A** (source node) as starting point (visit)
- Calculate the weight (distance) from **A** to its adjacent vertices (**B**, **C**)
- Select the vertex with minimum distance (**B**), and add the vertex **B** to the path

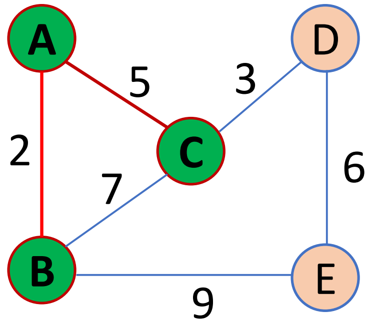


Distance
A→ A: 0
A→ B: 2
A→ C: 5
A→ D:
A→ E:

Path
A→ A: {A}
A→ B: {A, B}
A→ C:
A→ D:
A→ E:

Step 2:

- Calculate the distance from **A** to its adjacent vertices (not visited)
- Select the vertex with minimum distance (**C**), and add the vertex (**C**) to the path

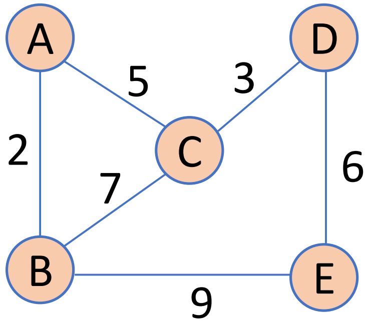


Distance
A→ A: 0
A→ B: 2
A→ C: 5 or **A→ B→C:** 9
A→ D:
A→ E:

Path
A→ A: {A}
A→ B: {A, B}
A→ C: {A, C}
A→ D:
A→ E:

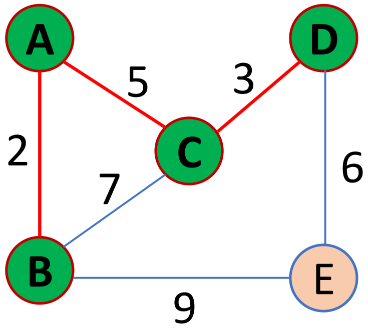
❑ Shortest Path Algorithm

Example: Consider the following weighted graph to perform the Dijkstra's algorithm.



Step 3:

- Calculate the distance from **C** to its adjacent vertices (**D**)
- Select the vertex with minimum distance (**D**), and add the vertex **D** to the path



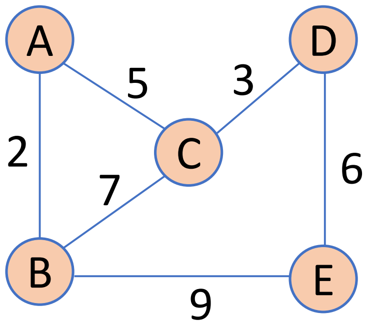
Distance
A→ A: 0
A→ B: 2
A→ C: 5 or **A→B→C:** 9
A→ D: **A→C→ D**: 8
A→ E:

Path
A→ A: {A}
A→ B: {A, B}
A→ C: {A, C}
A→ D: {A, C, D}
A→ E:

Graph Traversal Algorithm

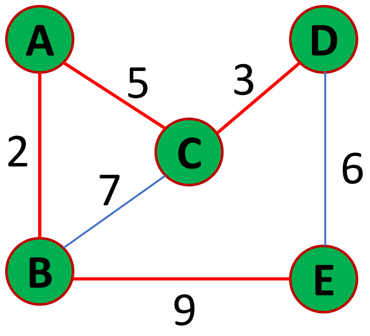
❑ Shortest Path Algorithm

Example: Consider the following weighted graph to perform the Dijkstra's algorithm.



Step :

- Calculate the distance from **D** to its adjacent vertices (**E)**
- Select the vertex with minimum distance (**E**), and add the vertex **E** to the path



Distance
A→ A: 0
A→ B: 2
A→ C: 5 or **A→B→C:** 9
A→ D: **A→C→ D**: 8
A→ E: **A→C→D→E**:14
Or **A→B→ E:** 11

Path
A→ A: {A}
A→ B: {A, B}
A→ C: {A, C}
A→ D: {A, C, D}
A→ E: {A, B, E}

Graph Traversal Algorithm

- **Bellman-Ford Algorithm:** It finds the shortest paths between nodes in a weighted graph.

✓ **Basics of the Bellman Ford Algorithm**

Step 1: Start with the weighted graph.

Step 2: Select a starting vertex and assign the distance 0 at the vertex, and infinity for other vertices.

Step 3: Update/relax the path values based on the following condition for $(n-1)$ times, where u, v are the vertices, and n is the total vertices in the graph.

$$if\ (d[u] + c(u,v) < d[v], then\ d[v] = d[u] + c(u,v)$$

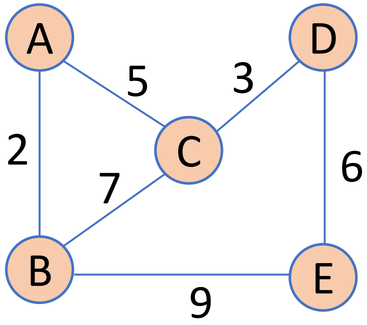
Step 4: The process continues until the relaxation of the path values are stopped.

✓ **Requirement of the Shortest Path**

This algorithm can work with graphs that have positive and negative weights.

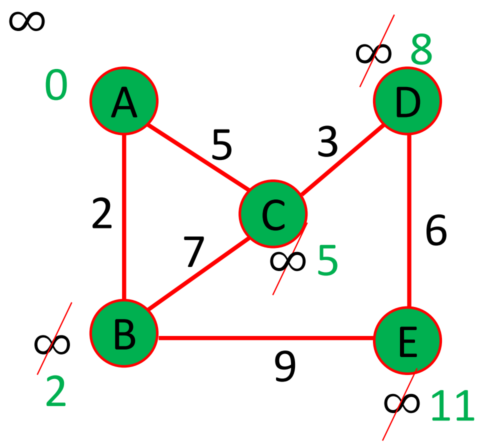
Bellman-Ford Algorithm

Example: Consider the following weighted graph to perform the Bellman-Ford algorithm to find a shortest path. We will have the shortest path from a vertex to all other vertices in the graph.



Step 1:

- Start from a source node **A**, assign the initial distance 0 at **A**, and for other vertices ∞
- Calculate the cost of all the edges, and relaxes the path values

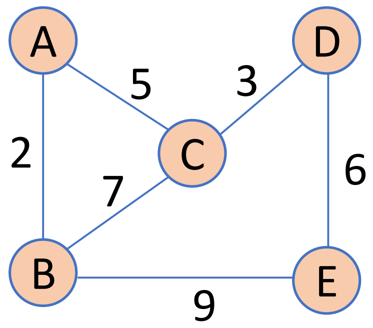


Vertices	Cost A	Cost B	Cost C	Cost D	Cost E
A	0	0 + 2 = 2	0 + 5 = 5		
B		0	2 + 7 = 9		2 + 9 = 11
C			0	5 + 3 = 8	
D				0	8 + 6 = 14
E					0

E = {(A,B), (A,C), (B,C), (B,E), (C, D), (D, E)}

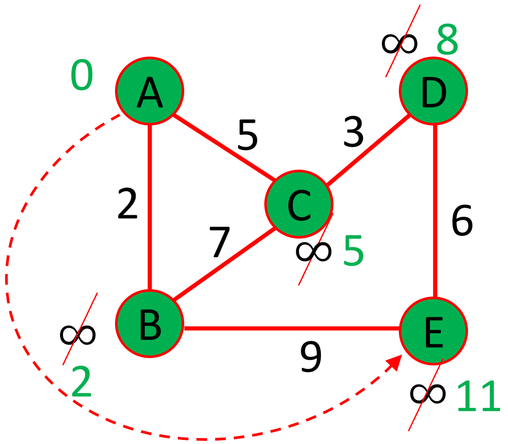
Bellman-Ford Algorithm

Example: Consider the following weighted graph to perform the Bellman-Ford algorithm to find a shortest path. We will have the shortest path from a vertex to all other vertices in the graph.



Step 2:

- Calculate the cost of all the edges, and relaxes the path values

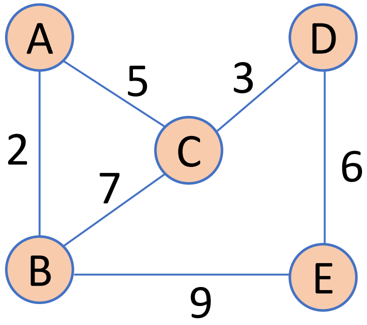


Vertices	Cost A	Cost B	Cost C	Cost D	Cost E
A	0	0 + 2 = 2	0 + 5 = 5		
B		0	2 + 7 = 9		2 + 9 = 11
C			0	5 + 3 = 8	
D				0	8 + 6 = 14
E					0

E = {(A,B), (A,C), (B,C), (B,E), (C, D), (D, E)}

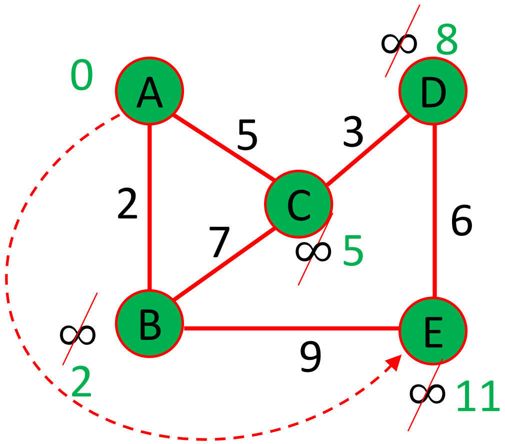
❑ Bellman-Ford Algorithm

Example: Consider the following weighted graph to perform the Bellman-Ford algorithm to find a shortest path. We will have the shortest path from a vertex to all other vertices in the graph.



Step 3:

- Calculate the cost of all the edges, and relaxes their path values
- Found no changes in the path values, the process is stopped.



Vertices	Cost A	Cost B	Cost C	Cost D	Cost E
A	0	0 + 2 = 2	0 + 5 = 5		
B		0	2 + 7 = 9		2 + 9 = 11
C			0	5 + 3 = 8	
D				0	8 + 6 = 14
E					0

The shortest path from A to E is (A, B, E)

E = {(A,B), (A,C), (B,C), (B,E), (C, D), (D, E)}

Graph Traversal Algorithm

Time and Space Complexity

Algorithm	Time Complexity	Space Complexity
Dijkstra's	$O\big((E+V)\log V\big)$ $= O(n\ log\ n)$	$O(V) = O(n)$
Bellman-Ford	$O(VE)\ =\ O(n^2)$	$O(V) = O(n)$

Where, E is the number of edges and V is the number vertices in the graph

Comparison

- ✓ Dijkstra's algorithm has a better time complexity compared to Bellman-Ford algorithm.
- ✓ Dijkstra's algorithm requires a priority queue which increased its space complexity unlike the Bellman-Ford where a simple array is required.
- ✓ Dijkstra's algorithm is generally preferred when the graph is sparse (i.e., E is much less than V^2) and non-negative weights are used.
- ✓ Bellman-Ford algorithm is preferred when the graph is dense (i.e., E is close to V^2) and negative weights are allowed.

Applications of Graph Data Structure and Algorithms

- The common uses of graph data structure are as follows:
- ✓ **Computer Science:** Graphs are used to represent the flow of computations.
- ✓ **Mapping Systems (Google Maps):** Intelligent traffic management, and finding shortest route from source to destination.
- ✓ **Various Social Media (e.g., Facebook, Linkedin):** Facebook friend requests use graph data structure.
- ✓ **Operating System:** Graph is used in resource and process allocation.
- ✓ **World Wide Web:** Accumulates knowledge from difference interlinked sources.

Graph Data Structure and Algorithms Implementation in Python (1)

- Directed graph construction and finding a path between two nodes:

```
# Implementing Graph in python
# Example: the grpah has six nodes and eight links
#    A-->B
#    A-->C
#    C-->D
#    D-->C
#    E-->F
#    F-->C
#.........................................................
# Initialize the grpah by defining python dictionary:
graph = {'A':['B', 'C'],
         'B':['C', 'D'],
         'C':['D'],
         'D':['C'],
         'E':['F'],
         'F':['C']
        }
print('The following graph has been initialized:\n')
print(graph) # Display the grpah

# Finding a path from a source node to any given node:
def find_path(graph, start, end, path=[]):
        path = path + [start]
        if start == end:
            return path
        if start not in graph:
            return None
        for node in graph[start]:
            if node not in path:
                newpath = find_path(graph, node, end, path)
                if newpath: return newpath
        return None

# Call the function 'find_path' to find the path between two nodes:
print("The path between two nodes is:")
find_path(graph, 'A', 'D')
```

```
The following graph has been initialized:

{'A': ['B', 'C'], 'B': ['C', 'D'], 'C': ['D'], 'D': ['C'], 'E': ['F'], 'F': ['C']}


Output:

The path between two nodes is:

['A', 'B', 'C', 'D']
```

Graph Data Structure and Algorithms Implementation in Python (2)

- Finding all possible paths between two nodes:

```python
# Initialize the grpah by defining python dictionary:
graph = {'A':['B', 'C'],
         'B':['C', 'D'],
         'C':['D'],
         'D':['C'],
         'E':['F'],
         'F':['C']
        }
print('The following graph has been initialized:\n')
print(graph) # Display the grpah
print('\n')

# Finding all possible paths between two nodes:
def find_all_paths(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return [path]
    if start not in graph:
        return []
    paths = []

    for node in graph[start]:
        if node not in path:
            newpaths = find_all_paths(graph, node, end, path)
            for newpath in newpaths:
                paths.append(newpath)

    return paths
# Call the function 'find_all_paths' to find the all paths between two nodes:
print('Output:\n')
print("The paths between two nodes are:")
find_all_paths(graph, 'A', 'D')
```

```
The following graph has been initialized:

{'A': ['B', 'C'], 'B': ['C', 'D'], 'C': ['D'], 'D': ['C'], 'E': ['F'], 'F': ['C']}


Output:

The paths between two nodes are:

[['A', 'B', 'C', 'D'], ['A', 'B', 'D'], ['A', 'C', 'D']]
```

Graph Data Structure and Algorithms Implementation in Python (3)

- Finding the shortest path between two nodes using Dijkstra's algorithm:

```
# Initialize the grpah by defining python dictionary:
graph = {'A':['B', 'C'],
         'B':['C', 'D'],
         'C':['D'],
         'D':['C'],
         'E':['F'],
         'F':['C']
        }
print('The following graph has been initialized:\n')
print(graph) # Display the grpah
print('\n')

# Finding the shortest path between two nodes:
def find_shortest_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if start not in graph:
        return None
    shortest = None
    for node in graph[start]:
        if node not in path:
            newpath = find_shortest_path(graph, node, end, path)
            if newpath:
                if not shortest or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest
# Call the function 'find_shortest_path' to find the path between two nodes:
print('Output:\n')
print("The shortest path between two nodes is:")
find_shortest_path(graph, 'A', 'C')
```

```
The following graph has been initialized:

{'A': ['B', 'C'], 'B': ['C', 'D'], 'C': ['D'], 'D': ['C'], 'E': ['F'], 'F': ['C']}


Output:

The shortest path between two nodes is:

['A', 'C']
```

Graph Data Structure and Algorithms Implementation in Python (4)

- Finding the shortest path between two nodes using Bellman-Ford algorithm:

```
# Find the shortest path between two nodes using Bellman Ford Algorithm
# Defining a class Graph
class Graph:
    def __init__(self, vertices):
        self.V = vertices    # The number of vertices in the Graph
        self.graph = []      # List of edges

    # Define a function to add edges in the Graph
    def addEdge(self, src, dest, weight):
        self.graph.append([src, dest, weight])

    # Display the shortest path
    def displayPaths(self, distance):
        print("The shortest distance from a source to other vertices:")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, distance[i]))

    # Define the Bellman-Ford Algorithm using the function bellmanFord
    def bellmanFord(self, source):
        # Step 1: Initilize the path values by inifinity
        distance = [float("Inf")] * self.V
        # Mark the source vertex
        distance[source] = 0

        # Step 2:Relax/update edges |V| - 1 times
        for _ in range(self.V - 1):
            for src, dest, weight in self.graph:
                if distance[src] != float("Inf") and distance[src] + weight < distance[dest]:
                    distance[dest] = distance[src] + weight

        # Step 3: if there is negetive cycle exists in the graph
        # The path values often change even after the number of passes
        # Then we can not find the shortest path
        for src, dest, weight in self.graph:
            if distance[src] != float("Inf") and distance[src] + weight < distance[dest]:
                print("Negetive cycle exists in the graph")
                return

        # No negetive cycle exists
        # Display the distance from source to the vertices
        self.displayPaths(distance)
```

```
# Define the graph
g = Graph(6)
g.addEdge(0, 1, 7)
g.addEdge(0, 2, 6)
g.addEdge(1, 3, 5)
g.addEdge(2, 1, 8)
g.addEdge(3, 2, 4)
g.addEdge(3,5,7)

print('Output:')
g.bellmanFord(0)
```

```
Output:
The shortest distance from a source to other vertices:
0		0
1		7
2		6
3		12
4		inf
5		19
```