CS302 - Data Structures
using C++

Topic: Shortest Paths

Kostas Alexis

Singe Source Shortest Path

Austin	
	160 miles
Houston	
	800 miles
Atlanta	
	600 miles
Washington	
Total miles	1560 miles

Austin	
	200 miles
Dallas	
	780 miles
Denver	
	1400 miles
Atlanta	
	600 miles
Washington	
Total Miles	2980 miles

Shortest Paths

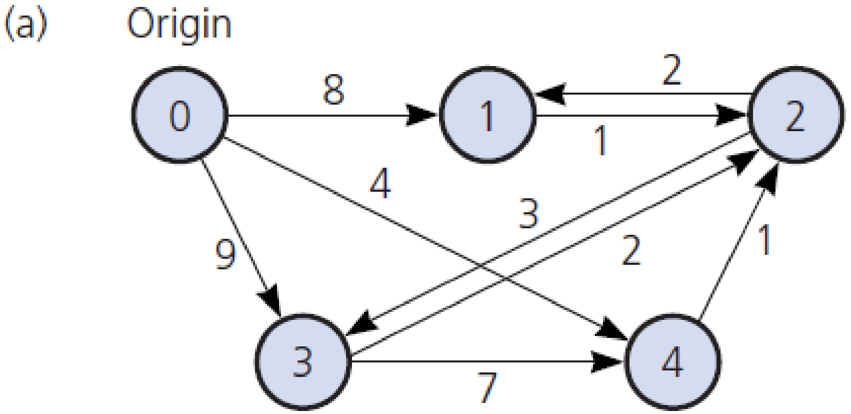
- There might be multiple paths from a source to a destination vertex
- Shortest path: the path whose total weight (i.e., sum of edge weights) is minimum
- Austin → Houston → Atlanta → Washington: 1560 miles
- Austin → Dallas → Denver → Atlanta → Washington: 2980 miles

Variants of Shortest Path

- **Single-pair shortest path**
  - Find a shortest path from u to v (for given vertices u and v)
- **Single-source shortest paths**
  - $G = (V,E)$ → find a shortest path from a given source vertex s to each vertex $v \in V$
- **Single-destination shortest paths**
  - Find a shortest path to a given destination vertex t from each vertex v
  - Reversing the direction of each edge → single source
- **All-pairs shortest paths**
  - Find a shortest path from u to v for every pair of vertices u and v

Shortest Paths

- The Shorted path between two vertices in a weighted graph
  - Has the smallest edge-weight sum
- A weighted directed graph and its adjacency matrix



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Shortest Paths

- Dijkstra’s Algorithm

Shortest Paths

- Dijkstra's Algorithm
    1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

Shortest Paths

- Dijkstra’s Algorithm
  1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
  2. Keep a set of visited nodes. This set starts with just the initial node.

Shortest Paths

- Dijktsra's Algorithm
  1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
  2. Keep a set of visited nodes. This set starts with just the initial node.
  3. For the current node, consider all of its unvisited neighbors and calculate (distance to the current node) + (distance from current node to the neighbor). If this is less than their current tentative distance, replace it with this new value.

Shortest Paths

- Dijkstra’s Algorithm
    1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
    2. Keep a set of visited nodes. This set starts with just the initial node.
    3. For the current node, consider all of its unvisited neighbors and calculate (distance to the current node) + (distance from current node to the neighbor). If this is less than their current tentative distance, replace it with this new value.
    4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.

Shortest Paths

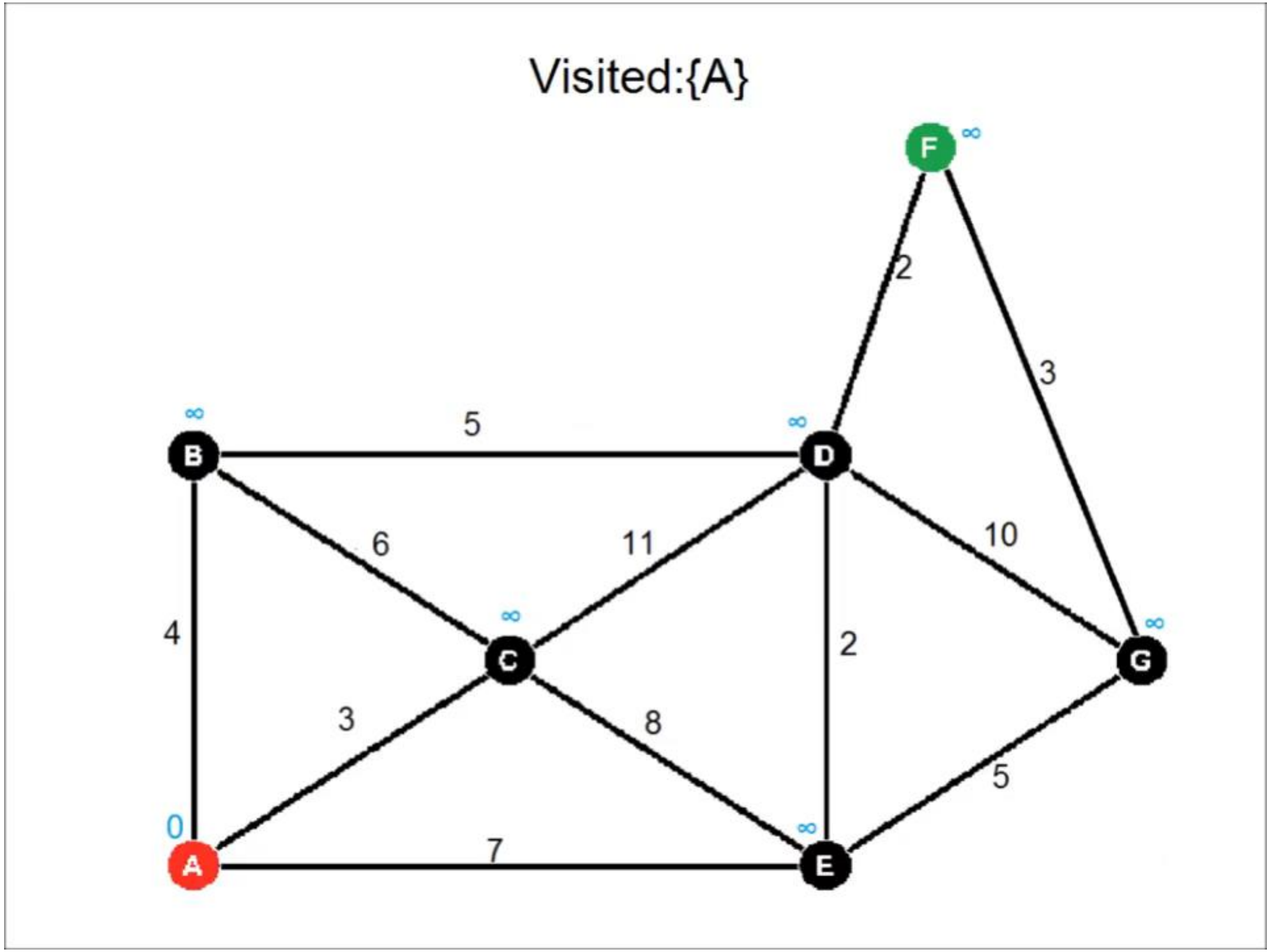
- Dijkstra's Algorithm
    1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
    2. Keep a set of visited nodes. This set starts with just the initial node.
    3. For the current node, consider all of its unvisited neighbors and calculate (distance to the current node) + (distance from current node to the neighbor). If this is less than their current tentative distance, replace it with this new value.
    4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
    5. If the destination node has been marked visited, the algorithm has finished.

Shortest Paths

- Dijkstra's Algorithm
  1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
  2. Keep a set of visited nodes. This set starts with just the initial node.
  3. For the current node, consider all of its unvisited neighbors and calculate (distance to the current node) + (distance from current node to the neighbor). If this is less than their current tentative distance, replace it with this new value.
  4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
  5. If the destination node has been marked visited, the algorithm has finished.
  6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3

Shortest Paths

- Dijkstra's shortest path

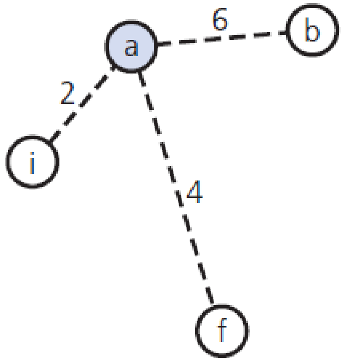


Shortest Paths

```
// Finds the minimum-cost paths between an origin vertex (vertex 0) and all other vertices in a weighted
// directed graph theGraph; theGraph’s weights are >=0
shortestPath(theGraph: Graph, weight: WeightArray)
{
    // Step 1: Initialization
    Create a set vertexSet that contains only vertex 0
    n = number of vertices in theGraph
    for (v = 0 through n-1)
          weight[v] = matrix[0][v]
    // Steps 2 through n
    // Invariant: For v not in vertexSet, weight[v] is the smallest weight of all points from 0 to v that pass through
    // only vertices in  vertexSet before reaching v. For v in vertexSet, weight[v] is the smallest weight of all parts
    // from 0 to v (including paths outside vertexSet), and the shortest path from 0 to v lies entirely in vertexSet
    for (step = 2 through n)
    {
          Find the smallest weight[v] such that v is not in vertexSet
          Add v to vertexSet
          // Check weight[u] for all u not in vertexSet
          for (all vertices u not in vertexSet)
                    if (weight[u] > weight[v] + matrix[v][u])
                              weight[u] = weight[v] + matrix[v][u]
    }
}
```

Shortest Paths

- A trace of the shortest-path algorithm applied to the graph in

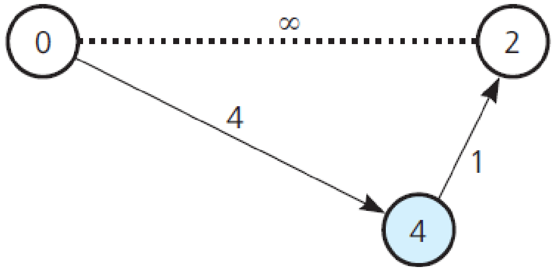


Step	**v**	**vertexSet**	Weight [0]	**Weight [1]**	**Weight [2]**	**Weight [3]**	**Weight [4]**
1	-	0,	0	8		9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2,1, 3	0	7	5	8	4

Shortest Paths

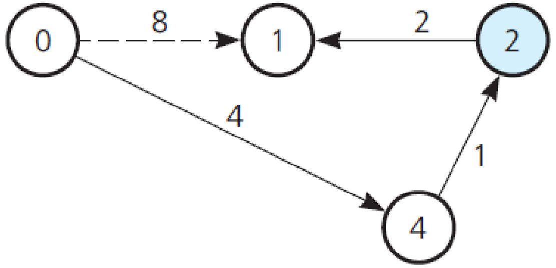
- Checking weight[u] by examining the graph: (a) weight[2] in step 2, (b) weight[1] in step 3

(a) *weight*[2] in step 2



Step 2. The path 0–4–2 is shorter than 0–2

(b) *weight*[1] in step 3

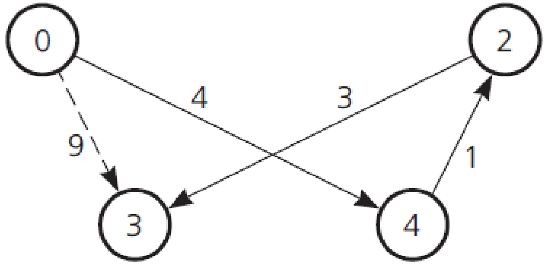


Step 3. The path 0–4–2–1 is shorter than 0–1

Shortest Paths

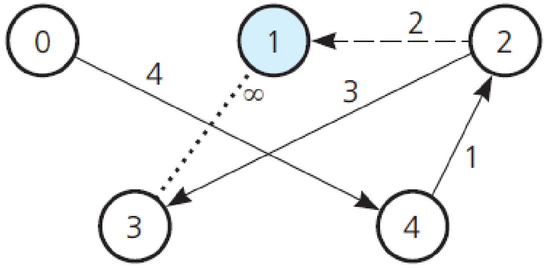
- Checking weight[u] by examining the graph: (c) weight[3] in step 3; (d) weight[3] in step 4.

(c) *weight*[3] in step 3



Step 3 continued. The path 0–4–2–3 is shorter than 0–3

(d) *weight*[3] in step 4



Step 4. The path 0–4–2–3 is shorter than 0–4–2–1–3

Shortest Paths

- C++ Implementation of Dijkstra’s algorithm

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
                min = dist[v], min_index = v;

    return min_index;
}
```

Shortest Paths

- C++ Implementation of Dijkstra’s algorithm

```
// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; i++)
    printf("%d tt %d\n", i, dist[i]);
}
```

Shortest Paths

- C++ Implementation of Dijkstra's algorithm

```
// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
      int dist[V];         // The output array. dist[i] will hold the shortest
                                                    // distance from src to i

      bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
                                                    // path tree or shortest distance from src to i is finalized

      // Initialize all distances as INFINITE and stpSet[] as false
      for (int i = 0; i < V; i++)
             dist[i] = INT_MAX, sptSet[i] = false;

      // Distance of source vertex from itself is always 0
      dist[src] = 0;

      // Find shortest path for all vertices
      for (int count = 0; count < V-1; count++)
      {
      // Pick the minimum distance vertex from the set of vertices not
      // yet processed. u is always equal to src in the first iteration.
      int u = minDistance(dist, sptSet);
```

Shortest Paths

- C++ Implementation of Dijkstra’s algorithm

```
    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the picked vertex.
    for (int v = 0; v < V; v++)

         // Update dist[v] only if is not in sptSet, there is an edge from
         // u to v, and total weight of path from src to v through u is
         // smaller than current value of dist[v]
         if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]+graph[u][v] < dist[v])
                  dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}
```

https://ide.geeksforgeeks.org/jM5ZlfsAEv

Shortest Paths

- C++ Implementation of Dijkstra’s algorithm

```
// driver program to test above function
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                         {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                         {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                         {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                         {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                         {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                         {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                         {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                         {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                         };

    dijkstra(graph, 0);

    return 0;
}
```

Thank you