Lecture 12

Christopher Godley

CSCI 2270 Data Structures

July 16th, 2018

Graphs

- Throughout this course, we’ve slowly built up more complex data structures
  - Linked lists maintained one pointer
  - Doubly Linked Lists maintained two pointers
  - BSTs maintain 3 pointers
  - 234 Trees maintain an array of up to 4 children pointers, not including the parent
- Pointers allow us to travel between data nodes

Graphs

- A graph utilizes a similar structure to organize data
- Think of a graph like a map:
  - If we want to get between two locations, we travel along the roads, or edges, between the two points
  - This may take us through multiple other nodes along the way

Adjacency Matrix

- A structure for representing direct connections between entities in a graph
  - i.e. locations
- Lets extend the map analogy and generate an adjacency matrix for a set of cities

Adjacency Matrix

- We start with an empty matrix such as this:

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver									
Colo Springs									
Pueblo									
Ft. Collins									
Lincoln									
Omaha									
Kansas City									
Lawren ce									
Wichita									

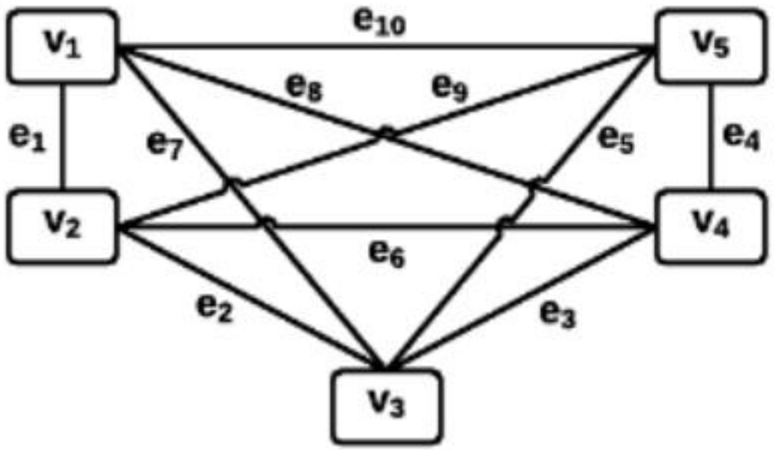
Adjacency Matrix

- We add a 1 if two locations share a road, or a 0 if they don't

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver	0	1	0	1	1	0	0	1	1
Colo Springs	1	0	1	0	0	0	0	0	0
Pueblo	0	1	0	0	0	0	0	0	0
Ft. Collins	1	0	0	0	1	0	0	0	0
Lincoln	1	0	0	1	0	1	0	0	0
Omaha	0	0	0	0	1	0	1	0	0
Kansas City	0	0	0	0	0	1	0	1	1
Lawrence	1	0	0	0	0	0	1	0	1
Wichita	1	0	0	0	0	0	1	1	0

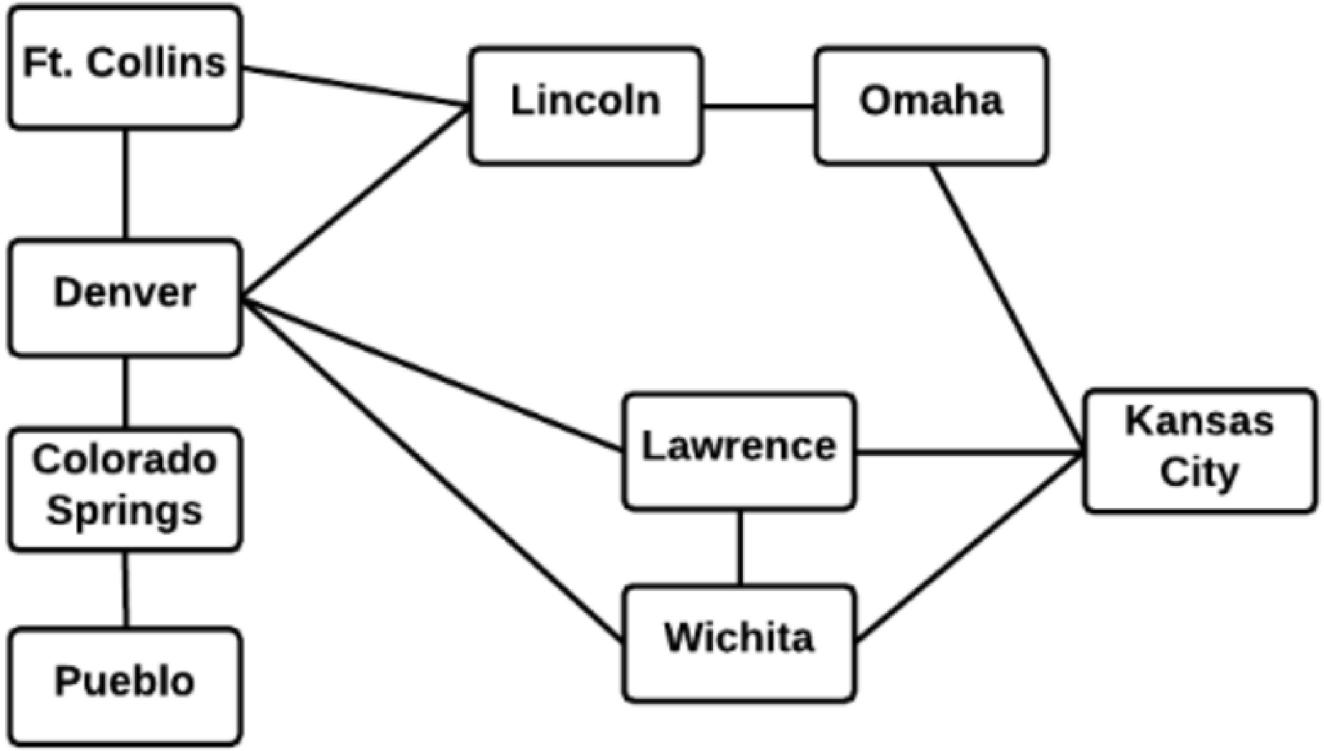
Back to Graphs

- We can represent an adjacency matrix with a graph
- A graph is defined as G = (V, E)
  - V is a set of vertices
  - E is a set of edges
- Ex:



Graph from Adjacency Matrix

- Using our example before, we get the following graph



Graphs

- Undirected Graph
  - Edges are bidirectional
  - Adjacency matrix is symmetric
- Directed Graph
  - Edges are unidirectional
  - Each edge in the adjacency matrix is an outgoing edge
  - Adjacency matrix may not symmetric

Graphs

- Weighted graphs
  - In the previous adjacency matrix shown, each edge is binary
    - An edge is either there or it isn’t
  - In a lot of graphs, edges may not be equivalent
  - Weights may be applied to signify the inequality of edges
    - Distances may be applied to a map
    - Cost may be applied in a supply chain
    - Latency may be applied in a networking graph

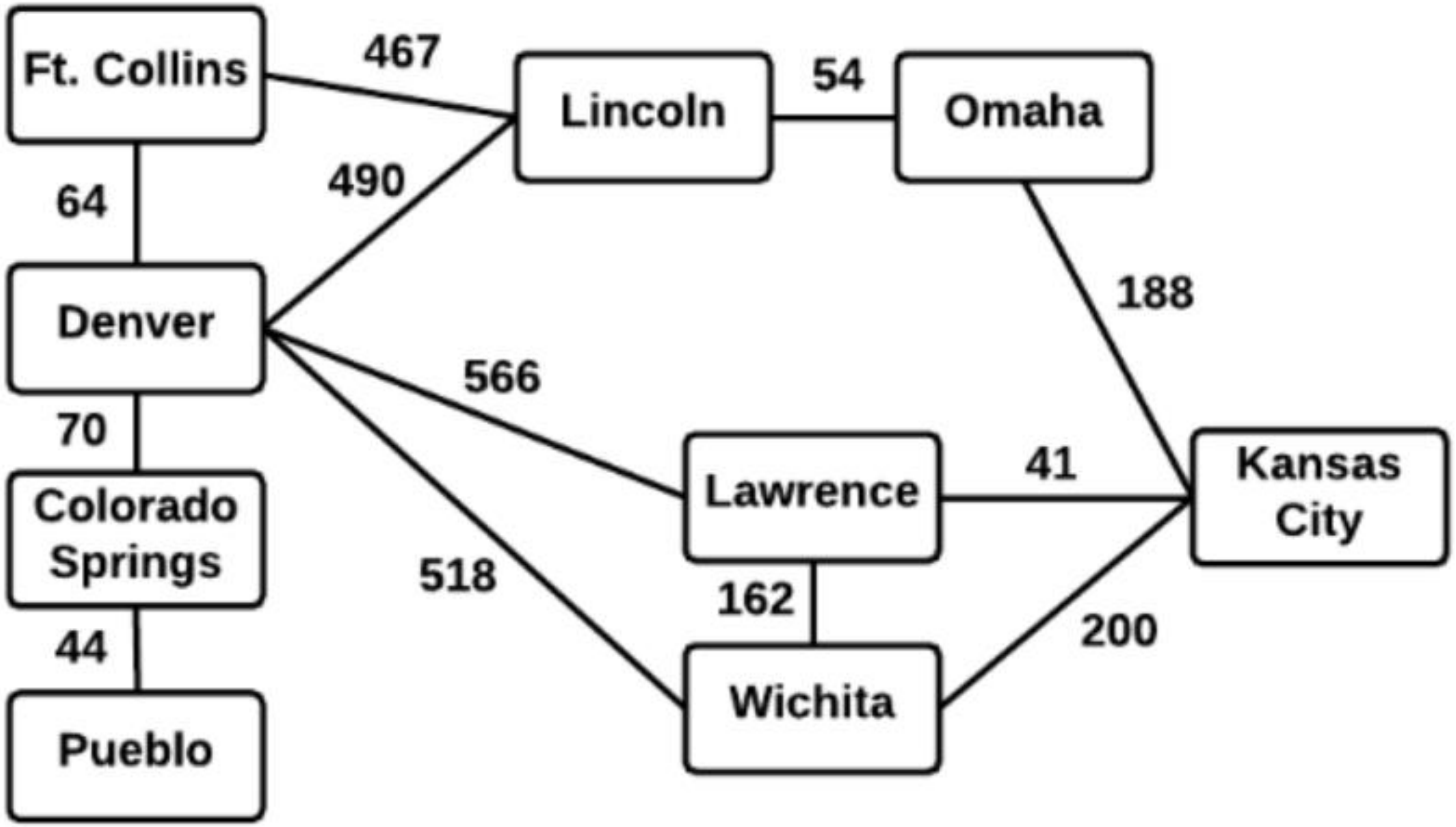
Graphs

- Weighted adjacency matrix from previous example:

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver	0	70	-1	64	490	-1	-1	566	518
Colo Springs	70	0	44	-1	-1	-1	-1	-1	-1
Pueblo	-1	44	0	-1	-1	-1	-1	-1	-1
Ft. Collins	64	-1	-1	0	467	-1	-1	-1	-1
Lincoln	490	-1	-1	467	0	54	-1	-1	-1
Omaha	-1	-1	-1	-1	54	0	188	-1	-1
Kansas City	-1	-1	-1	-1	-1	188	0	41	200
Lawrence	566	-1	-1	-1	-1	-1	41	0	162
Wichita	518	-1	-1	-1	-1	-1	200	162	0

Graphs

- Weighted graph



Graphs: ADT

Structs:

vertex:

key

adj

adjVertex:

vertex*

weight

Graphs: ADT

```
Graph:
        private:
                vertices
        public:
                Init()
                insertVertex(value)
                insertEdge(startValue, endValue, weight)
                deleteVertex(value)
                deleteEdge(startValue, endValue)
                printGraph()
                search(value)
```

Graphs: Insert Vertex

```
insertVertex(value)
    found = false
    for (int i=0; i<vertices.size(); i++)
        if (vertices[i].key == value)
            found = true
            break
    if (found == false)
        vertex v
        v.key = value
        vertices.add(v)
```

Graphs: Insert Edge

```
insertEdge(v1, v2, weight)
    for (int x=0; x<vertices.size(); x++)
        if (vertices[x].key == v1) // found v1
            for (int y=0; y<vertices.size(); y++)
                if (vertices[y].key == v2 and x!=y) // found v2
                    adjVertex av;
                    av.v = &vertices[y];
                    av.weight = weight
                    vertices[x].adjacent.push_back(av)
```

Graphs: Search

```
search(value)
    for x=0 to vertices.end
        if vertices[x].key == value
            return vertices[x]
    return NULL
```

Graphs: ADT

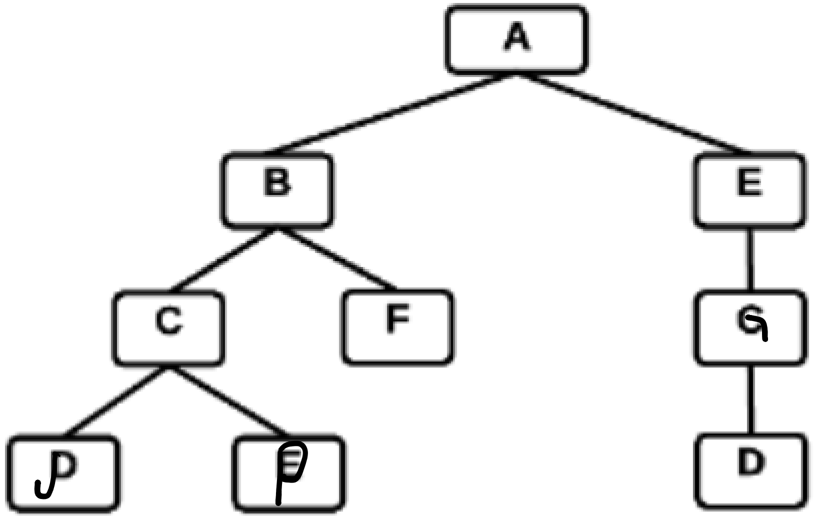
- The textbook shows more pseudocode for the Graph ADT
  - See chapter 12.7

Graph Traversals

- A graphs edges are its traversable roadways
- Counting the number of edges will give you the number of steps to get from one vertex to another
- Adding edge weights rather than just counting the edges will provide an exact distance, or the cost from traversing from v1 to v2

Graphs: Breadth First Search

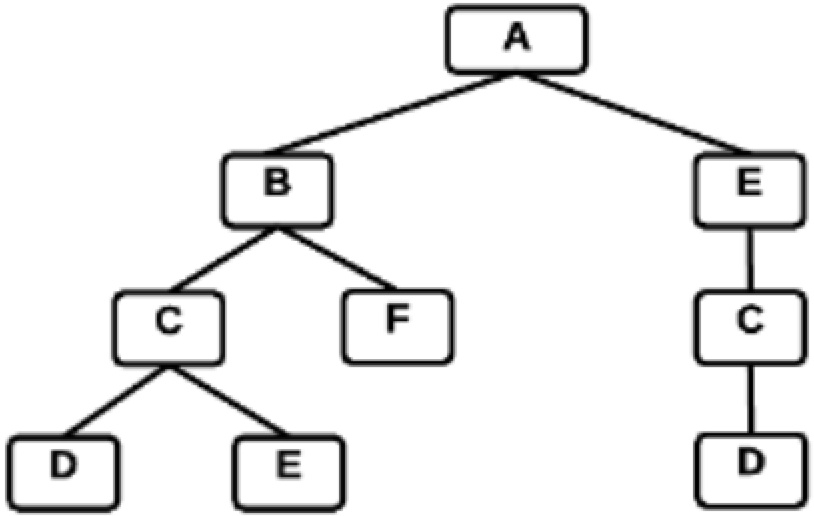
- Breadth First Search (BFS)
  - To simplify this description, lets think of a tree
    - Each node in the tree is visited once,
    - All nodes at any depth are seen before any nodes that are deeper



d
0
1
2
3

Graphs: Breadth First Search

- Breadth First Search (BFS)
  - To simplify this description, lets think of a tree
    - Once a node is visited, its children are added to a queue
  - Will find shortest path of *unweighted* graph

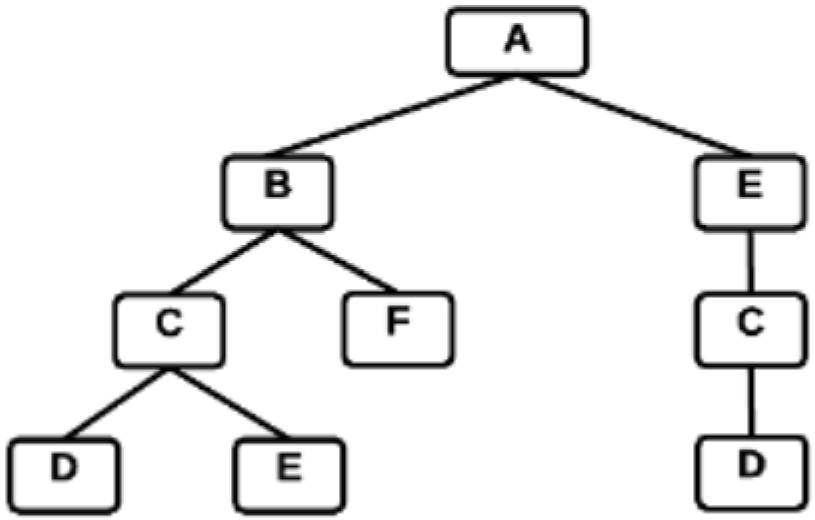


Graphs: Breadth First Search

```
breadthFirstSearch(startValue, endValue)
        vertex = search(startValue)
        vertex.visited = true
        vertex.distance = 0
        queue = new queue()
        queue.enqueue(vertex)
        while(!queue.isEmpty())
                n = queue.dequeue()
                for x=0 to v.adjacent.end
                        if (!n->adjacent[x]->v->visited)
                                n.adjacent[x].v.distance = n.distance + 1
                                n.adjacent[x].v.parent = n
                                if (n.adjacent[x].v.key == endValue)
                                        return n.adjacent[x].v
                                else
                                        n.adjacent[x].v.visited = true
                                        queue.enqueue(n.adjacent[x].v)
        return NULL
```

Graphs: Depth First Search (DFS)

- We can also use a tree to simplify the picture for learning DFS
  - Similar to our tree traversal
  - Finds *a* path, not necessarily the shortest path



Graphs: Depth First Search

```
DFS(vertex)
        vertex.visited = true
        for x=0 to vertex.adjacent.end
                if (!vertex.adjacent[x].v.visited)
                        print(vertex.adjacent[x].v.key)
                        DFS(vertex.adjacent[x].v)

depthFirstSearch(value)
        vertex = search(value)
        print(vertex.key)
        DFS(vertex)
```

Graphs: Depth First Search

```
depthFirstSearchNonRecursive(value)
        vertex = search(value)
        vertex.visited = true
        vertex.distance = 0
        stack.push(vertex)
        while (!stack.isEmpty())
                ve = stack.pop()
                print(ve.key)
                for x=0 to ve.adjacent.end
                        if (!ve.adjacent[x].v.visited)
                                ve.adjacent[x].v.visited = true
                                stack.push(ve.adjacent[x].v)
```

Dijkstra’s Algorithm

- Edsger W. Dijkstra (1956)
- Breadth first search finds shortest distance of unweighted graph
  - This finds the minimum number of edges to the destination
- Distance in weighted graphs is calculated by adding edge weights
- Instead of traveling along a few expensive edges there may be a path along many cheaply traversable edges
  - BFS will always return the few expensive edges

Dijkstra’s Algorithm

- Struct for Disjkstra’s vertex:

    string key

    vector<adjVertex> adjacent

    bool solved // as opposed to simply ‘visited’

    int distance

    vertex* parent

Dijkstra’s Algorithm

- Mark the start node **solved** and set it’s distance as **0**.
  - Look to all unsolved adjacent vertices
  - The vertex with shortest distance is marked solved with the distance from the start vertex
- No solved vertex will be solved again

Dijkstra’s Algorithm

- Instead of looking at adjacent vertices of a single vertex
  - we look to the set of adjacent vertices to any already solved vertex

- Find the shortest path from this set and mark that vertex solved
- Keep distance in reference to the start vertex
  - When marking a node solved, distance is the edge weight in plus the distance at the vertex the edge leaves

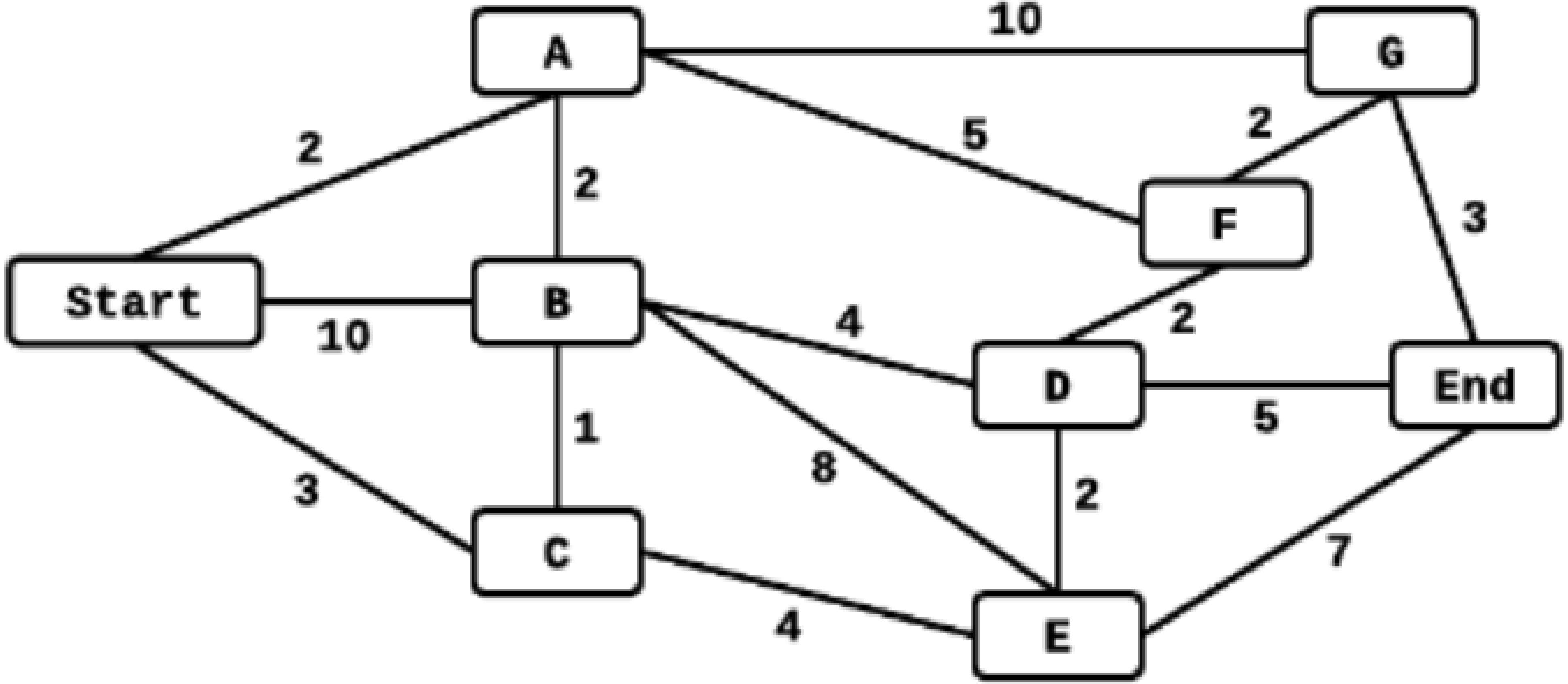
Dijkstra's Algorithm

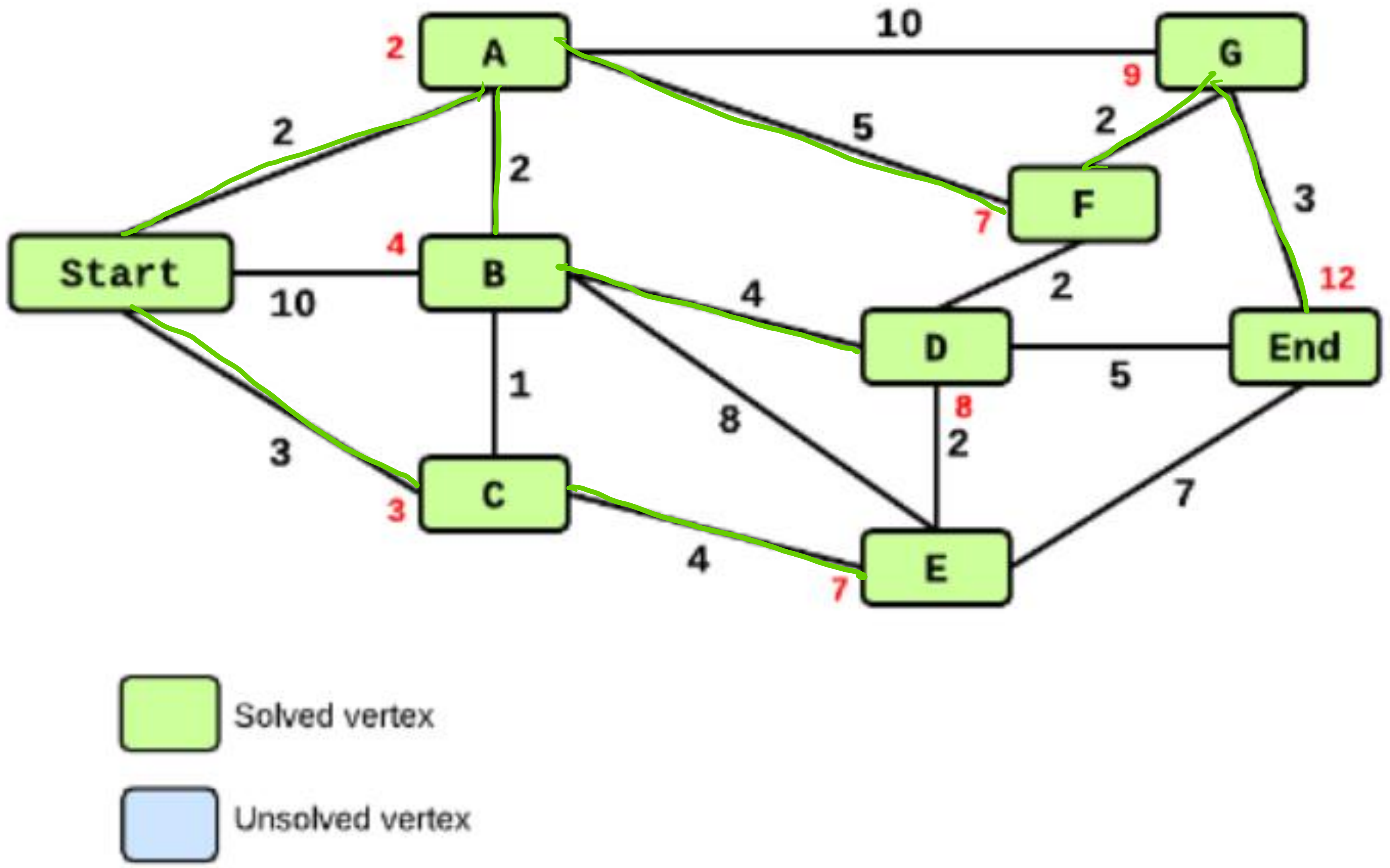
```
Dijkstra(start, end)
        // Find the start and end node
        startV = search(start)
        endV = search(end)

        // Mark the start as solved with distance 0
        startV.solved = true
        startV.distance = 0

        // Store list of solved vertices
        solved = {startV}
        …
```

Dijkstra’s Algorithm

```
Dijkstra(start, end)
    …
    while (!endV.solved)
        minDistance = INT_MAX // arbitrarily large weight
        solvedV = NULL
        for x=0 to solved.end // for all solved vertices
            s = solved[x]
            for y=0 to s.adjacent.end // look at all adjacent
                if (!s.adjacent[y].v.solved)
                    dist = s.distance + s.adjacent[y].weight
                    if (dist < minDistance)
                        solvedV = s.adjacent[y].v
                        minDistance = dist
                        parent = s
        solvedV.distance = minDistance
        solvedV.parent = parent
        solvedV.solved = true
        solved.add(solvedV)
    return endV
```

Dijkstra’s Example





Start
A
B
C
D
E
F
G
End
2
4
3
8
7
7
9
12
10
2
2
10
1
3
4
8
4
5
2
2
2
5
3
7
Solved vertex
Unsolved vertex

Dijkstra's Example

Solved

Start	A	C	B	F	E	D	End