

# Data Structures and Algorithms

*CS245-2015S-17*

## *Shortest Path Dijkstra's Algorithm*

David Galles

Department of Computer Science

University of San Francisco

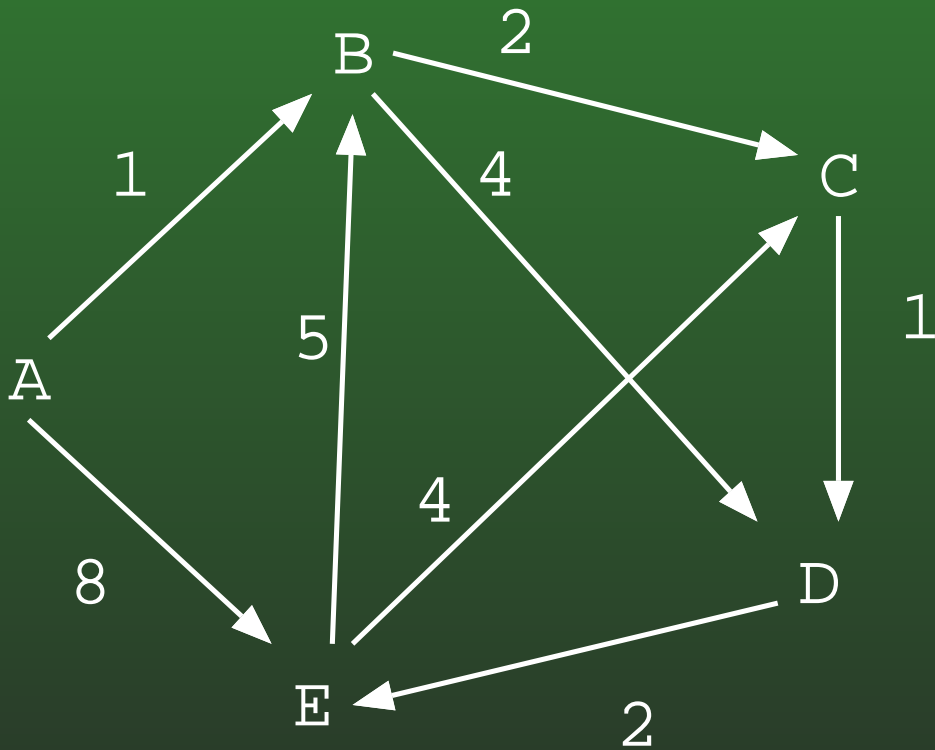
# 17-0: Computing Shortest Path

---

- Given a directed weighted graph  $G$  (all weights non-negative) and two vertices  $x$  and  $y$ , find the least-cost path from  $x$  to  $y$  in  $G$ .
  - Undirected graph is a special case of a directed graph, with symmetric edges
- Least-cost path may not be the path containing the fewest edges
  - “shortest path” == “least cost path”
  - “path containing fewest edges” = “path containing fewest edges”

# 17-1: Shortest Path Example

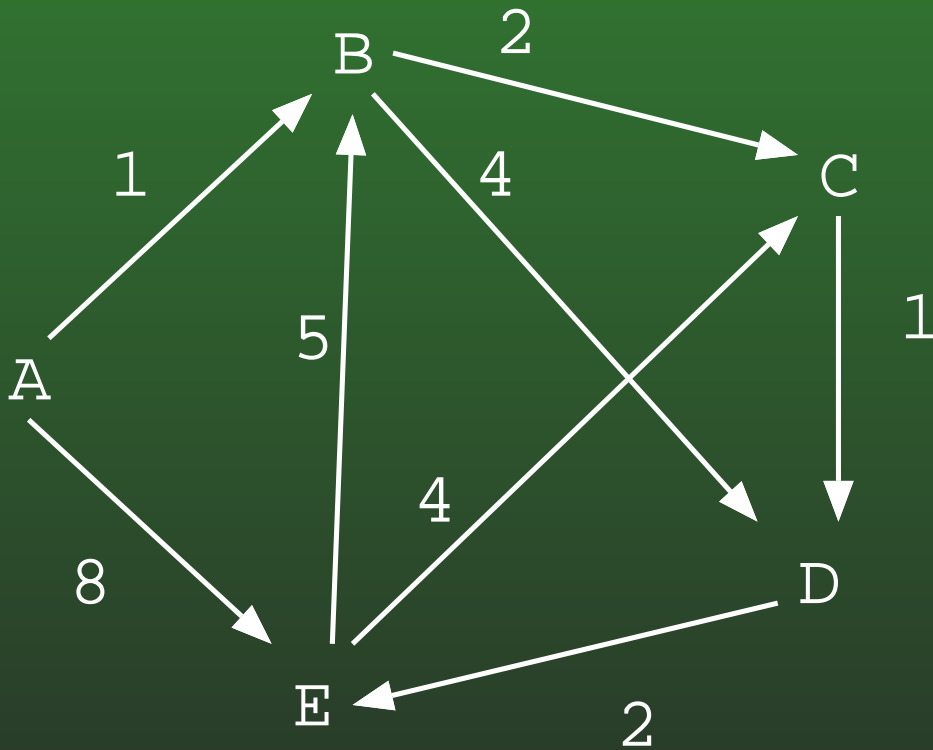
- Shortest path  $\neq$  path containing fewest edges



- Shortest Path from A to E?

# 17-2: Shortest Path Example

- Shortest path  $\neq$  path containing fewest edges



- Shortest Path from A to E:
  - A, B, C, D, E

# 17-3: Single Source Shortest Path

---

- To find the shortest path from vertex  $x$  to vertex  $y$ , we need (worst case) to find the shortest path from  $x$  to *all* other vertices in the graph
  - Why?

# 17-4: Single Source Shortest Path

---

- To find the shortest path from vertex  $x$  to vertex  $y$ , we need (worst case) to find the shortest path from  $x$  to *all* other vertices in the graph
  - To find the shortest path from  $x$  to  $y$ , we need to find the shortest path from  $x$  to all nodes on the path from  $x$  to  $y$
  - Worst case, *all* nodes will be on the path

# 17-5: Single Source Shortest Path

---

- If all edges have unit weight ...

# 17-6: Single Source Shortest Path

---

- If all edges have unit weight,
- We can use Breadth First Search to compute the shortest path
- BFS Spanning Tree contains shortest path to each node in the graph
  - Need to do some more work to create & save BFS spanning tree
- When edges have differing weights, this obviously will not work

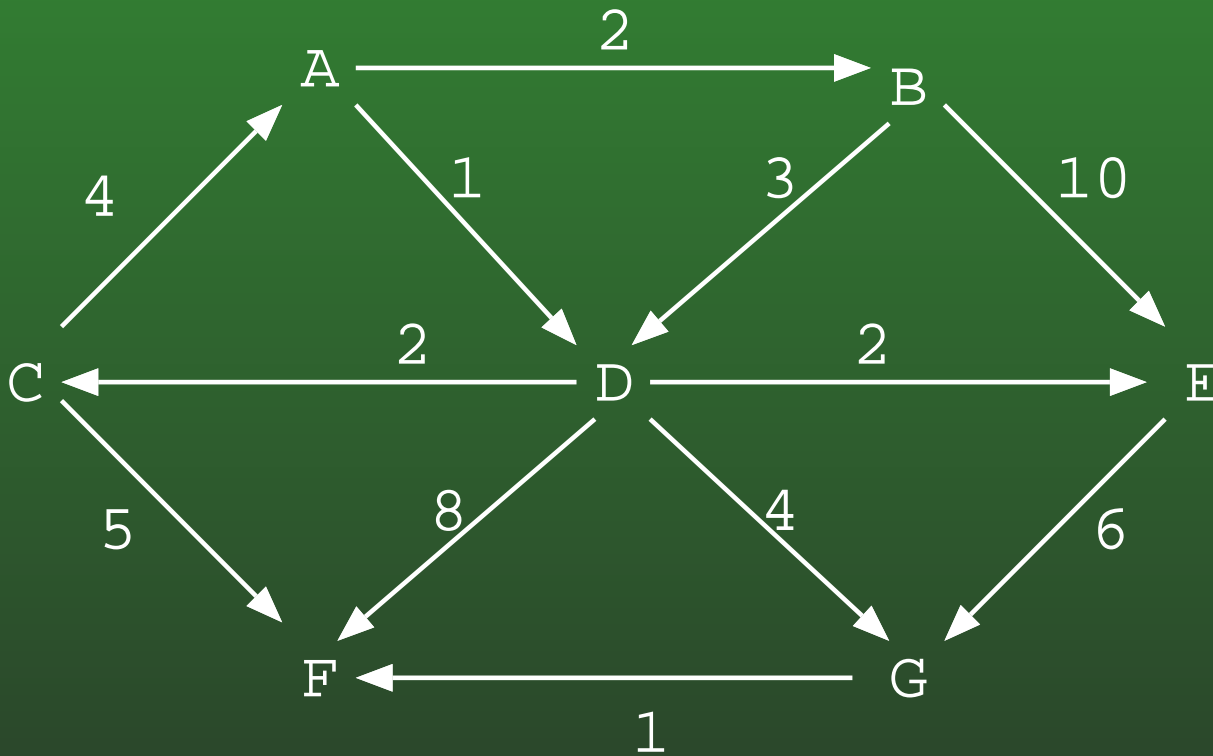


# 17-7: Single Source Shortest Path

---

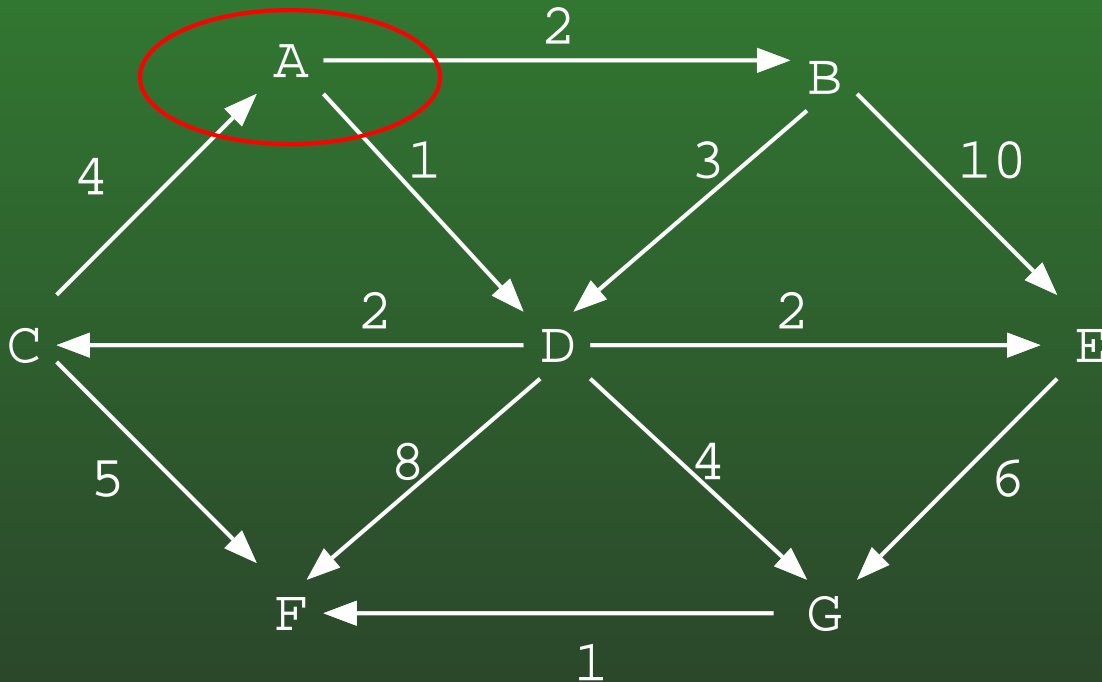
- Divide the vertices into two sets:
  - Vertices whose shortest path from the initial vertex is known
  - Vertices whose shortest path from the initial vertex is not known
- Initially, only the initial vertex is known
- Move vertices one at a time from the unknown set to the known set, until all vertices are known

# 17-8: Single Source Shortest Path



- Start with the vertex A

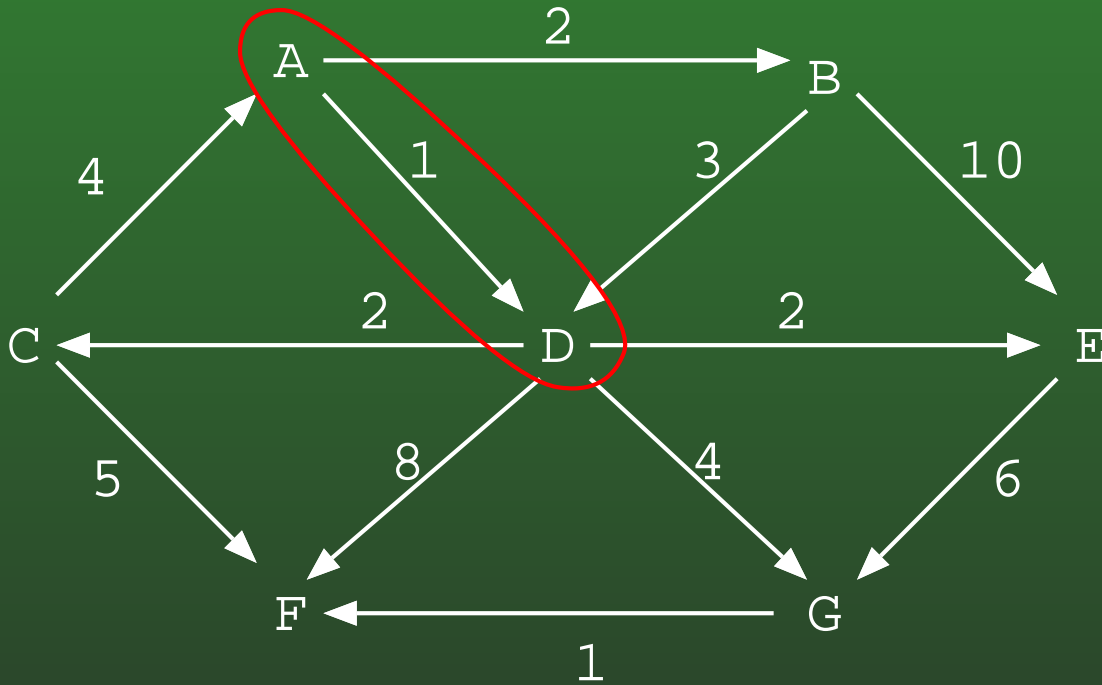
# 17-9: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    |          |
| C    |          |
| D    |          |
| E    |          |
| F    |          |
| G    |          |

- Known vertices are circled in red
- We can now extend the known set by 1 vertex

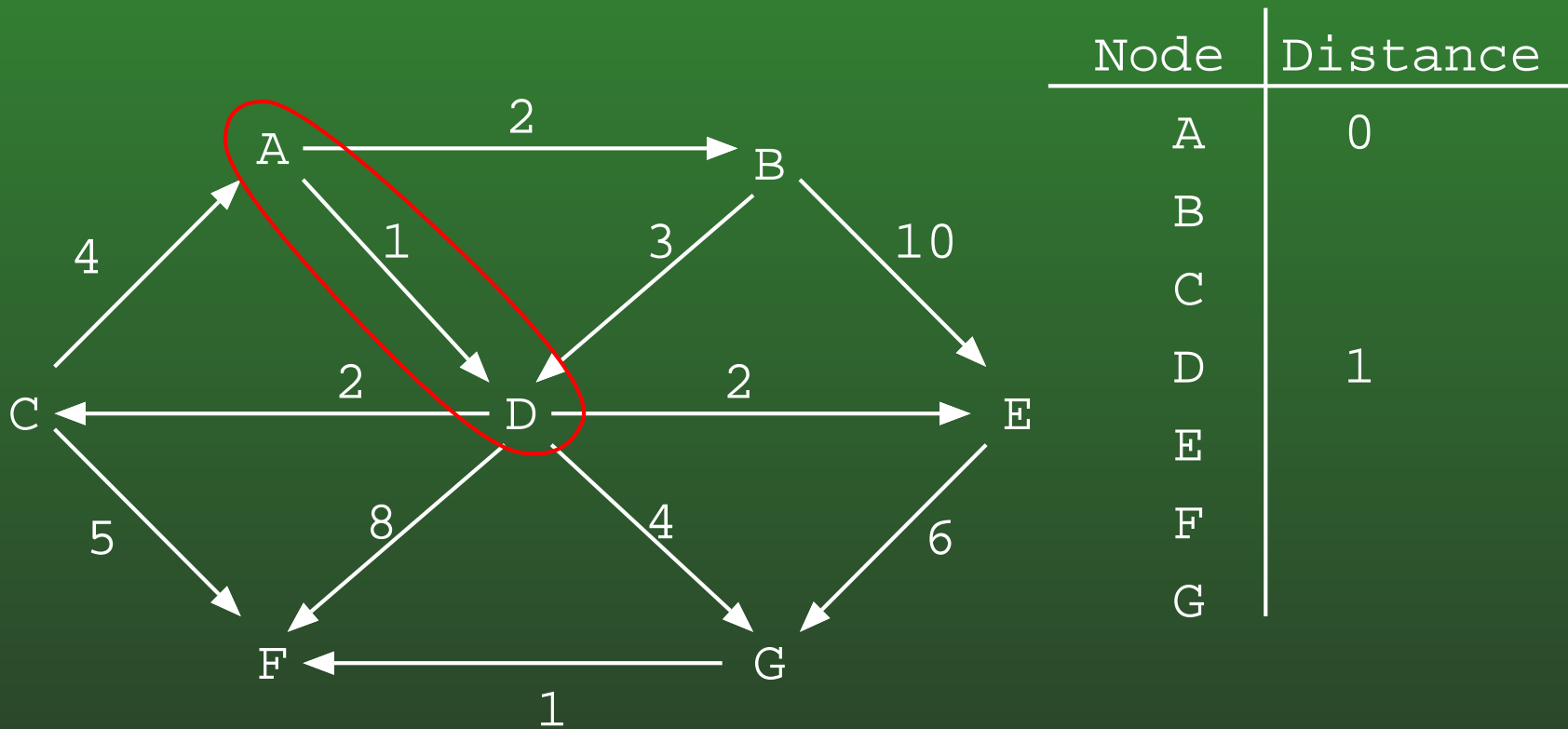
# 17-10: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    |          |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

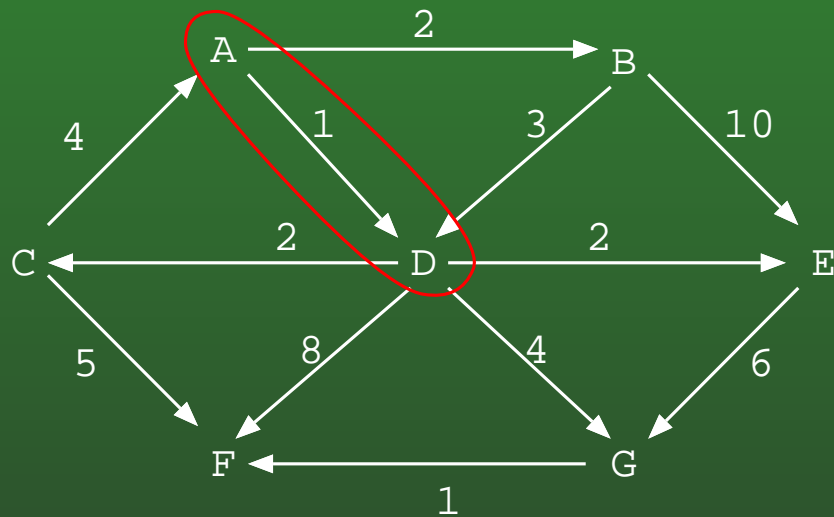
- Why is it safe to add D, with cost 1?

# 17-11: Single Source Shortest Path



- Why is it safe to add D, with cost 1?
  - Could we do better with a more roundabout path?

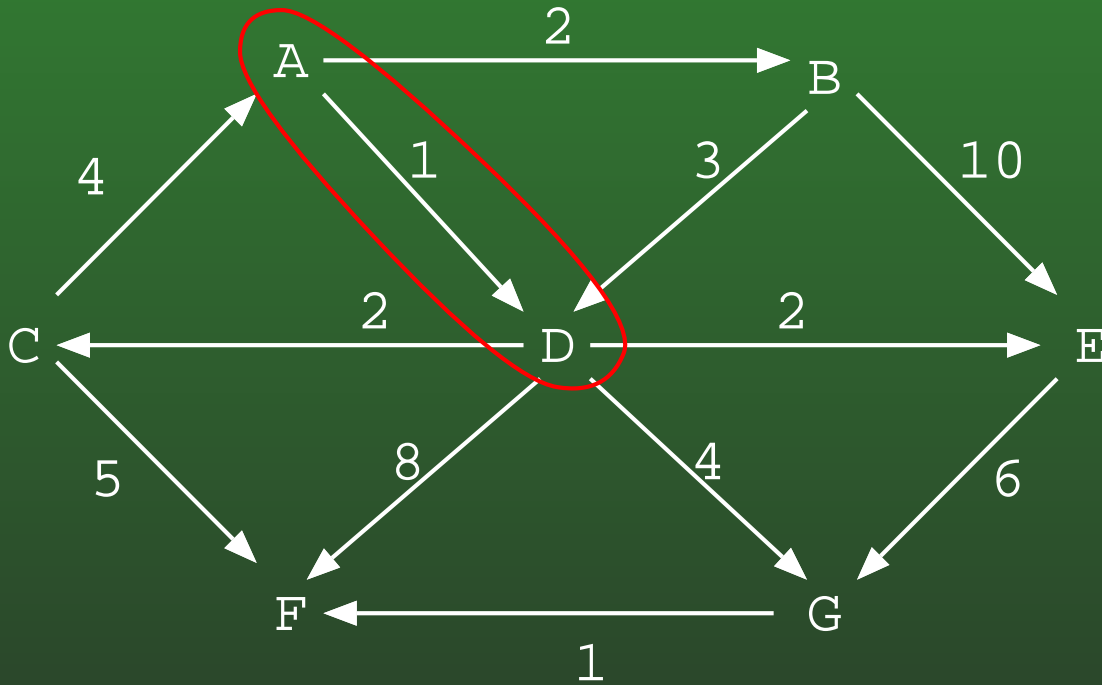
# 17-12: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    |          |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- Why is it safe to add D, with cost 1?
  - Could we do better with a more roundabout path?
  - No – to get to any other node will cost at least 1
  - No negative edge weights, can't do better than 1

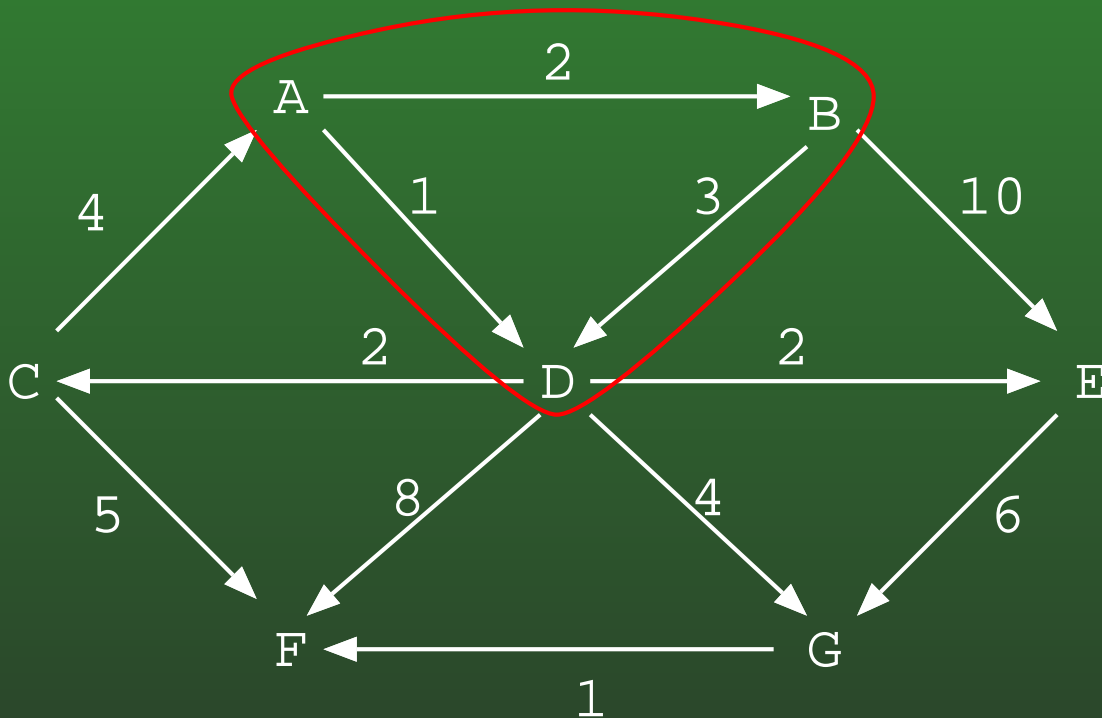
# 17-13: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    |          |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- We can now add another vertex to our known list ...

# 17-14: Single Source Shortest Path

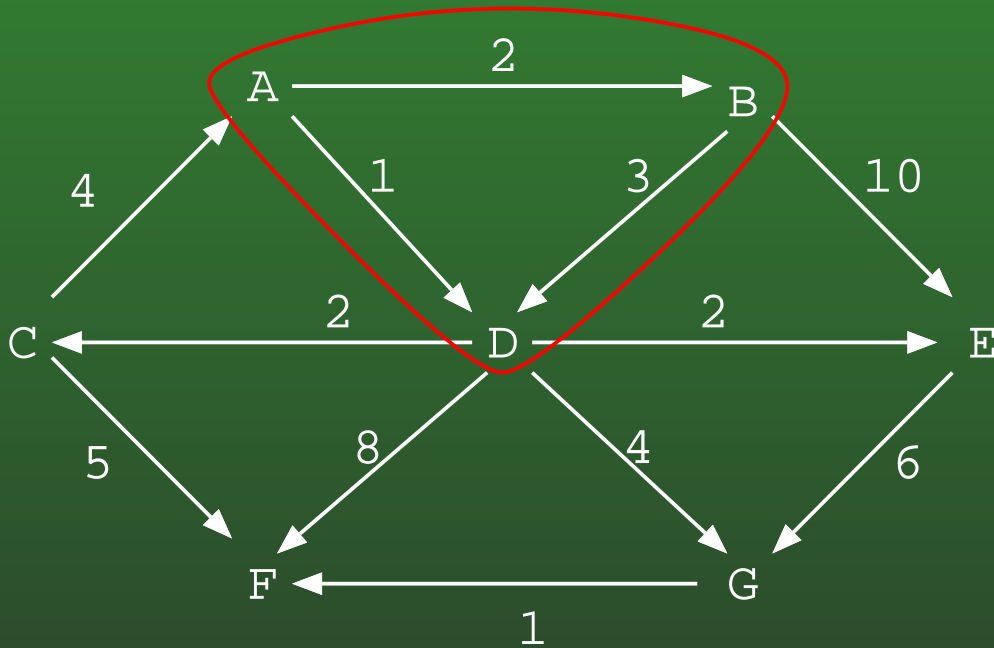


| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- How do we know that we could not get to B cheaper than by going through D?



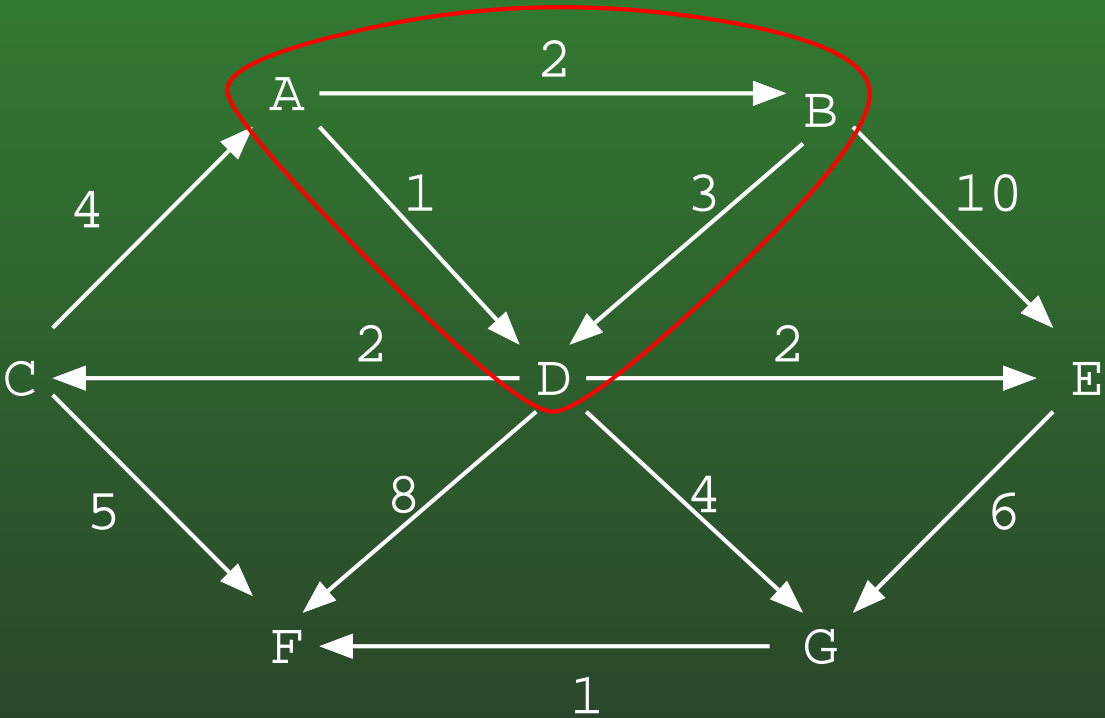
# 17-15: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- How do we know that we could not get to B cheaper than by going through D?
  - Costs 1 to get to D
  - Costs at least 2 to get anywhere from D
    - Cost *at least*  $(1+2 = 3)$  to get to B through D

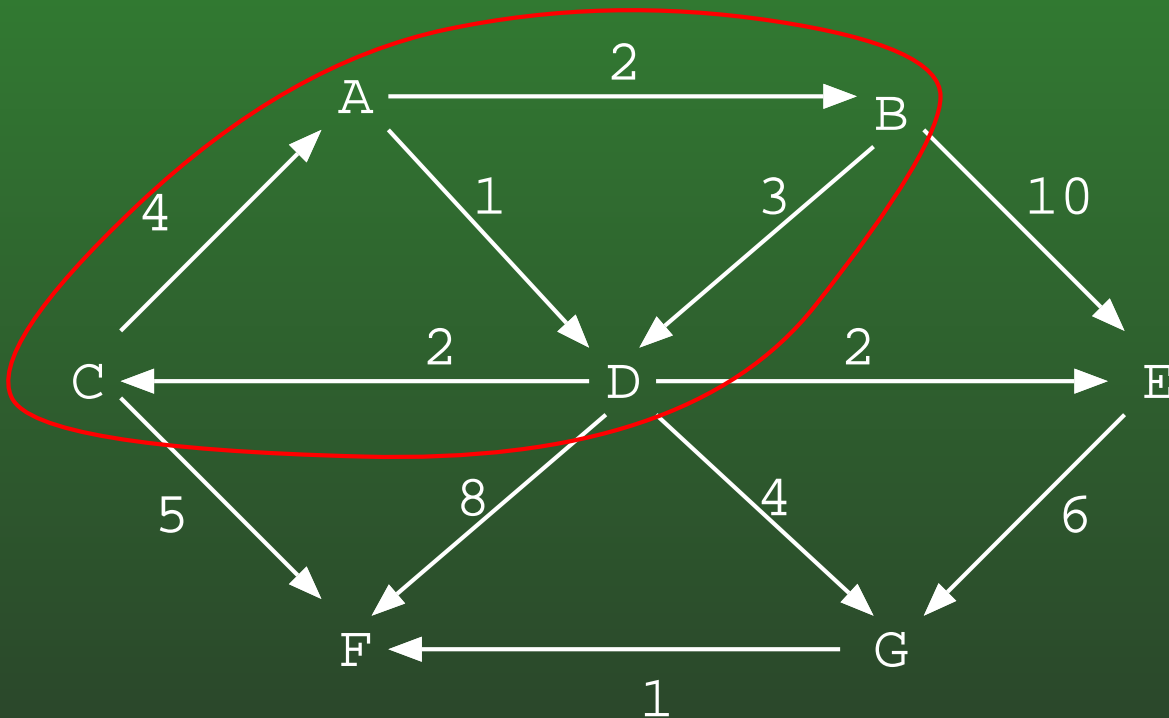
# 17-16: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    |          |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- Next node we can add ...

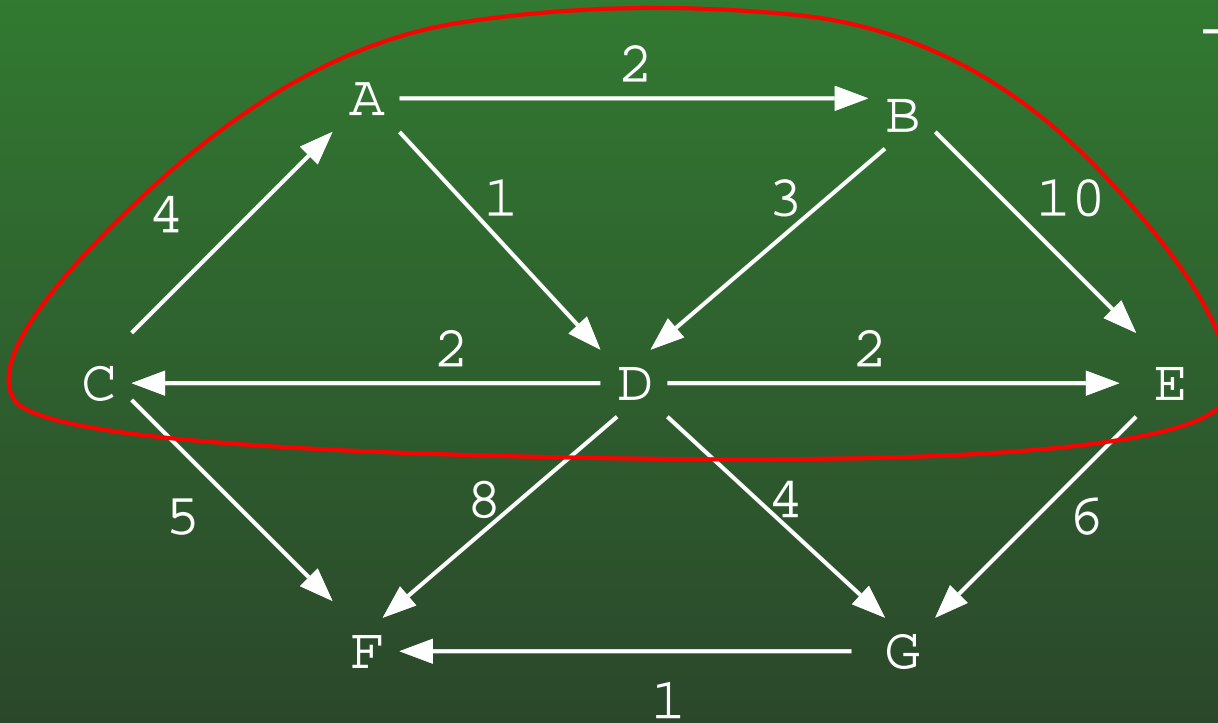
# 17-17: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    | 3        |
| D    | 1        |
| E    |          |
| F    |          |
| G    |          |

- (We also could have added E for this step)
- Next vertex to add to Known ...

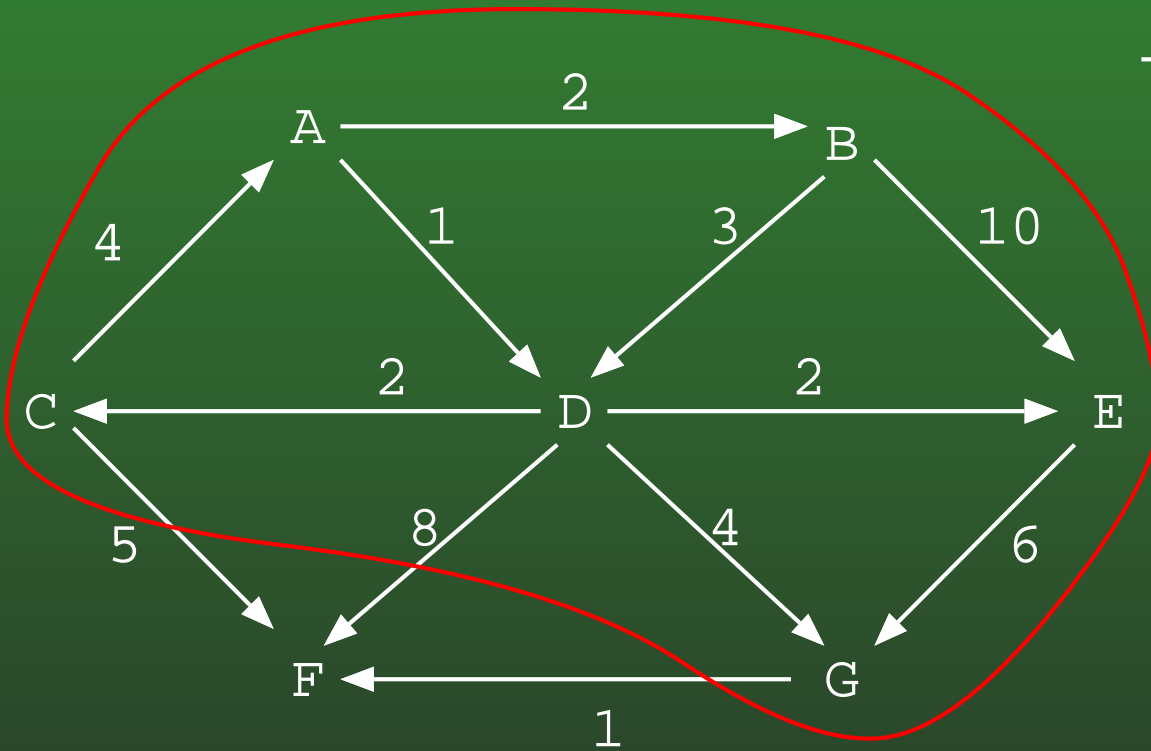
# 17-18: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    | 3        |
| D    | 1        |
| E    | 3        |
| F    |          |
| G    |          |

- Cost to add F is 8 (through C)
- Cost to add G is 5 (through D)

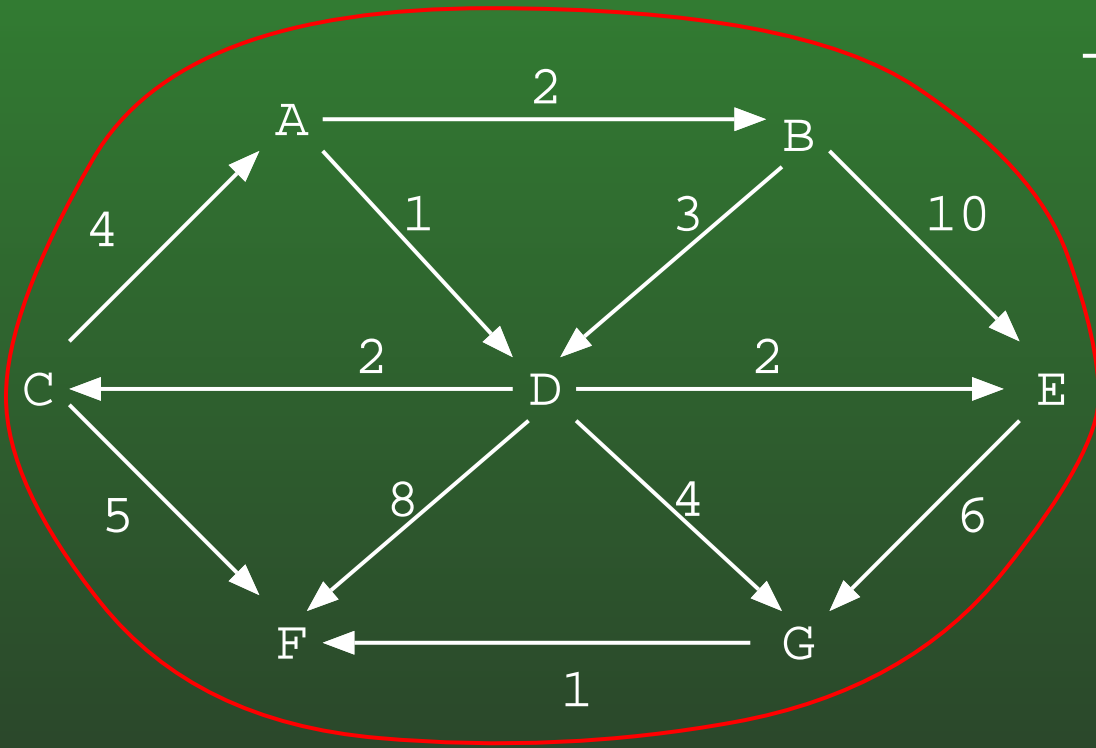
# 17-19: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    | 3        |
| D    | 1        |
| E    | 3        |
| F    | 5        |
| G    |          |

- Last node ...

# 17-20: Single Source Shortest Path



| Node | Distance |
|------|----------|
| A    | 0        |
| B    | 2        |
| C    | 3        |
| D    | 1        |
| E    | 3        |
| F    | 5        |
| G    | 6        |

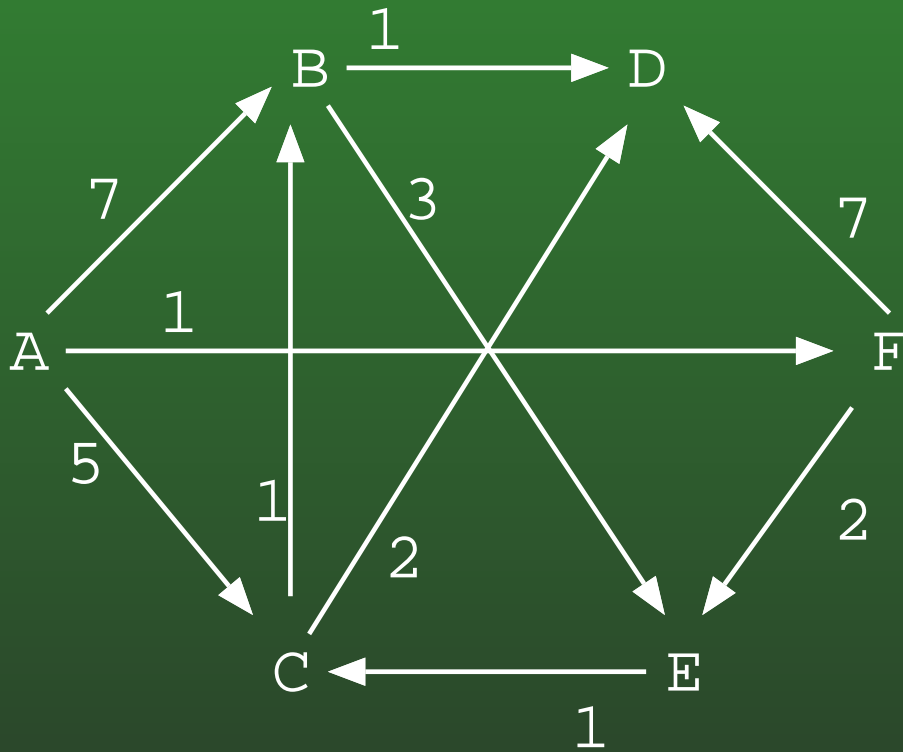
- We now know the length of the shortest path from *A* to all other vertices in the graph

# 17-21: Dijkstra's Algorithm

---

- Keep a table that contains, for each vertex
  - Is the distance to that vertex known?
  - What is the best distance we've found so far?
- Repeat:
  - Pick the smallest unknown distance
  - mark it as known
  - update the distance of all unknown neighbors of that node
- Until all vertices are known

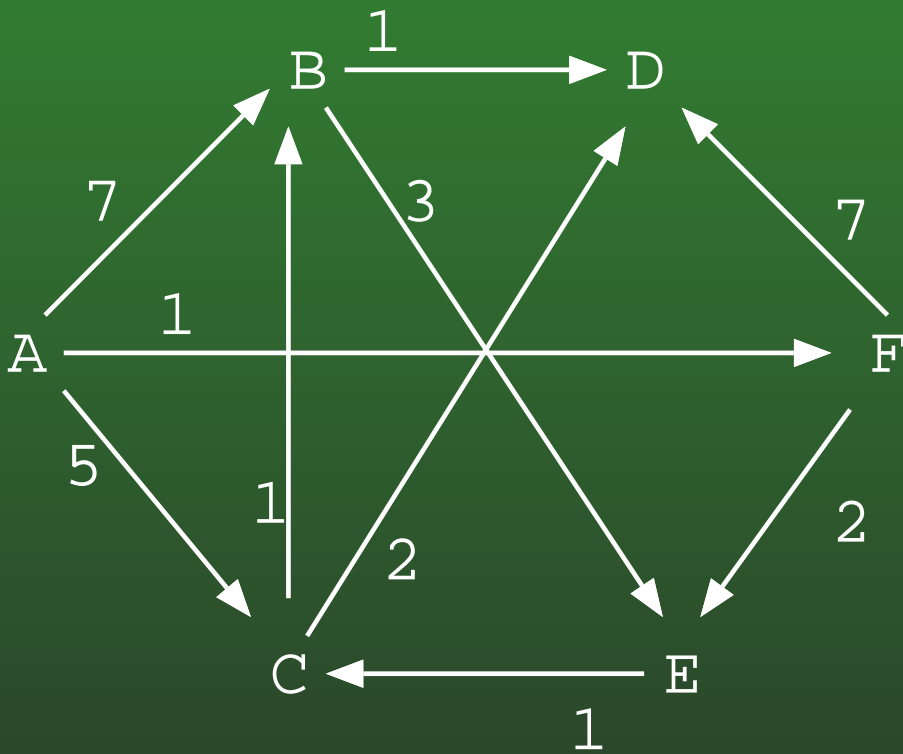
# 17-22: Dijkstra's Algorithm Example



| Node | Known | Distance |
|------|-------|----------|
| A    | false | 0        |
| B    | false | $\infty$ |
| C    | false | $\infty$ |
| D    | false | $\infty$ |
| E    | false | $\infty$ |
| F    | false | $\infty$ |

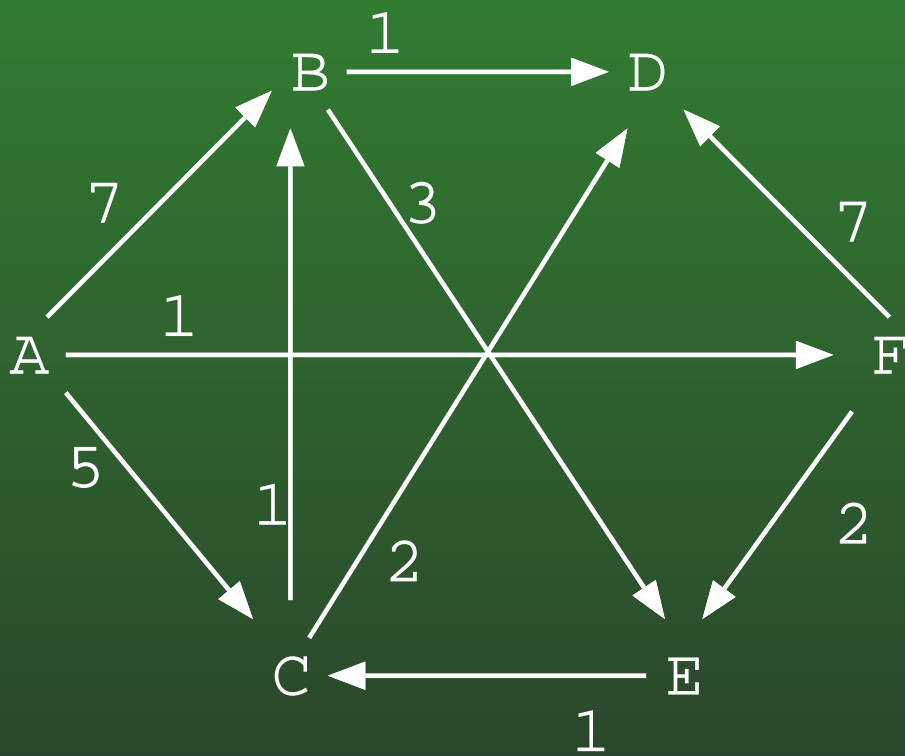


# 17-23: Dijkstra's Algorithm Example



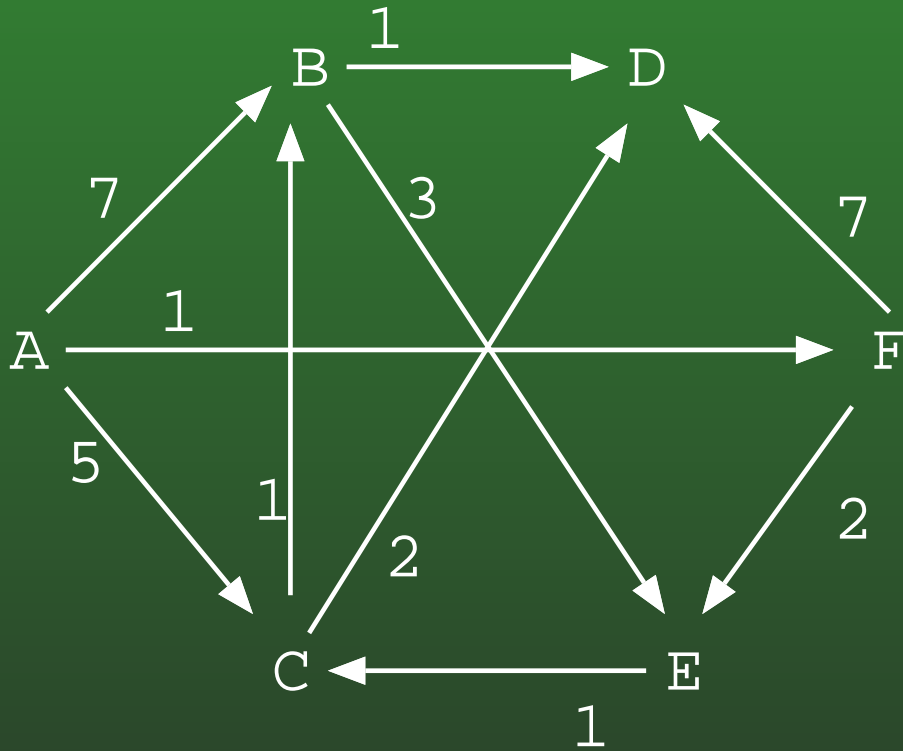
| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | false | 7        |
| C    | false | 5        |
| D    | false | $\infty$ |
| E    | false | $\infty$ |
| F    | false | 1        |

# 17-24: Dijkstra's Algorithm Example



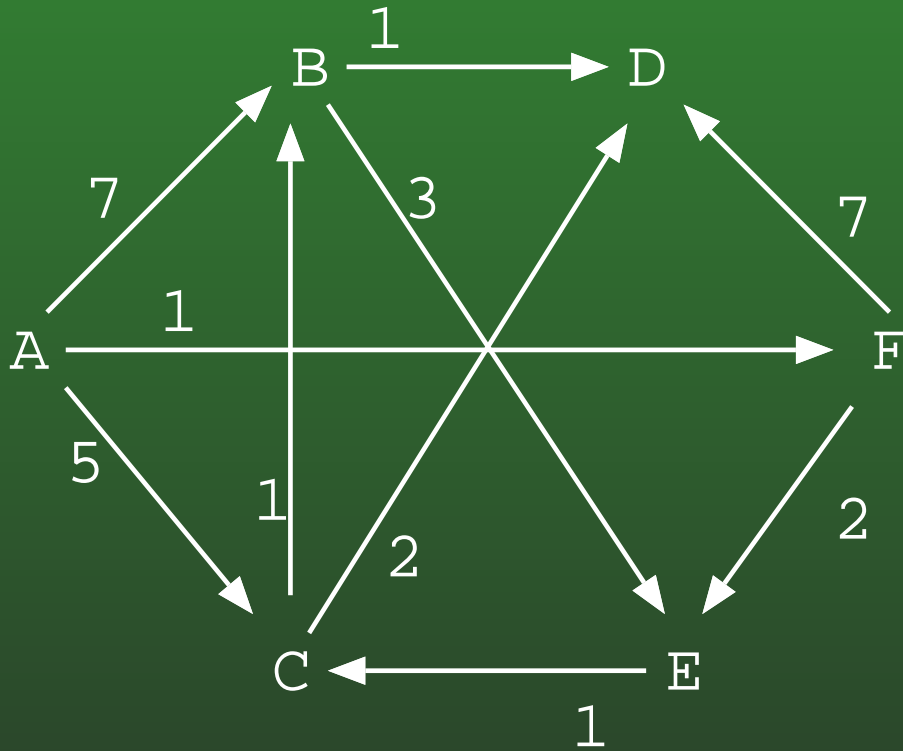
| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | false | 7        |
| C    | false | 5        |
| D    | false | 8        |
| E    | false | 3        |
| F    | true  | 1        |

# 17-25: Dijkstra's Algorithm Example



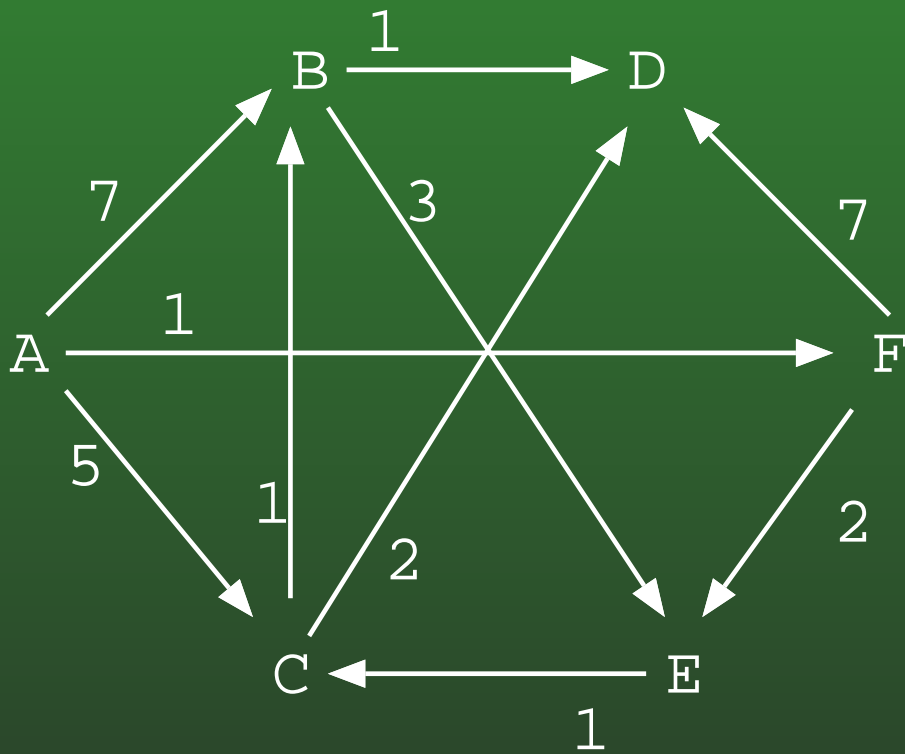
| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | false | 7        |
| C    | false | 4        |
| D    | false | 8        |
| E    | true  | 3        |
| F    | true  | 1        |

# 17-26: Dijkstra's Algorithm Example



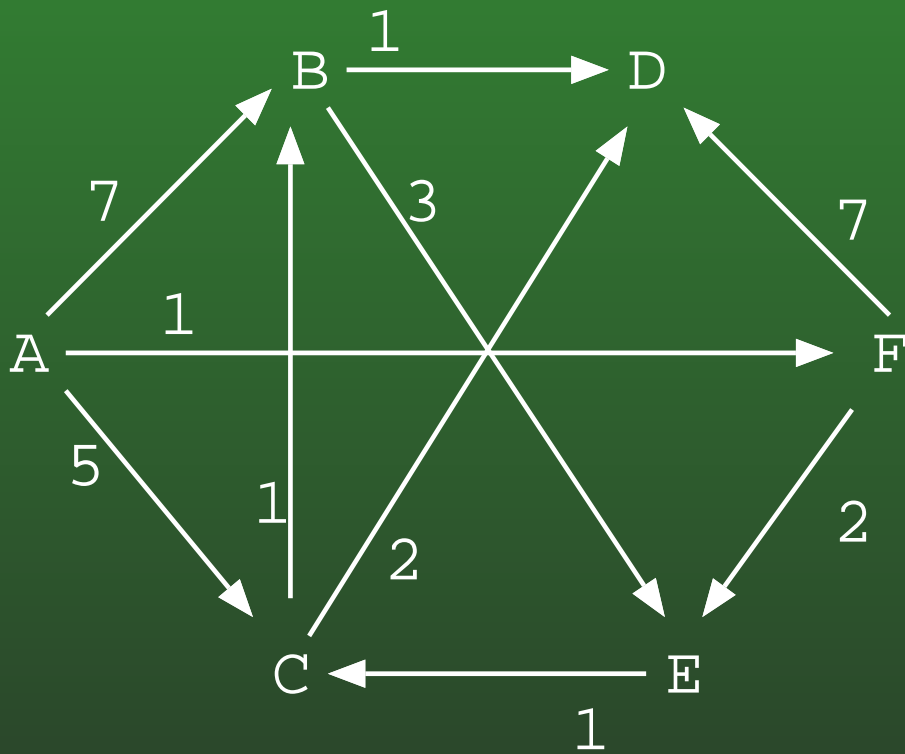
| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | false | 5        |
| C    | true  | 4        |
| D    | false | 6        |
| E    | true  | 3        |
| F    | true  | 1        |

# 17-27: Dijkstra's Algorithm Example



| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | true  | 5        |
| C    | true  | 4        |
| D    | false | 6        |
| E    | true  | 3        |
| F    | true  | 1        |

# 17-28: Dijkstra's Algorithm Example



| Node | Known | Distance |
|------|-------|----------|
| A    | true  | 0        |
| B    | true  | 5        |
| C    | true  | 4        |
| D    | true  | 6        |
| E    | true  | 3        |
| F    | true  | 1        |

# 17-29: Dijkstra's Algorithm

---

- After Dijkstra's algorithm is complete:
  - We know the *length* of the shortest path
  - We do not know *what* the shortest path is
- How can we modify Dijkstra's algorithm to compute the path?

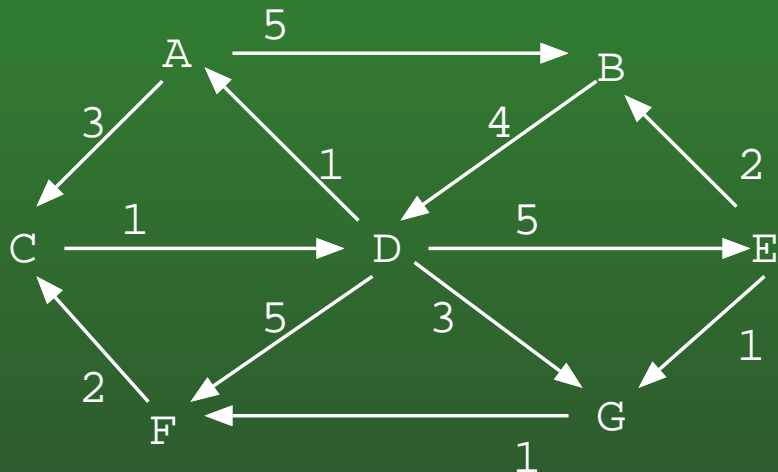
# 17-30: Dijkstra's Algorithm

---

- After Dijkstra's algorithm is complete:
  - We know the *length* of the shortest path
  - We do not know *what* the shortest path is
- How can we modify Dijkstra's algorithm to compute the path?
  - Store not only the distance, but the immediate parent that led to this distance

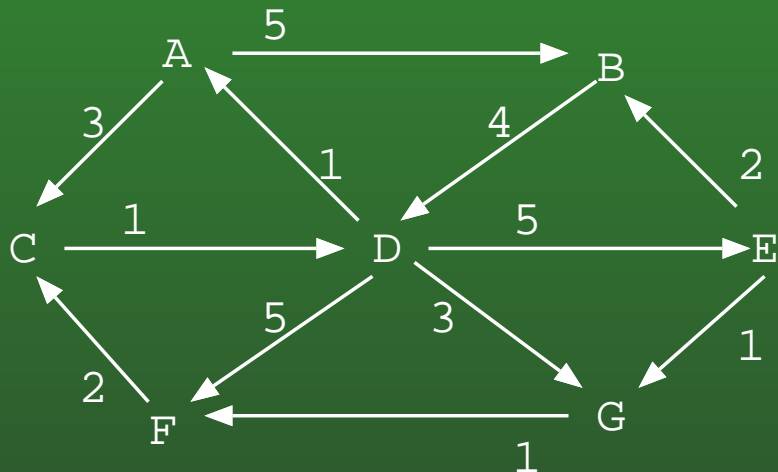


# 17-31: Dijkstra's Algorithm Example



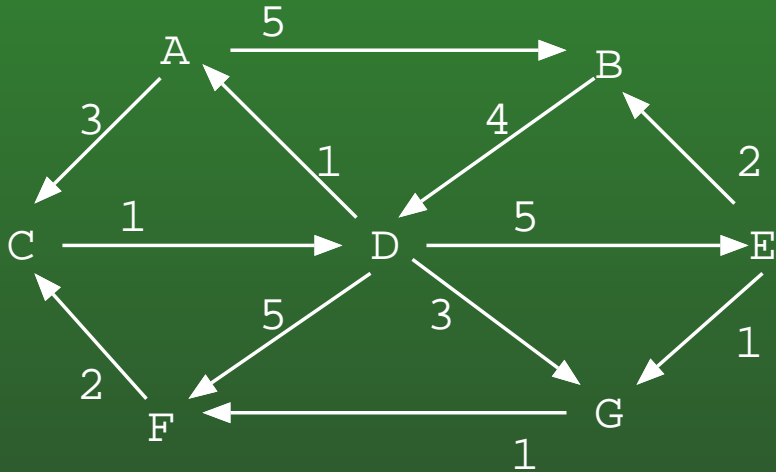
| Node | Known | Dist     | Path |
|------|-------|----------|------|
| A    | false | 0        |      |
| B    | false | $\infty$ |      |
| C    | false | $\infty$ |      |
| D    | false | $\infty$ |      |
| E    | false | $\infty$ |      |
| F    | false | $\infty$ |      |
| G    | false | $\infty$ |      |

# 17-32: Dijkstra's Algorithm Example



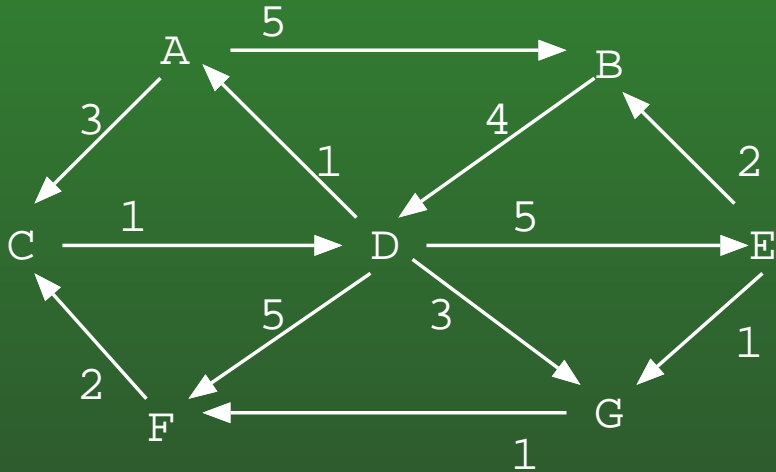
| Node | Known | Dist     | Path |
|------|-------|----------|------|
| A    | true  | 0        |      |
| B    | false | 5        | A    |
| C    | false | 3        | A    |
| D    | false | $\infty$ |      |
| E    | false | $\infty$ |      |
| F    | false | $\infty$ |      |
| G    | false | $\infty$ |      |

# 17-33: Dijkstra's Algorithm Example



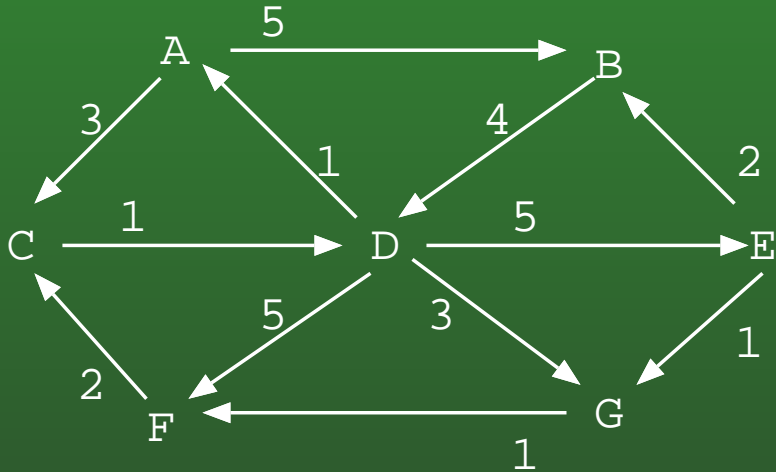
| Node | Known | Dist     | Path |
|------|-------|----------|------|
| A    | true  | 0        |      |
| B    | false | 5        | A    |
| C    | true  | 3        | A    |
| D    | false | 4        | C    |
| E    | false | $\infty$ |      |
| F    | false | $\infty$ |      |
| G    | false | $\infty$ |      |

# 17-34: Dijkstra's Algorithm Example



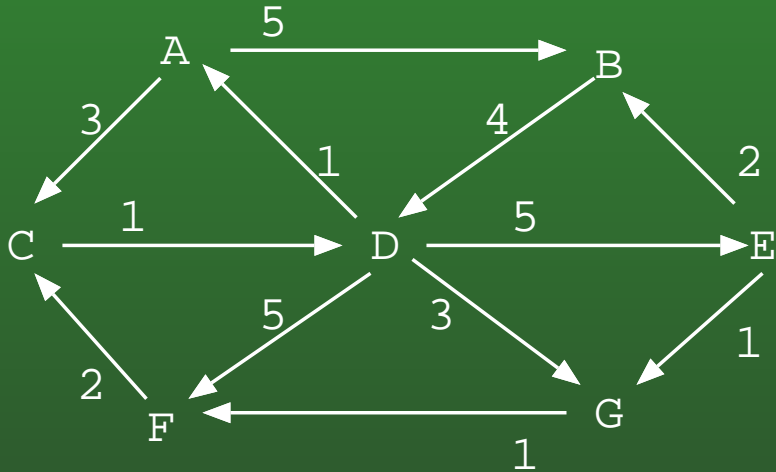
| Node | Known | Dist | Path |
|------|-------|------|------|
| A    | true  | 0    |      |
| B    | false | 5    | A    |
| C    | true  | 3    | A    |
| D    | true  | 4    | C    |
| E    | false | 9    | D    |
| F    | false | 9    | D    |
| G    | false | 7    | D    |

# 17-35: Dijkstra's Algorithm Example



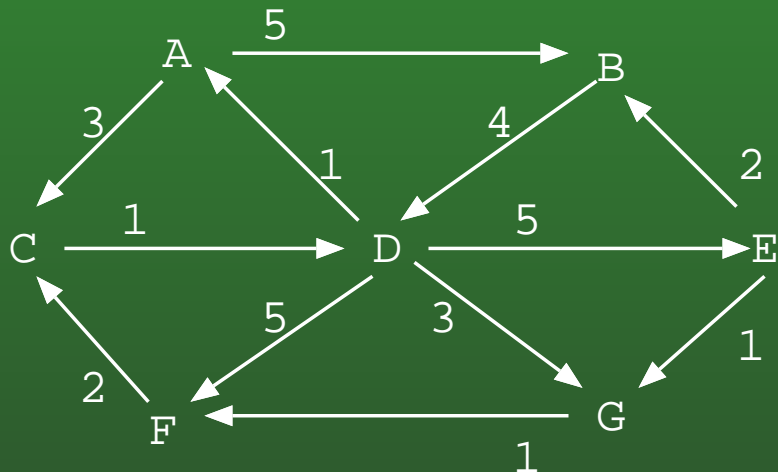
| Node | Known | Dist | Path |
|------|-------|------|------|
| A    | true  | 0    |      |
| B    | true  | 5    | A    |
| C    | true  | 3    | A    |
| D    | true  | 4    | C    |
| E    | false | 9    | D    |
| F    | false | 9    | D    |
| G    | false | 7    | D    |

# 17-36: Dijkstra's Algorithm Example



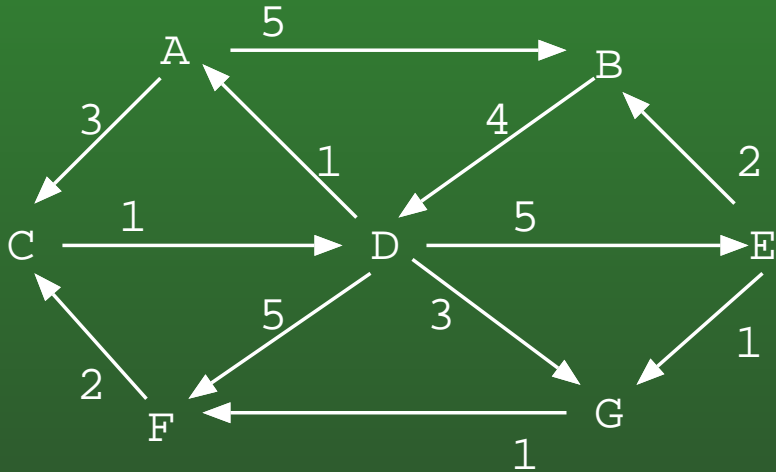
| Node | Known | Dist | Path |
|------|-------|------|------|
| A    | true  | 0    |      |
| B    | true  | 5    | A    |
| C    | true  | 3    | A    |
| D    | true  | 4    | C    |
| E    | false | 9    | D    |
| F    | false | 8    | G    |
| G    | true  | 7    | D    |

# 17-37: Dijkstra's Algorithm Example



| Node | Known | Dist | Path |
|------|-------|------|------|
| A    | true  | 0    |      |
| B    | true  | 5    | A    |
| C    | true  | 3    | A    |
| D    | true  | 4    | C    |
| E    | false | 9    | D    |
| F    | true  | 8    | G    |
| G    | true  | 7    | D    |

# 17-38: Dijkstra's Algorithm Example



| Node | Known | Dist | Path |
|------|-------|------|------|
| A    | true  | 0    |      |
| B    | true  | 5    | A    |
| C    | true  | 3    | A    |
| D    | true  | 4    | C    |
| E    | true  | 9    | D    |
| F    | true  | 8    | G    |
| G    | true  | 7    | D    |



# 17-39: Dijkstra's Algorithm

---

- Given the “path” field, we can construct the shortest path
  - Work backward from the end of the path
  - Follow the “path” pointers until the start node is reached
    - We can use a sentinel value in the “path” field of the initial node, so we know when to stop

# 17-40: Dijkstra Code

---

```
void Dijkstra(Edge G[], int s, tableEntry T[]) {
    int i, v;
    Edge e;
    for(i=0; i<G.length; i++) {
        T[i].distance = Integer.MAX_VALUE;
        T[i].path = -1;
        T[i].known = false;
    }
    T[s].distance = 0;
    for (i=0; i < G.length; i++) {
        v = minUnknownVertex(T);
        T[v].known = true;
        for (e = G[v]; e != null; e = e.next) {
            if (T[e.neighbor].distance >
                T[v].distance + e.cost) {
                T[e.neighbor].distance = T[v].distance + e.cost;
                T[e.neighbor].path = v;
            }
        }
    }
}
```

## 17-41: minUnknownVertex

---

- Calculating minimum distance unknown vertex:

```
int minUnknownVertex(tableEntry T[]) {
    int i;
    int minVertex = -1;
    int minDistance = Integer.MAX_VALUE;
    for (i=0; i < T.length; i++) {
        if ((!T[i].known) &&
            (T[i].distance < minDistance)) {
            minVertex = i;
            minDistance = T[i].distance;
        }
    }
    return minVertex;
}
```

# 17-42: Dijkstra Running Time

---

- Time for initialization:

```
for(i=0; i<G.length; i++) {  
    T[i].distance = Integer.MAX_VALUE;  
    T[i].path = -1;  
    T[i].known = false;  
}  
T[s].distance = 0;
```

# 17-43: Dijkstra Running Time

---

- Time for initialization:

```
for(i=0; i<G.length; i++) {  
    T[i].distance = Integer.MAX_VALUE;  
    T[i].path = -1;  
    T[i].known = false;  
}  
T[s].distance = 0;
```

- $\Theta(V)$

# 17-44: Dijkstra Running Time

- Total time for all calls to `minUnknownVertex`, and setting `T[v].known = true` (for all iterations of the loop)

```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);          < These two lines
    T[v].known = true;                < -----
    for (e = G[v]; e != null; e = e.next) {
        if (T[e.neighbor].distance >
            T[v].distance + e.cost) {
            T[e.neighbor].distance = T[v].distance + e.cost;
            T[e.neighbor].path = v;
        }
    }
}
```

# 17-45: Dijkstra Running Time

- Total time for all calls to `minUnknownVertex`, and setting `T[v].known = true` (for all iterations of the loop)

```
for (i=0; i < G.length; i++) {  
    v = minUnknownVertex(T);           < These two lines  
    T[v].known = true;                 < -----  
    for (e = G[v]; e != null; e = e.next) {  
        if (T[e.neighbor].distance >  
            T[v].distance + e.cost) {  
            T[e.neighbor].distance = T[v].distance + e.cost;  
            T[e.neighbor].path = v;  
        }  
    }  
}
```

- $\Theta(V^2)$

# 17-46: Dijkstra Running Time

---

- Total # of times the if statement will be executed:

```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
|>     if (T[e.neighbor].distance >
|>         T[v].distance + e.cost) {
|>         T[e.neighbor].distance = T[v].distance + e.cost;
|>         T[e.neighbor].path = v;
        }
    }
}
```



# 17-47: Dijkstra Running Time

---

- Total # of times the if statement will be executed:

```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
|>     if (T[e.neighbor].distance >
|>         T[v].distance + e.cost) {
|>         T[e.neighbor].distance = T[v].distance + e.cost;
|>         T[e.neighbor].path = v;
        }
    }
}
```

- $E$

# 17-48: Dijkstra Running Time

---

- Total running time for all iterations of the inner for statement:

```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
|>   for (e = G[v]; e != null; e = e.next) {
|>       if (T[e.neighbor].distance >
|>           T[v].distance + e.cost) {
|>           T[e.neighbor].distance = T[v].distance + e.cost;
|>           T[e.neighbor].path = v;
        }
    }
}
```

# 17-49: Dijkstra Running Time

- Total running time for all iterations of the inner for statement:

```
for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
|> for (e = G[v]; e != null; e = e.next) {
|>     if (T[e.neighbor].distance >
|>         T[v].distance + e.cost) {
|>         T[e.neighbor].distance = T[v].distance + e.cost;
|>         T[e.neighbor].path = v;
|>     }
|> }
}
```

- $\Theta(V + E)$ 
  - Why  $\Theta(V + E)$  and not just  $\Theta(E)$ ?

# 17-50: Dijkstra Running Time

---

- Total running time:
- Sum of:
  - Time for initialization
  - Time for executing all calls to `minUnknownVertex`
  - Time for executing all distance / path updates
- $= \Theta(V + V^2 + (V + E)) = \Theta(V^2)$

# 17-51: Improving Dijkstra

---

- Can we do better than  $\Theta(V^2)$
- For *dense* graphs, we can't do better
  - To ensure that the shortest path to all vertices is computed, need to look at all edges in the graph
  - A dense graph can have  $\Theta(V^2)$  edges
- For *sparse* graphs, we can do better
  - Where should we focus our attention?

# 17-52: Improving Dijkstra

---

- Can we do better than  $\Theta(V^2)$
- For *dense* graphs, we can't do better
  - To ensure that the shortest path to all vertices is computed, need to look at all edges in the graph
  - A dense graph can have  $\Theta(V^2)$  edges
- For *sparse* graphs, we can do better
  - Where should we focus our attention?
  - Finding the unknown vertex with minimum cost!

# 17-53: Improving Dijkstra

---

- To improve the running time of Dijkstra:
  - Place all of the vertices on a min-heap
    - Key value for min-heap = distance of vertex from initial
  - While min-heap is not empty:
    - Pop smallest value off min-heap
    - Update table
- Problems with this method?

# 17-54: Improving Dijkstra

---

- To improve the running time of Dijkstra:
  - Place all of the vertices on a min-heap
    - Key value for min-heap = distance of vertex from initial
  - While min-heap is not empty:
    - Pop smallest value off min-heap
    - Update table
- Problems with this method?
  - When we update the table, we need to rearrange the heap



# 17-55: Rearranging the heap

---

- Store a pointer for each vertex back into the heap
- When we update the table, we need to do a decrease-key operation
- Decrease-key can take up to time  $O(\lg V)$ .
- (Examples!)

# 17-56: Rearranging the heap

---

- Total time:
  - $O(V)$  remove-mins –  $O(V \lg V)$
  - $O(E)$  decrease-keys –  $O(E \lg V)$
  - Total time:  $O(V \lg V + E \lg V) \in O(E \lg V)$

# 17-57: Improving Dijkstra

---

- Store vertices in heap
- When we update the table, we need to rearrange the heap
- Alternate Solution:
  - When the cost of a vertex decreases, add a *new copy* to the heap

# 17-58: Improving Dijkstra

---

- Create a new priority queue, add start node
- While the queue is not empty:
  - Remove the vertex  $v$  with the smallest distance in the heap
  - If  $v$  is not known
    - Mark  $v$  as known
    - For each neighbor  $w$  of  $v$ 
      - If  $\text{distance}[w] > \text{distance}[v] + \text{cost}((v, w))$
      - Set  $\text{distance}[w] = \text{distance}[v] + \text{cost}((v, w))$
      - Add  $w$  to priority queue with priority  $\text{distance}[w]$

# 17-59: Improved Dijkstra Time

---

- Each vertex can be added to the heap once for each incoming edge
- Size of the heap can then be up to  $\Theta(E)$ 
  - $E$  inserts, on heap that can be up to size  $E$
  - $E$  delete-mins, on heap that can be up to size  $E$
- Total:  $\Theta(E \lg E) \in \Theta(E \lg V)$

# 17-60: Improved? Dijkstra Time

---

- Don't use priority queue, running time is  $\Theta(V^2)$
- Do use a priority queue, running time is  $\Theta(E \lg E)$
- Which is better?

# 17-61: Improved? Dijkstra Time

---

- Don't use priority queue, running time is  $\Theta(V^2)$
- Do use a priority queue, running time is  $\Theta(E \lg E)$
- Which is better?
  - For dense graphs, ( $E \in \Theta(V^2)$ ),  $\Theta(V^2)$  is better
  - For sparse graphs ( $E \in \Theta(V)$ ),  $\Theta(E \lg E)$  is better

# 17-62: Improved! Dijkstra Time

---

- If we use a data structure called a Fibonacci heap instead of a standard heap, we can implement decrease-key in constant time (on average).
- Total time:
  - $O(V)$  remove-mins –  $O(V \lg V)$
  - $O(E)$  decrease-keys –  $O(E)$  (each decrease key takes  $O(1)$  on average)
  - Total time:  $O(V \lg V + E)$



# 17-63: Negative Edges

---

- What if our graph has negative-weight edges?
  - Think of the cost of the edge as the amount of energy consumed for a segment of road
  - A downhill segment could have negative energy consumed for a hybrid
- Will Dijkstra's algorithm still work correctly?
  - Examples

# 17-64: Negative Edges

---

- What happens if there is a negative-weight cycle?
- What does the shortest path even mean?

# 17-65: Negative Edges

---

- What happens if there is a negative-weight cycle?
- What does the shortest path even mean?
  - Finding shortest paths in graphs that contain negative edges, assume that there are no negative weight cycles
  - Hybrid example

# 17-66: All-Source Shortest Path

---

- What if we want to find the shortest path from all vertices to all other vertices?
- How can we do it?

# 17-67: All-Source Shortest Path

---

- What if we want to find the shortest path from all vertices to all other vertices?
- How can we do it?
  - Run Dijkstra's Algorithm  $V$  times
  - How long will this take?
  - What about negative edges?

# 17-68: All-Source Shortest Path

---

- What if we want to find the shortest path from all vertices to all other vertices?
- How can we do it?
  - Run Dijkstra's Algorithm  $V$  times
  - How long will this take?
    - $\Theta(V E \lg E)$  (using priority queue)
      - for sparse graphs,  $\Theta(V^2 \lg V)$
      - for dense graphs,  $\Theta(V^3 \lg V)$
    - $\Theta(V^3)$  (not using a priority queue)
  - What about negative edges?
    - Doesn't work correctly

# 17-69: Floyd's Algorithm

---

- Alternate solution to all pairs shortest path
- Yields  $\Theta(V^3)$  running time for all graphs
- Works for graphs with negative edges
- Can detect negative-weight cycles

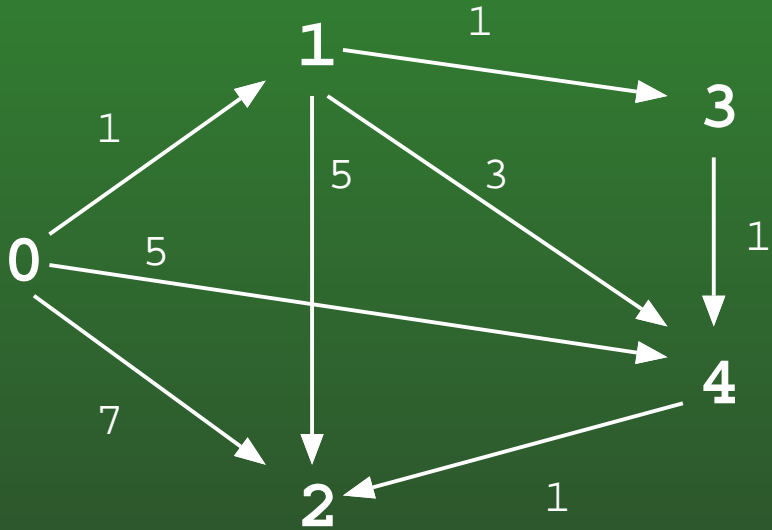
# 17-70: Floyd's Algorithm

---

- Vertices numbered from  $0..(n-1)$
- $k$ -path from vertex  $v$  to vertex  $u$  is a path whose intermediate vertices (other than  $v$  and  $u$ ) contain only vertices numbered less than or equal to  $k$
- -1-path is a direct link

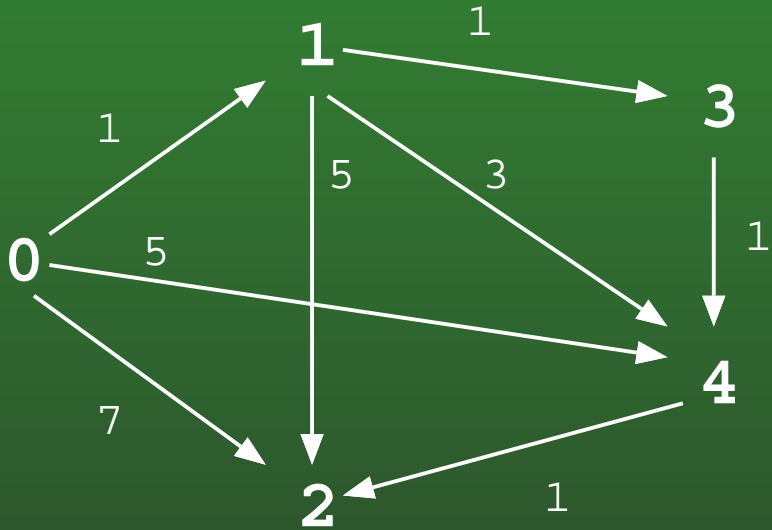


# 17-71: k-path Examples



- Shortest  $-1$ -path from 0 to 4: 5
- Shortest 0-path from 0 to 4: 5
- Shortest 1-path from 0 to 4: 4
- Shortest 2-path from 0 to 4: 4
- Shortest 3-path from 0 to 4: 3

# 17-72: k-path Examples



- Shortest -1-path from 0 to 2: 7
- Shortest 0-path from 0 to 2: 7
- Shortest 1-path from 0 to 2: 6
- Shortest 2-path from 0 to 2: 6
- Shortest 3-path from 0 to 2: 6
- Shortest 4-path from 0 to 2: 4

# 17-73: Floyd's Algorithm

---

- Shortest  $n$ -path = Shortest path
- Shortest -1-path:
  - $\infty$  if there is no direct link
  - Cost of the direct link, otherwise

# 17-74: Floyd's Algorithm

---

- Shortest  $n$ -path = Shortest path
- Shortest -1-path:
  - $\infty$  if there is no direct link
  - Cost of the direct link, otherwise
- If we could use the shortest  $k$ -path to find the shortest  $(k + 1)$  path, we would be set

# 17-75: Floyd's Algorithm

---

- Shortest  $k$ -path from  $v$  to  $u$  either goes through vertex  $k$ , or it does not
- If not:
  - Shortest  $k$ -path = shortest  $(k - 1)$ -path
- If so:
  - Shortest  $k$ -path = shortest  $k - 1$  path from  $v$  to  $k$ , followed by the shortest  $k - 1$  path from  $k$  to  $w$

# 17-76: Floyd's Algorithm

---

- If we had the shortest  $k$ -path for all pairs  $(v, w)$ , we could obtain the shortest  $k + 1$ -path for all pairs
  - For each pair  $v, w$ , compare:
    - length of the  $k$ -path from  $v$  to  $w$
    - length of the  $k$ -path from  $v$  to  $k$  appended to the  $k$ -path from  $k$  to  $w$
  - Set the  $k + 1$  path from  $v$  to  $w$  to be the minimum of the two paths above

# 17-77: Floyd's Algorithm

---

- Let  $D_k[v, w]$  be the length of the shortest  $k$ -path from  $v$  to  $w$ .
- $D_0[v, w]$  = cost of arc from  $v$  to  $w$  ( $\infty$  if no direct link)
- $D_k[v, w] = \text{MIN}(D_{k-1}[v, w], D_{k-1}[v, k] + D_{k-1}[k, w])$
- Create  $D_{-1}$ , use  $D_{-1}$  to create  $D_0$ , use  $D_0$  to create  $D_1$ , and so on – until we have  $D_{n-1}$

# 17-78: Floyd's Algorithm

---

- Use a doubly-nested loop to create  $D_k$  from  $D_{k-1}$ 
  - Use the same array to store  $D_{k-1}$  and  $D_k$  – just overwrite with the new values
- Embed this loop in a loop from 1..k



# 17-79: Floyd's Algorithm

---

```
Floyd(Edge G[], int D[][]) {
    int i,j,k

    Initialize D, D[i][j] = cost from i to j

    for (k=0; k<G.length; k++;
        for(i=0; i<G.length; i++)
            for(j=0; j<G.length; j++)
                if ((D[i][k] != Integer.MAX_VALUE) &&
                    (D[k][j] != Integer.MAX_VALUE) &&
                    (D[i][j] > (D[i,k] + D[k,j])))
                    D[i][j] = D[i][k] + D[k][j]
}
```

# 17-80: Floyd's Algorithm

---

- We've only calculated the *distance* of the shortest path, not the path itself
- We can use a similar strategy to the PATH field for Dijkstra to store the path
  - We will need a 2-D array to store the paths:  
 $P[i][j]$  = last vertex on shortest path from  $i$  to  $j$