Single-Source Shortest Paths

CSE 2320 – Algorithms and Data Structures

Vassilis Athitsos
University of Texas at Arlington

Terminology

- A <u>**network**</u> is a <u>**directed graph**</u>. We will use both terms interchangeably.

- The <u>**weight of a path**</u> is the sum of weights of the edges that make up the path.

- The <u>**shortest path**</u> between two vertices ***s*** and ***t*** in a directed graph is a directed path from ***s*** to ***t*** with the property that no other such path has a lower weight.

Shortest Paths

- We will consider two problems:
  - **Single-source**: find the shortest path from the source vertex s to all other vertices in the graph.
    - These shortest paths will form a tree, with s as the root.
  - **All-pairs**: find the shortest paths for all pairs of vertices in the graph.

- Assumptions:
  - Directed graphs
  - Edges cannot have non-negative weights.

Assumptions

- For directed graphs.
  - In all our shortest path algorithms, we will allow graphs to be directed.
  - Obviously, any algorithm that works on directed graphs will also work on undirected graphs. Why?

- Negative edge weights are not allowed. Why?

Assumptions

- Obviously, any algorithm that works on directed graphs will also work on undirected graphs. Why?
    - Undirected graphs are a special case of directed graphs.

- Negative edge weights are not allowed. Why?
  - With negative weights, "shortest paths" may not be defined.
  - If a cycle has negative weight, then repeating that cycle infinitely on a path make it "shorter" and "shorter"
    - Note that the weight of the whole path needs to be negative (not just an edge) for this to happen.
  - If all weights are nonnegative, a shortest path never needs to include a cycle.

Shortest-Paths Spanning Tree

- Given a network G and a designated vertex s, a **shortest-paths spanning tree** (SPST) for s is a tree that contains s and all vertices reachable from s, such that:
  - Vertex s is the root of this tree.
  - Each tree path is a shortest path in G.

Dijkstra's Algorithm

```
Dijkstra(G,w,s) // N = |V|
int d[N], p[N]
For v =0 -> N-1
    d[v]=inf    //total weight from s to v
    p[v]=-1     //predecessor of v on path from s to v
d[s]=0
Q = PriorityQueue(d)
While notEmpty(Q)
    u = removeMin(Q,w)
    for each v adjacent to u
        if v in Q and (d[u]+w(u,v))<d[v]
            p[v]=u
            d[v] = d[u]+w(u,v);  //total weight of path from s to v through u
            decreasedKeyFix(Q,v,d)  //v is neither index nor key
```

Add to the MST the vertex, u, with the shortest distance.

For each vertex, *v*, record the shortest distance from *s* to it and the edge that connects it (like Prim).

Dijkstra's Algorithm: Runtime

Time complexity: O(ElgV)
O(V + VlgV + E lgV) = O(ElgV)
Assuming V=O(E)

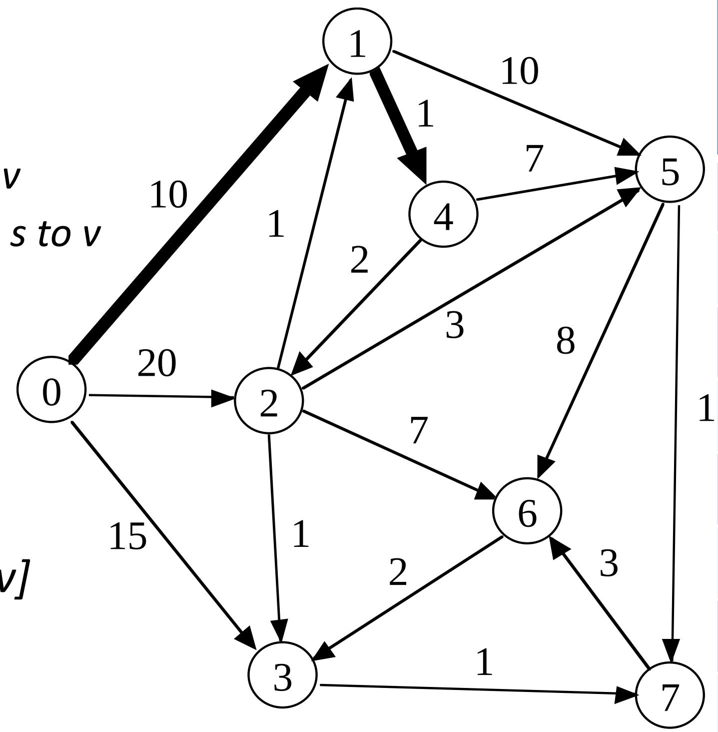
```
Dijkstra(G,w,s) // N = |V|
1   int d[N], p[N]
2   For v =0 -> N-1      ---------------------> Θ(V)
3       d[v]=inf    //total weight from s to v
4       p[v]=-1     //predecessor of v on path from s to v
5   d[s]=0
6   Q = PriorityQueue(d) ---------------------> Θ(V)
7   While notEmpty(Q) ---------------------> O(V)
8       u = removeMin(Q,w) ---------------------> O(lgV)  --> O(VlgV) (lines 7 & 8)
9       for each v adjacent to u -----------------> O(E) (from lines 7 & 9)
10          if v in Q and (d[u]+w(u,v))<d[v]
11              p[v]=u
12              d[v] = d[u]+w(u,v);  //total weight of path from s to v through u
13              decreasedKeyFix(Q,v,d)  //v is neither index nor key ---> O(lgV) --> O(ElgV)
                                  (aggregate from
                                  both for-loop and
                                  while-loop
                                  Lines: 7,9,13)
```

Dijkstra's Algorithm

- Computes an SPST for a graph G and a source s.
- Very similar to Prim's algorithm, but:
  - First vertex to add is the source, s.
  - Works with directed graphs as well, whereas Prim's only works with undirected graphs.
  - Requires edge weights to be non-negative.
  - **It looks at the total path weight, not just the weight of the current edge.**
- Time complexity: O(E lg V) using a heap for the priority-queue and adjacency list for edges.

Dijkstra’s Algorithm: SPST(G,0)

```
Dijkstra(G,w,s) // N = |V|
int d[N], p[N]
For v =0 -> N-1
    d[v]=inf    //total weight from s to v
    p[v]=-1   //v's predecessor on path s to v
d[s]=0
Q = PriorityQueue(d)
While notEmpty(Q)
    u = removeMin(Q,w)
    for each v adjacent to u
        if v in Q and (d[u]+w(u,v))<d[v]
            p[v]=u
            d[v] = d[u]+w(u,v);
            decreasedKeyFix(Q,v,d)
```



Added Vertex , v	Edge	Dis- tance from s to v

Vertex										
Work										

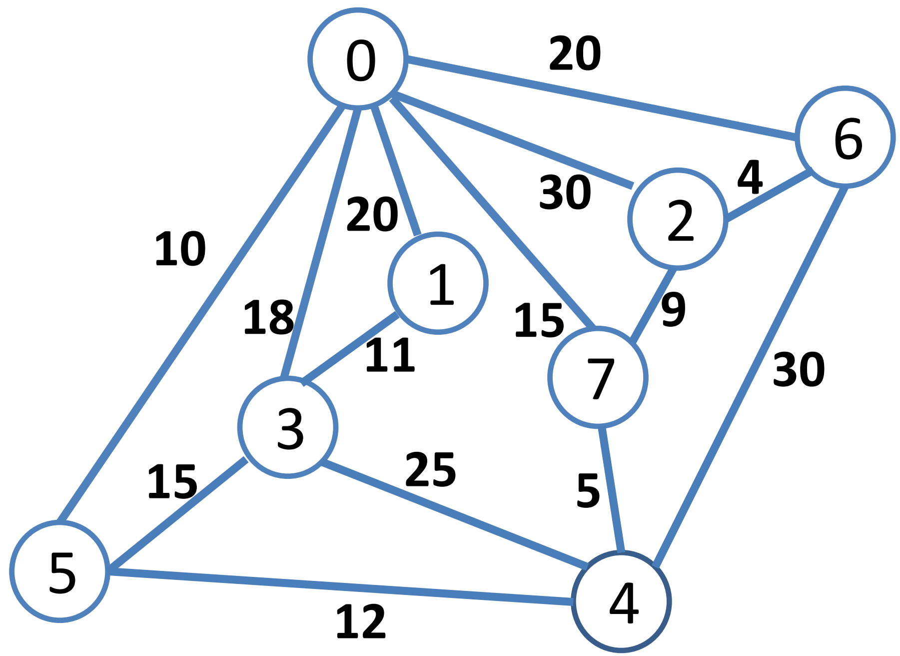
Dist (parent)

Dijkstra's Algorithm

```
Dijkstra(G,w,s) // N = |V|
    int d[N], p[N]
    For v =0 -> N-1
        d[v]=inf    //total weight from s to v
        p[v]=-1   //v's predecessor on path s to v
    d[s]=0
    Q = PriorityQueue(d)
    While notEmpty(Q)
        u = removeMin(Q,w)
        for each v adjacent to u
            if v in Q and (d[u]+w(u,v))<d[v]
                p[v]=u
                d[v] = d[u]+w(u,v);
                decreasedKeyFix(Q,v,d)
```

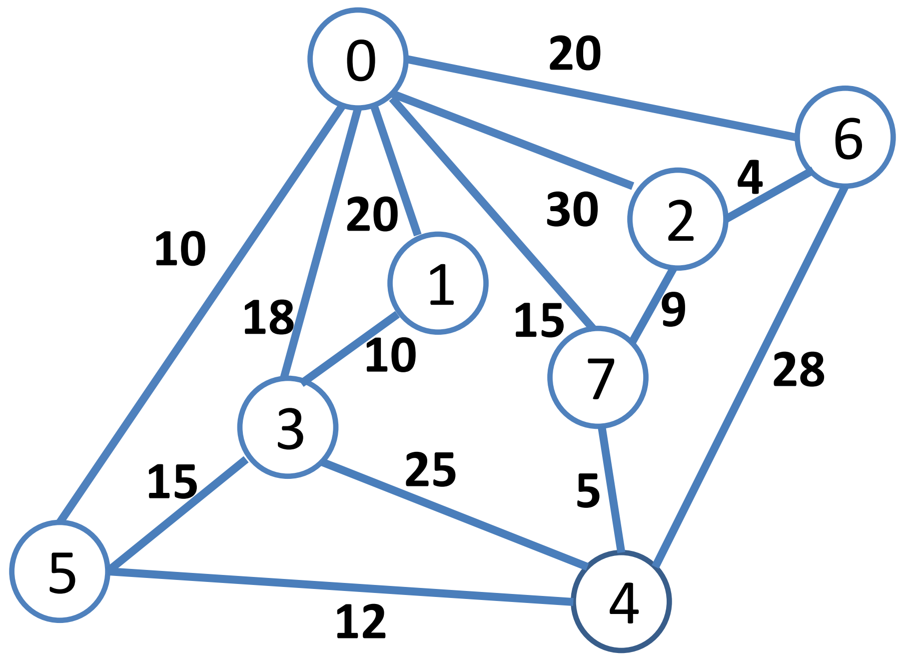
- Find the SPST(G,5).

Show the distance and the parent for each vertex.



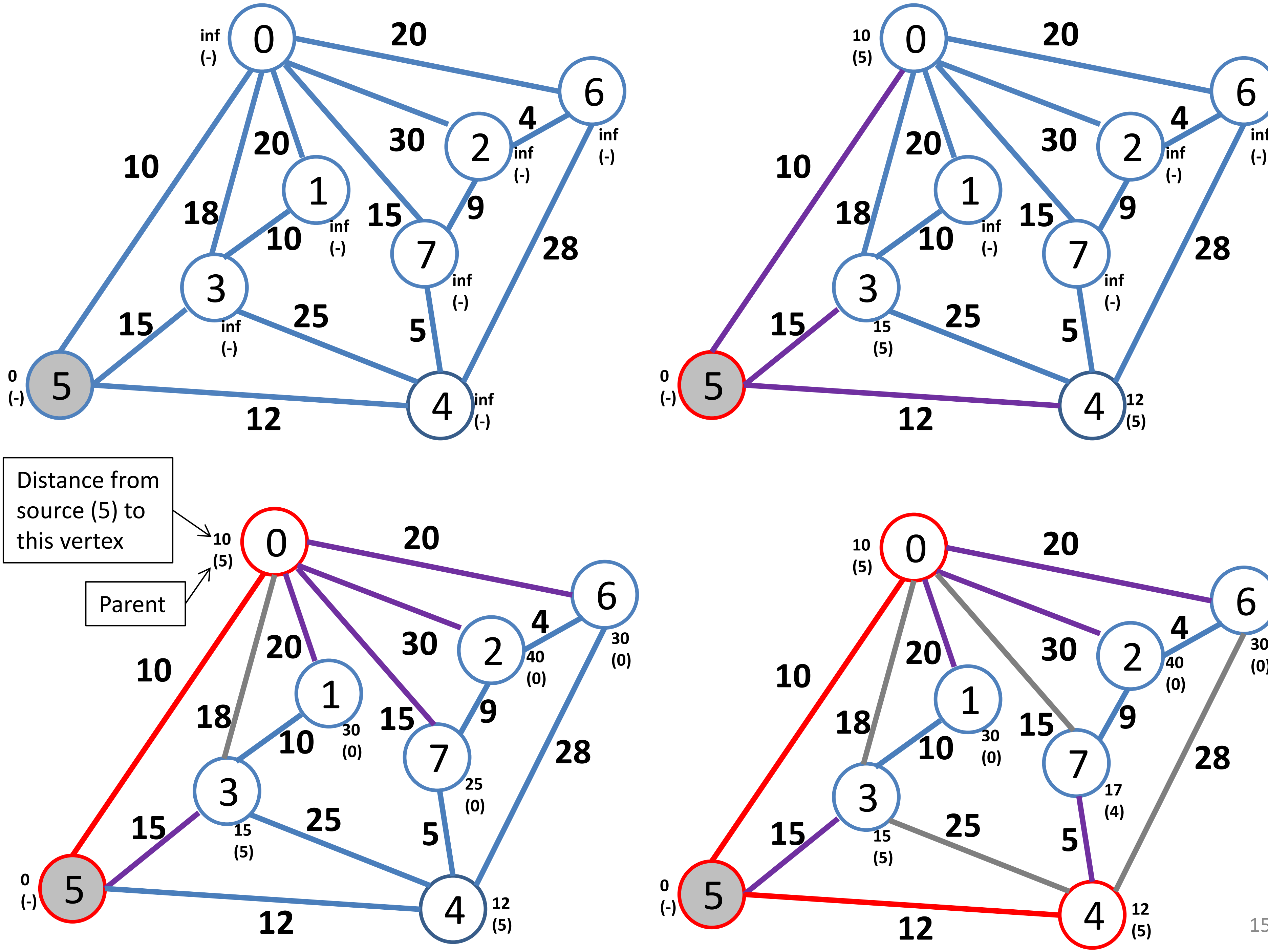
Dijkstra's Algorithm

```
Dijkstra(G,w,s) // N = |V|
    int d[N], p[N]
    For v =0 -> N-1
        d[v]=inf    //total weight from s to v
        p[v]=-1   //v's predecessor on path s to v
    d[s]=0
    Q = PriorityQueue(d)
    While notEmpty(Q)
        u = removeMin(Q,w)
        for each v adjacent to u
            if v in Q and (d[u]+w(u,v))<d[v]
                p[v]=u
                d[v] = d[u]+w(u,v);
                decreasedKeyFix(Q,v,d)
```

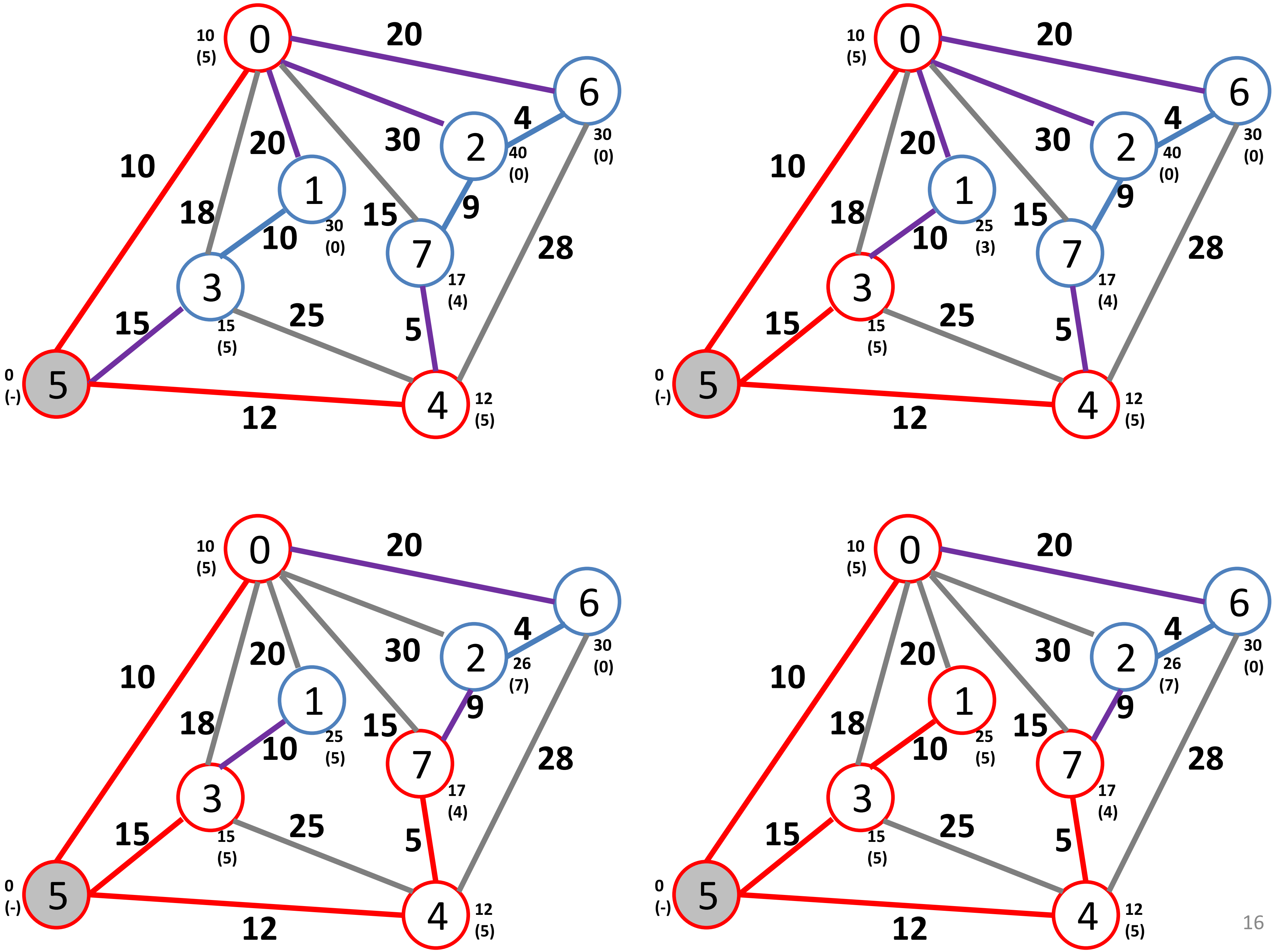


Worked-out Dijkstra example

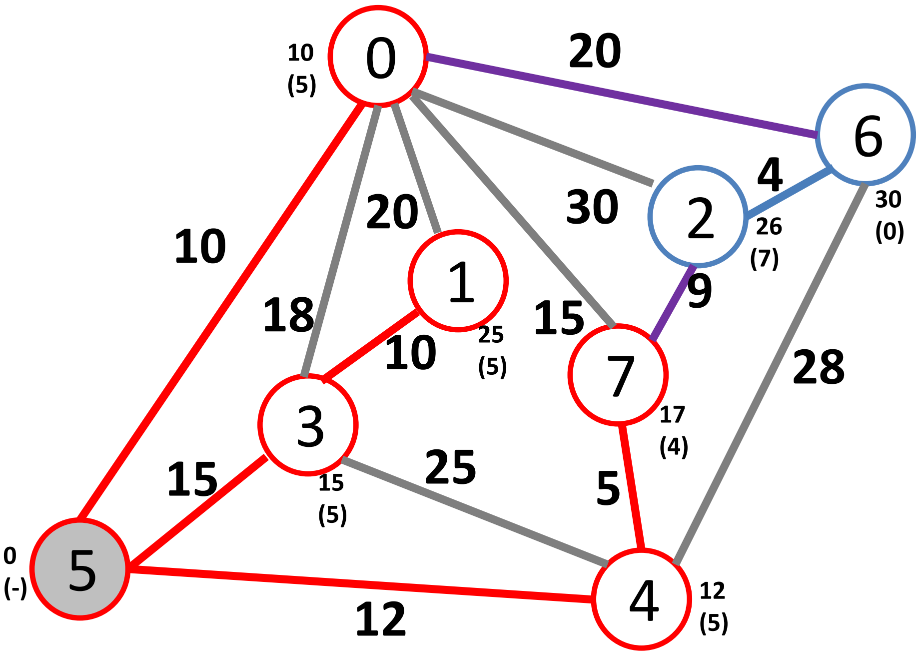
- Note that this example is for an undirected graph. The same algorithm will be applied to a directed graph (going in the direction of the arrows).
- When moving to a new page, the last state of the graph (bottom right) is copied first (top left).
- Purple edges – edges that change the distance from source to vertex (best edge to take to get to that vertex).
- Gray edges – edges discovered that do not provide a shorter path to the vertex (discovered, but not used).
- Red edges and vertices – shortest path spanning tree (SPST) built with Dijkstra.

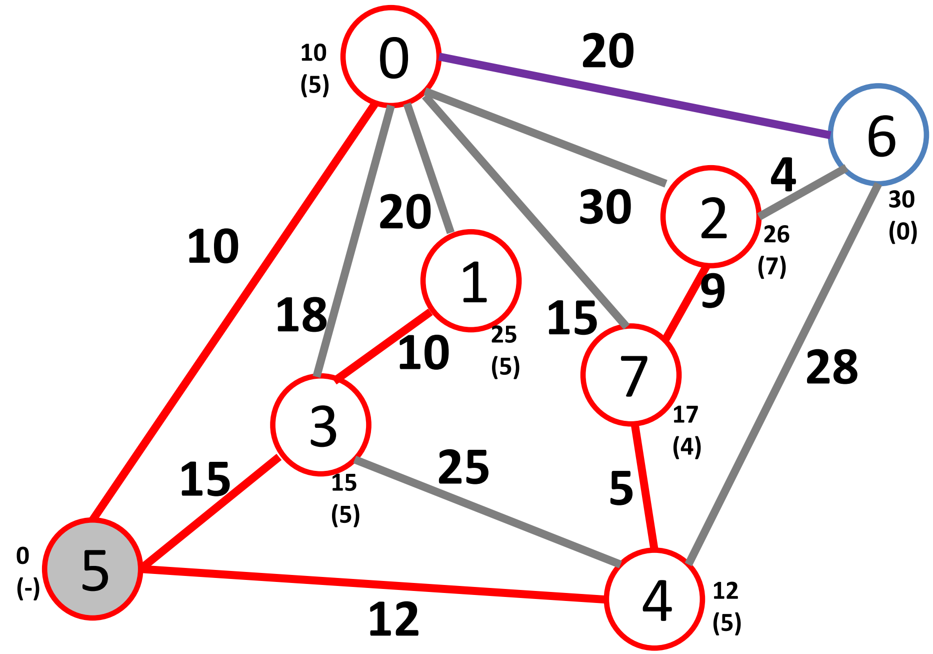


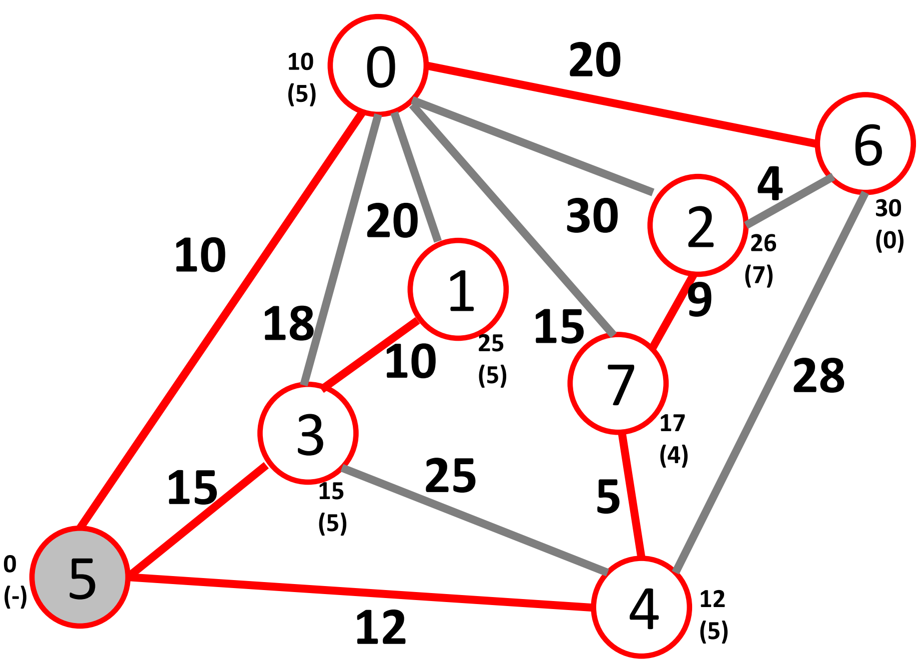
Distance from source (5) to this vertex
Parent



0
10
(5)
20
6
30
(0)
4
2
40
(0)
30
20
10
1
30
(0)
18
10
15
9
7
17
(4)
28
3
15
(5)
15
25
5
5
0
(-)
4
12
(5)
12
0
10
(5)
20
6
30
(0)
4
2
40
(0)
30
20
10
1
25
(3)
18
10
15
9
7
17
(4)
28
3
15
(5)
15
25
5
5
0
(-)
4
12
(5)
12
0
10
(5)
20
6
30
(0)
4
2
26
(7)
30
20
10
1
25
(5)
18
10
15
9
7
17
(4)
28
3
15
(5)
15
25
5
5
0
(-)
4
12
(5)
12
0
10
(5)
20
6
30
(0)
4
2
26
(7)
30
20
10
1
25
(5)
18
10
15
9
7
17
(4)
28
3
15
(5)
15
25
5
5
0
(-)
4
12
(5)
12







Vertex, as added	Edge (parent,vertex)	Distance from source to vertex
5	-	0
0	(5,0)	10
4	(5,4)	12
3	(5,3)	15
7	(4,7)	17
1	(3,1)	25
2	(7,2)	26
6	(0,6)	30

All-Pairs Shortest Paths

- Run Dijkstra to compute the Shortest Path Spanning Tree (SPST) for each vertex used as source.
  - Note that the array of predecessors completely specifies the SPST.

- Trick to get the successor (rather than the predecessor/parent) on a path: reverse direction of arrows, solve problem in reversed graph. That solution gives the predecessor in the reversed graph which correspond to successors in the original graph.
  - A 2D table will be needed to store the all pair shortest paths (one row for each SPTS).