Lecture # 17

Shortest Paths

Shortest Paths

- In the *shortest -paths problem* We are given a weighted, directed graph G=(V, E) The weight of path p = < v_0, v_1,..., v_k > is the sum of the constituent edges:

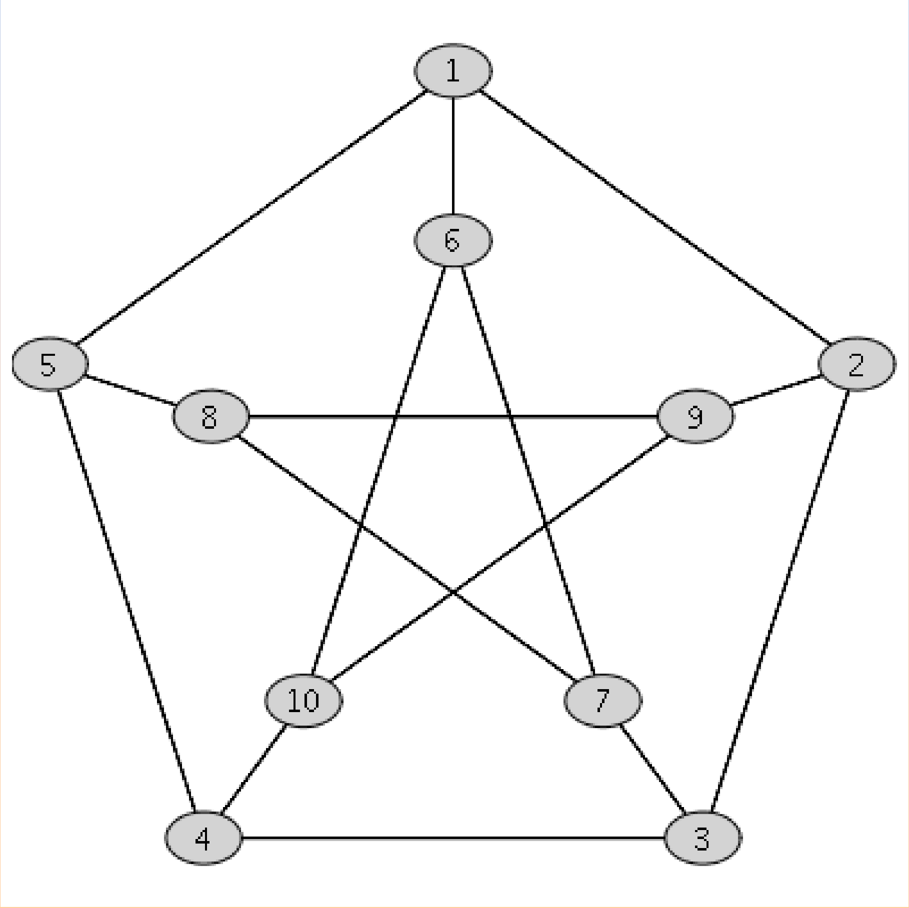
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- We define the *shortest-path weight* from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Shortest Paths

- The **breadth-first-search** algorithm we discussed earlier is a shortest-path algorithm that works on un-weighted graphs. An un-weighted graph can be considered as a graph in which every edge has weight one unit.



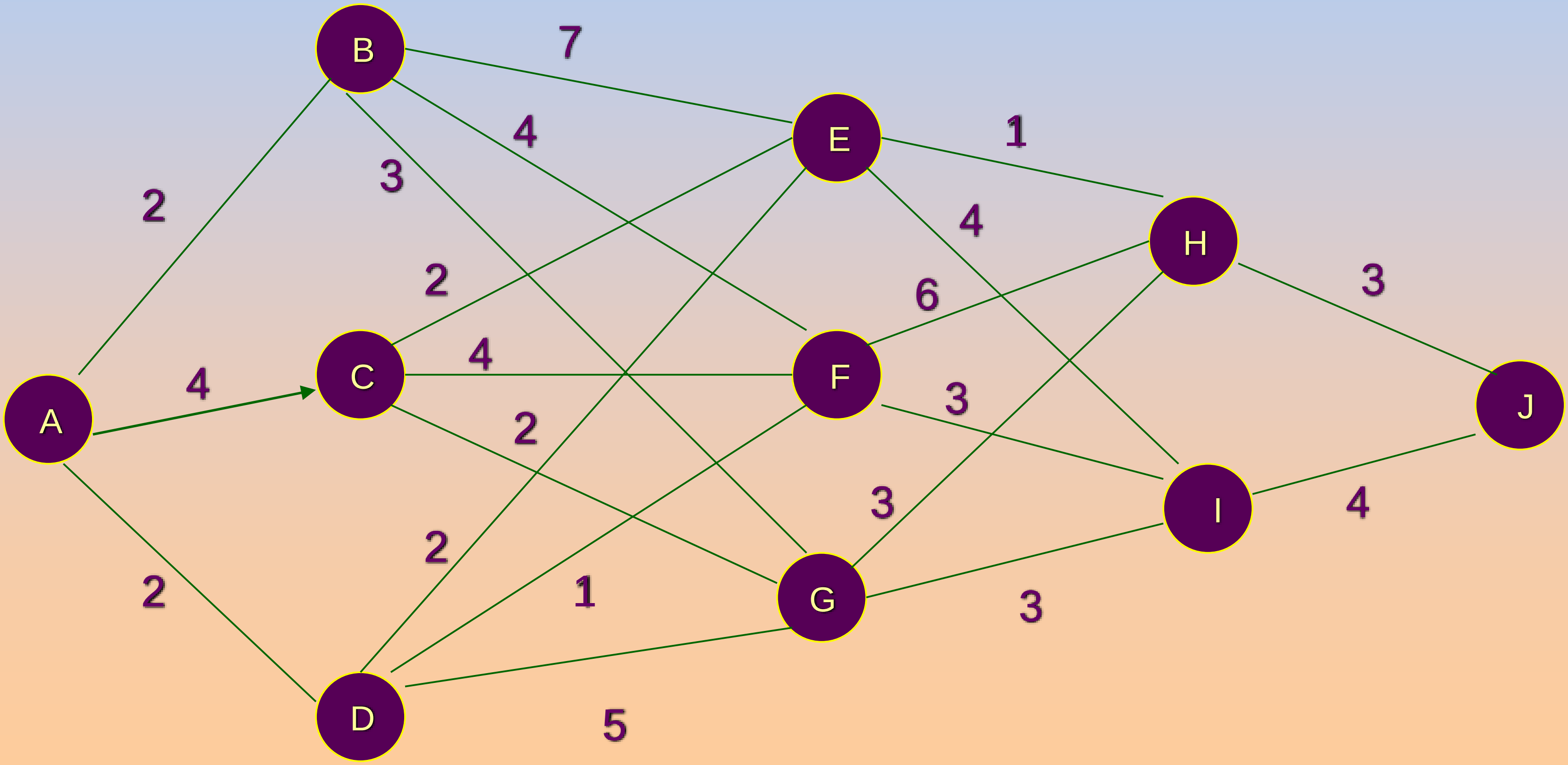
Shortest Paths

- There are a few variants of the shortest path problem. We will cover their definitions and then discuss algorithms for some.

Shortest Paths

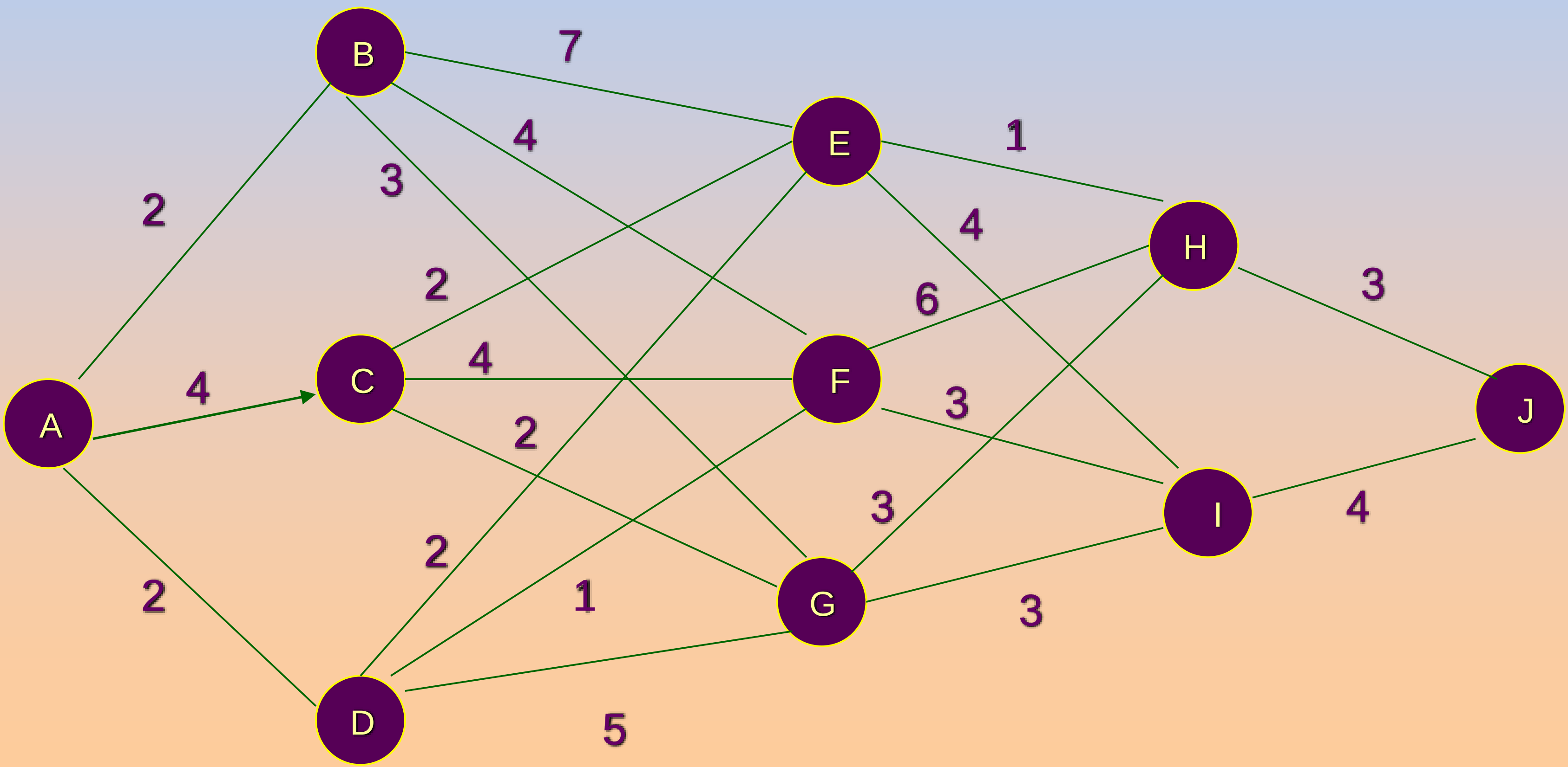
- **Single-source shortest-path problem:** Find shortest paths from a given (single) *source* vertex

s € V to every other vertex v € V in the graph G.



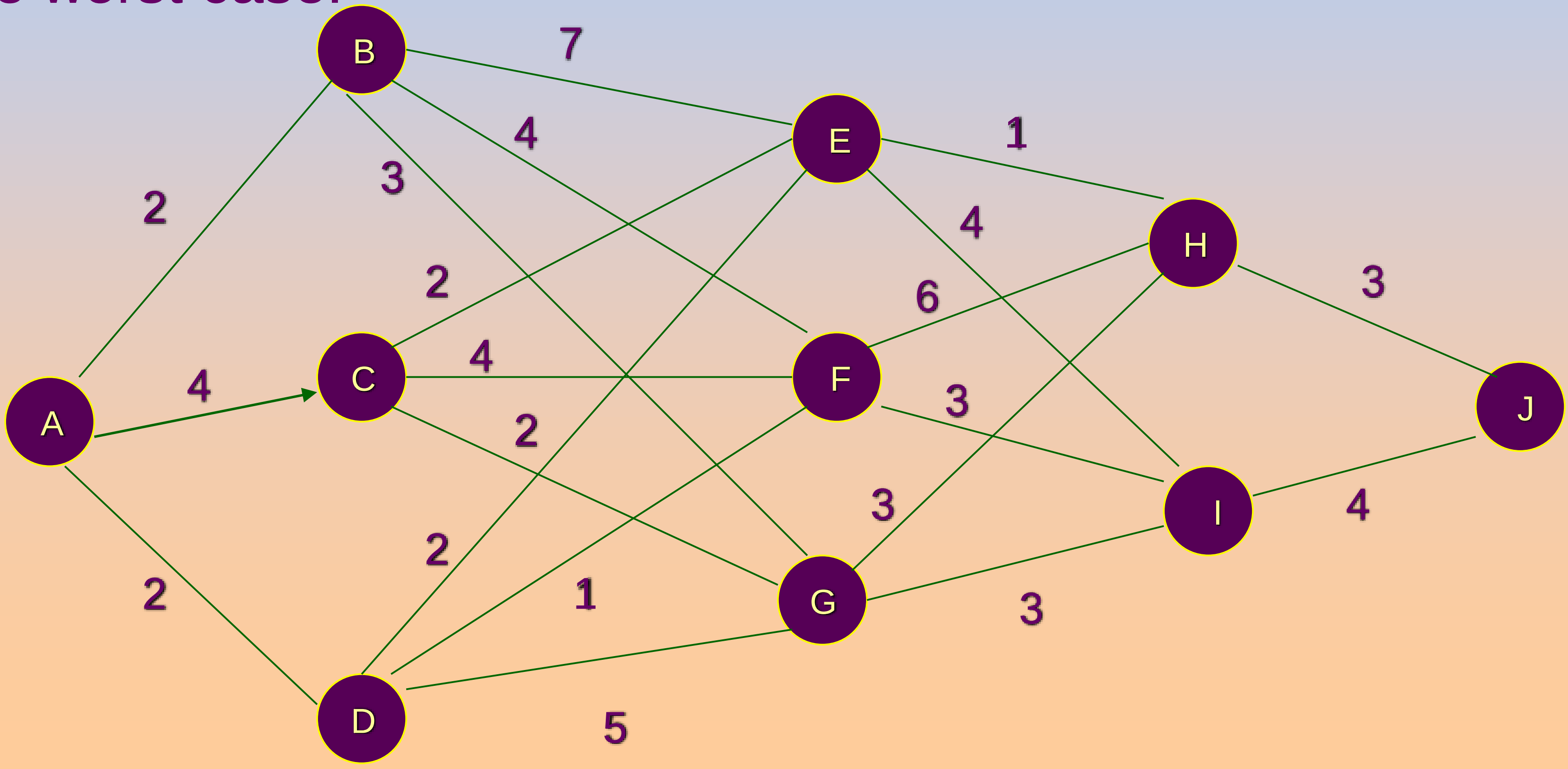
Shortest Paths

- **Single-destination shortest-paths problem:** Find a shortest path to a given destination vertex ***t*** from each vertex ***v***. We can reduce the problem to a single-source problem by reversing the direction of each edge in the graph.



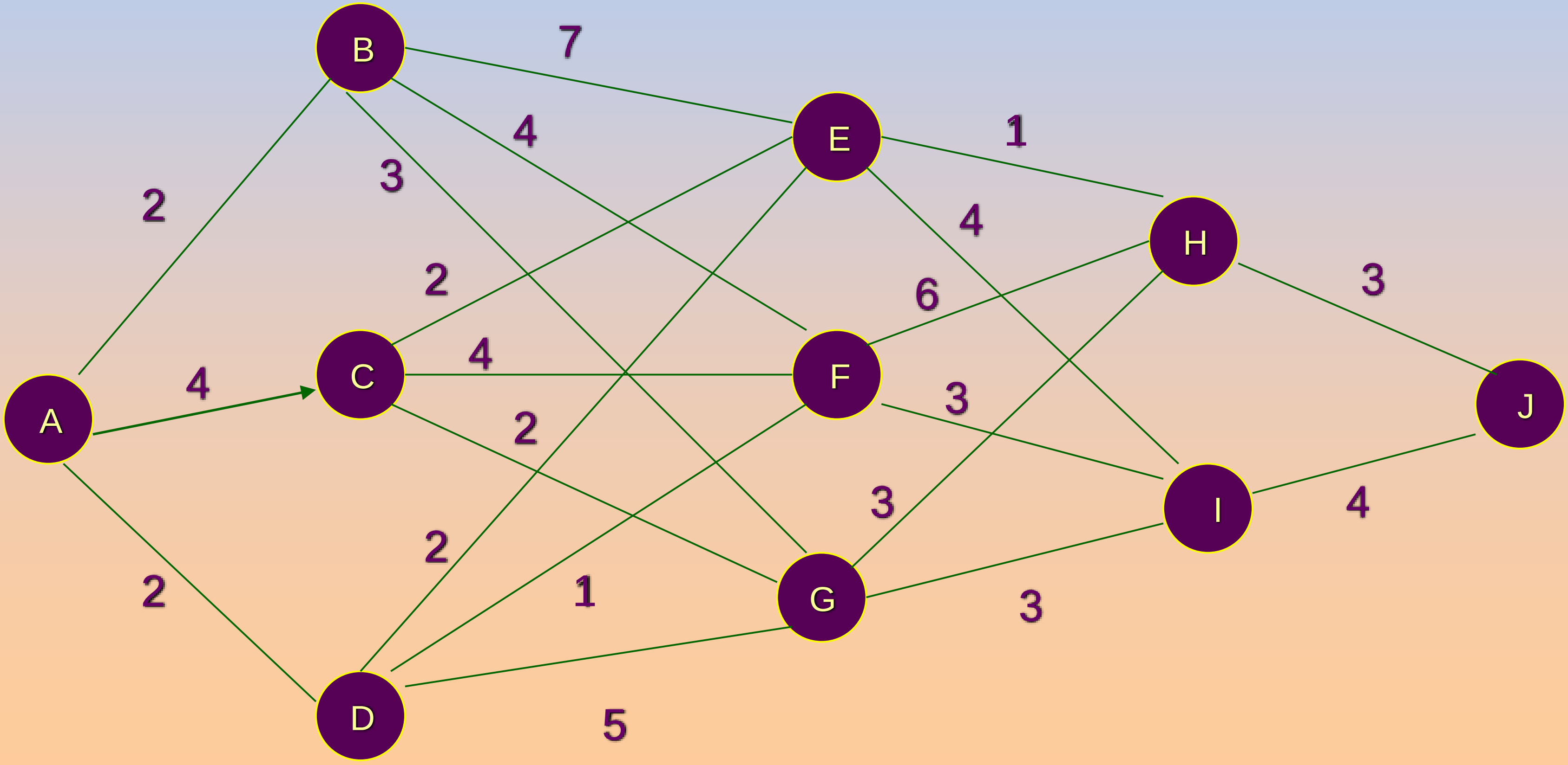
Shortest Paths

- **Single-pair shortest-path problem:** Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. No algorithms for this problem are known to run asymptotically faster than the best single-source algorithms in the worst case.



Shortest Paths

- **All-pairs shortest-paths problem:** Find a shortest path from u to v for *every pair* of vertices u and v. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster.



Dijkstra’s Algorithm

- Dijkstra’s algorithm is a simple *greedy* algorithm for computing the **single-source shortest-paths** to all other vertices.

- Dijkstra’s algorithm works on a **weighted directed** graph $G = (V, E)$ in which all edge **weights are non-negative**, i.e., $w(u, v) \geq 0$ for each edge (u, v) € E.

Dijkstra’s Algorithm

- Negative edges weights may be counter to intuition but this can occur in real life problems.
- However, we will *not allow negative cycles* because then there is no shortest path. If there is a negative cycle between, say, s and t, then we can always find a shorter path by going around the cycle one more time.

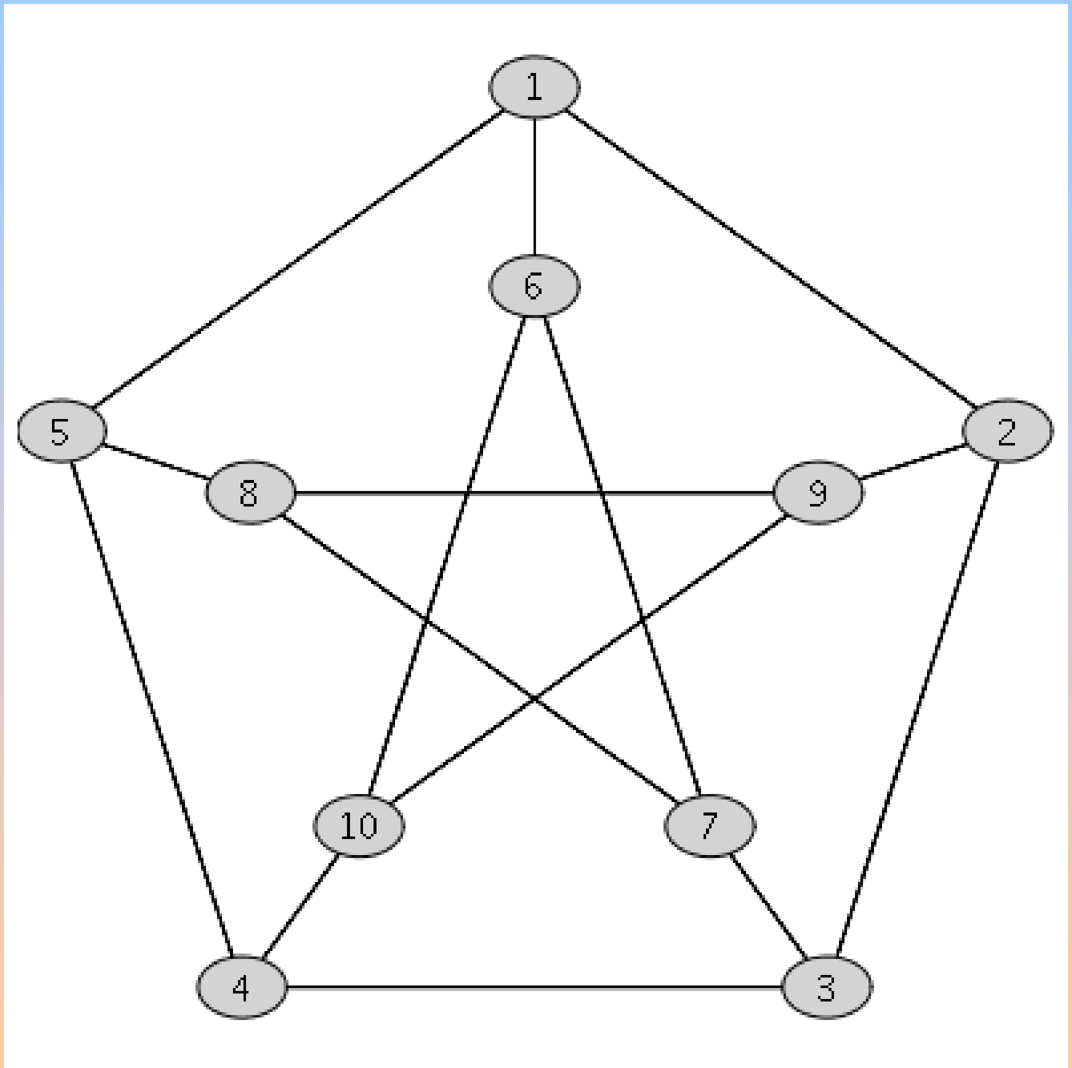
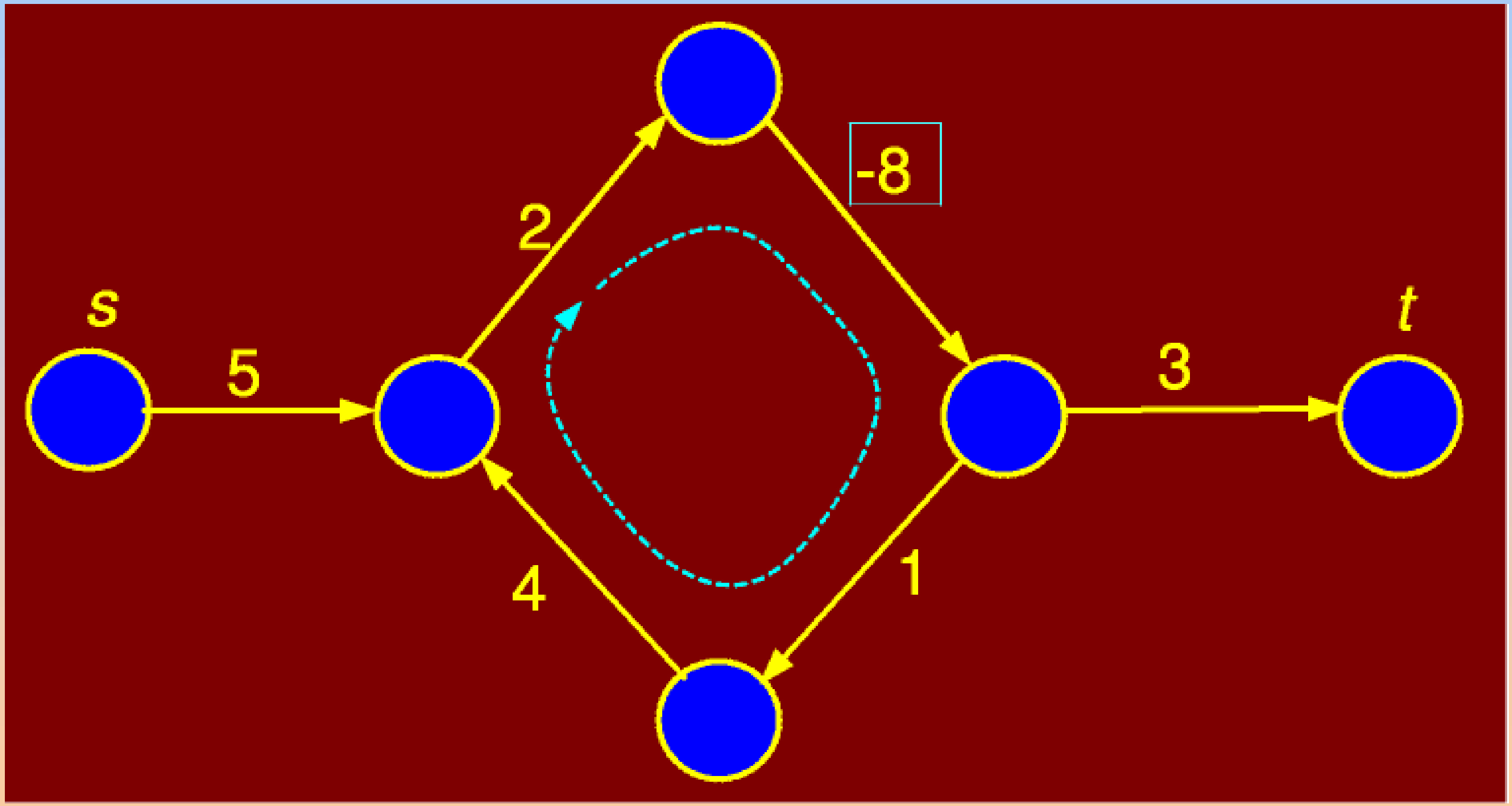

1
6
5
2
8
9
10
7
4
3

Figure : Negative weight cycle



Dijkstra's Algorithm

- The basic structure of Dijkstra's algorithm is to maintain an ***estimate* of the shortest path** from the source vertex to each vertex in the graph. Call this estimate **d[v].**
- Intuitively, d[v] will be the length of the shortest path *that the algorithm knows of* from **s** to **v**.
- This value will always be greater than or equal to the *true shortest path distance* from **s** to **v**. I.e., $d[v] \geq \delta(u, v)$.
- Initially, we do not know the paths, so $d[v] = \infty$ Moreover, $d[s] = 0$ for the source vertex.
- As the algorithm goes on and sees more and more vertices, it attempts to update d[v] for each vertex in the graph. The process of updating estimates is called ***relaxation***.

Dijkstra's Algorithm

- Consider an edge from a vertex u to v whose weight is w(u, v). Suppose that we have already computed current estimates on d[u] and d[v].
- We know that there is a path from s to u of weight d[u].
- By taking this path and following it with the edge (u, v) we get a path to v of length d[u] + w(u, v).
- If this path is better than the existing path of length d[v] to v, we should update the value of d[v].
- We should also remember that the shortest way back to the source is through u by updating the predecessor pointer.
- The relaxation process is illustrated in the following figure.

Dijkstra's Algorithm

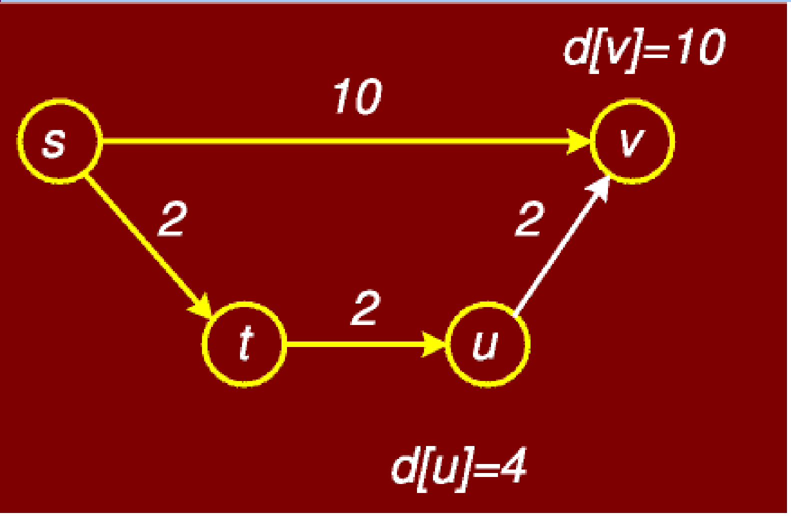


Figure 8.62: Vertex u relaxed

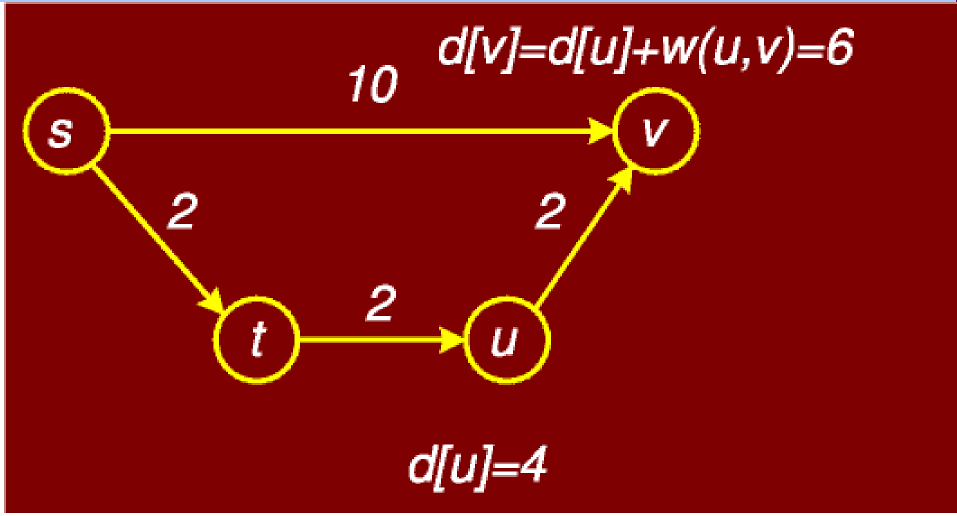


Figure 8.63: Vertex v relaxed

```
RELAX( (u, v) )
    if (d[u] +w(u,v) < d[v])
        then d[v] ← d[u] + w(u, v)
             pred[v] = u
```

Dijkstra's Algorithm

- Dijkstra's algorithm is based on the notion of performing repeated relaxations.

- The algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we *know* the true distance, d[v] = δ(s,v). Initially S = Ø, the empty set. We set d[u] = 0 and all others to ∞. One by one we select vertices from V-S to add to S.

Dijkstra's Algorithm

- How do we select which vertex among the vertices of V- S to add next to S? Here is ***greediness*** comes in.
- For each vertex u ϵ (V - S), we have computed a distance estimate d[u].
- The greedy thing to do is to take the vertex for which d[u] is minimum, i.e., take the unprocessed vertex that is closest by our estimate to s.
- Later, we justify why this is the proper choice. In order to perform this selection efficiently, we store the vertices of V-S in a *priority queue.*

Dijkstra's Algorithm

```
DIJKSTRA( (G, w, s) )
for ( each u ϵ V)
do d[u] ← ∞
      pq.insert (u, d[u] )
d[s] ← 0; pred [s] ← nil;  pq.decrease_key (s, d[s])
while ( pq.not_empty ( ) )
do  u ← pq.extract_min ( )
      for ( each v ϵ adj [u] )
      do if (d[u] + w(u,v) < d[v])
            then d[v] = d[u] + w(u,v)
                  pq.decrease_key (v, d[v] )
                  pred [v] = u
```

Analyzing Dijkstra's Algorithm

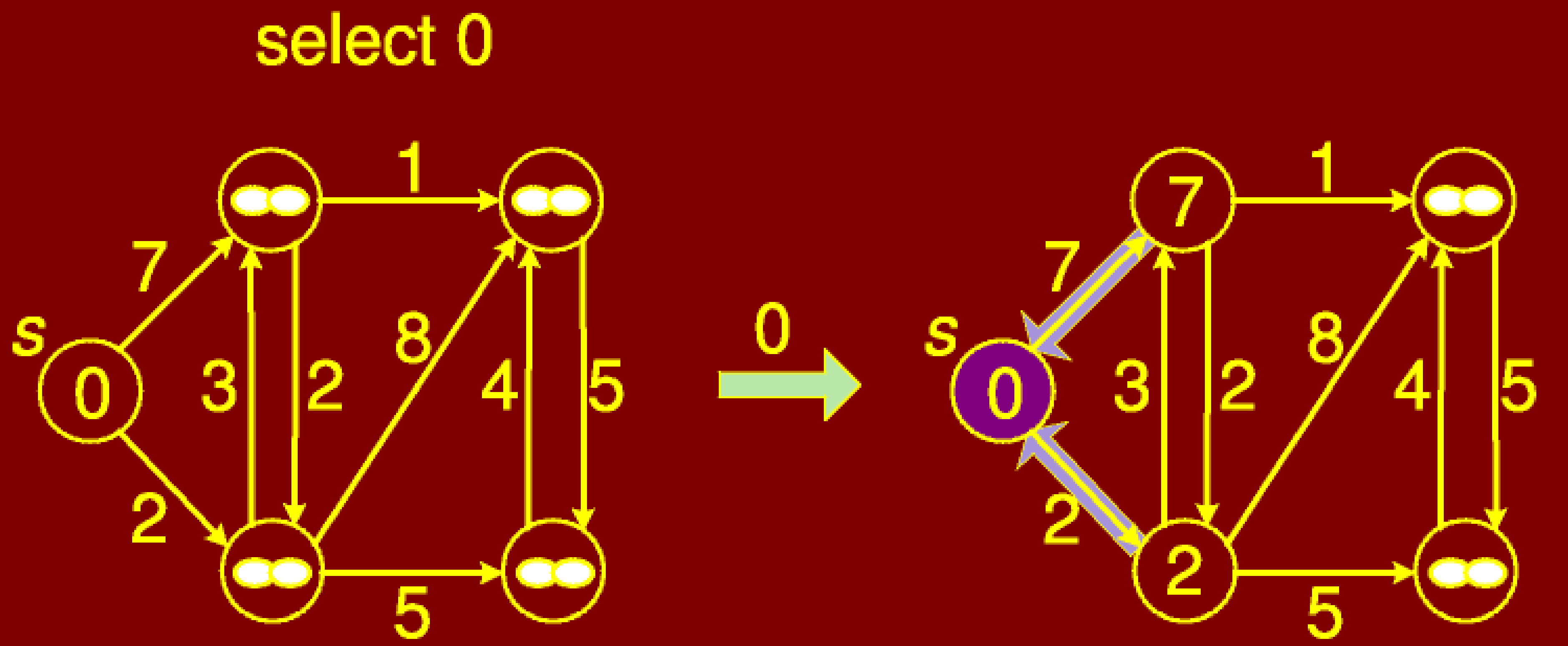
- The call to **`BuildHeap()`** takes O(n) time
- Each of the n - 1 calls to **`Heapify()`** takes O(lg n) time

- Thus the total time taken :
  = O(n) + (n - 1) O(lg n)
  = O(n) + O(n lg n)
  = **O(n lg n)**

Dijkstra’s Algorithm

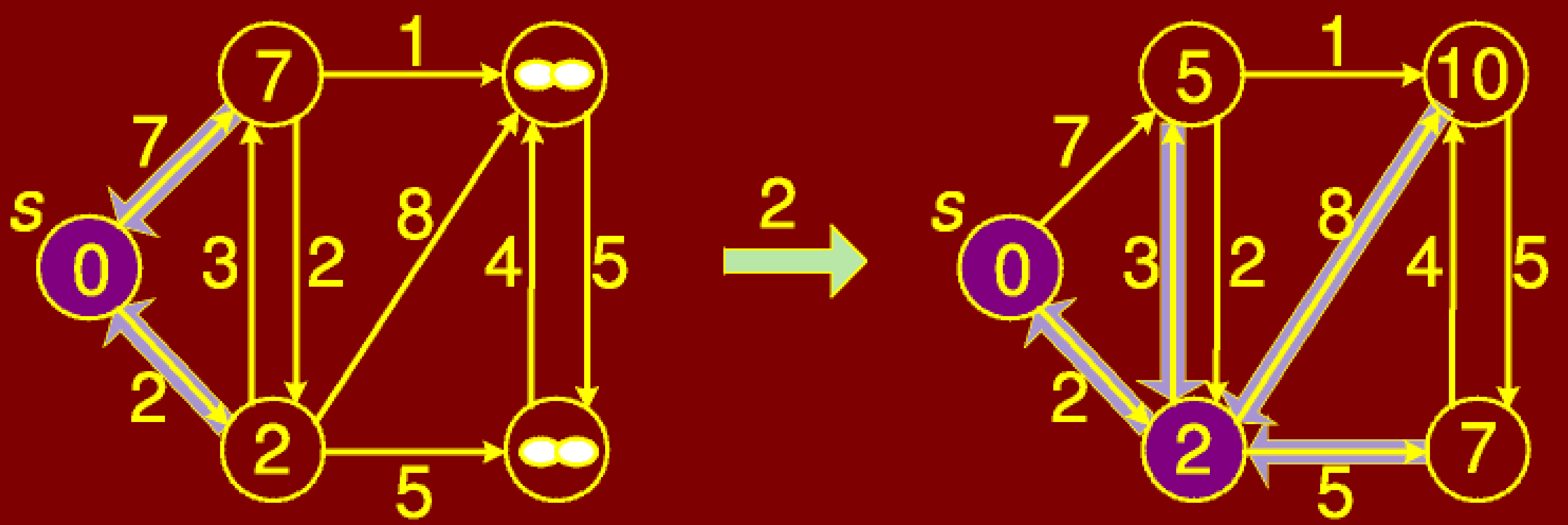
- Note the similarity with Prim’s algorithm, although a different key is used here. Therefore the running time is the same, i.e.. **Θ(E log V).**
- Next Figures demonstrate the algorithm applied to a directed graph with no negative weight edges.

Dijkstra's Algorithm



Dijkstra's Algorithm

select 2



Dijkstra’s Algorithm

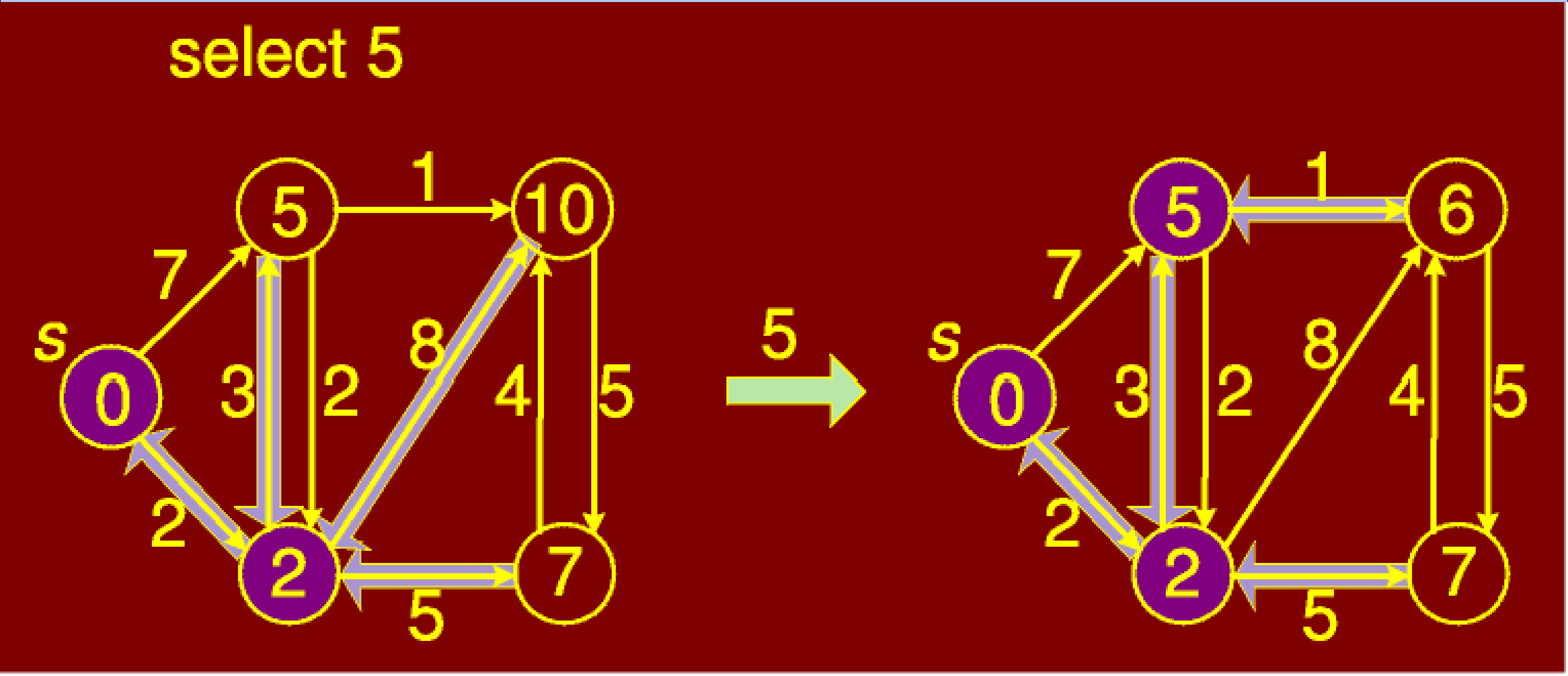
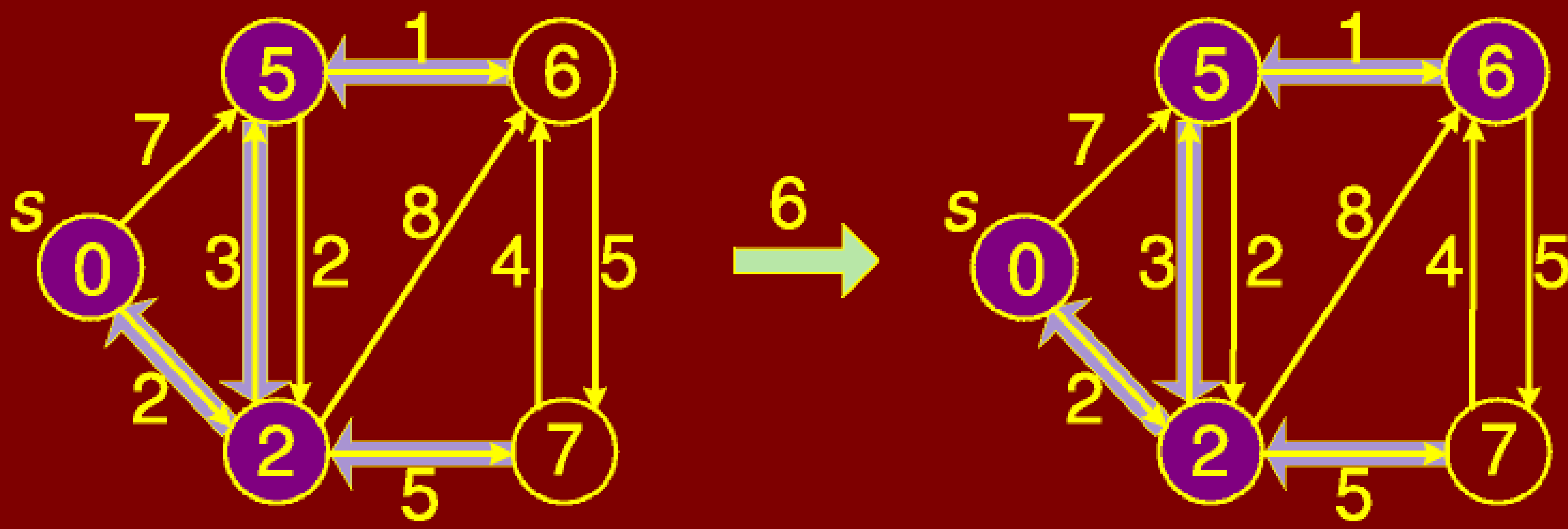


Figure 8.66: Dijkstra's algorithm: select 5

Dijkstra's Algorithm

select 6



Dijkstra's Algorithm