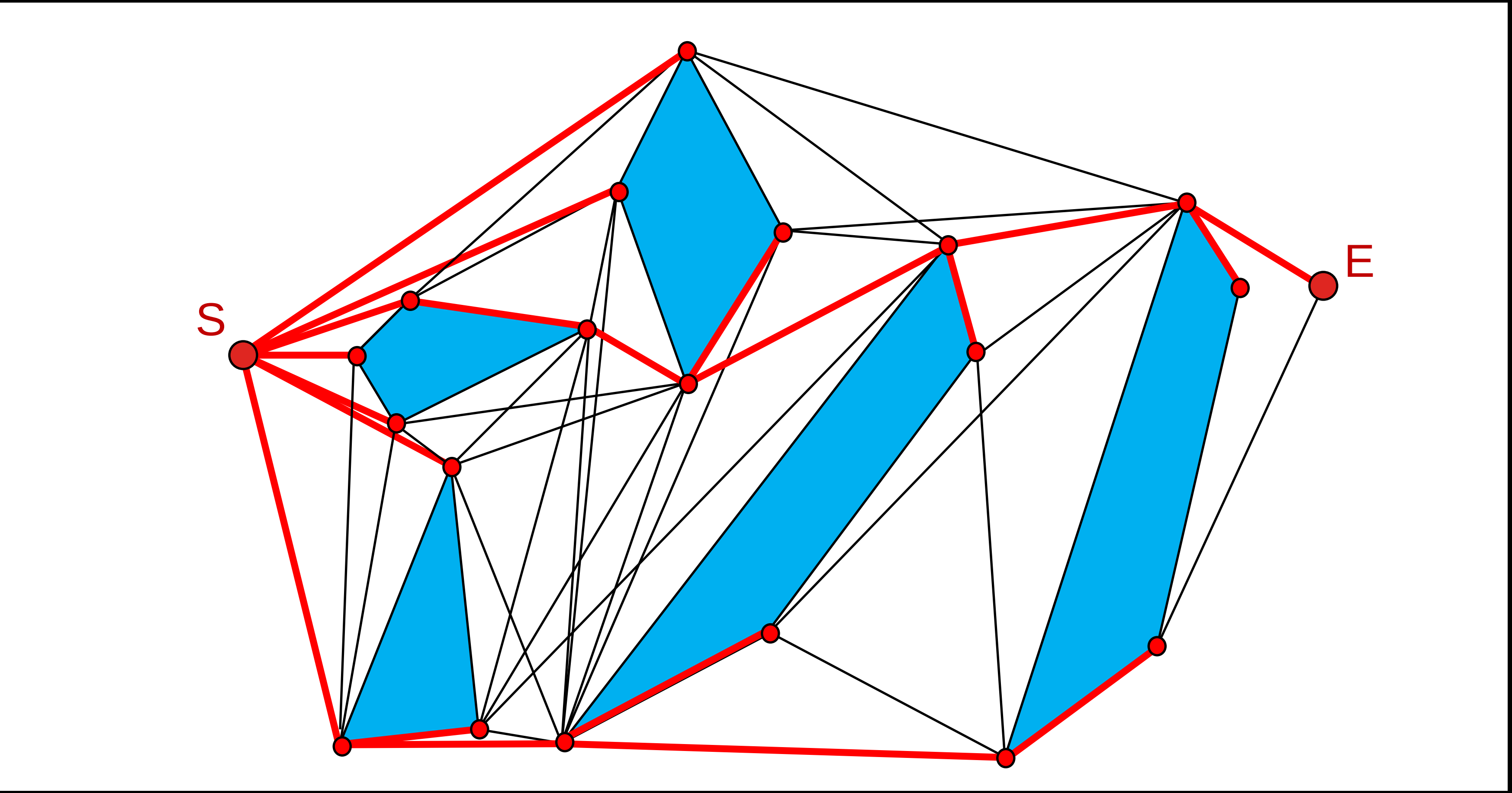
Computing Shortest Paths

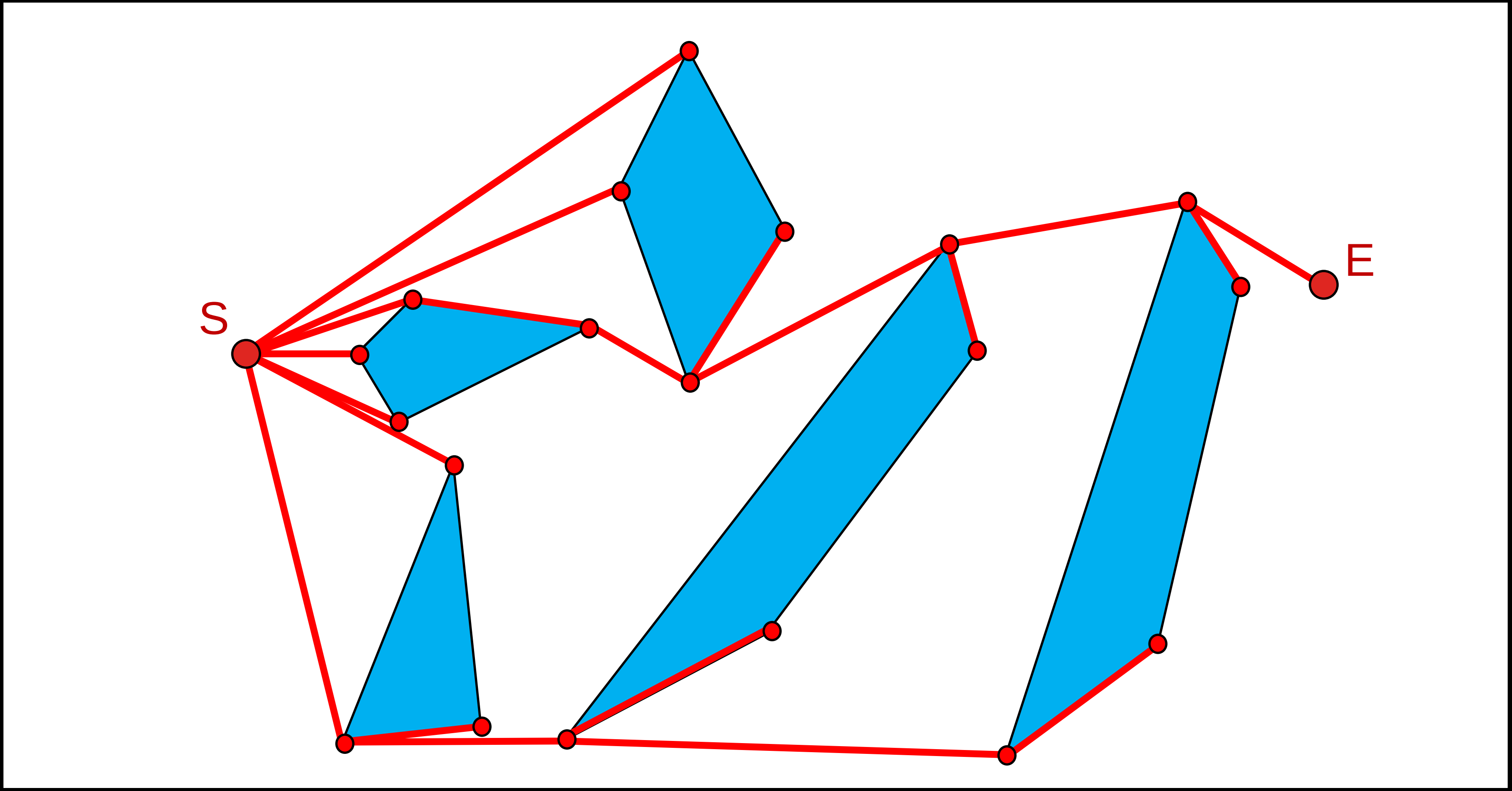
Shortest Paths

- The shortest path from the start to each vertex of the visibility graph will consist of edges of the graph:



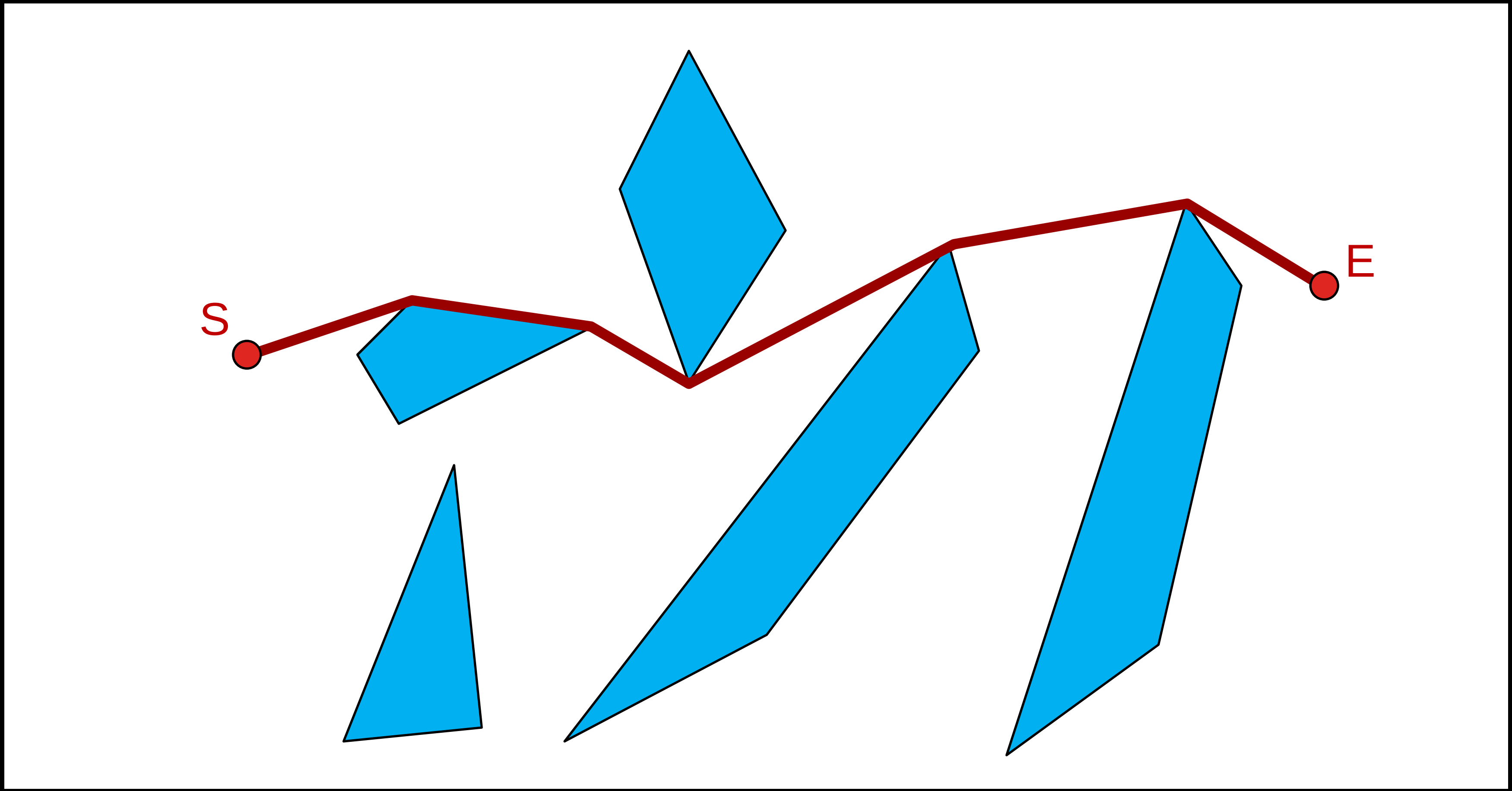
Shortest Path Algorithm

- Result is called the *Shortest Path Tree*:



Shortest Path Algorithm

- The shortest path to the goal is one of these paths:



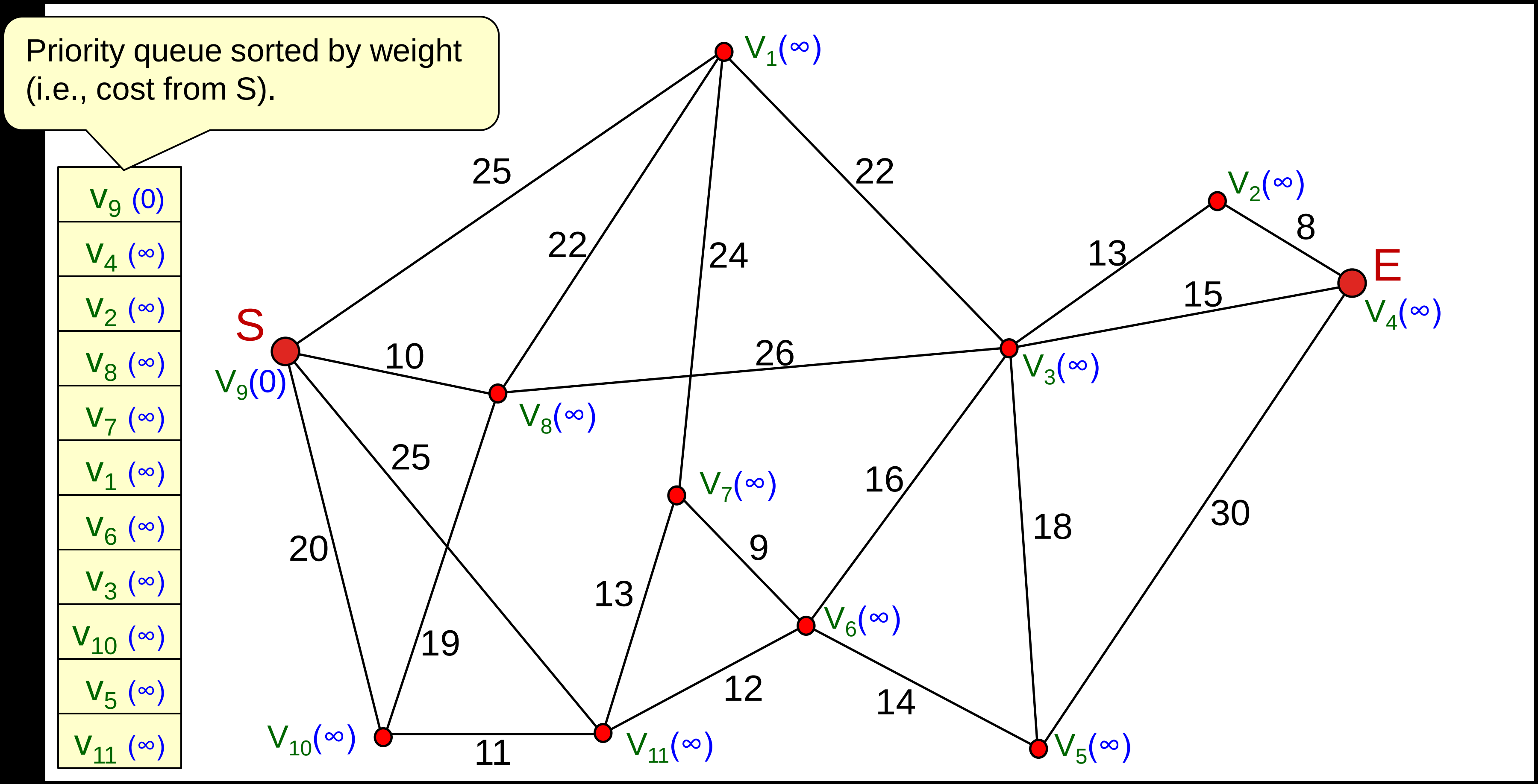
Dijkstra's Shortest Path Algorithm

- A popular algorithm for computing shortest paths in a graph is known as Dijkstra's Algorithm:
  - Starts with a *weight* of ZERO at the start node
  - Propagates outwards from the source (like a wavefront) to all graph edges.
    - Nodes "closer to" the source are visited before those further away.

  - Each time an edge is travelled along, the robot incurs a cost according to some metric (e.g., distance, time, battery usage, etc..) which is usually represented by a *weight* on the edge.
  - Once all nodes have been reached by the "wavefront", the algorithm is done, and each node will have a weight corresponding to the cost to get there from the source.

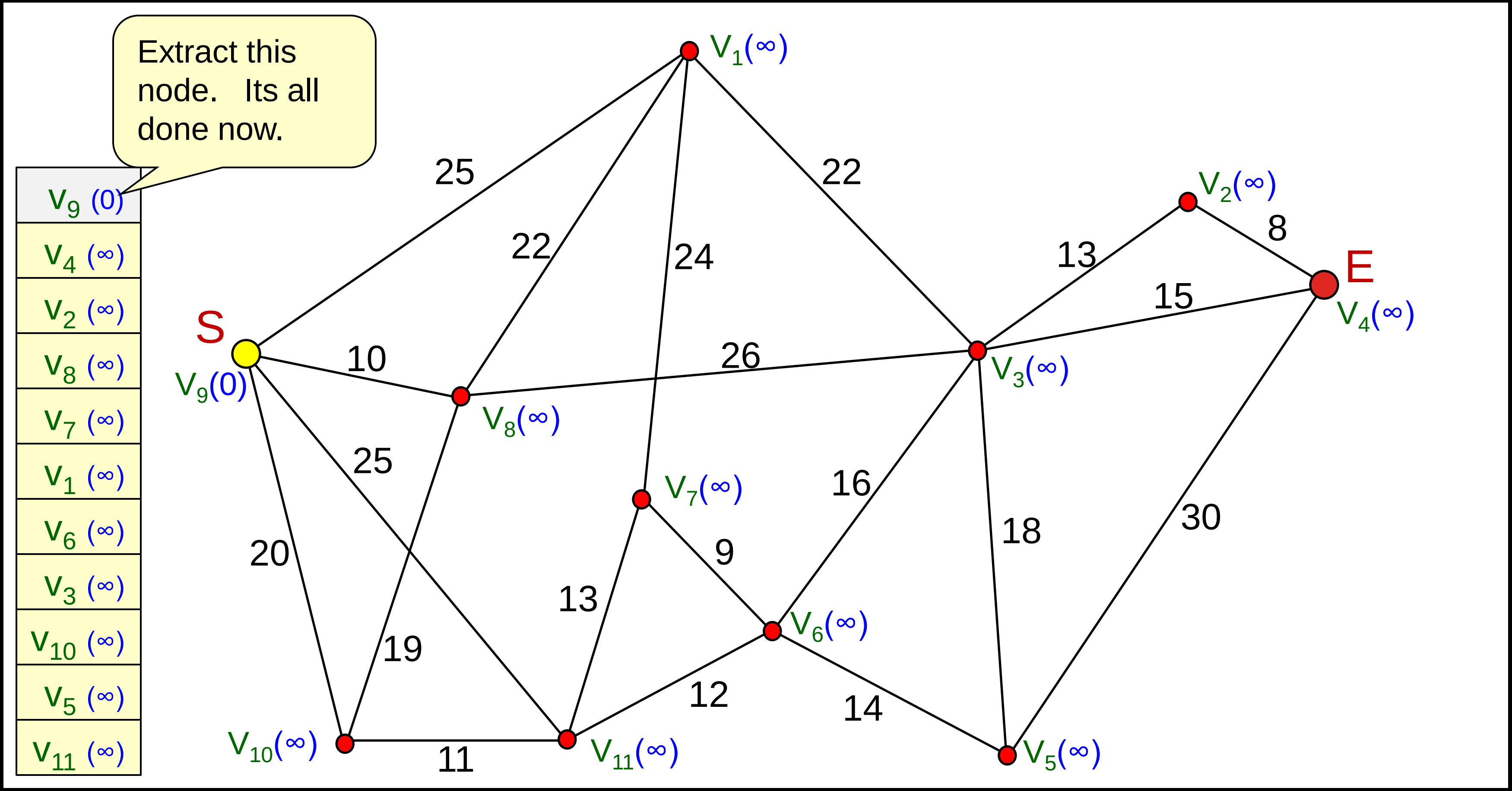
Dijkstra's Shortest Path Algorithm

- To start, source is given weight of 0, all other nodes a weight of ∞. Nodes are stored in priority queue.



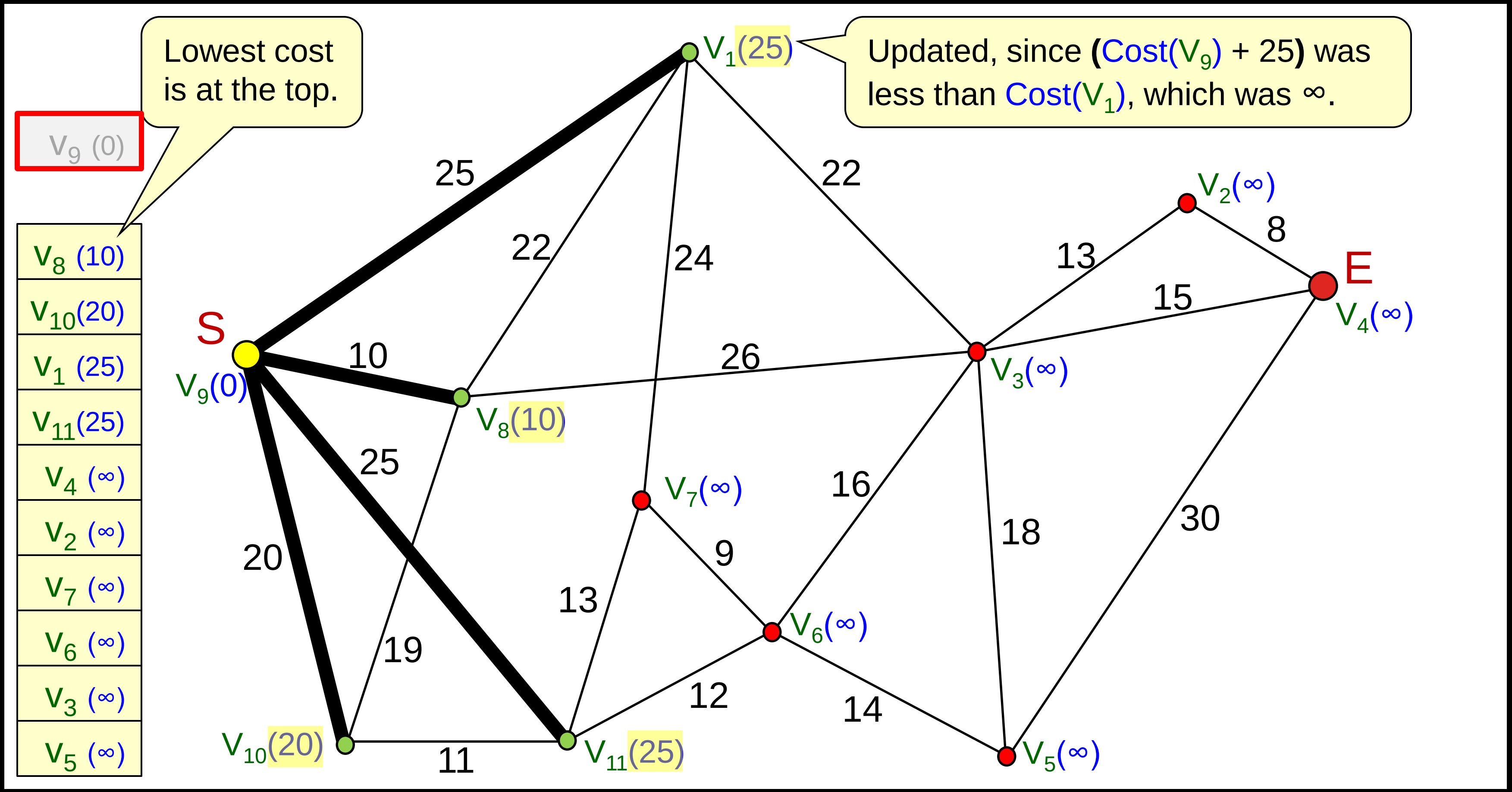
Dijkstra's Shortest Path Algorithm

- Algorithm repeatedly extracts top node from queue.
  - An extracted node is done being processed, has its final cost



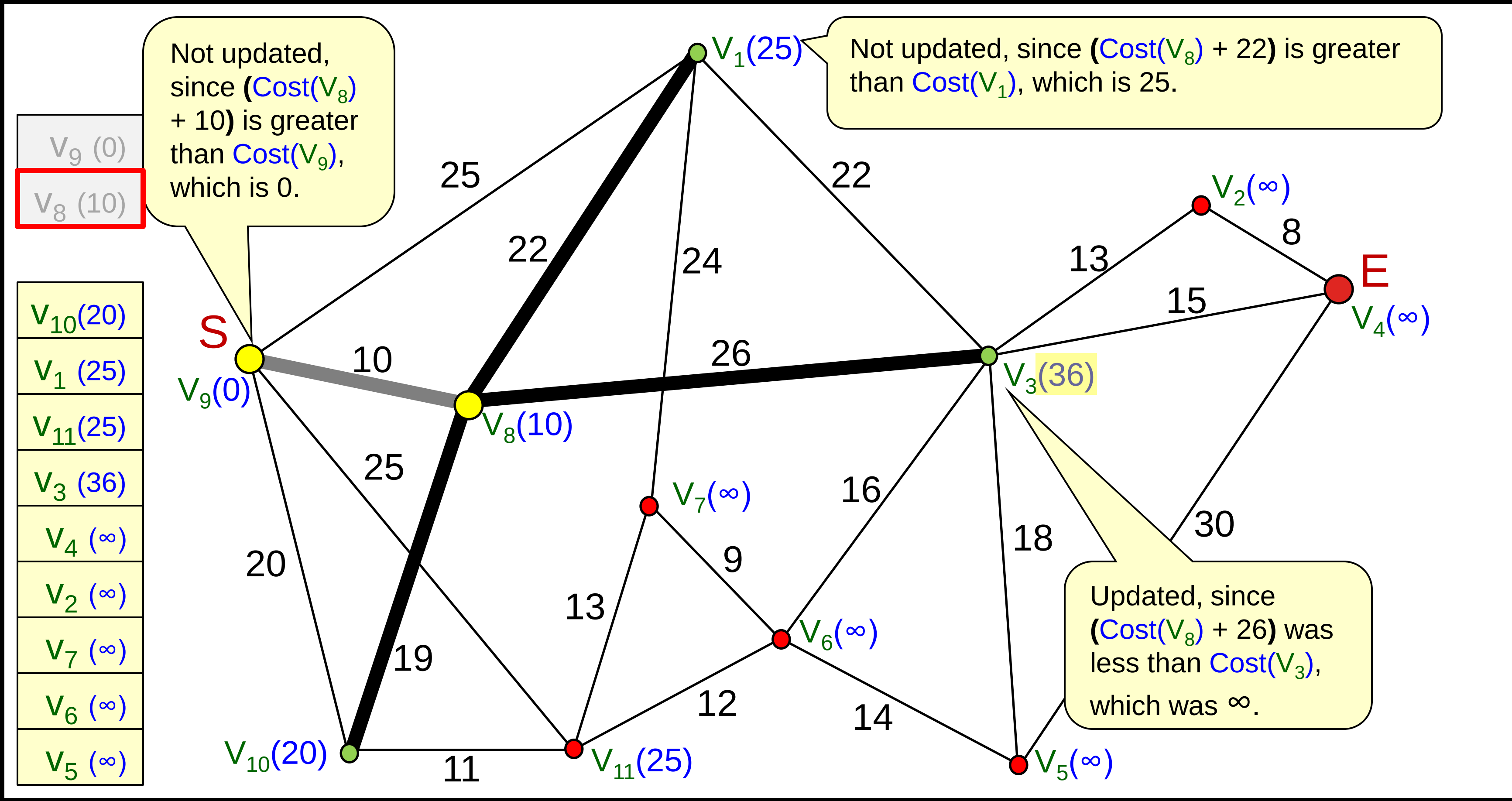
Dijkstra's Shortest Path Algorithm

- Visit all nodes connected to the extracted node
  - Update their cost if it is less (use e.g., edge length)



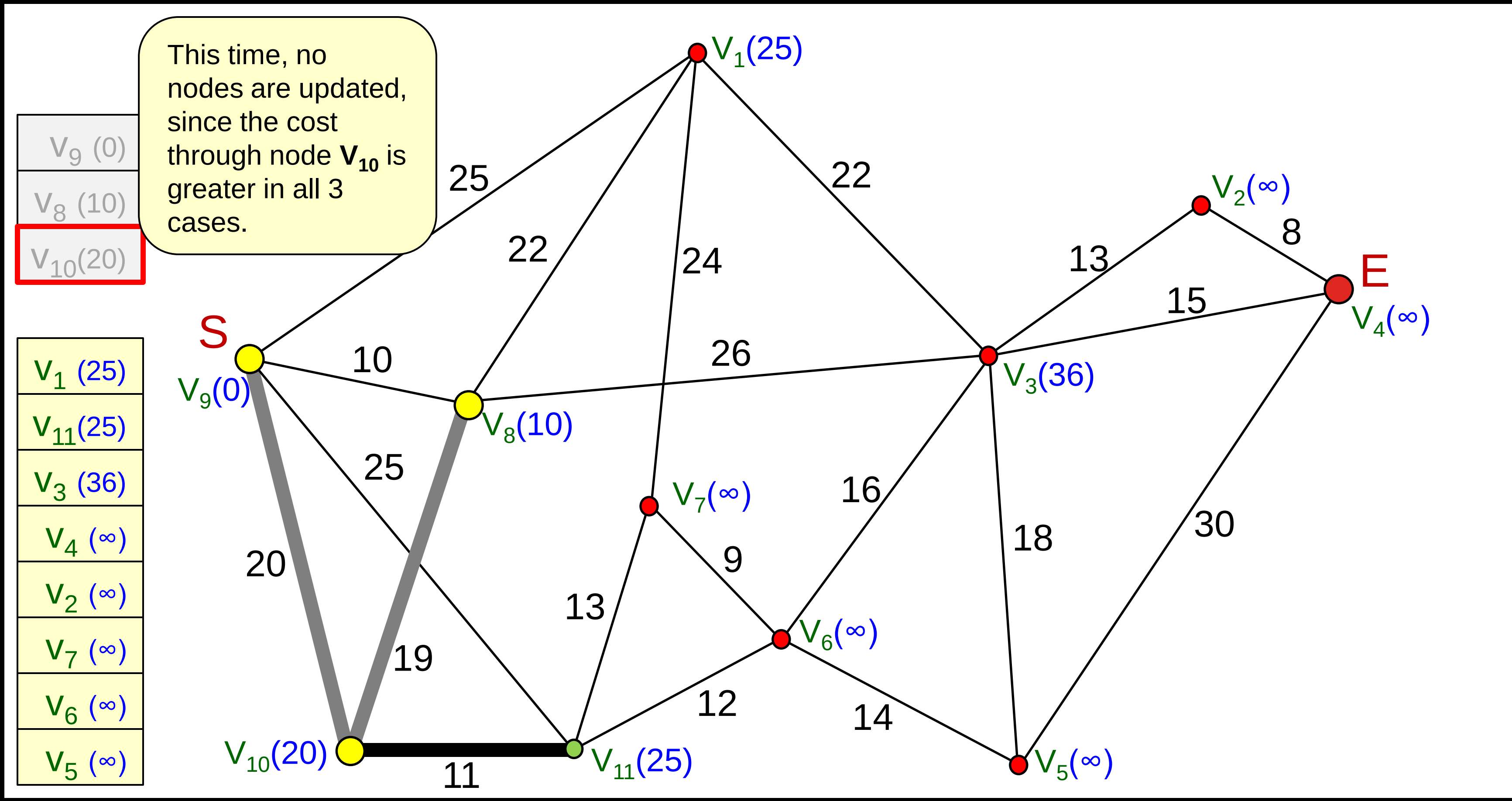
Dijkstra's Shortest Path Algorithm

- Repeat again, taking off the next closest node and check it's neighbours.



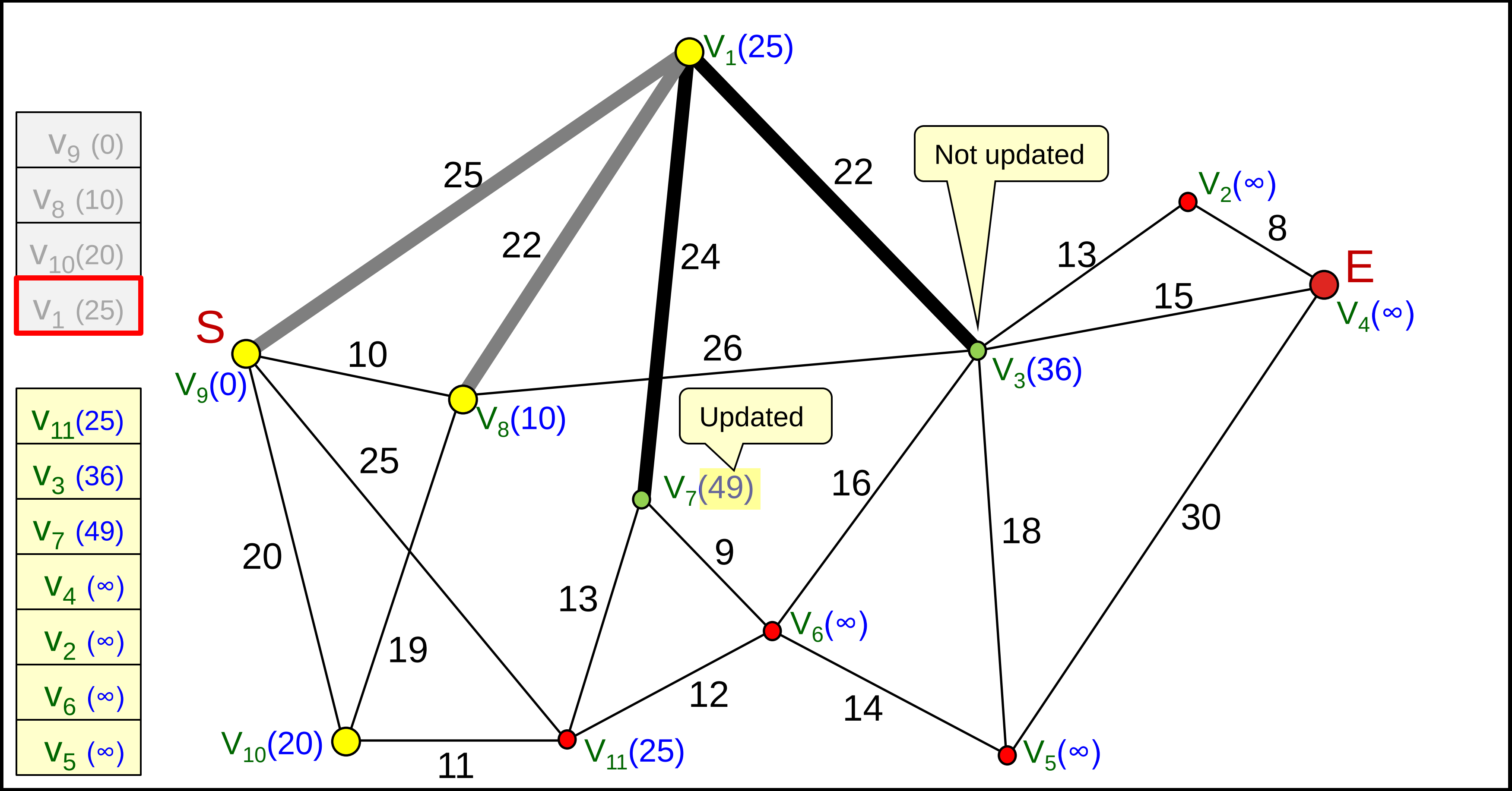
Dijkstra's Shortest Path Algorithm

- Repeat again, always updating nodes if the cost through this extracted node is lower.



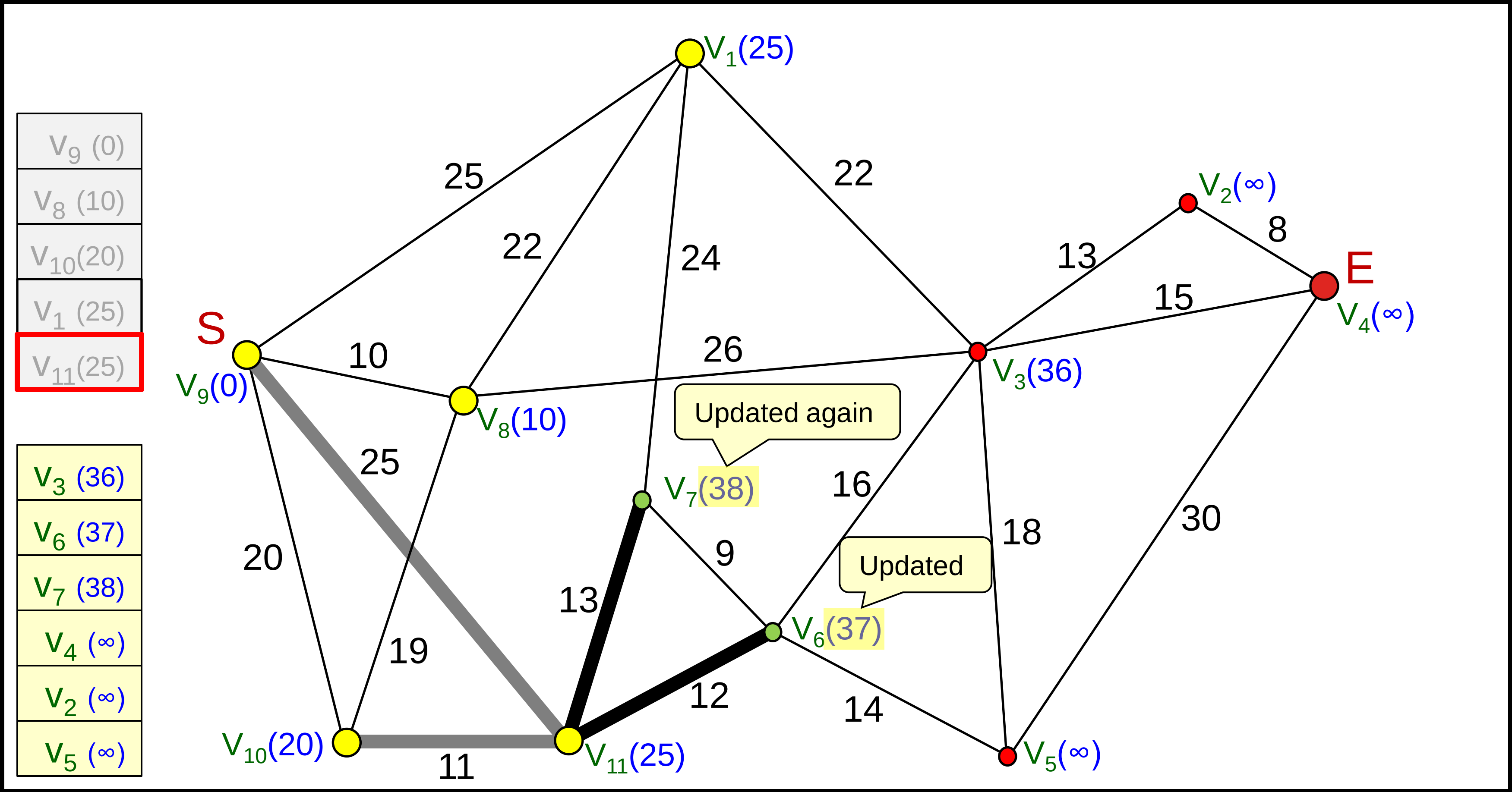
Dijkstra's Shortest Path Algorithm

- Repeat again …



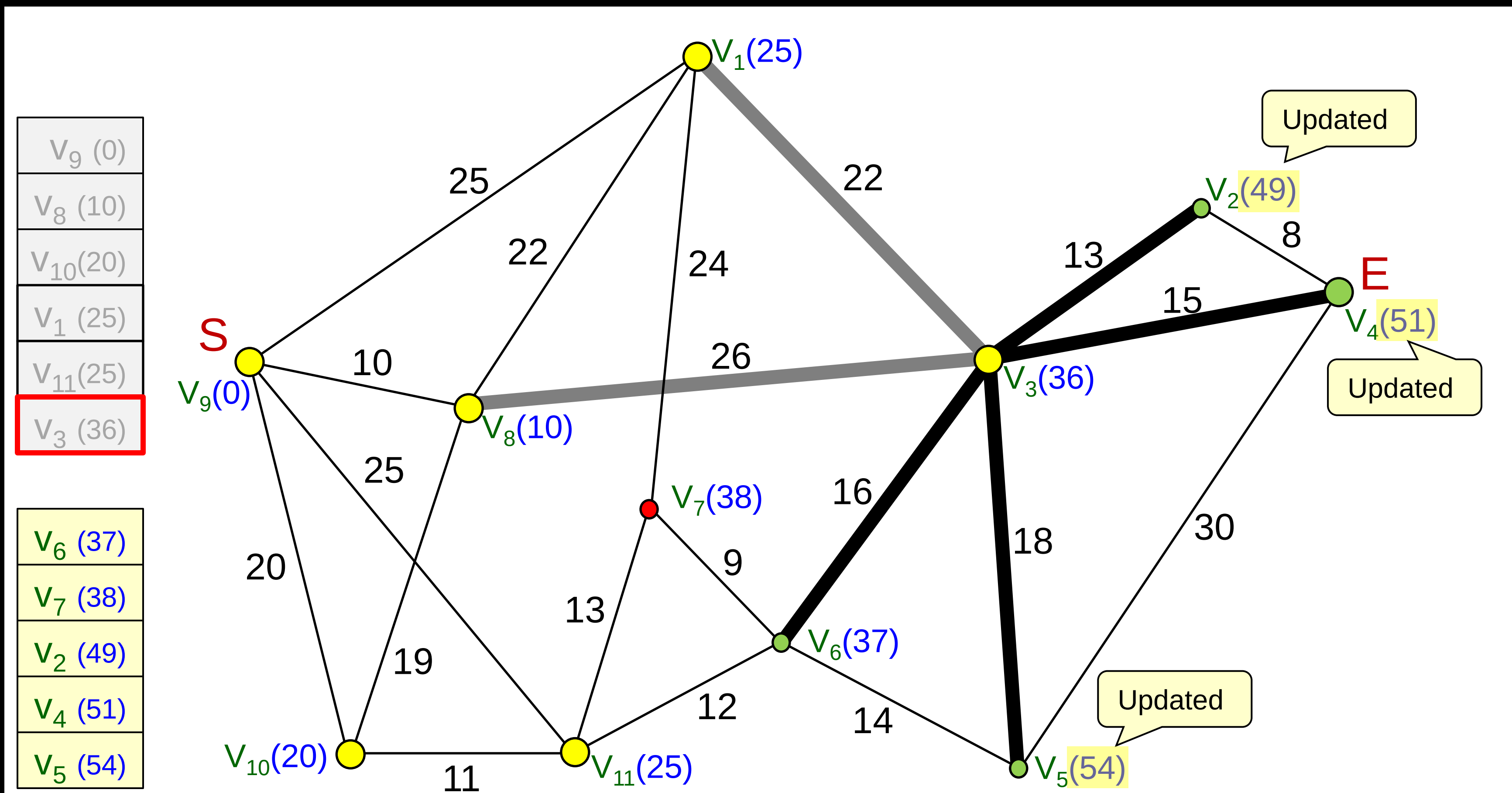
Dijkstra's Shortest Path Algorithm

- Repeat again …



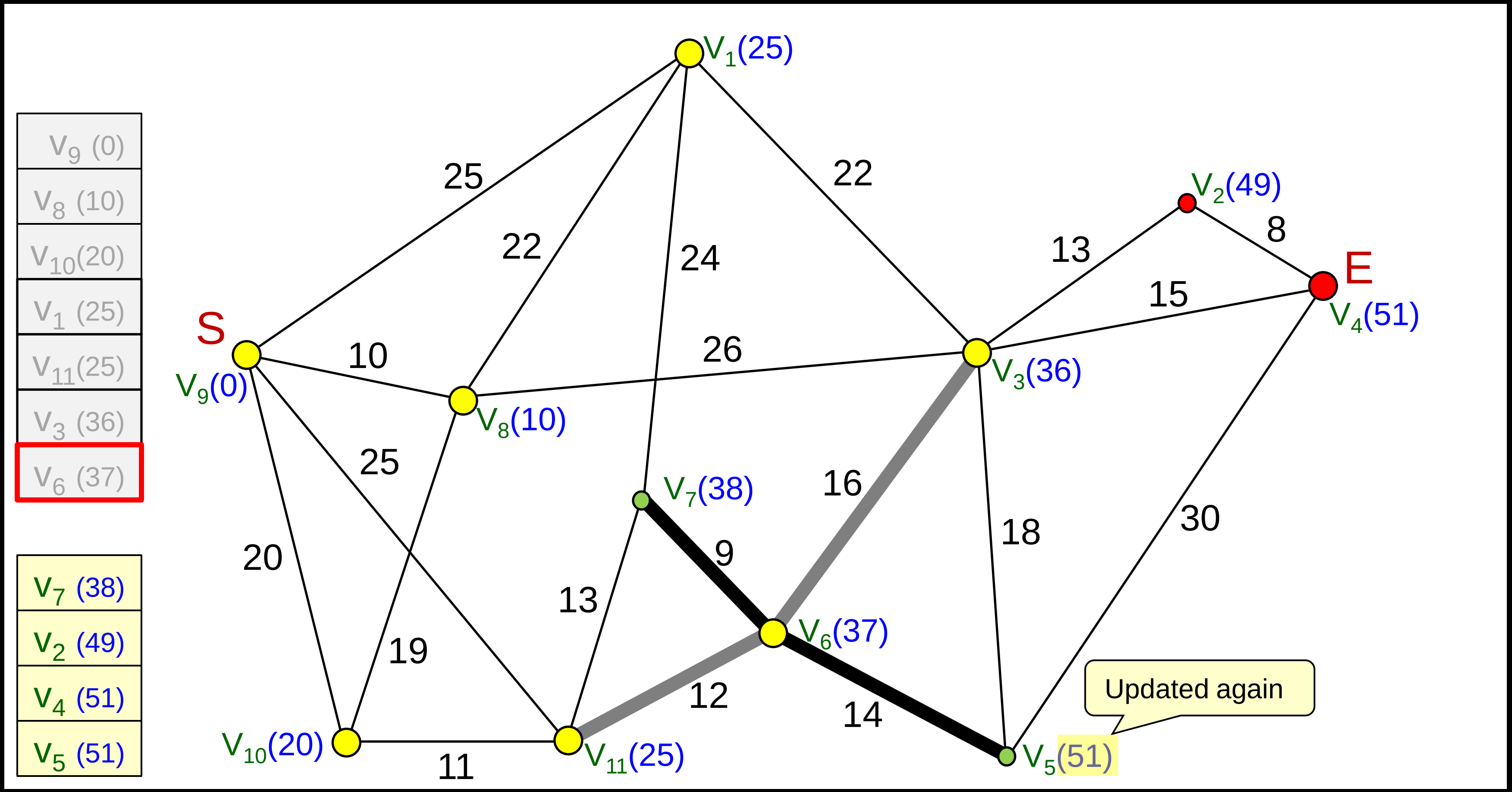
Dijkstra's Shortest Path Algorithm

- Keep going …



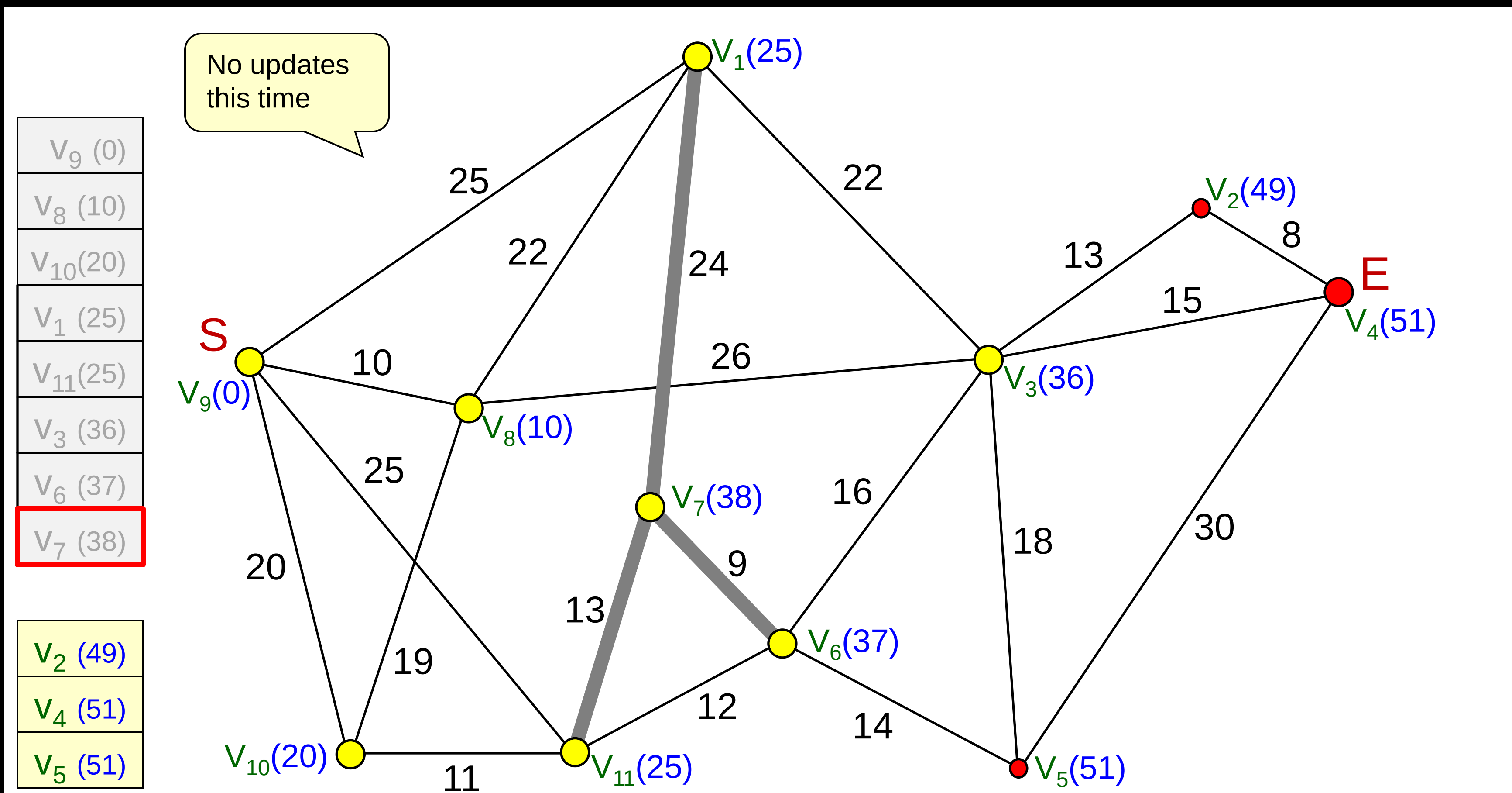
Dijkstra's Shortest Path Algorithm

- Keep going …



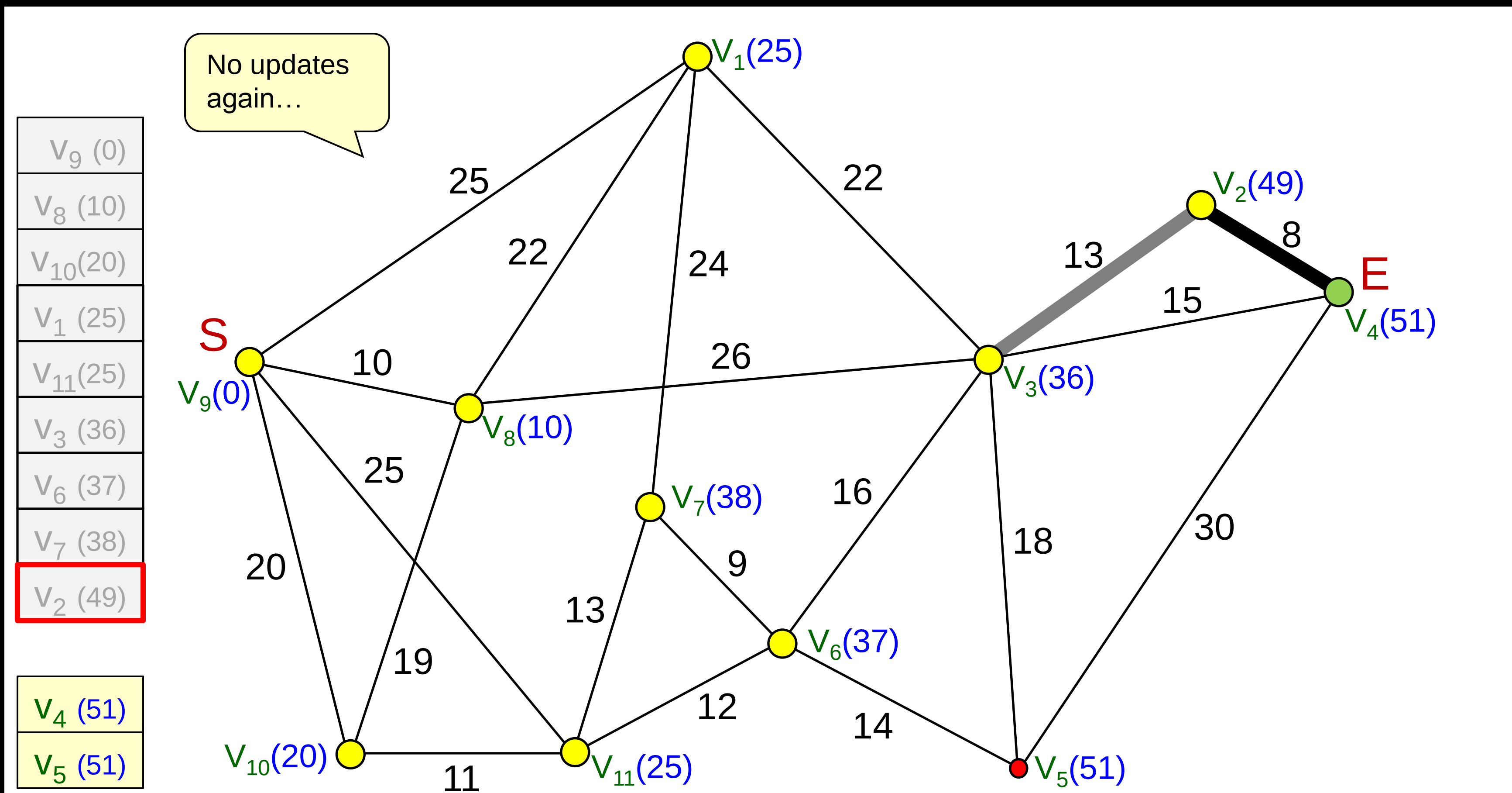
Dijkstra's Shortest Path Algorithm

- Almost done …



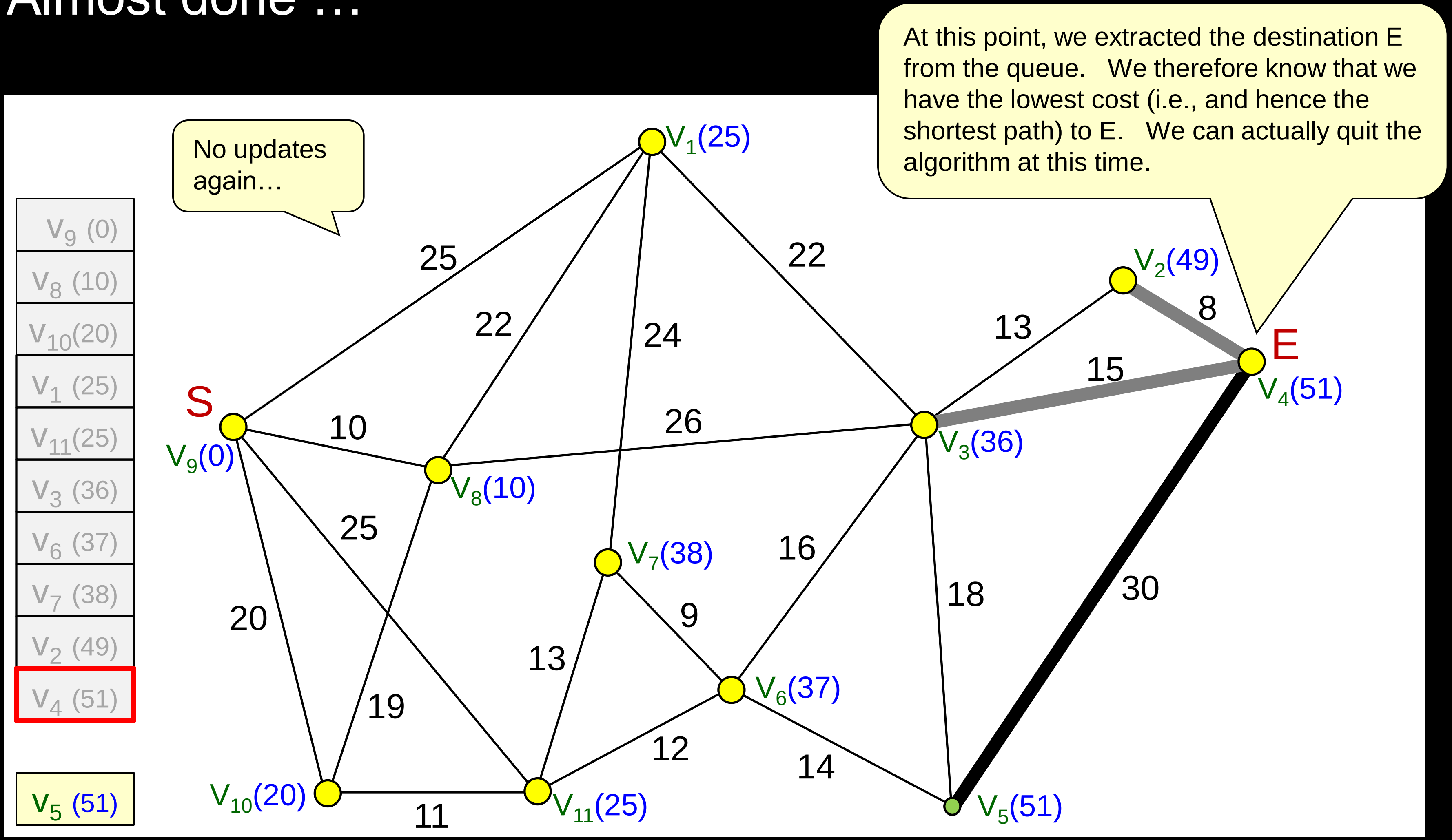
Dijkstra's Shortest Path Algorithm

- Almost done …



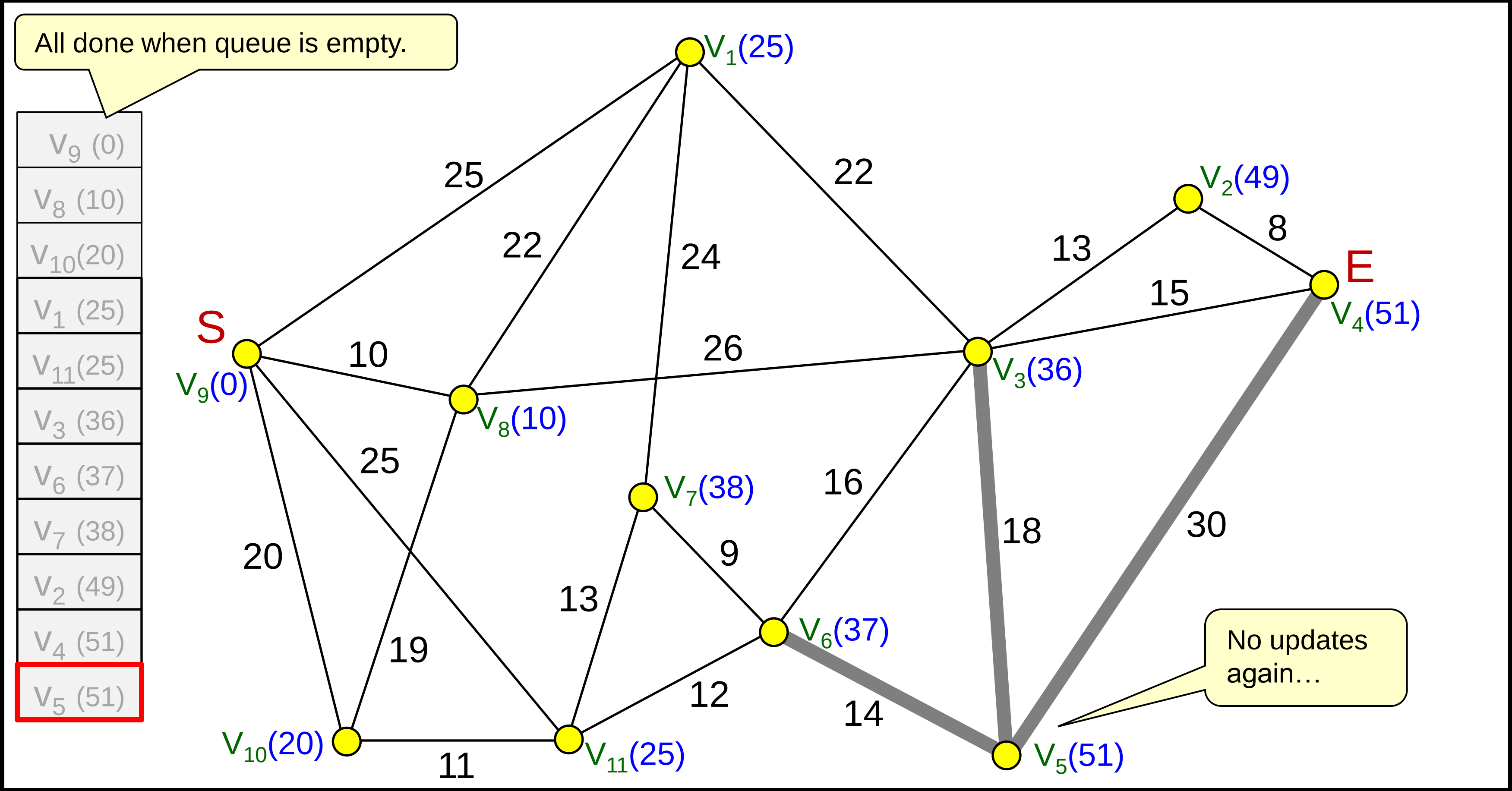
Dijkstra's Shortest Path Algorithm

- Almost done …



Dijkstra's Shortest Path Algorithm

- And this completes it. We now have the shortest path cost to each node, with respect to the source S.



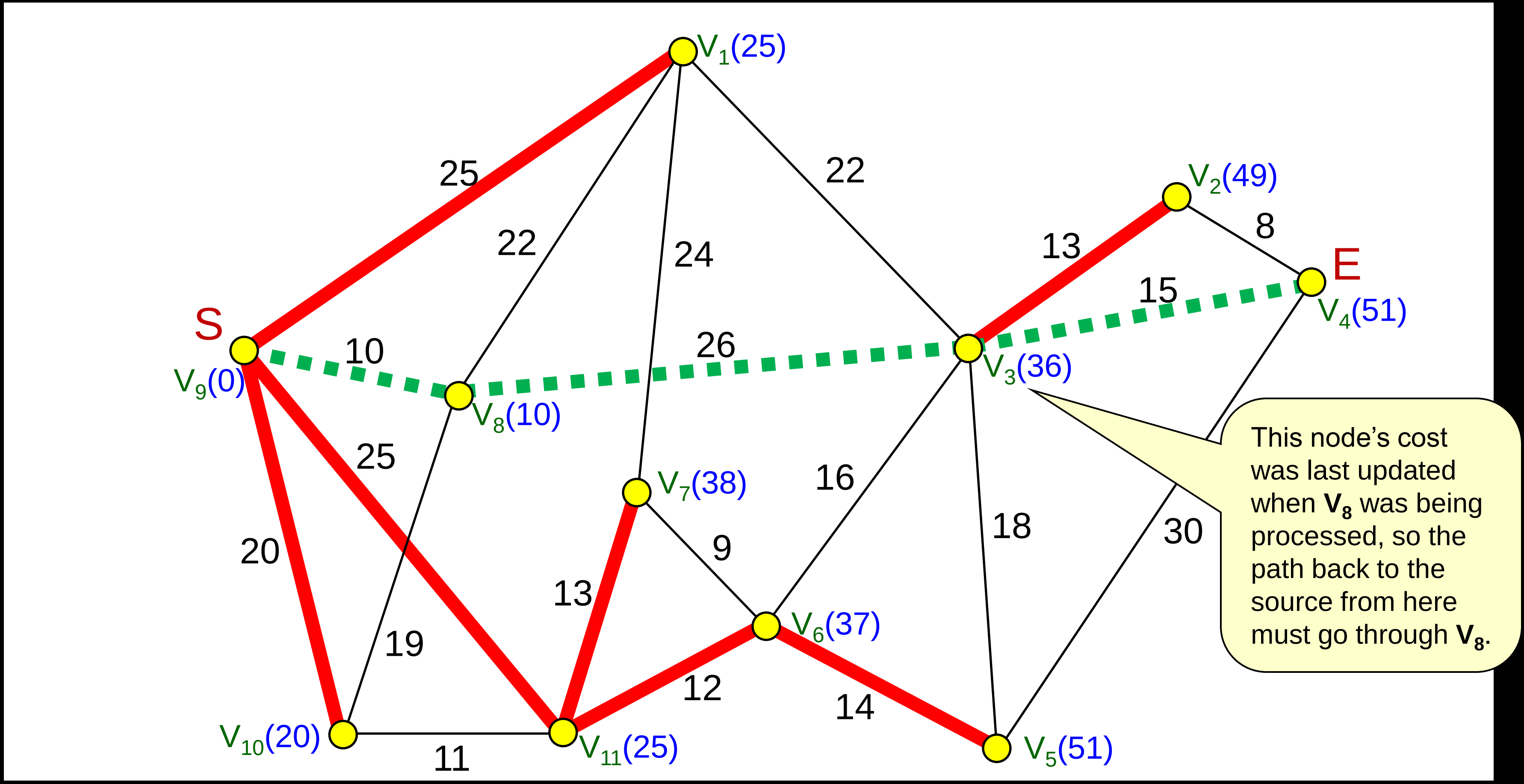
Computing the Shortest Path Tree

- We can use Dijkstra's shortest path algorithm to compute the shortest path tree from **s** in this graph.
  - Takes O(V log V + E) time for a V-vertex / E-edge graph

FUNCTION DijkstraShortestPathTree(G, **s**)
    Initialize **weight(v)** of each vertex **v** to ∞ but initialize **weight(s)** of **s** to **0**
    Q = a queue containing all vertices sorted by weights (lowest weight is at front)
    WHILE (**Q** is not empty) **DO**
        v = get and remove the vertex from **Q** with minimal weight
        FOR each edge $\overline{vu}$ outgoing from **v** **DO**
            IF (**weight(u)** > **weight(v)** + |$\overline{vu}$|) **THEN**
                weight(u) = **weight(v)** + |$\overline{vu}$|
                Re-sort node **u** in **Q** (because a weight has changed now)

Finding a Shortest Path

- Trace path from any node back to source by remembering node that updated the cost to it:



Remembering How We Got There

- When updating a node's cost to a better one, just add a line to remember which vertex led to that node in the path from **s**

FUNCTION DijkstraShortestPathTree(G, **s**, **e**)

Need to add parameter **e** if we don't want to compute the whole tree (e.g., if we just want to find the path to **e**)

    Initialize **weight(v)** of each vertex **v** to ∞ but initialize **weight(s)** of **s** to **0**
    Q = a queue containing all vertices sorted by weights (lowest weight is at front)
    WHILE (**Q** is not empty) **DO**
        v = get and remove the vertex from **Q** with minimal weight
        // **if** (**v** is the destination **e**) **then** break out of loop
        FOR each edge $\overline{\textbf{vu}}$ outgoing from **v** **DO**
            IF (**weight(u)** > **weight(v)** + |$\overline{\textbf{vu}}$|) **THEN**
                Set parent of **u** to **v**
                weight(u) = **weight(v)** + |$\overline{\textbf{vu}}$|
                Re-sort node **u** in **Q** (because a weight has changed now)

Only add this if we don't want to compute the whole tree (e.g., if we just want to find the path to **e**)

Store **v** as the node that let to **u** in the shortest path from **s** to **u**. So **v** is the parent of **u** in the shortest path tree from **s**.

Tracing the Path Back

- Finding the shortest path from **s** to **e** involves tracing the path back from **e** to **s**:

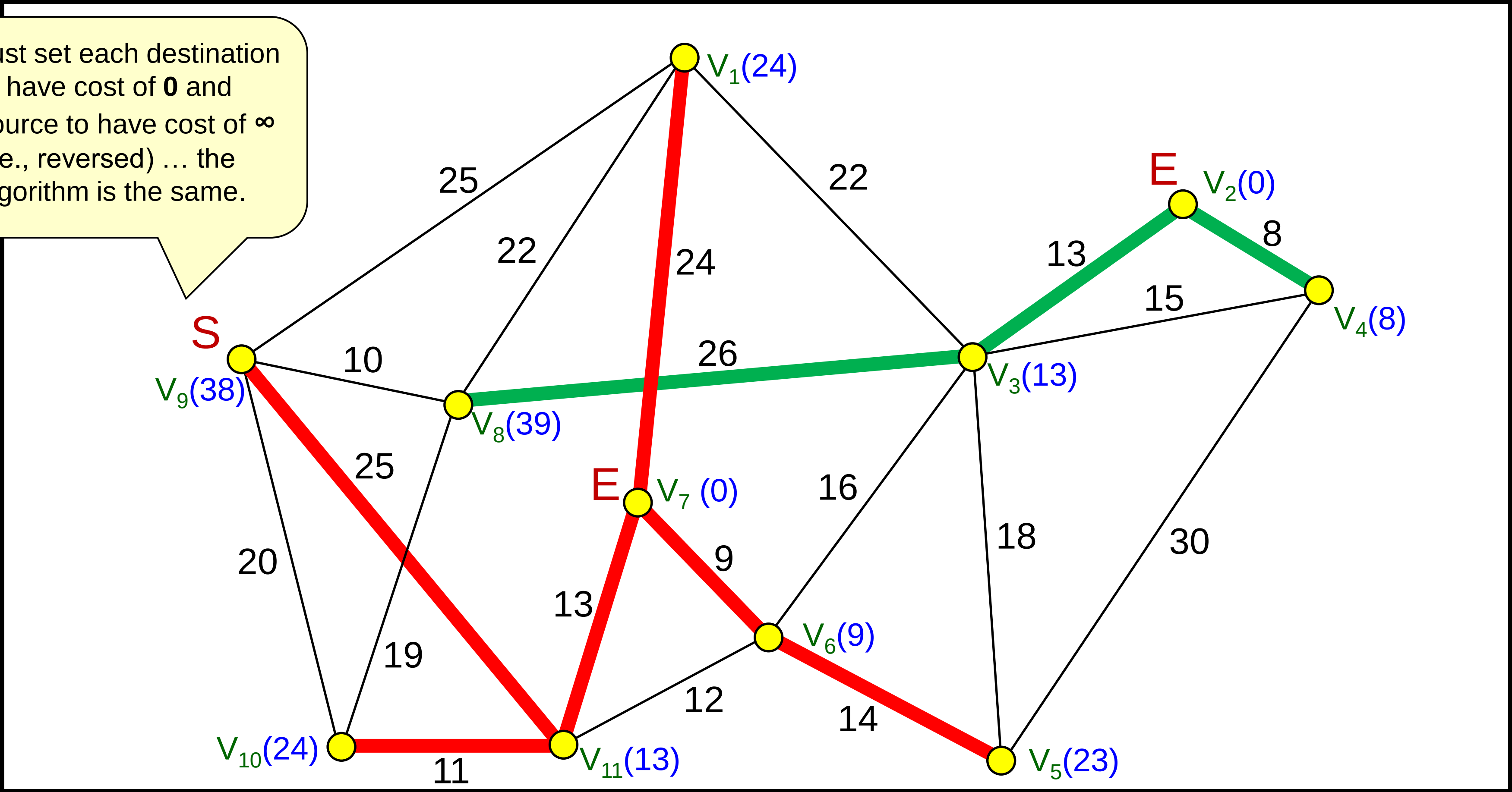
```
FUNCTION TraceBackPath(G, s, e)
    currentNode = e
    path = an empty list
    WHILE (currentNode is not s) DO
        add currentNode to front of path list
        currentNode = parent of currentNode
    Add s to front of path
```

Just get parent, then the parent of that parent, then that node's parent … etc … until we reach **s**.

Multiple Sources

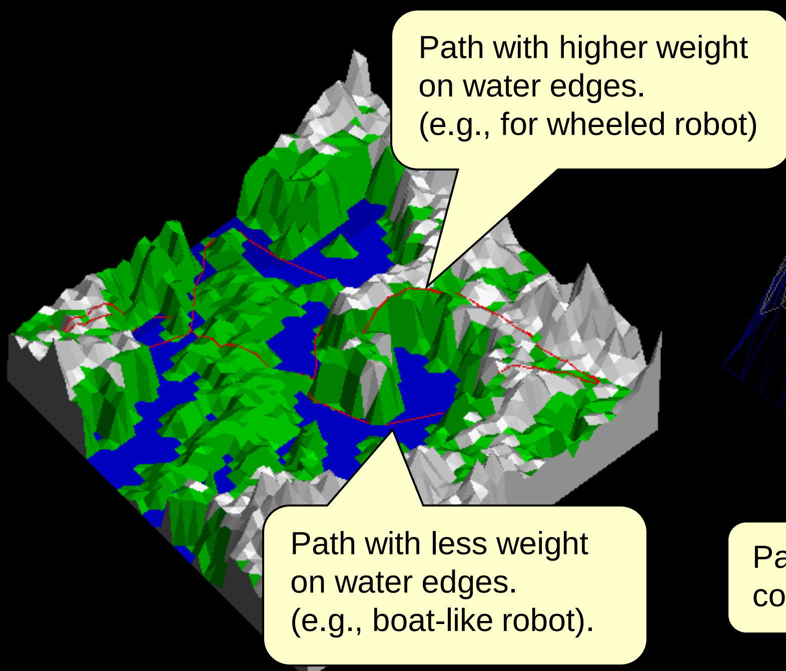
- Interestingly, the algorithm also works for multiple destinations.
  - Robot may wish to go to the closest of a set of destinations.

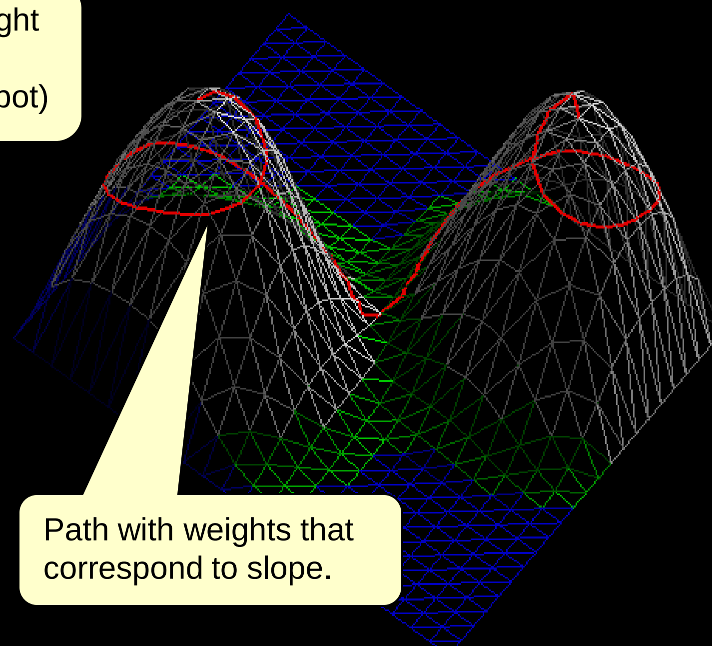
Just set each destination to have cost of **0** and source to have cost of ∞ (i.e., reversed) … the algorithm is the same.



Other Metrics

- Algorithm allows arbitrary weights on edges (as long as they are positive).
  - Allows some edges to be more “costly” than others
  - Can result in a kind of “weighted shortest path” that can go, for example, around obstacles (e.g., water).





Start the Lab …