# QEEE DSA05
# DATA STRUCTURES AND ALGORITHMS

G VENKATESH AND MADHAVAN MUKUND

LECTURE 7, 26 AUGUST 2014

# Recall that …

- BFS and DFS are two systematic ways to explore a graph
  - Both take time linear in the size of the graph with adjacency lists
- Recover paths by keeping parent information
- BFS can compute shortest paths, in terms of number of edges
- DFS numbering can reveal many interesting features

# Adding edge weights

# Adding edge weights

* Label each edge with a number—**cost**

# Adding edge weights

- Label each edge with a number—**cost**
  - Ticket price on a flight sector

# Adding edge weights

- Label each edge with a number—**cost**
  - Ticket price on a flight sector
  - Tolls on highway segment

# Adding edge weights

- Label each edge with a number—**cost**
  - Ticket price on a flight sector
  - Tolls on highway segment
  - Distance travelled between two stations

# Adding edge weights

- Label each edge with a number—**cost**
  - Ticket price on a flight sector
  - Tolls on highway segment
  - Distance travelled between two stations
  - Typical time between two locations during peak hour traffic

# Shortest paths

# Shortest paths

- **Weighted graph**

# Shortest paths

- **Weighted graph**
  - G=(V,E)  together with

# Shortest paths

- **Weighted graph**
  - G=(V,E) together with
  - **Weight function,** w : E→Reals

# Shortest paths

- **Weighted graph**
  - $G=(V,E)$ together with
  - **Weight function,** $w : E \rightarrow \text{Reals}$
- Let $e_1=(v_0,v_1)$, $e_2 = (v_1,v_2)$, …, $e_n = (v_{n-1},v_n)$ be a path from $v_0$ to $v_n$

# Shortest paths

- **Weighted graph**
  - G=(V,E) together with
  - **Weight function,** w : E→Reals
- Let $e_1=(v_0,v_1)$, $e_2 = (v_1,v_2)$, …, $e_n = (v_{n-1},v_n)$ be a path from $v_0$ to $v_n$
- Cost of the path is $w(e_1) + w(e_2) + \ldots + w(e_n)$

# Shortest paths

- **Weighted graph**
  - G=(V,E) together with
  - **Weight function,** $w : E \rightarrow \text{Reals}$
- Let $e_1=(v_0,v_1)$, $e_2 = (v_1,v_2)$, …, $e_n = (v_{n-1},v_n)$ be a path from $v_0$ to $v_n$
- Cost of the path is $w(e_1) + w(e_2) + \ldots + w(e_n)$
- **Shortest path** from $v_0$ to $v_n$ : minimum cost

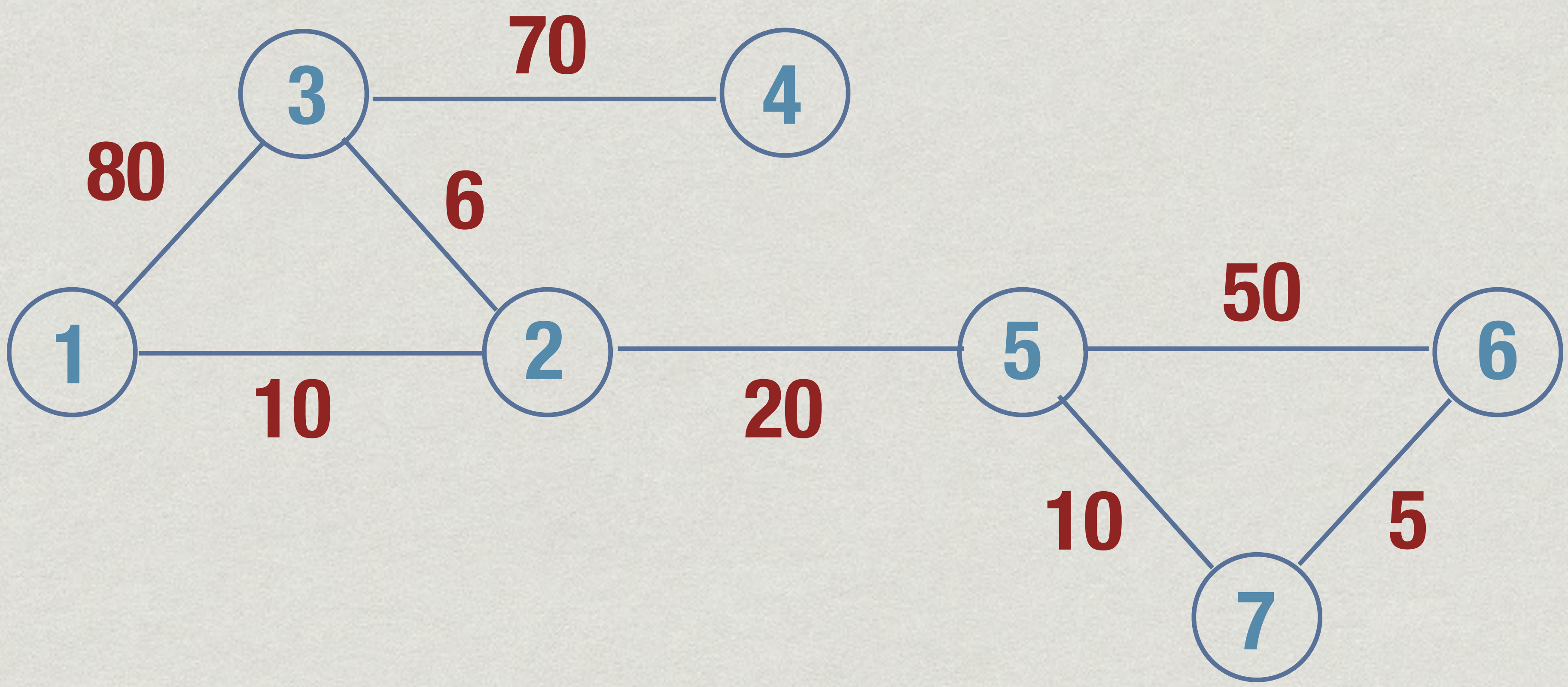
# Shortest paths …

# Shortest paths …

- BFS finds path with fewest number of edges

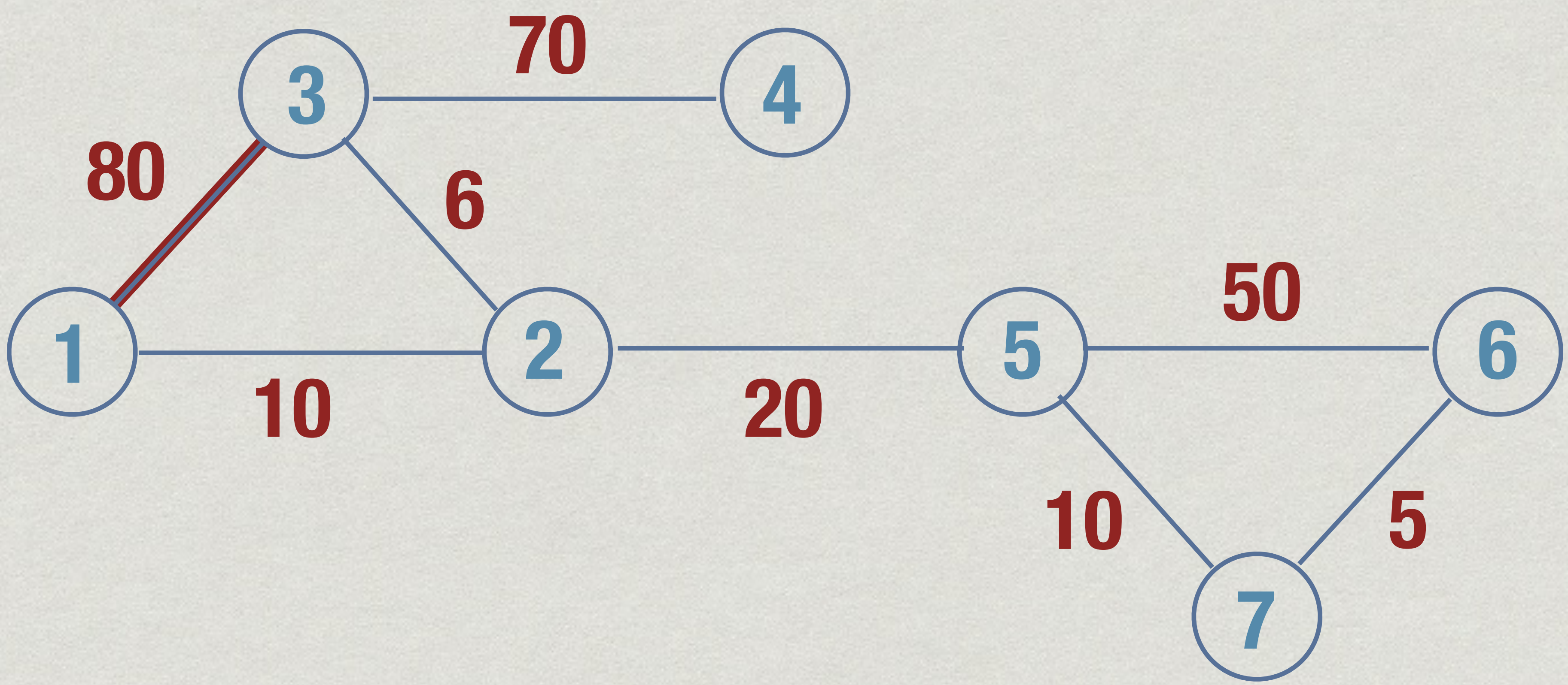
# Shortest paths …

- BFS finds path with fewest number of edges
- In a weighted graph, need not be the shortest path



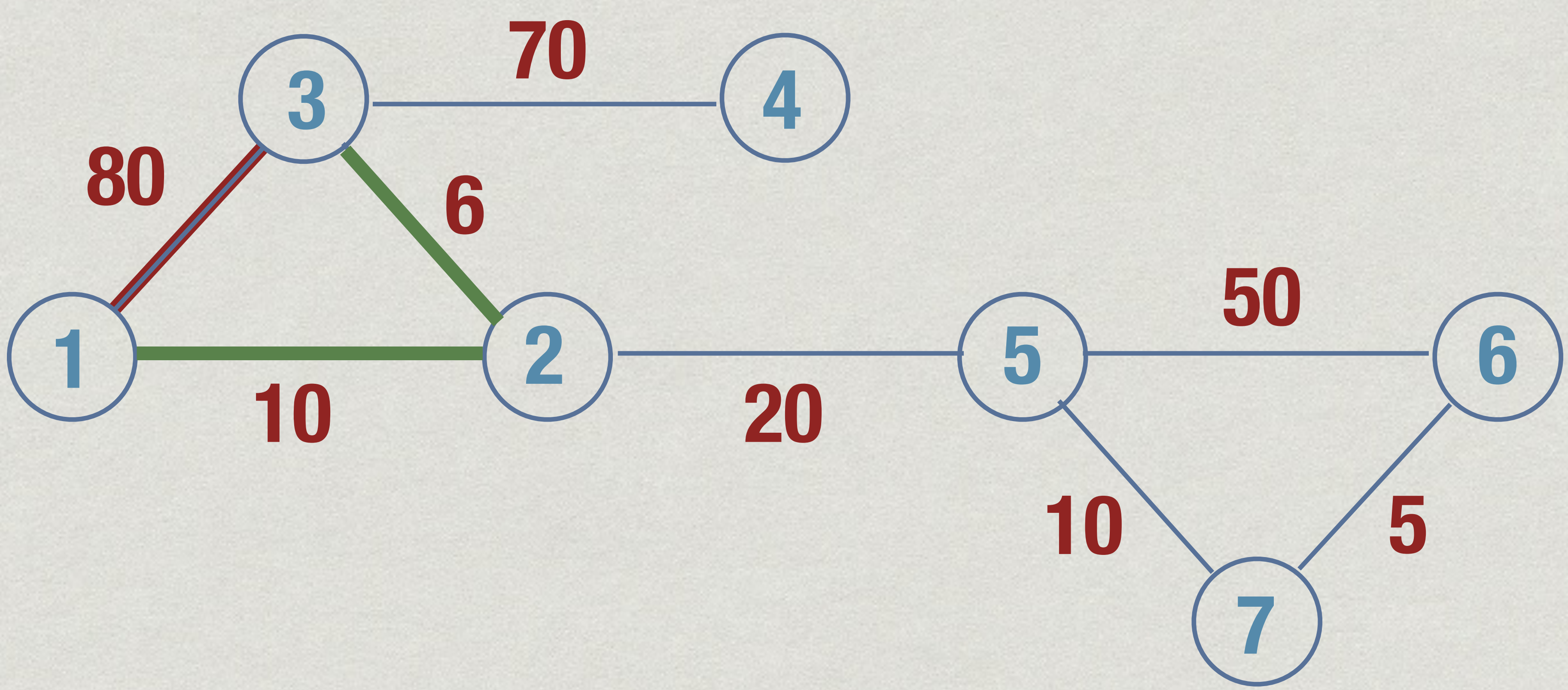
# Shortest paths …

- BFS finds path with fewest number of edges
- In a weighted graph, need not be the shortest path



# Shortest paths …

- BFS finds path with fewest number of edges
- In a weighted graph, need not be the shortest path



# Shortest path problems

# Shortest path problems

- **Single source**

# Shortest path problems

- **Single source**
    - Find shortest paths from some fixed vertex, say 1, to every other vertex

# Shortest path problems

- **Single source**
  - Find shortest paths from some fixed vertex, say 1, to every other vertex
  - Transport finished product from factory (single source) to all retail outlets

# Shortest path problems

- **Single source**
  - Find shortest paths from some fixed vertex, say 1, to every other vertex
  - Transport finished product from factory (single source) to all retail outlets
  - Courier company delivers items from distribution centre (single source) to addressees

# Shortest path problems

# Shortest path problems

- **All pairs**

# Shortest path problems

- **All pairs**
  - Find shortest paths between every pair of vertices i and j

# Shortest path problems

- **All pairs**
  - Find shortest paths between every pair of vertices i and j
  - Railway routes, shortest way to travel between any pair of cities

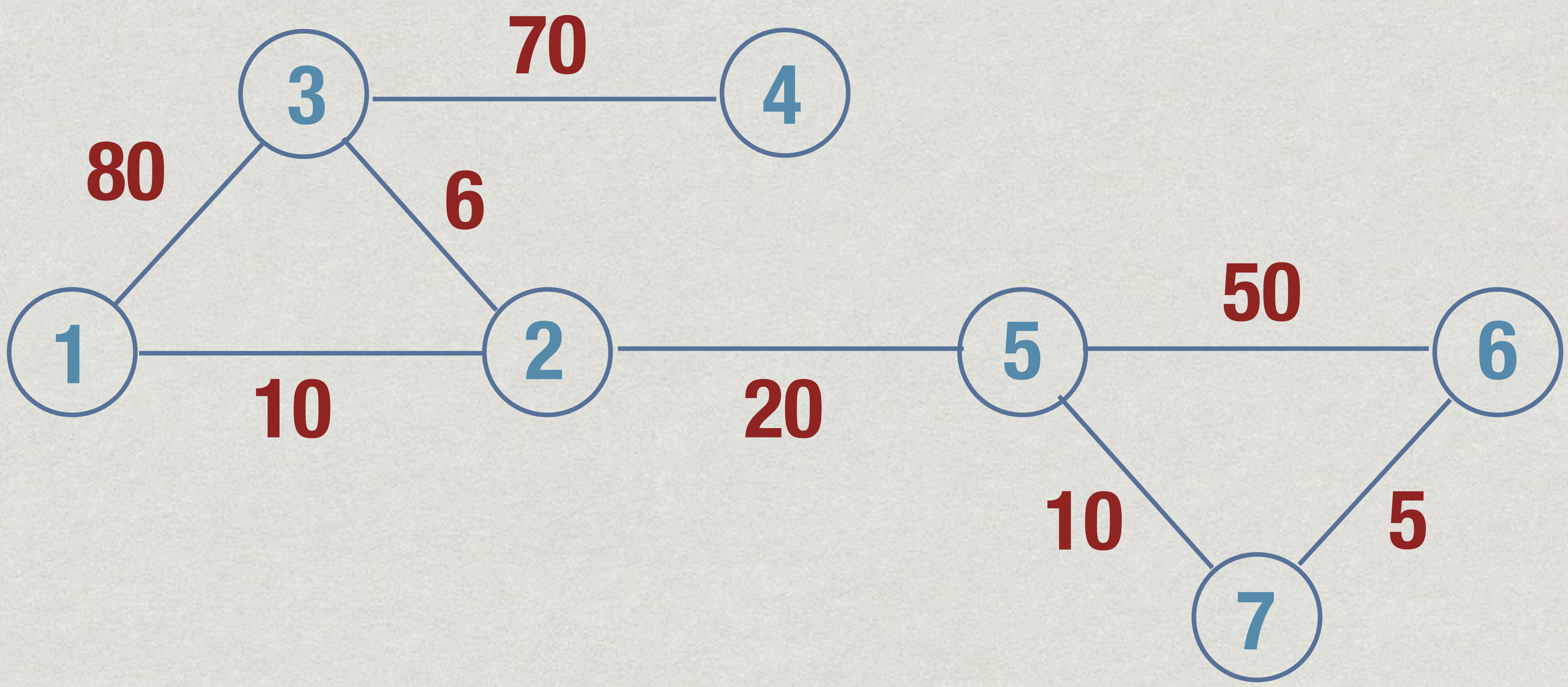
# Today…

# Today…

- Single source shortest paths

# Today…

- Single source shortest paths
- For instance, shortest paths from 1 to 2,3,…,7



# Single source shortest paths

# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines

# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1

# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
  - Fire travels at uniform speed along each pipeline

# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
    - Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 1 is nearest vertex

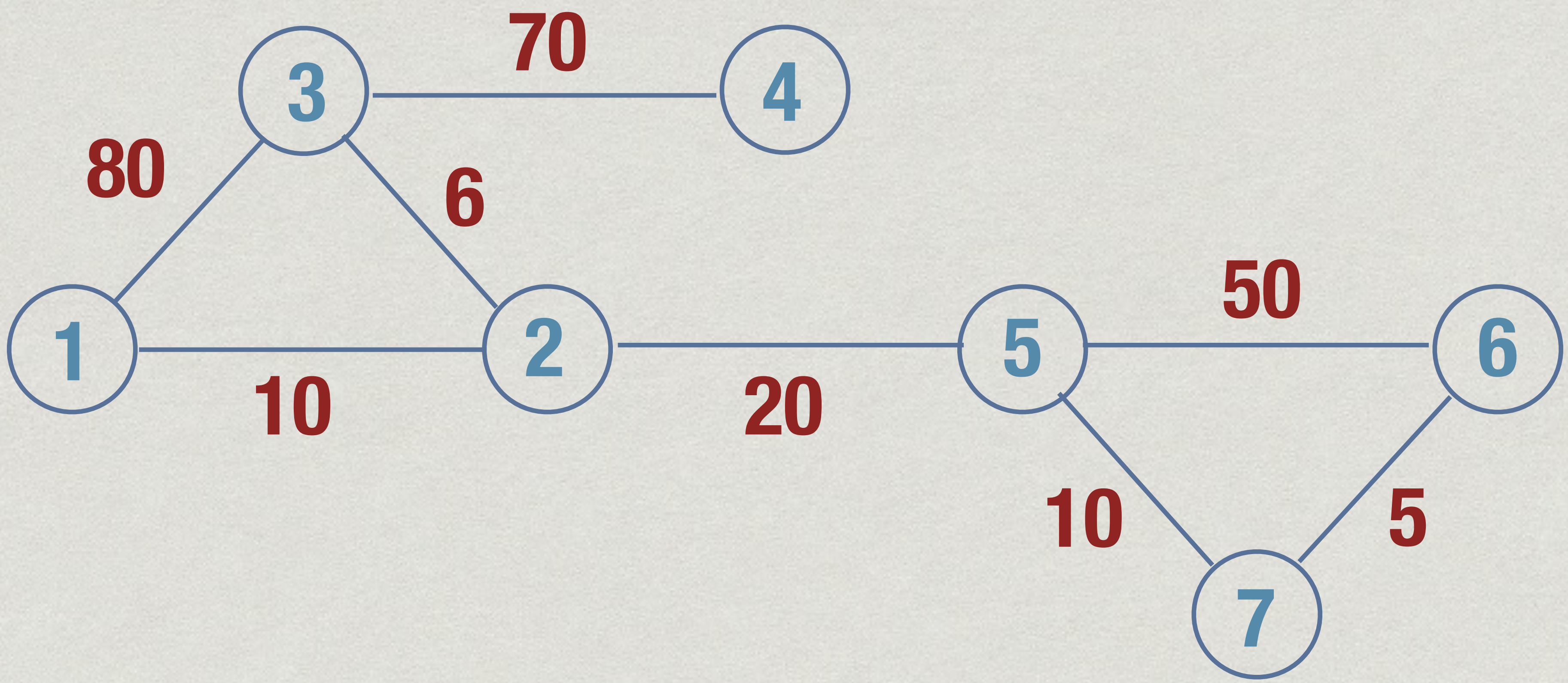
# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
  - Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 1 is nearest vertex
- Next oil depot is second nearest vertex

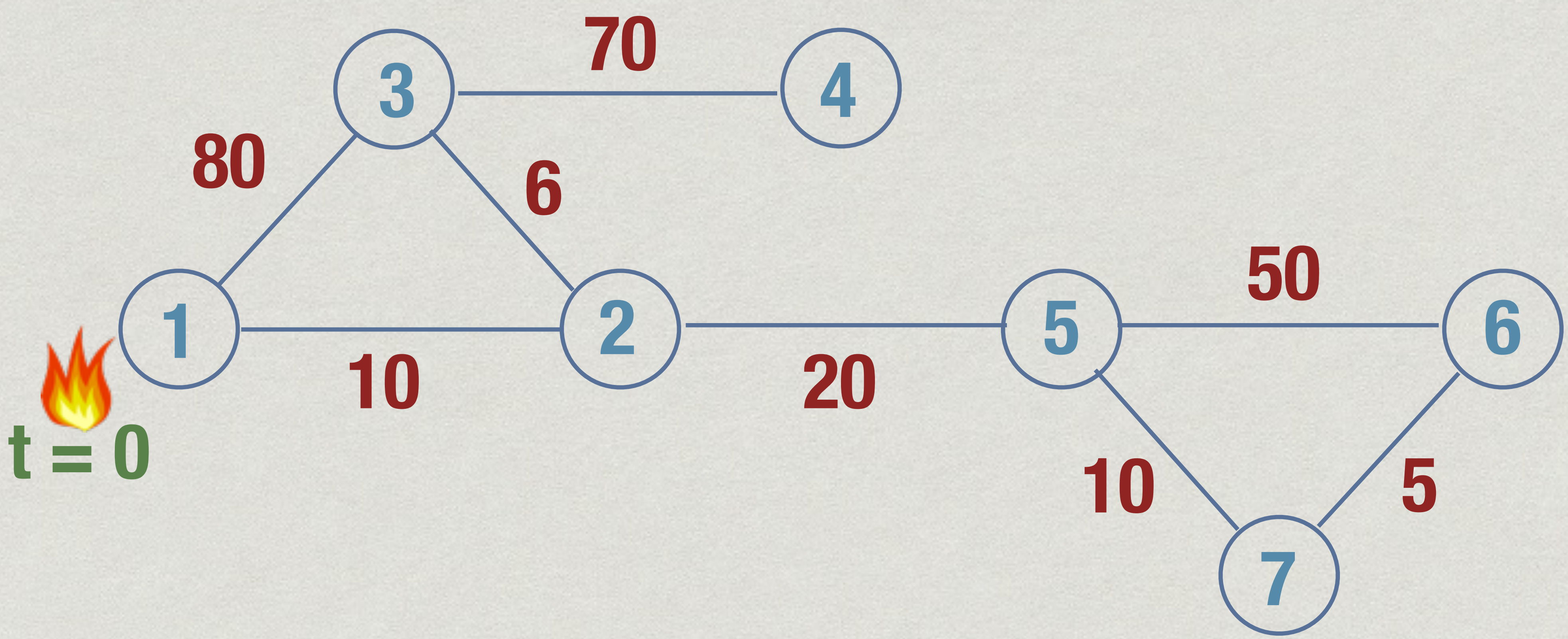
# Single source shortest paths

- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 1
  - Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 1 is nearest vertex
- Next oil depot is second nearest vertex
- ...

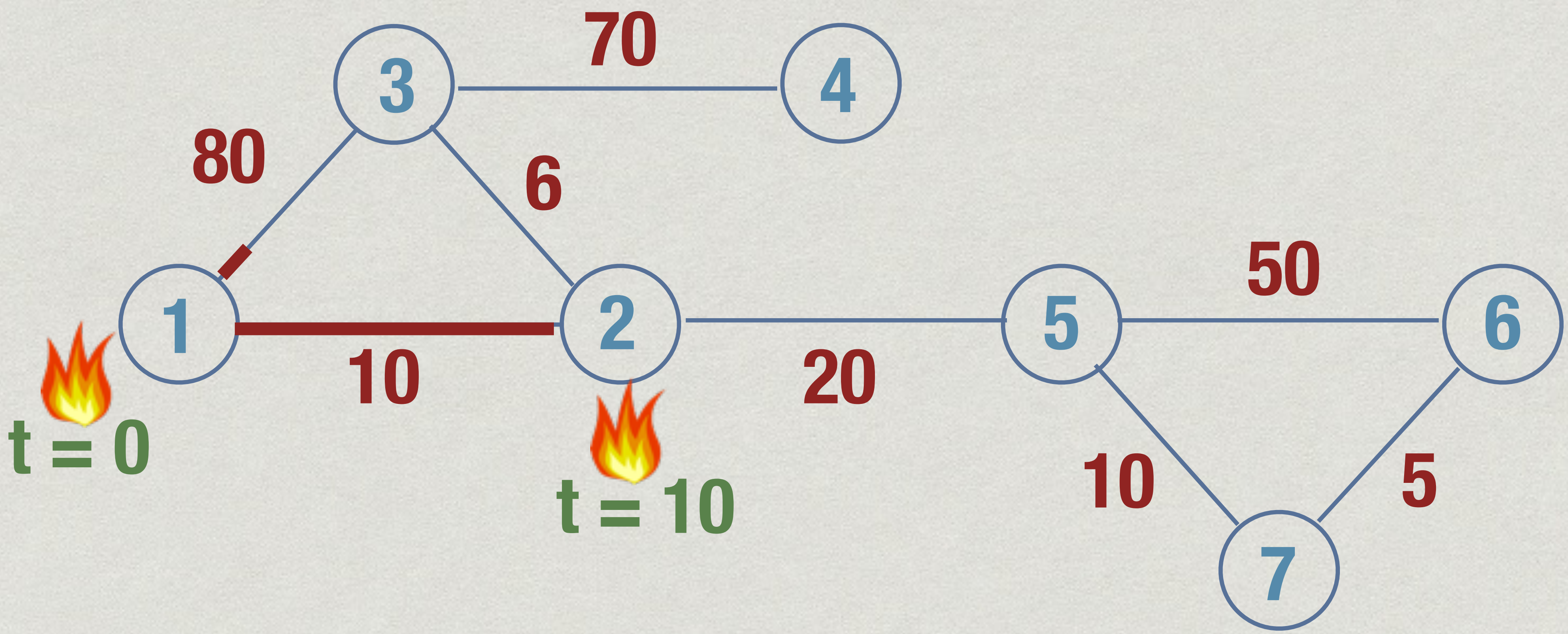
# Single source shortest paths



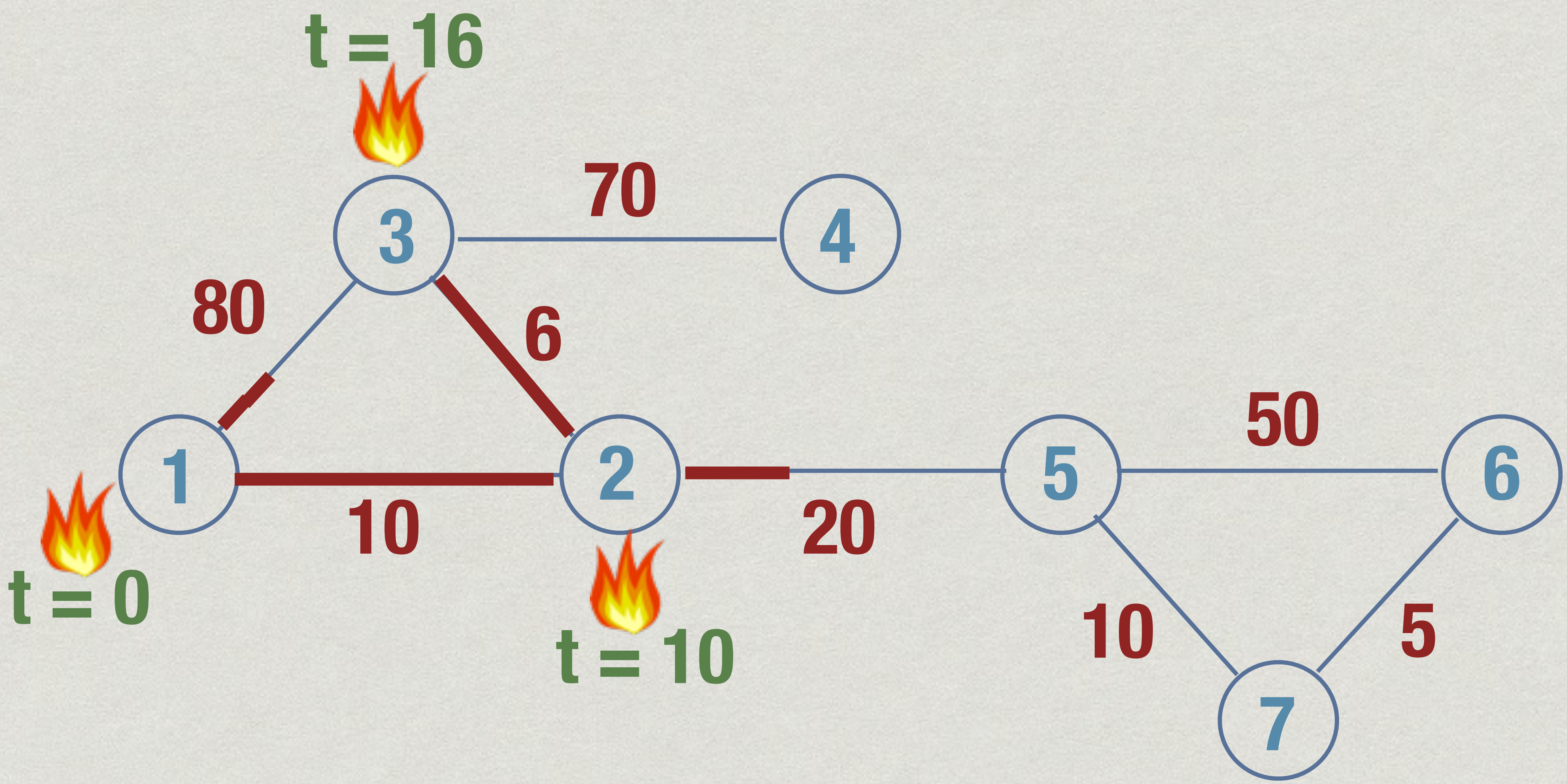
# Single source shortest paths



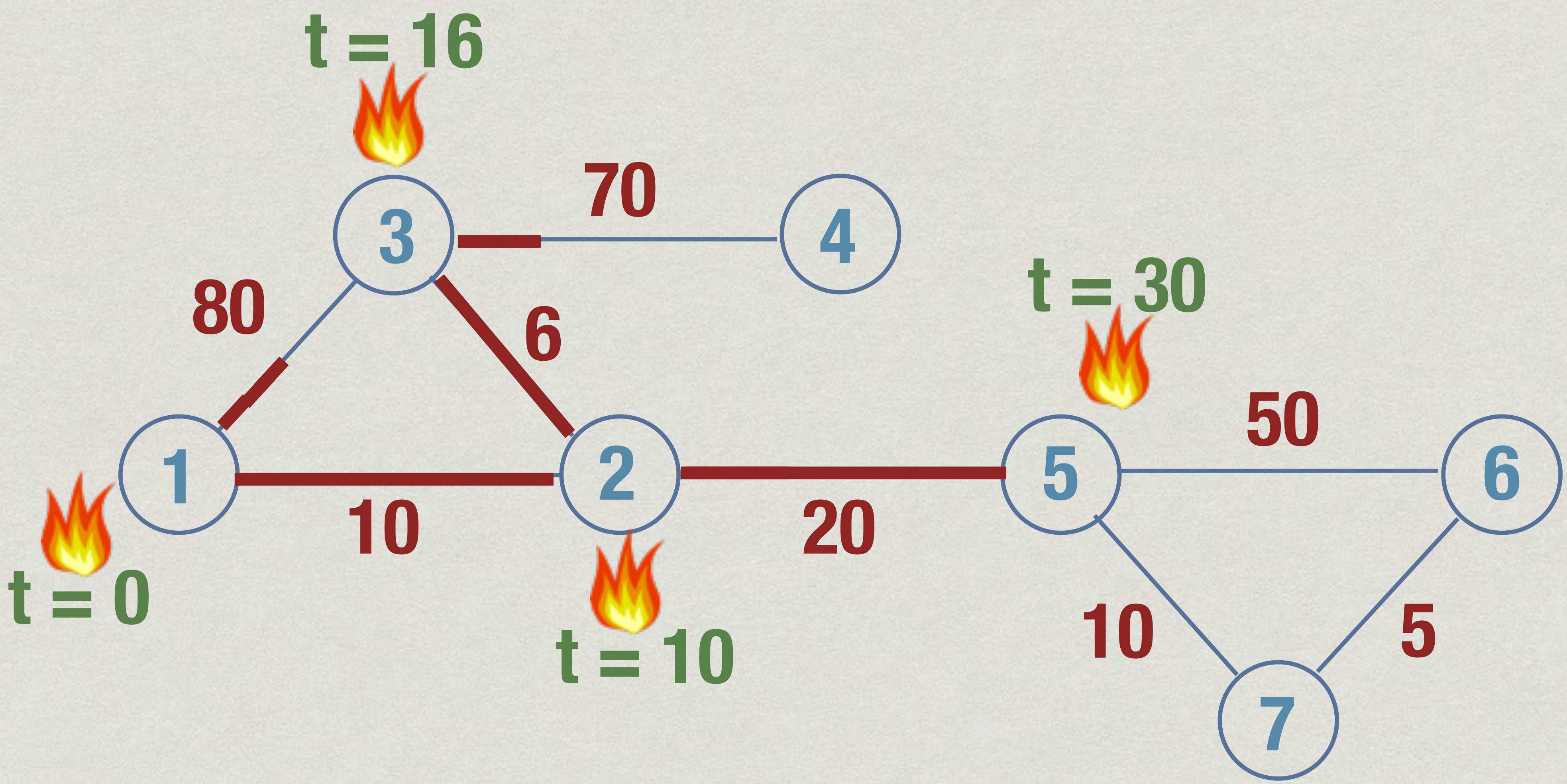
# Single source shortest paths



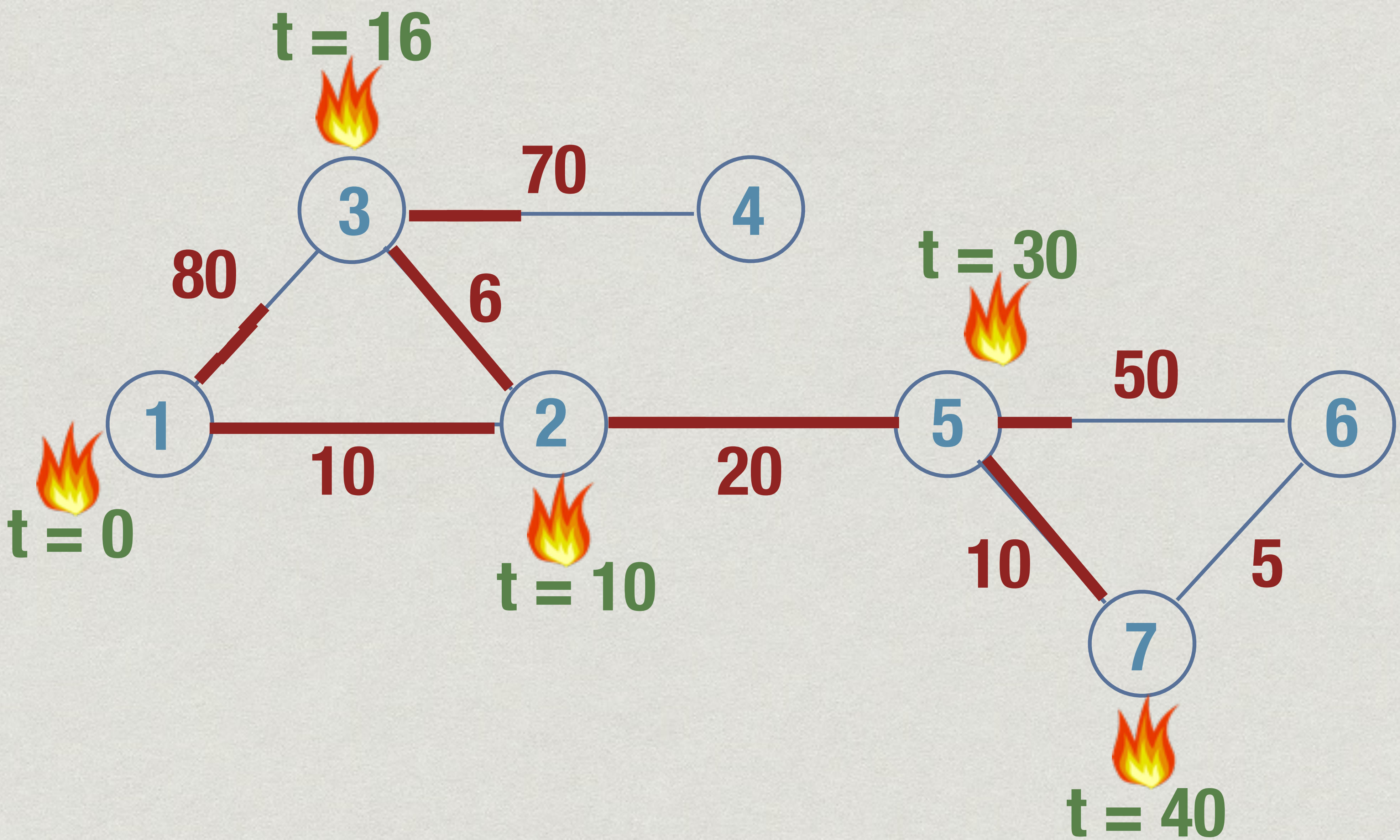
# Single source shortest paths



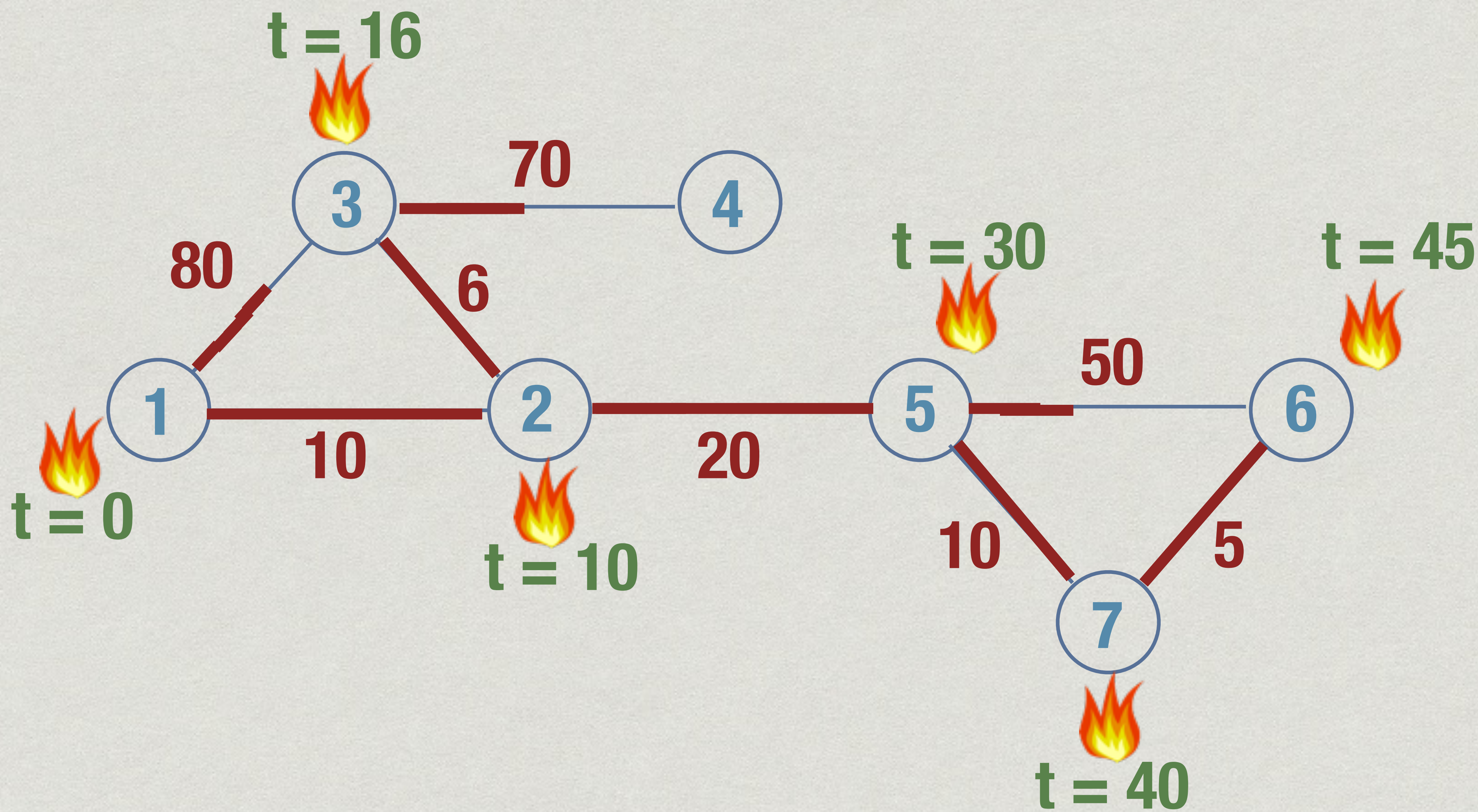
# Single source shortest paths



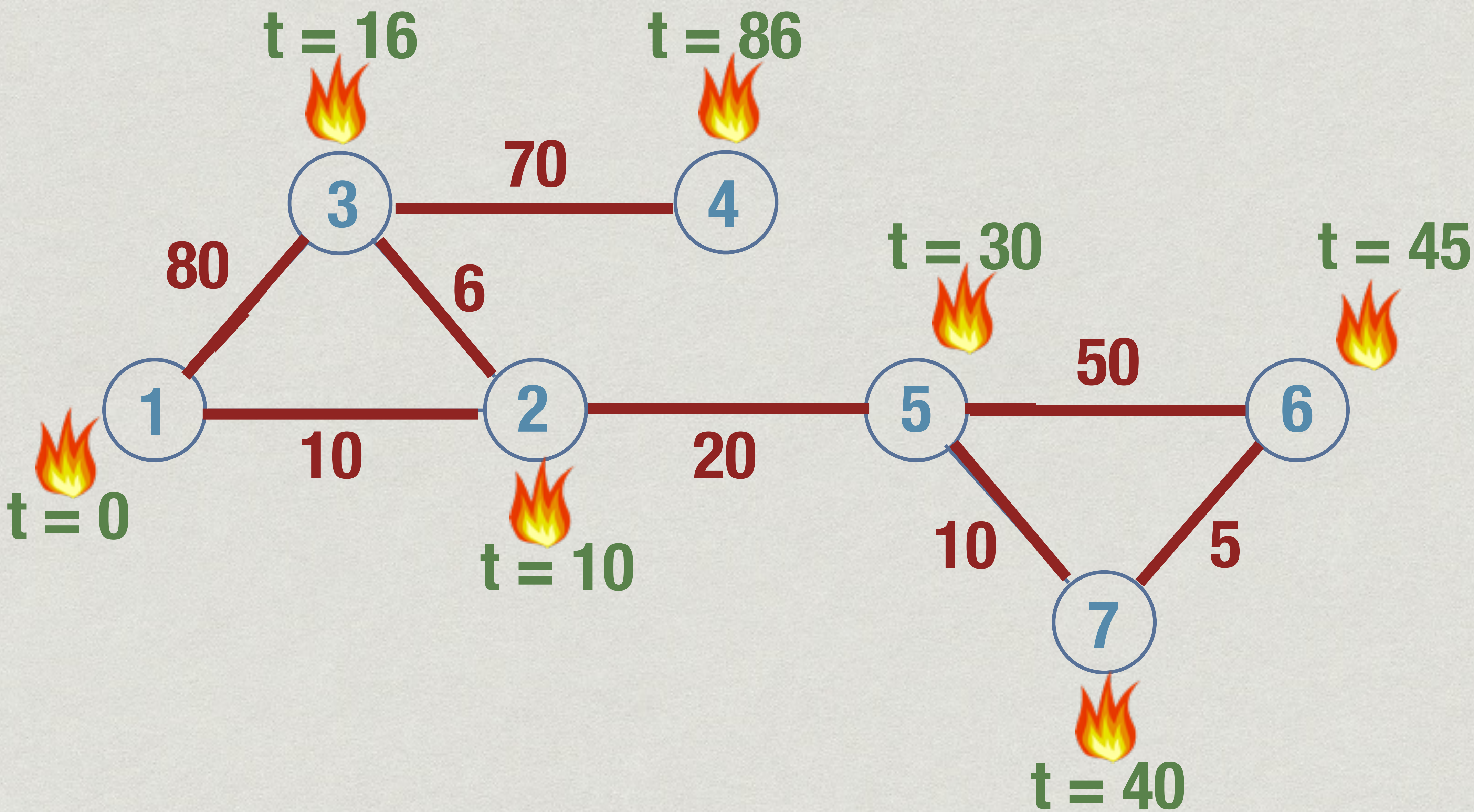
# Single source shortest paths



# Single source shortest paths

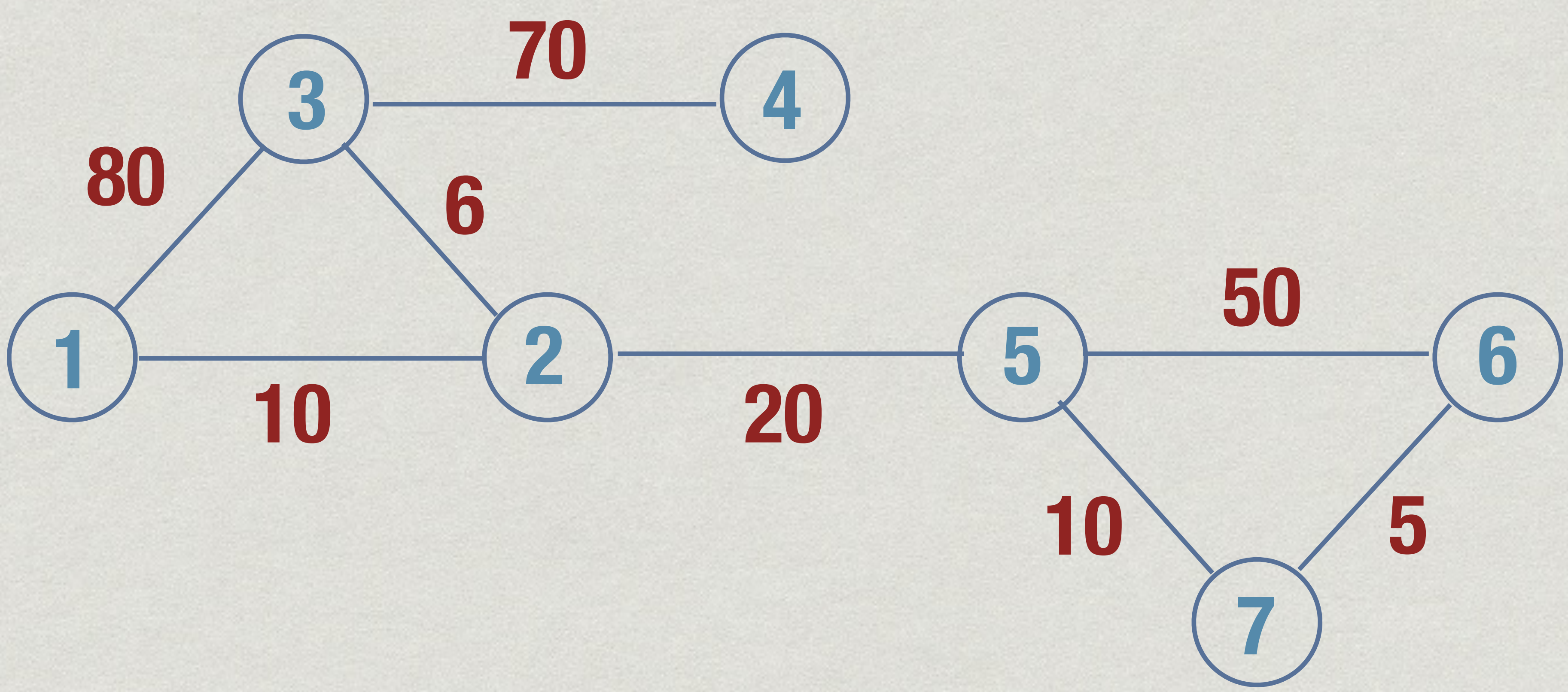


# Single source shortest paths



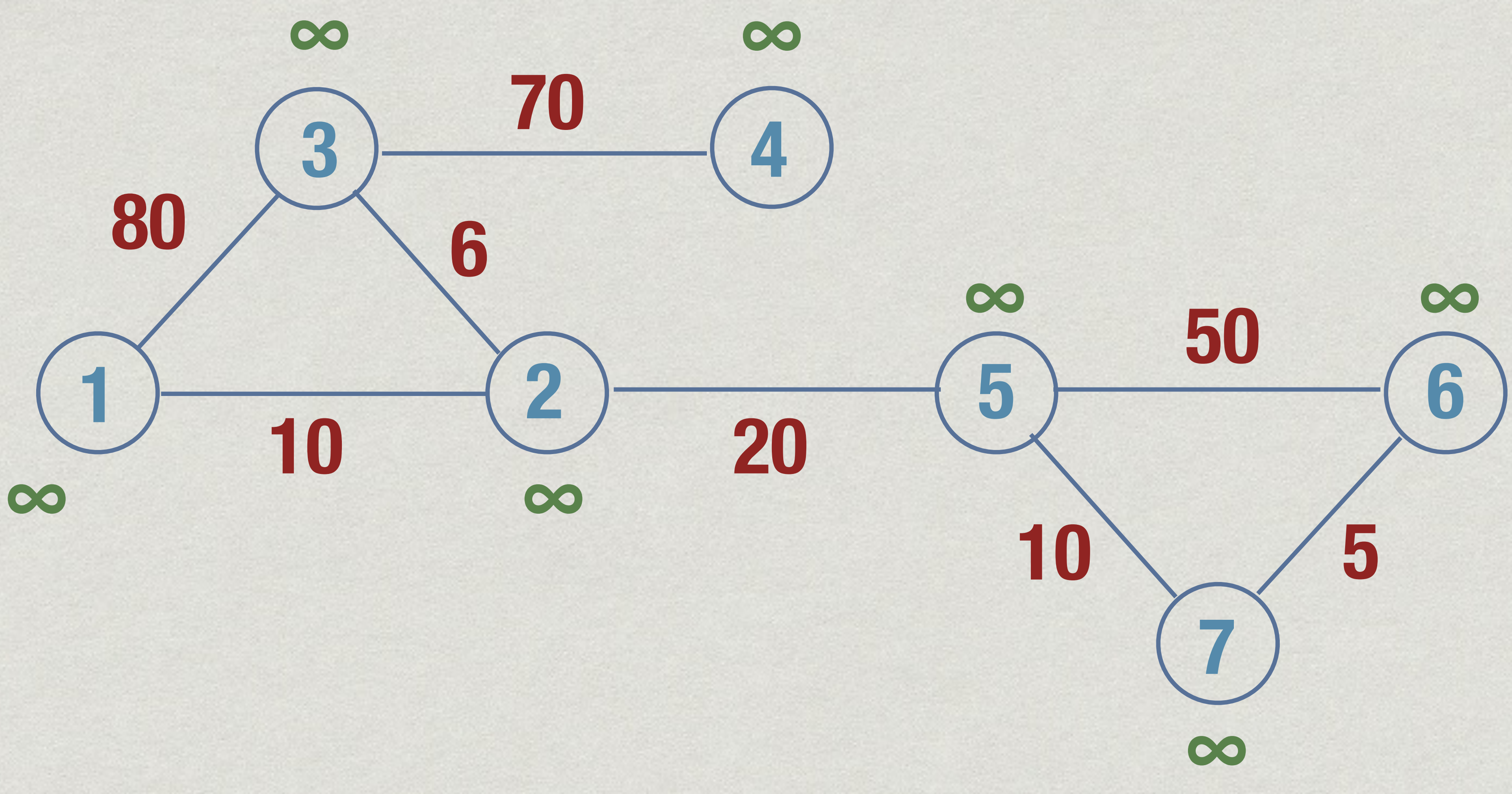
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



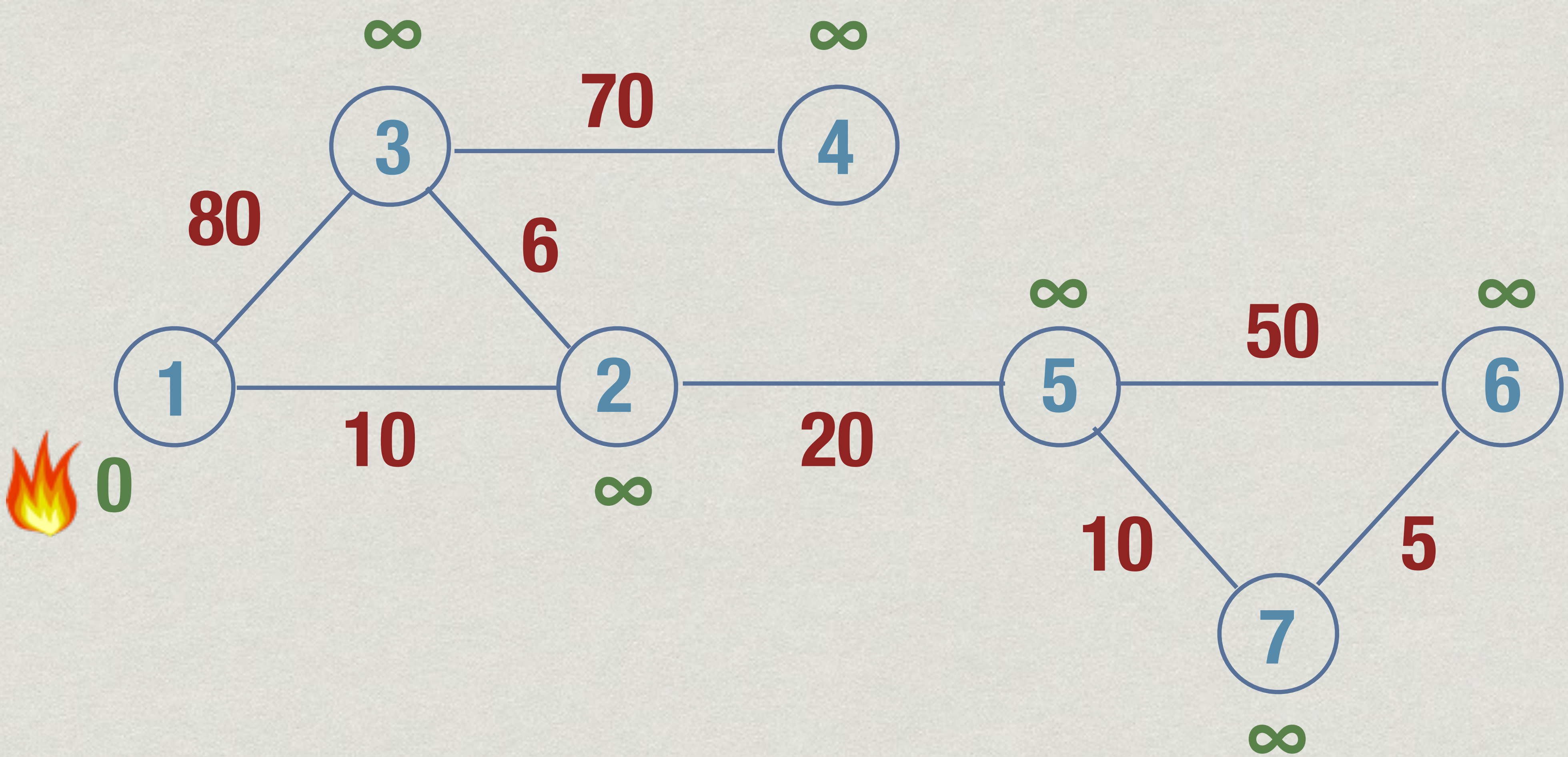
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



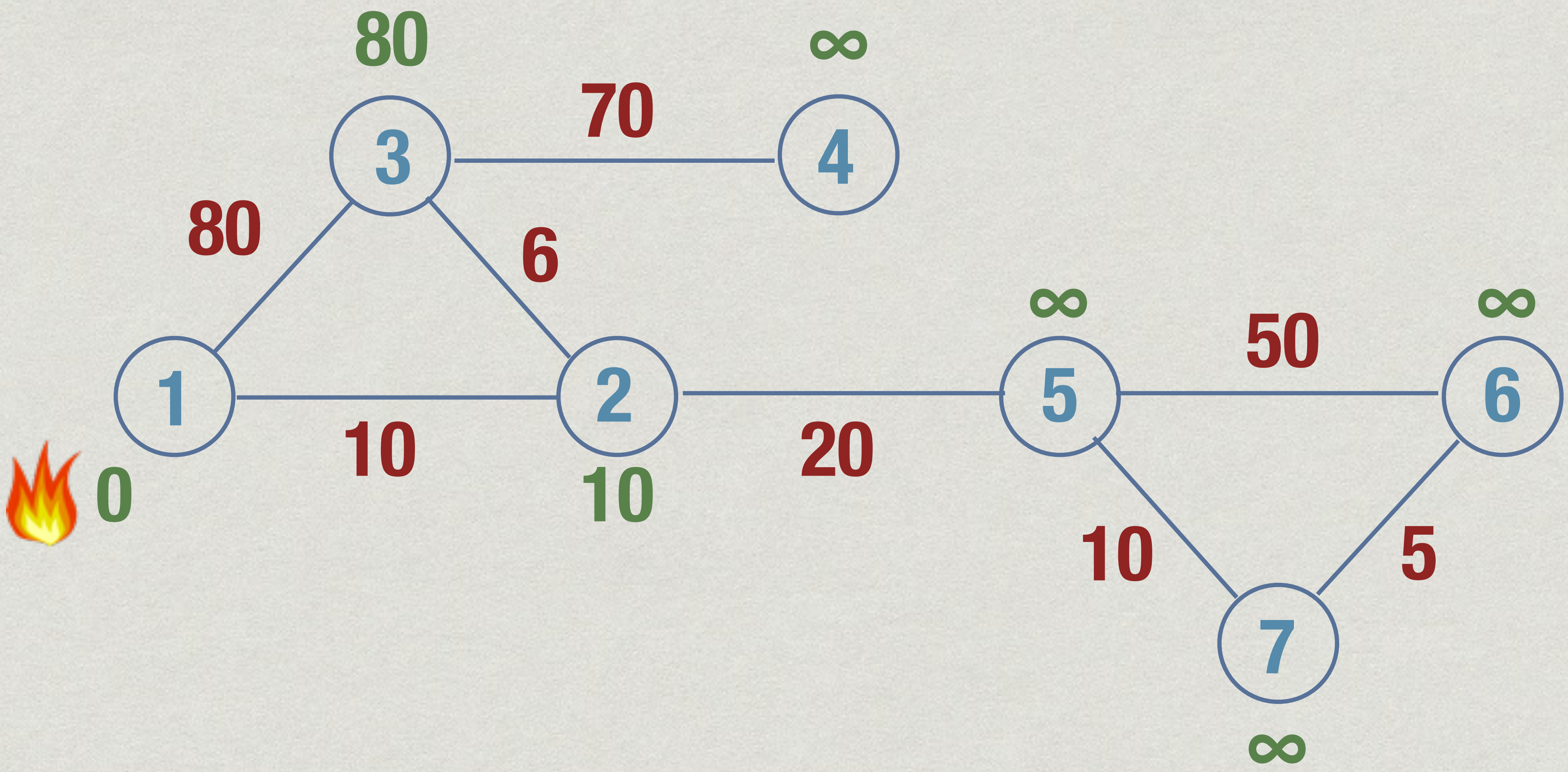
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



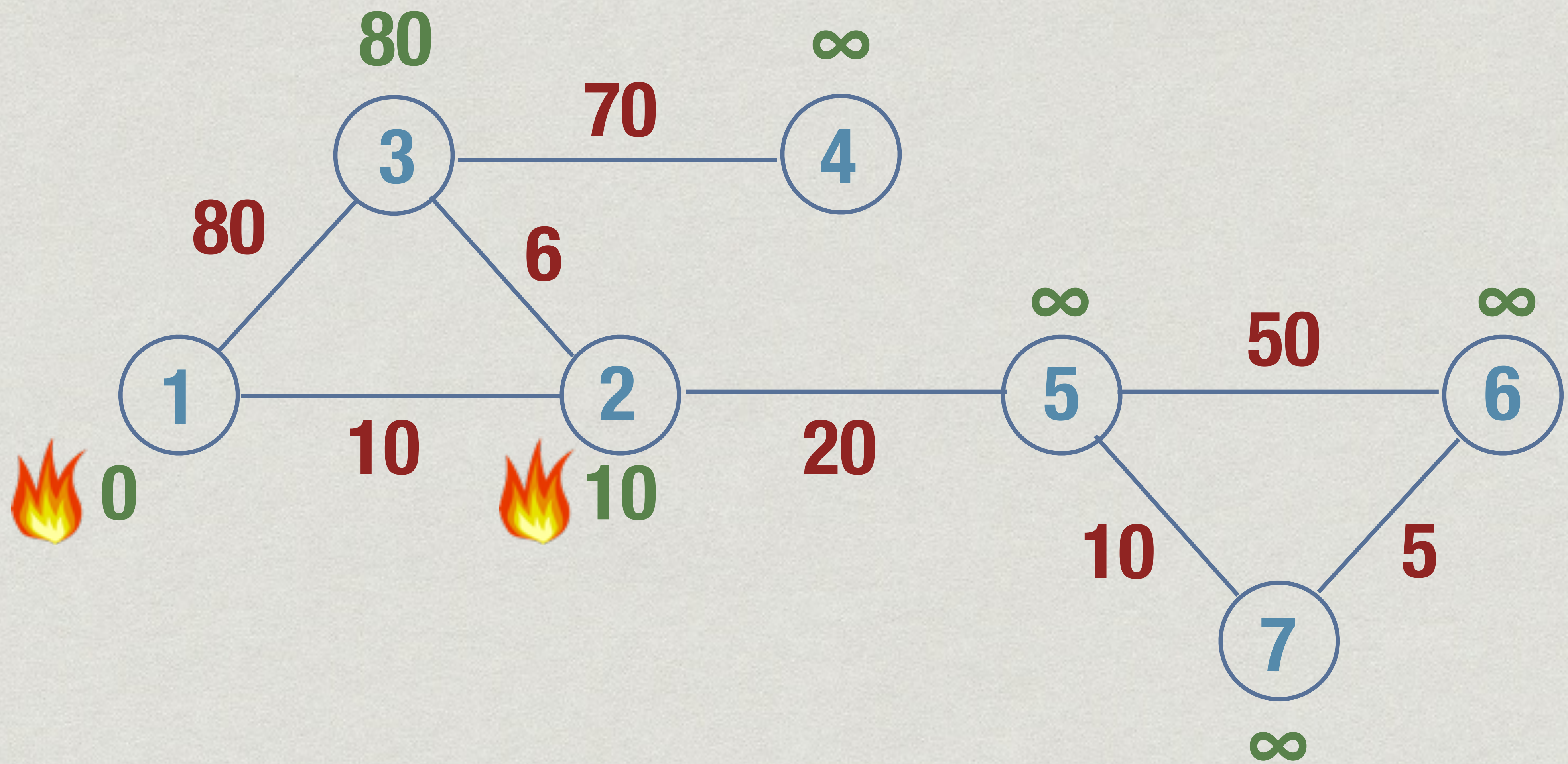
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



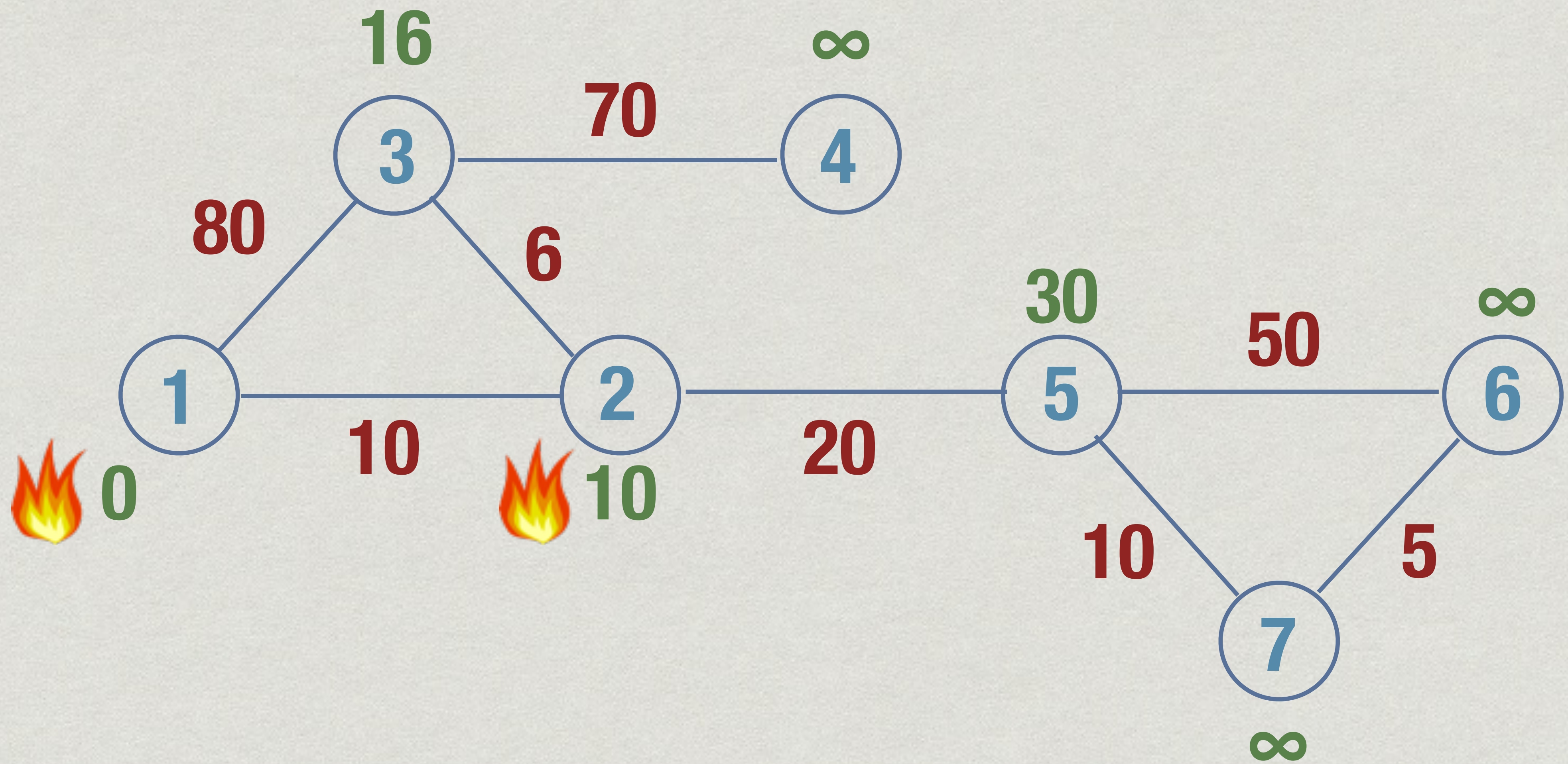
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



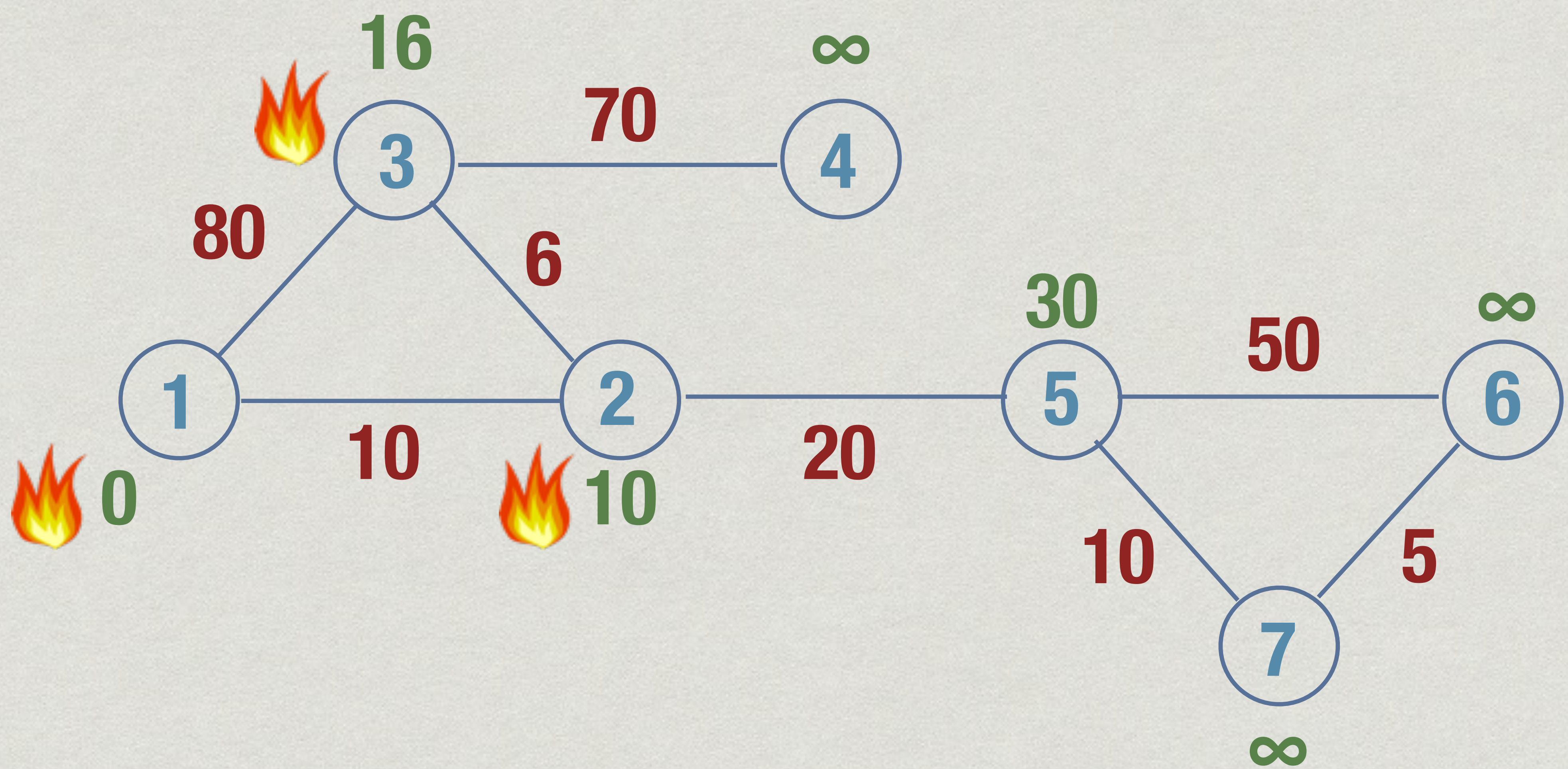
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



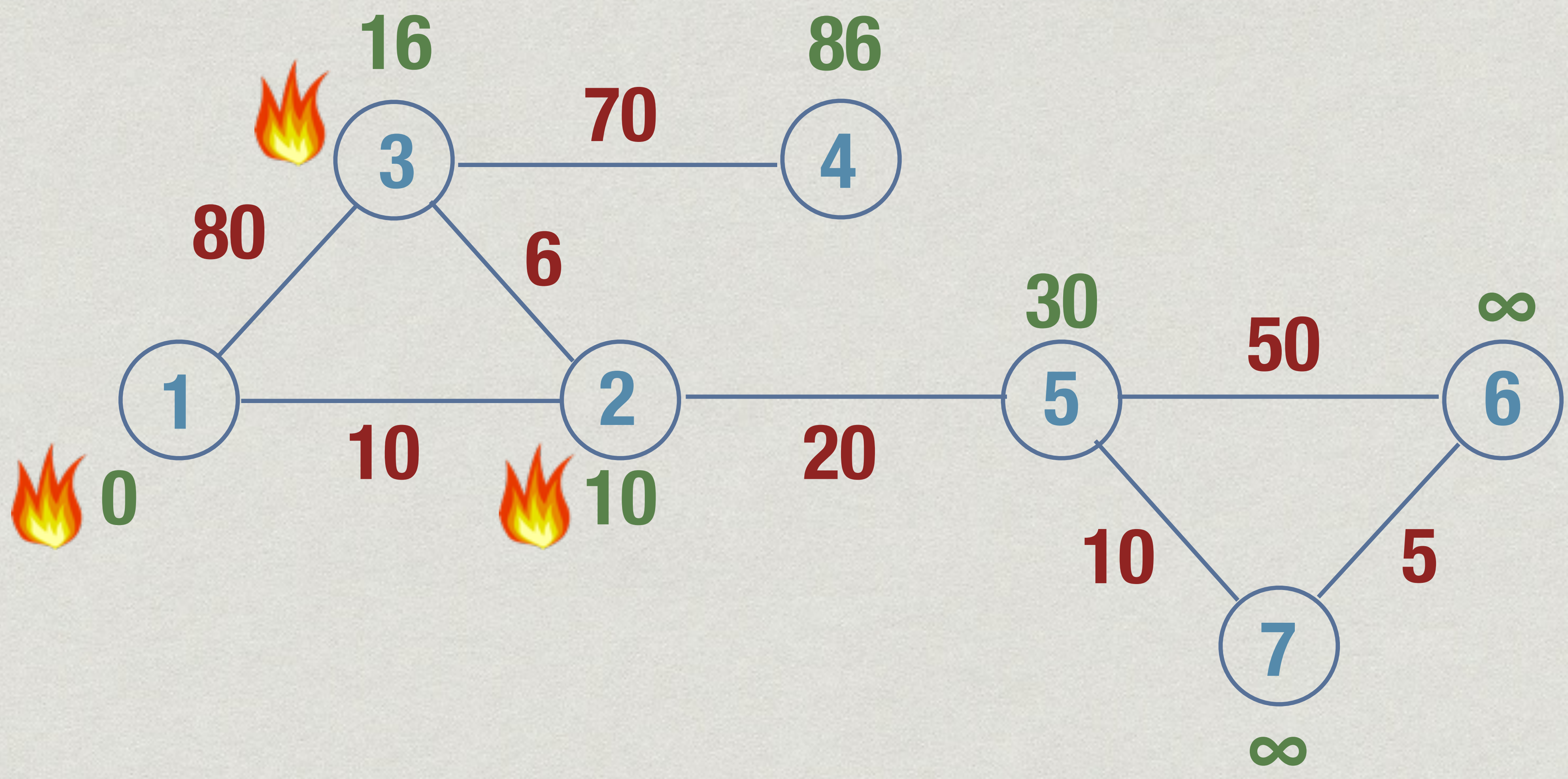
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



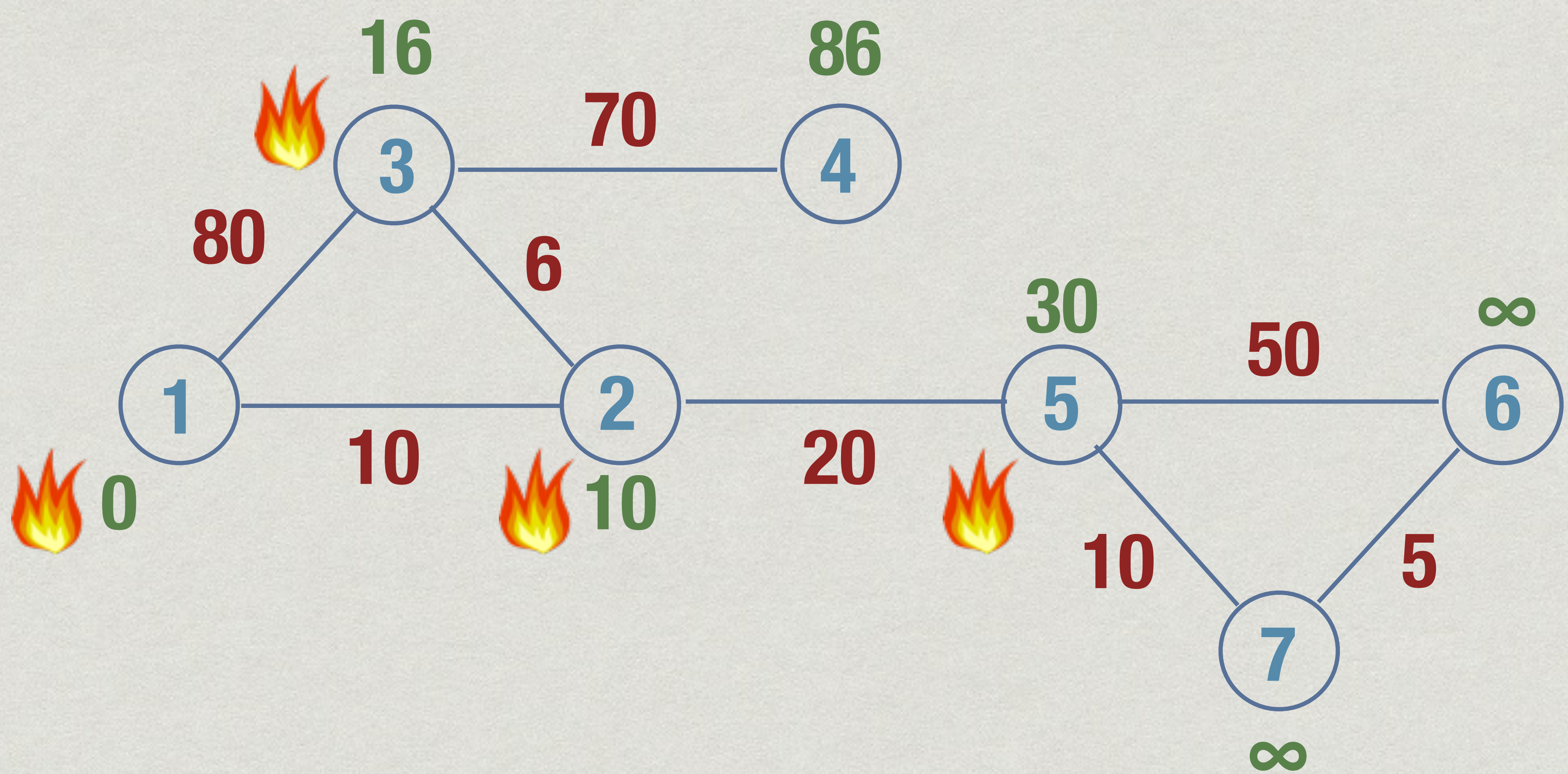
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



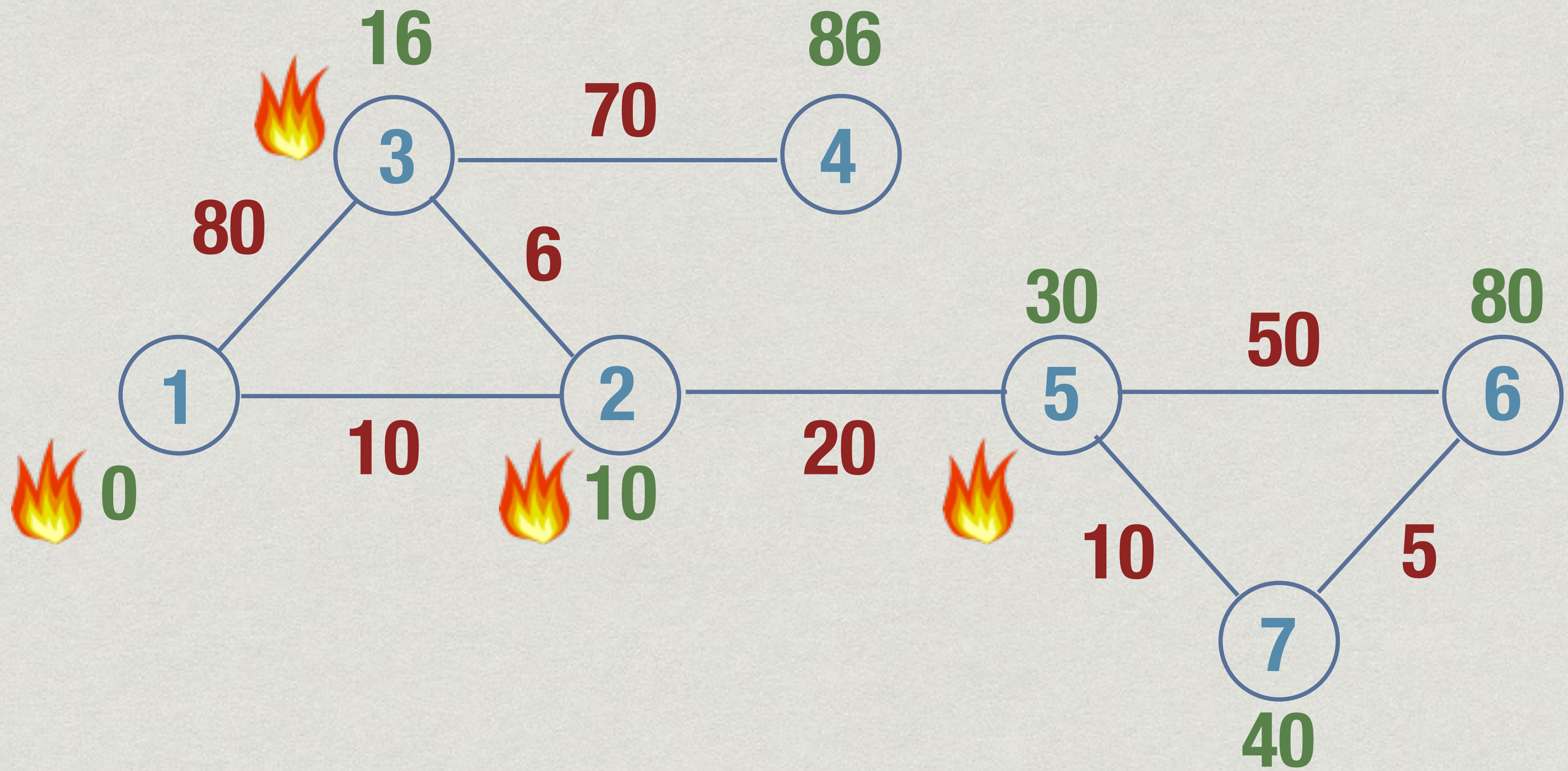
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



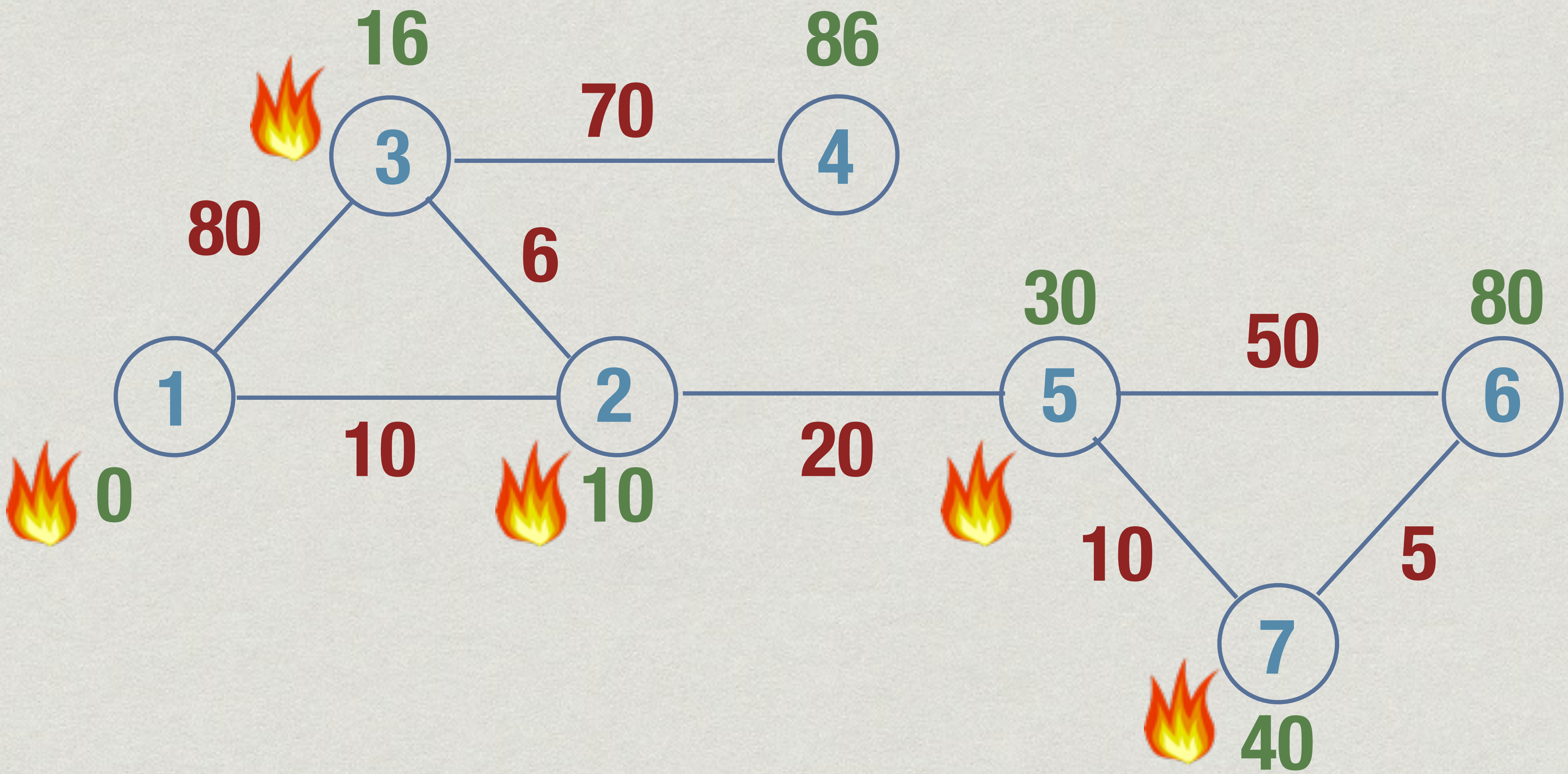
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



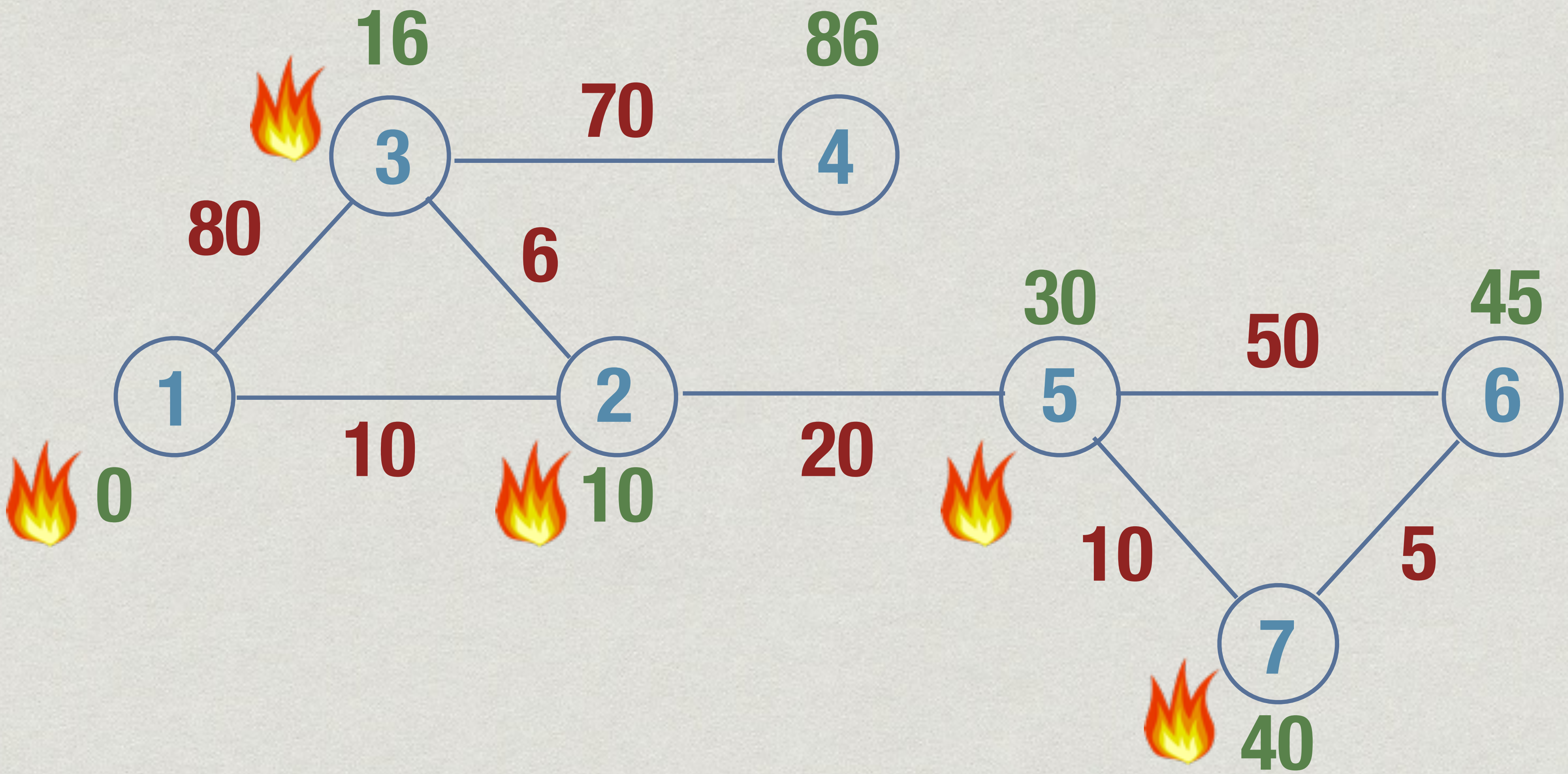
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



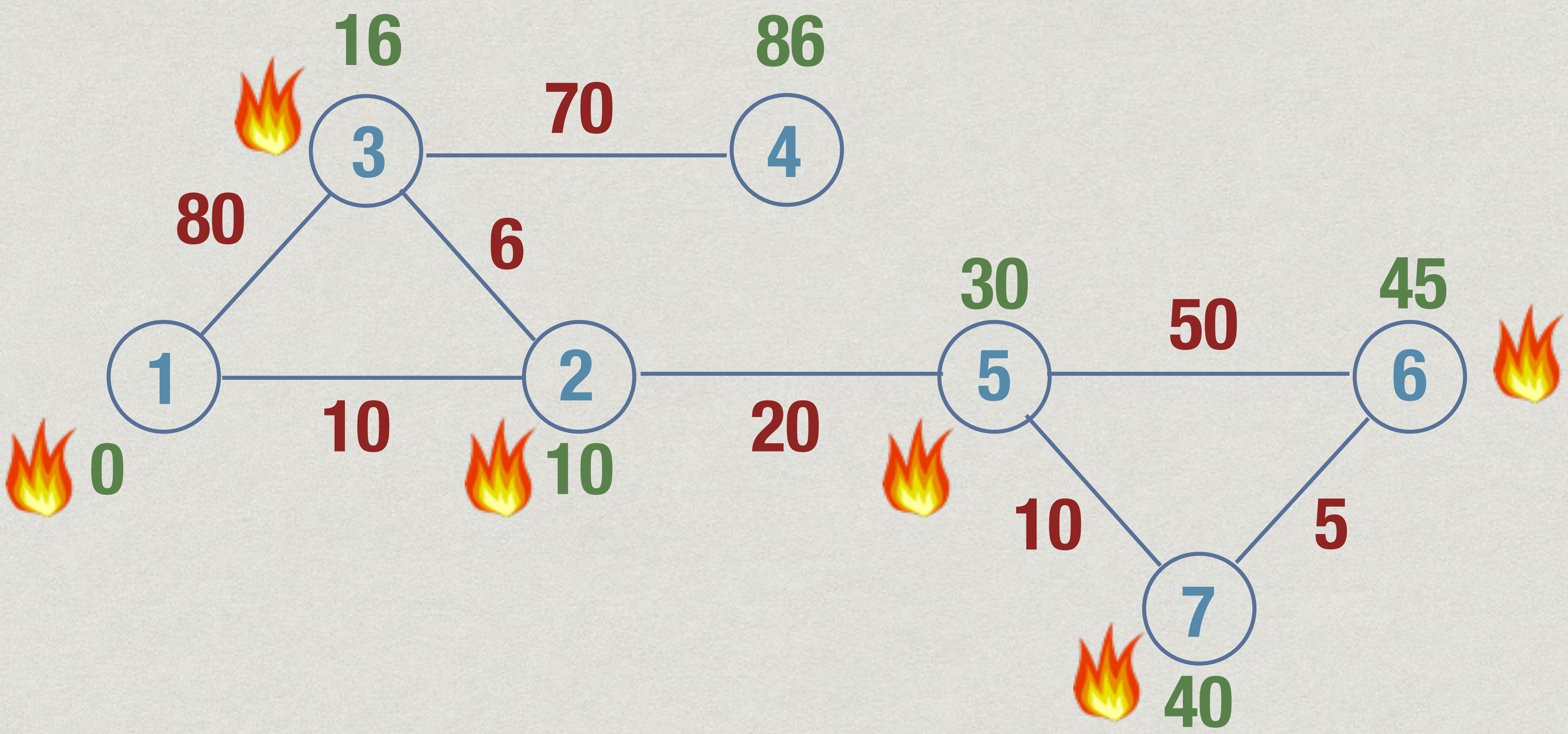
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



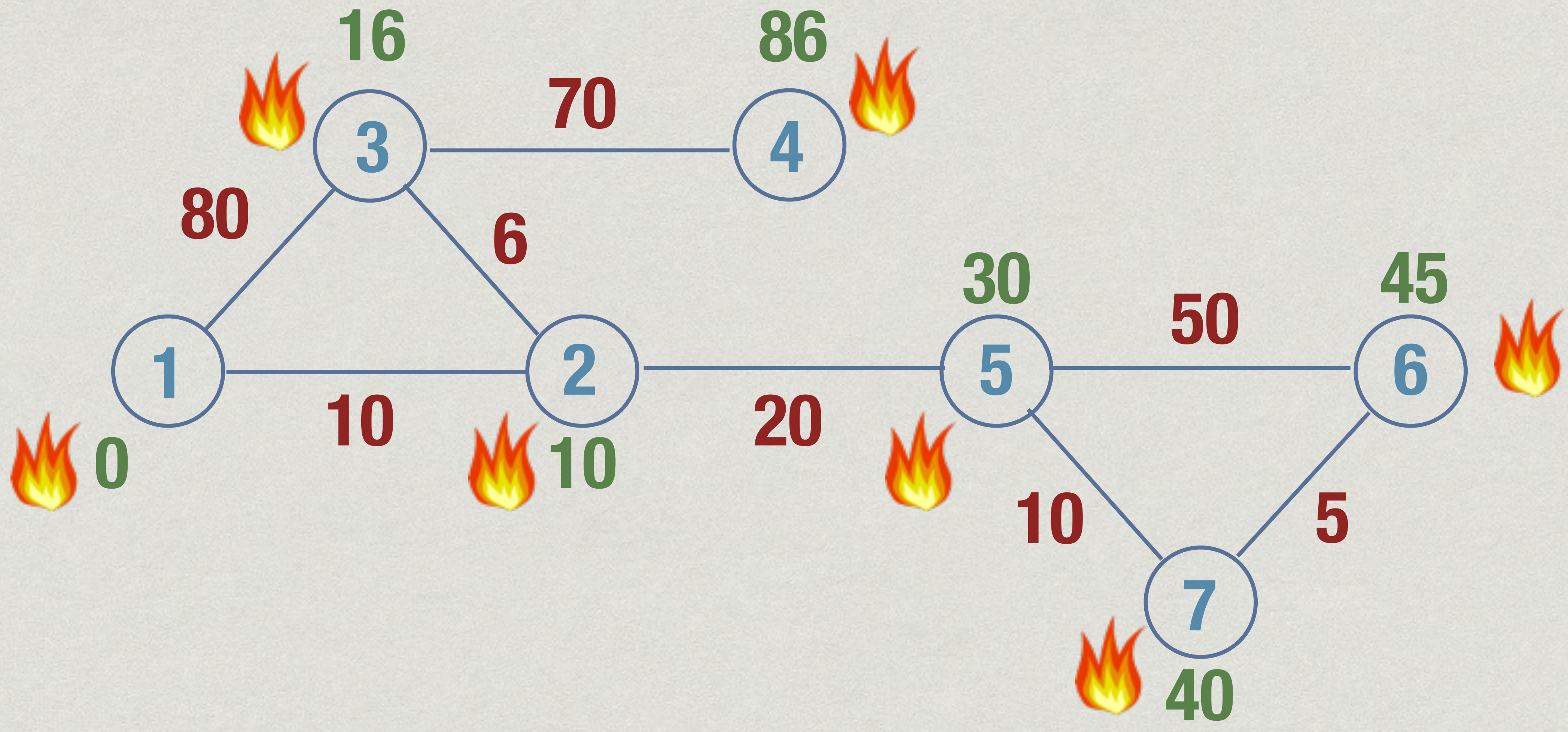
# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



# Single source shortest paths

- Compute expected time to burn of each vertex
- Update this each time a new vertex burns



# Algorithmically

# Algorithmically

- Maintain two arrays
  - `BurntVertices[ ]`, initially `False` for all i
  - `ExpectedBurnTime[ ]`, initially ∞ for all i
    - For ∞, use sum of all edge weights + 1

# Algorithmically

- Maintain two arrays
  - `BurntVertices[ ]`, initially `False` for all i
  - `ExpectedBurnTime[ ]`, initially ∞ for all i
    - For ∞, use sum of all edge weights + 1
- Set `ExpectedBurnTime[1] = 0`

# Algorithmically

- Maintain two arrays
  - `BurntVertices[ ]`, initially `False` for all i
  - `ExpectedBurnTime[ ]`, initially ∞ for all i
    - For ∞, use sum of all edge weights + 1
- Set `ExpectedBurnTime[1] = 0`
- Repeat, until all vertices are burnt
  - Find `j` with minimum `ExpectedBurnTime`
  - Set `BurntVertices[j] = True`
  - Recompute `ExpectedBurnTime[k]` for each neighbour `k` of `j`

# Dijkstra’s algorithm

```
function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    BV[i] = False; EBT[i] = infinity

  EBT[s] = 0

  for i = 1 to n
    Choose u such that BV[u] == False
                        and EBT[u] is minimum
    BV[u] = True
    for each edge (u,v) with BV[v] == False
      if EBT[v] > EBT[u] + weight(u,v)
        EBT[v] = EBT[u] + weight(u,v)
```

# Dijkstra's algorithm

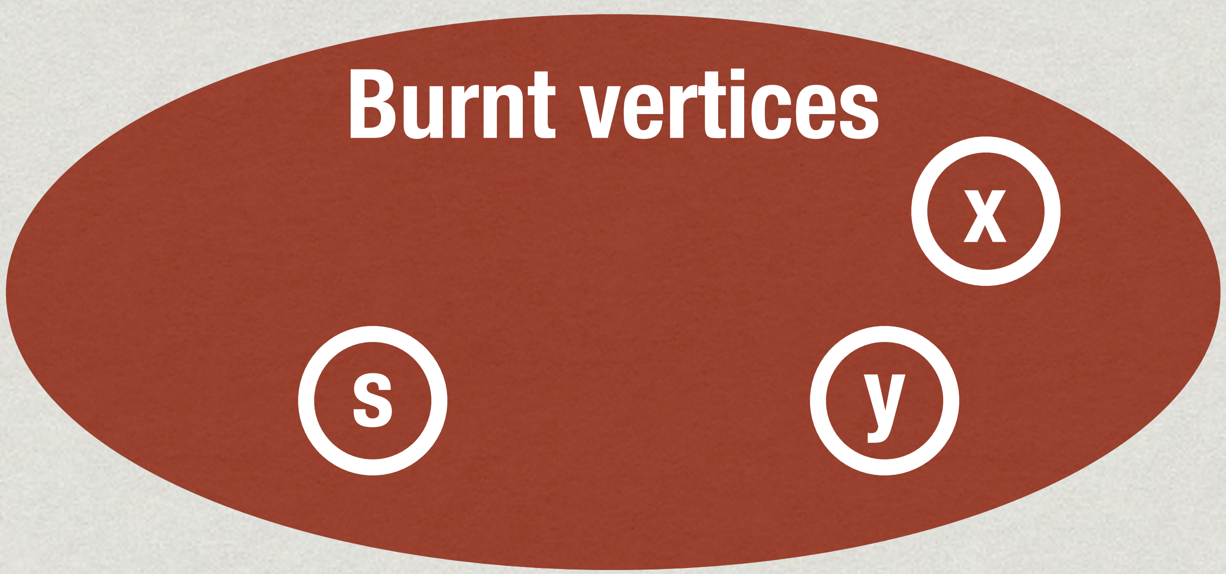
```
function ShortestPaths(s){ // assume source is s
  for i = 1 to n
    Visited[i] = False; Distance[i] = infinity

  Distance[s] = 0

  for i = 1 to n
    Choose u such that Visited[u] == False
                    and Distance[u] is minimum
    Visited[u] = True
    for each edge (u,v) with Visited[v] == False
      if Distance[v] > Distance[u] + weight(u,v)
        Distance[v] = Distance[u] + weight(u,v)
```

# Correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt

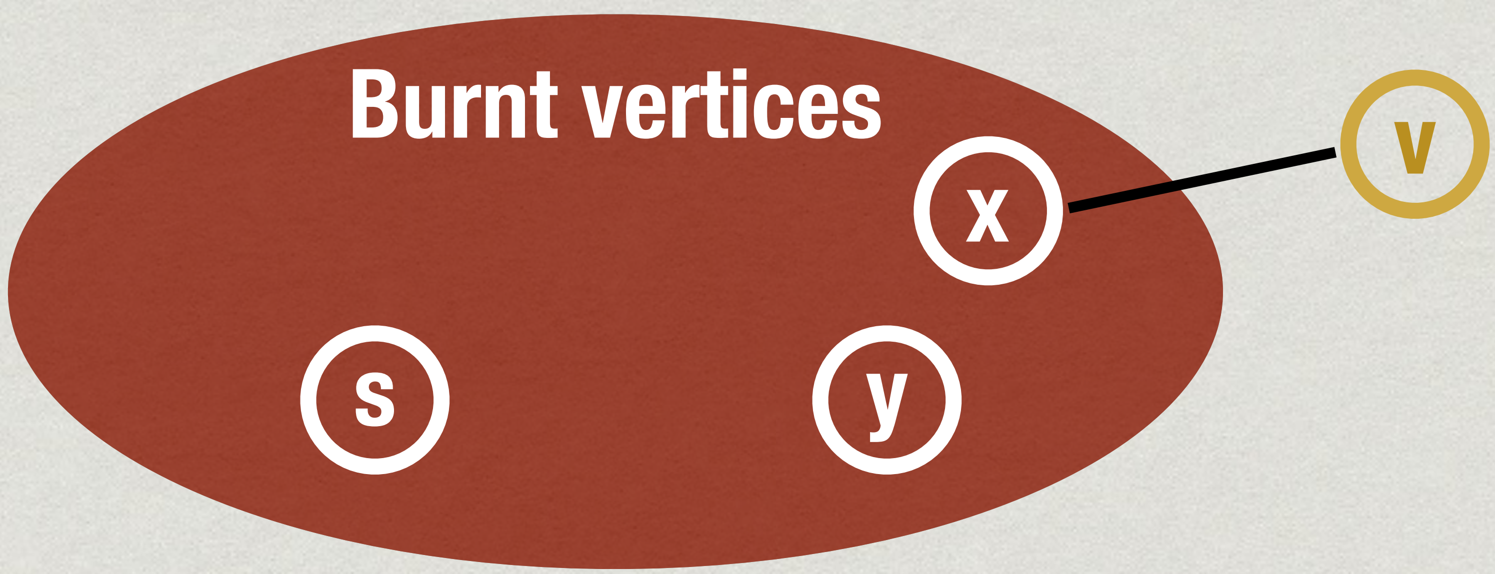
# Correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt



# Correctness

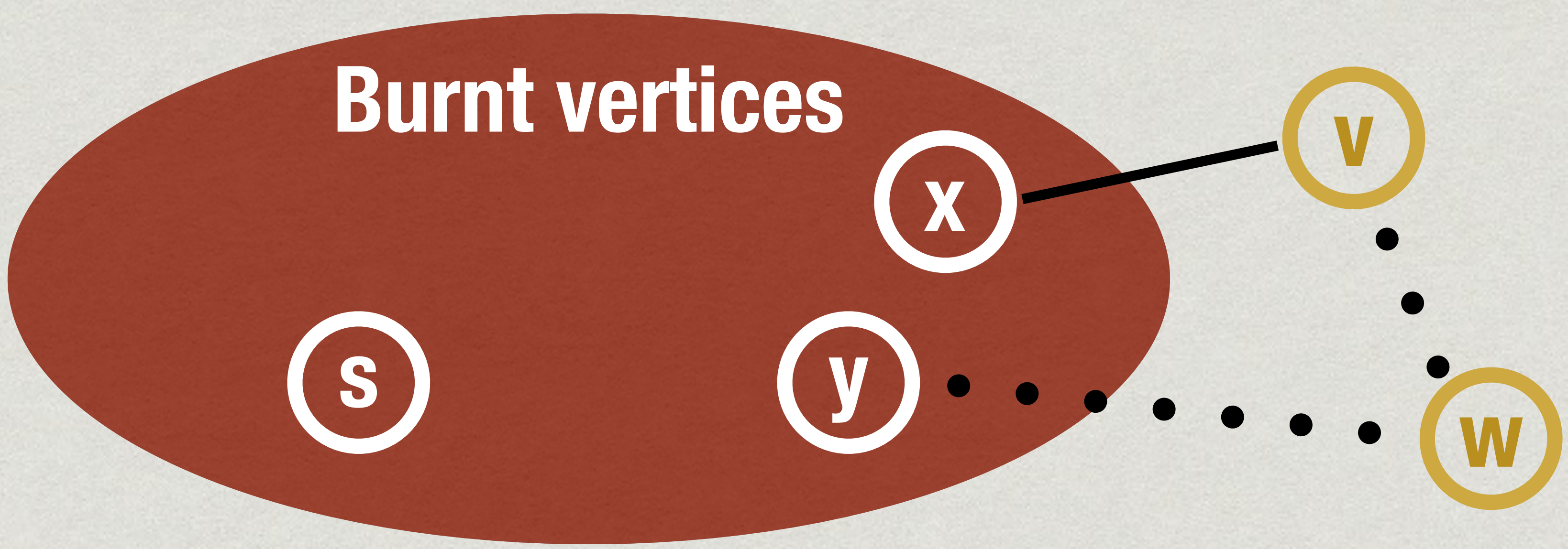
- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt




- Next vertex to burn is v, via x

# Correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have identified shortest paths to all vertices already burnt




- Next vertex to burn is v, via x
- Cannot later find a shorter path from y to w to v

# Complexity

# Complexity

- Outer loop runs n times

# Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex

# Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex
  - O(n) scan to find minimum burn time vertex

# Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex
  - O(n) scan to find minimum burn time vertex
- Each time we burn a vertex v, we have to scan all its neighbours to update burn times

# Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex
  - O(n) scan to find minimum burn time vertex
- Each time we burn a vertex v, we have to scan all its neighbours to update burn times
  - O(n) scan of adjacency matrix to find all neighbours

# Complexity

- Outer loop runs n times
  - In each iteration, we burn one vertex
  - O(n) scan to find minimum burn time vertex
- Each time we burn a vertex v, we have to scan all its neighbours to update burn times
  - O(n) scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

# Complexity

# Complexity

- Does adjacency list help?

# Complexity

- Does adjacency list help?
    - Scan neighbours to update burn times

# Complexity

- Does adjacency list help?
  - Scan neighbours to update burn times
  - O(m) across all iterations

# Complexity

- Does adjacency list help?
  - Scan neighbours to update burn times
  - $O(m)$ across all iterations
- However, identifying minimum burn time vertex still takes $O(n)$ in each iteration

# Complexity

- Does adjacency list help?
  - Scan neighbours to update burn times
  - O(m) across all iterations
- However, identifying minimum burn time vertex still takes O(n) in each iteration
- Still $O(n^2)$

# Complexity

# Complexity

* Can maintain `ExpectedBurnTime` in a more sophisticated data structure

# Complexity

- Can maintain `ExpectedBurnTime` in a more sophisticated data structure
    - Different types of trees (heaps, red-black trees) allow both of the following in O(log n) time

# Complexity

- Can maintain `ExpectedBurnTime` in a more sophisticated data structure
  - Different types of trees (heaps, red-black trees) allow both of the following in O(log n) time
    - find and delete minimum

# Complexity

- Can maintain `ExpectedBurnTime` in a more sophisticated data structure
  - Different types of trees (heaps, red-black trees) allow both of the following in O(log n) time
    - find and delete minimum
    - insert or update a value

# Complexity

# Complexity

* With such a tree

# Complexity

- With such a tree
  - Finding minimum burn time vertex takes O(log n)

# Complexity

- With such a tree
  - Finding minimum burn time vertex takes O(log n)
  - With adjacency list, updating burn times take O(log n) each, total O(m) edges

# Complexity

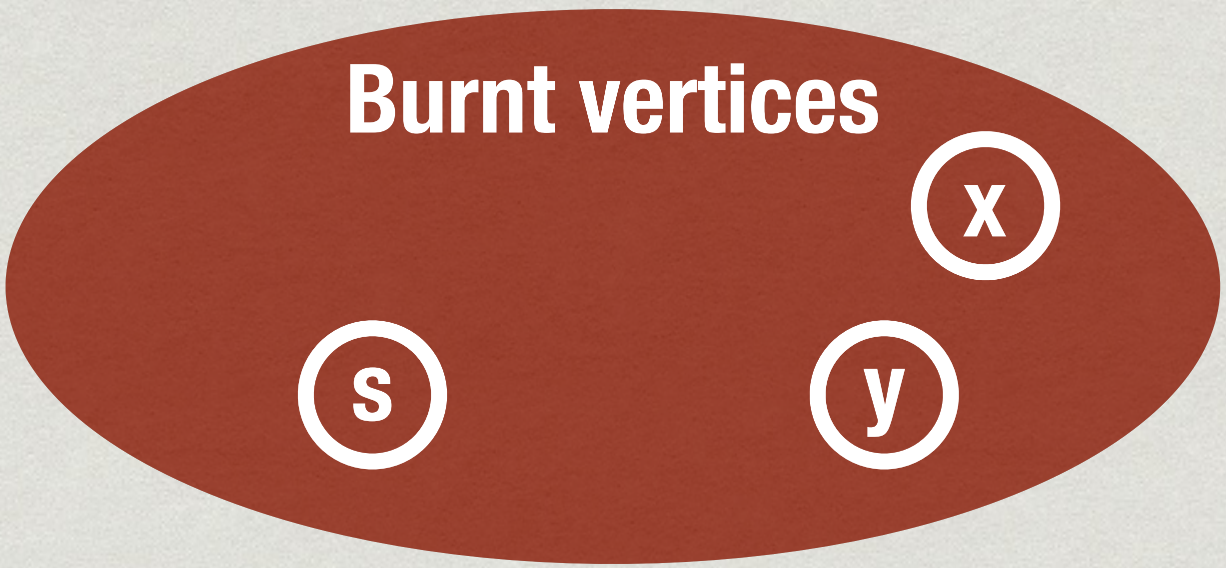
- With such a tree
  - Finding minimum burn time vertex takes O(log n)
  - With adjacency list, updating burn times take O(log n) each, total O(m) edges
- Overall O(n log n + m log n) = O((n+m) log n)

# Limitations

- What if edge weights can be negative?
- Our correctness argument is no longer valid

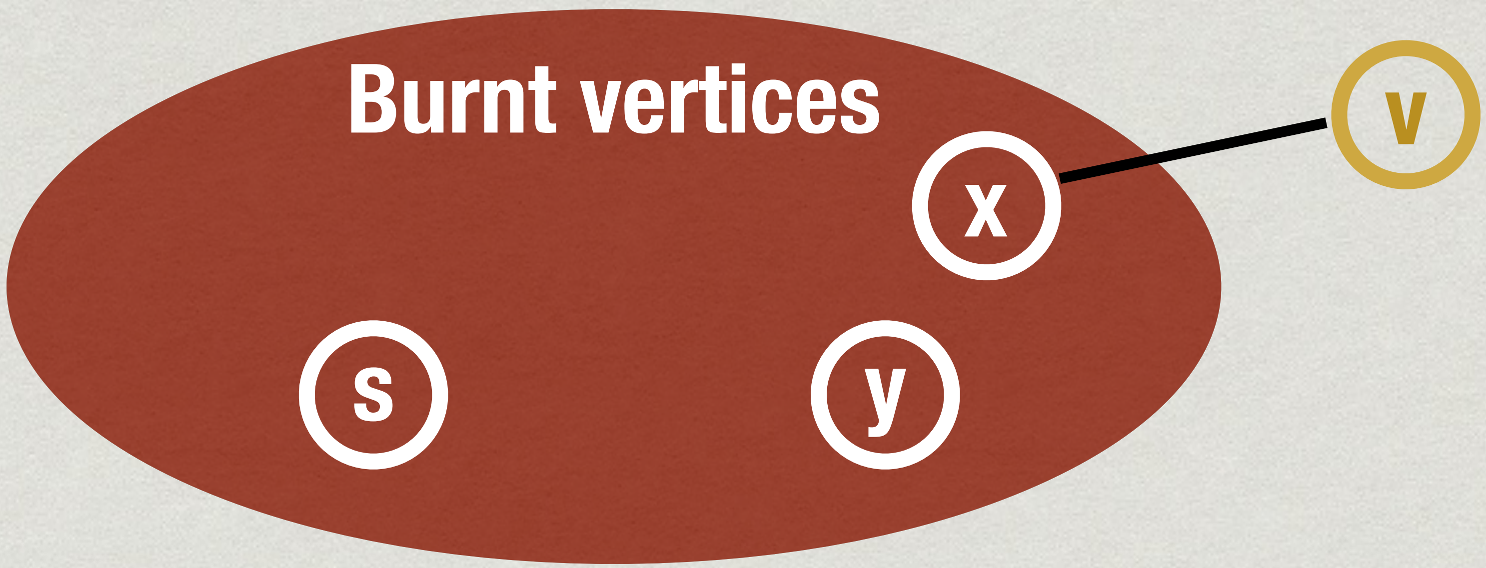
# Limitations

- What if edge weights can be negative?
- Our correctness argument is no longer valid



# Limitations

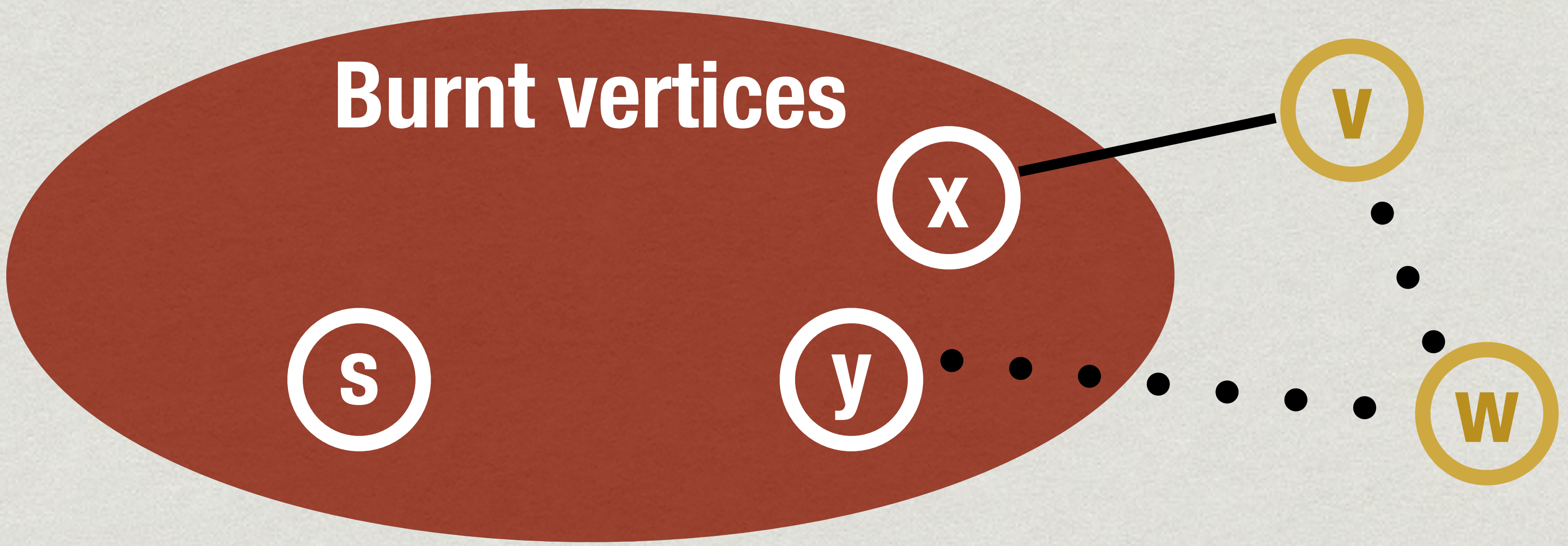
- What if edge weights can be negative?
- Our correctness argument is no longer valid




- Next vertex to burn is v, via x

# Limitations

- What if edge weights can be negative?
- Our correctness argument is no longer valid




- Next vertex to burn is v, via x
- Might find a shorter path later with negative weights from y to w to v

# Handling negative edges

# Handling negative edges

- **Negative cycle**: loop with a negative total weight

# Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles

# Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound

# Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, other algorithms exist

# Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, other algorithms exist
  - Bellman-Ford

# Handling negative edges

- **Negative cycle**: loop with a negative total weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost without a bound
- With negative edges, but no negative cycles, other algorithms exist
  - Bellman-Ford
  - Floyd-Warshall all pairs shortest path (will see later)

# Summary

- Dijkstra's algorithm solves the single source shortest path problem, assuming no negative weights
  - Simple implementation is $O(n^2)$
  - Using clever trees, reduce to $O((n+m) \log n)$
- With negative edges, but without negative cycles, need to use other strategies

# Greedy algorithms

# Greedy algorithms

- Algorithm makes a sequence of choices

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on “current best value”

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on "current best value"
    - Never go back and change a choice

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on “current best value”
  - Never go back and change a choice
- Dijkstra’s algorithm is greedy

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on “current best value”
  - Never go back and change a choice
- Dijkstra’s algorithm is greedy
  - Select vertex with minimum expected burn time

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on “current best value”
  - Never go back and change a choice
- Dijkstra’s algorithm is greedy
  - Select vertex with minimum expected burn time
- Need to prove that greedy strategy is optimal

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on "current best value"
    - Never go back and change a choice
- Dijkstra's algorithm is greedy
    - Select vertex with minimum expected burn time
- Need to prove that greedy strategy is optimal
- Most times, greedy approach fails

# Greedy algorithms

- Algorithm makes a sequence of choices
- Next choice is based on “current best value”
  - Never go back and change a choice
- Dijkstra’s algorithm is greedy
  - Select vertex with minimum expected burn time
- Need to prove that greedy strategy is optimal
- Most times, greedy approach fails
  - Current best choice may not be globally optimal