Graph Algorithms

Weighted Graphs

Weighted Graphs

- Recall the graph

Weighted Graphs

- Recall the graph
  - $G = (V, E)$
  - V: Set of vertices
  - E: Set of edges
    - E is a subset of pairs (v, v'): $E \subseteq V \times V$
    - Undirected graph: (v, v') and (v', v) are same edge
    - Directed graph:
      - (v, v') is an edge from v to v'
      - Does not guarantee that (v', v) is also an edge

Weighted Graphs …

- Adding edge weights

Weighted Graphs …

- Adding edge weights
  - Label each edge with a number - **cost**

Weighted Graphs …

- 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 location during peak hour traffic

Weighted Graphs …

- Weighted graph
  - $G = (V, E)$
  - **Weight function,** $w: E \rightarrow Reals$

Weighted Graphs …(Shortest paths)

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

Shortest path problems

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 center (single source) to addresses

Shortest path problems

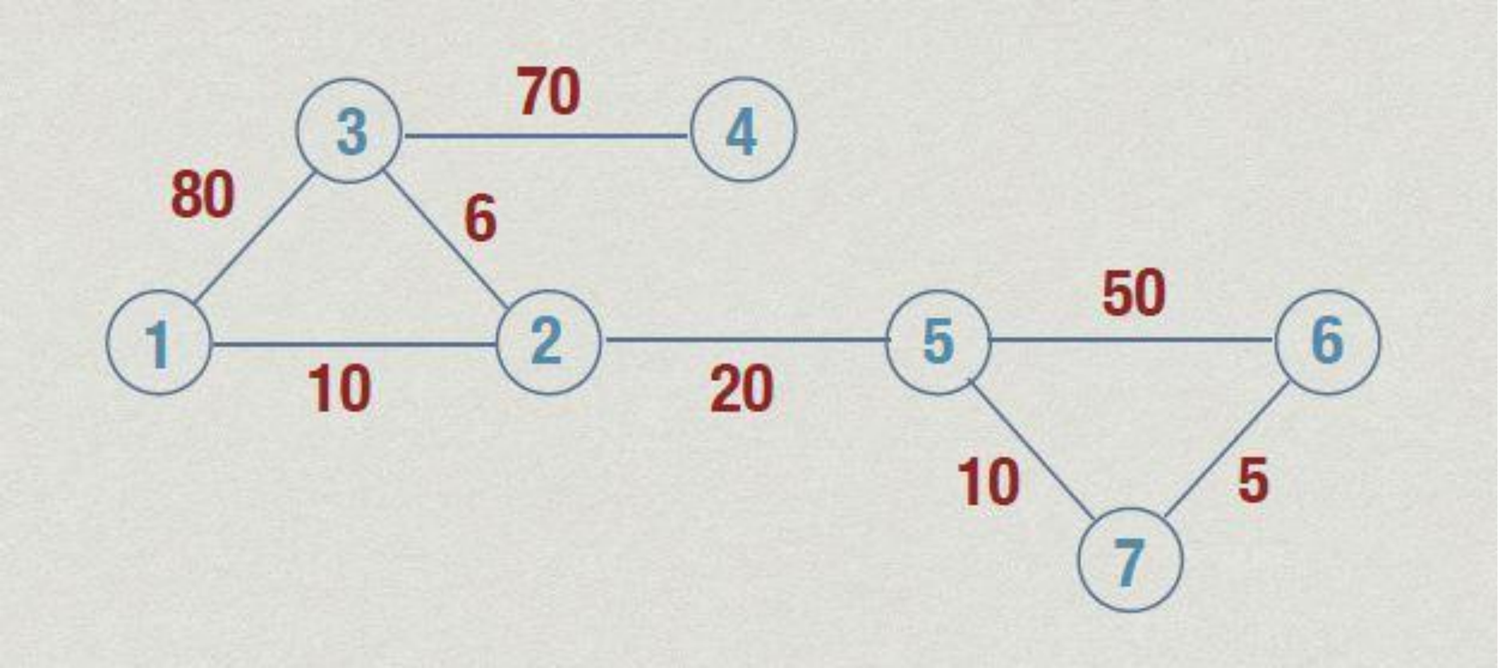
- **All pairs**
  - Find shortest paths between every pair of vertices i and j

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

Single source shortest paths

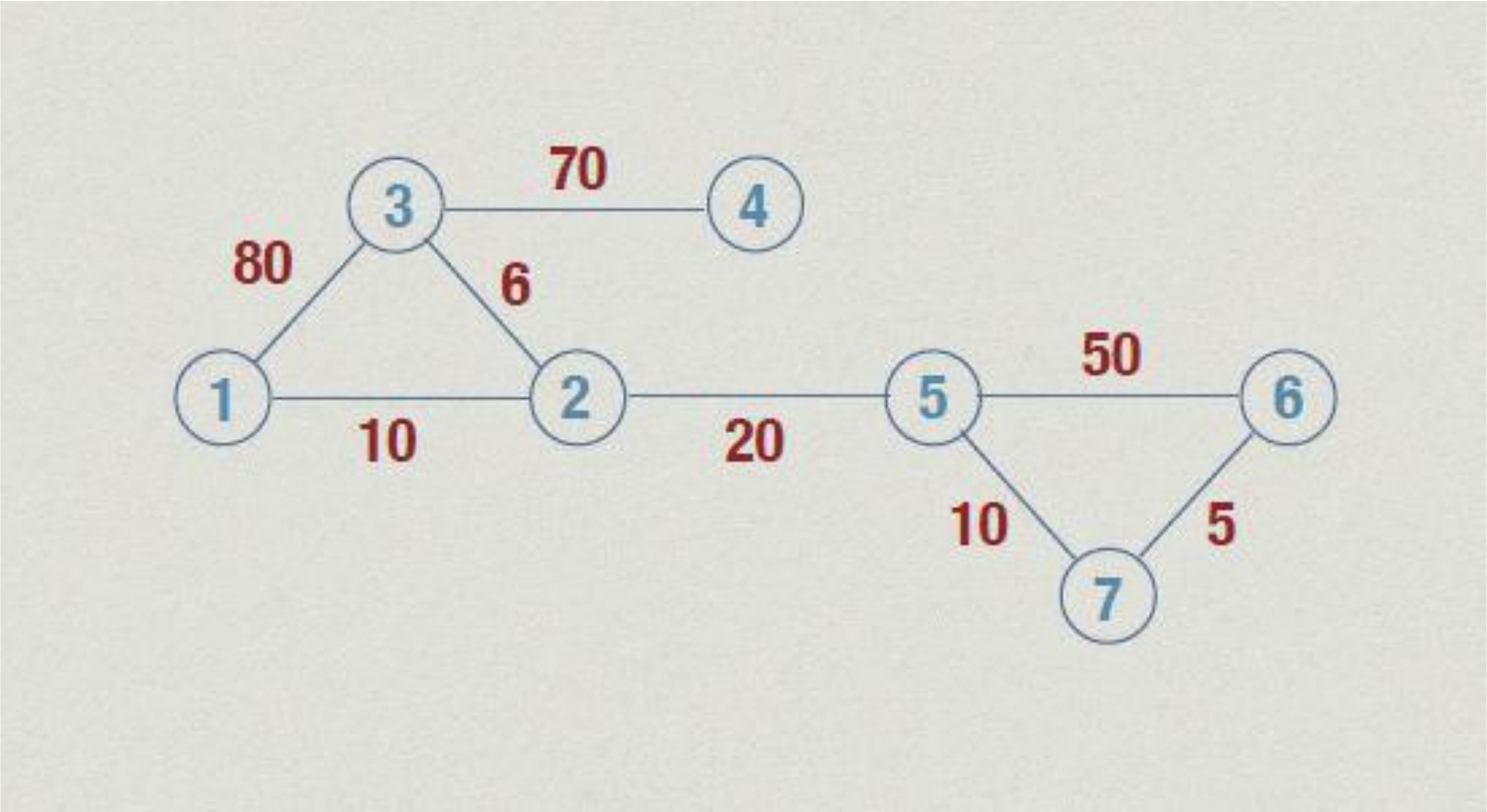
- For instance, shortest path from 1 to 2,3, … , 7



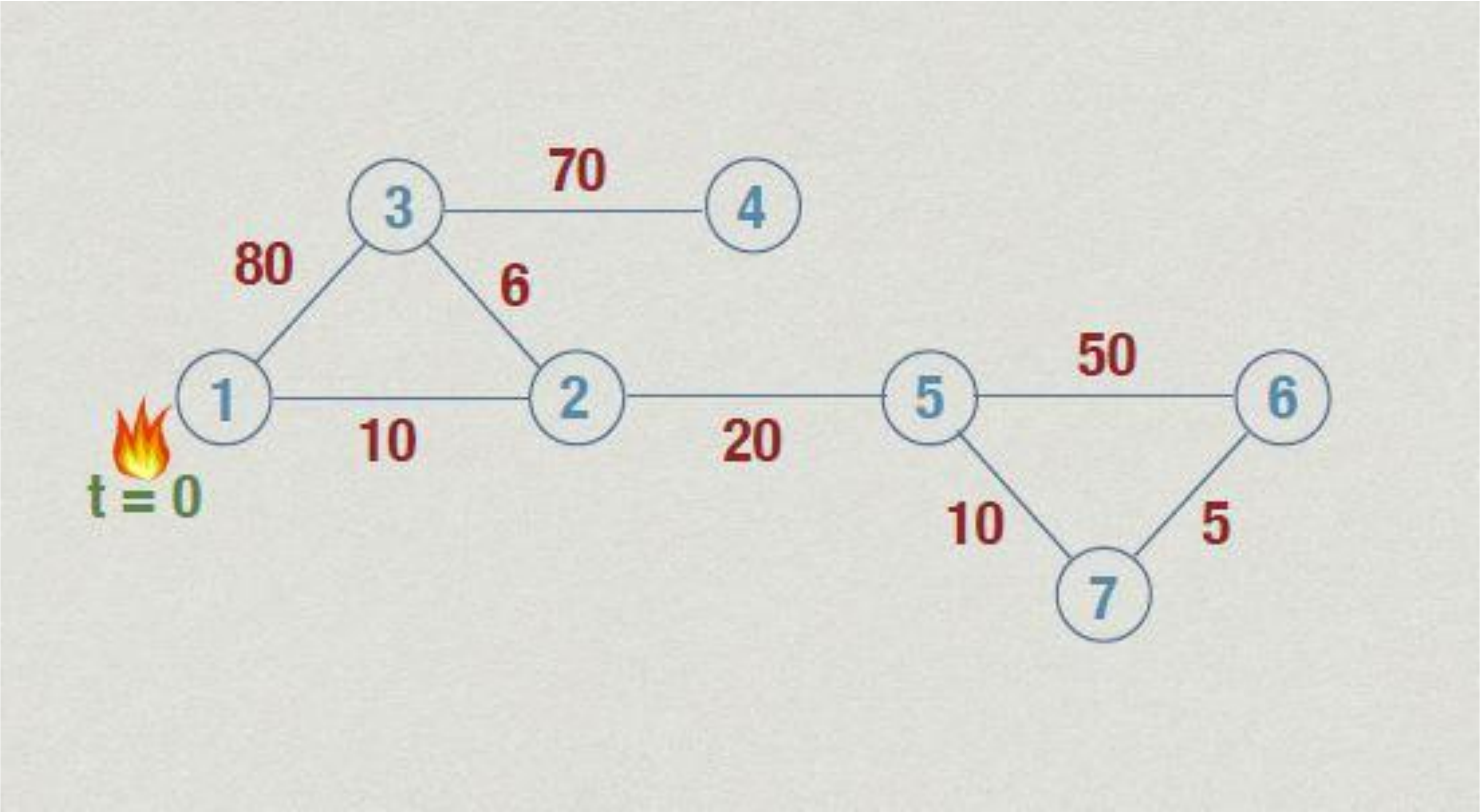
Single source shortest paths …

- 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
- …

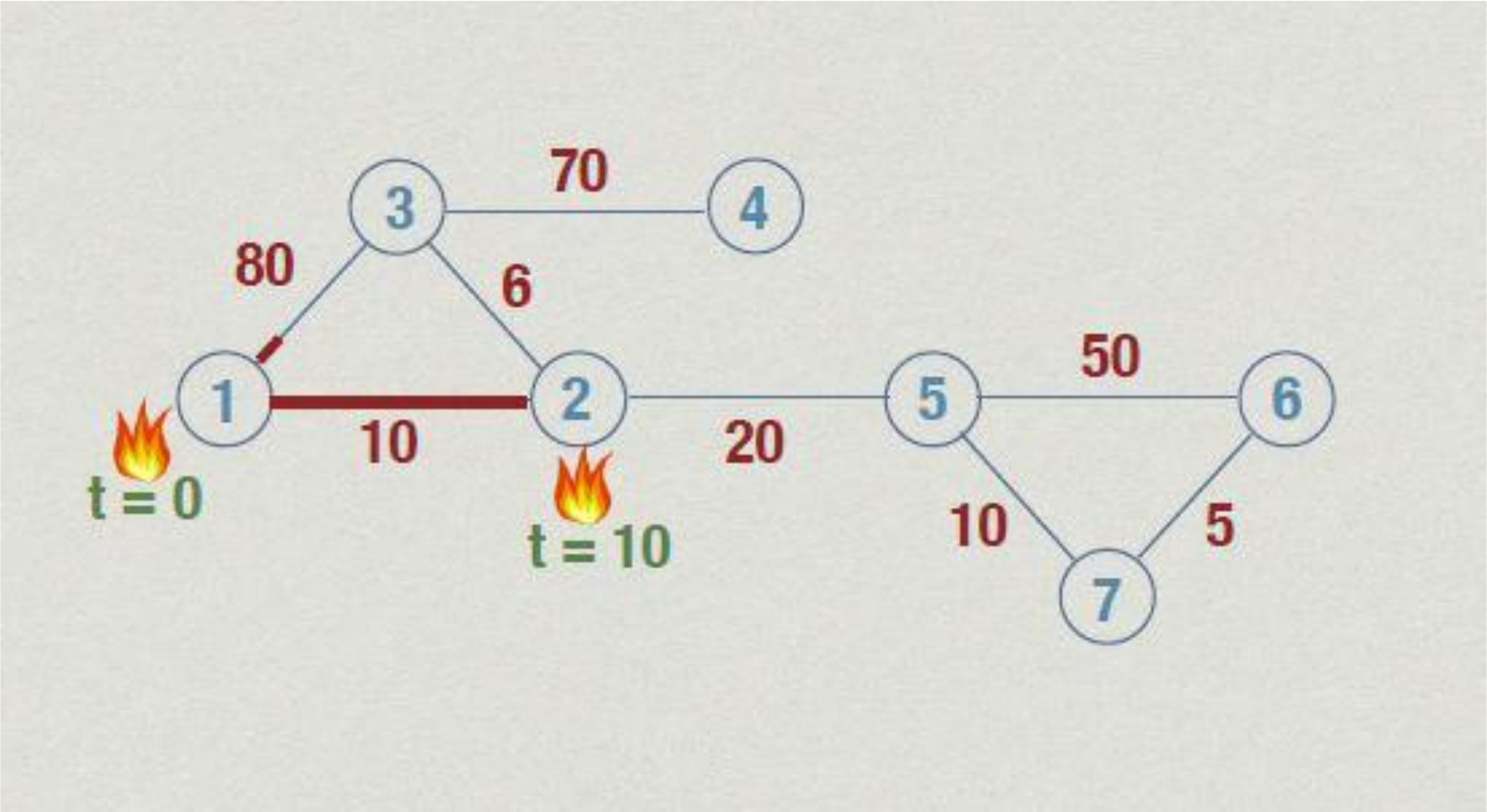
Single source shortest paths …



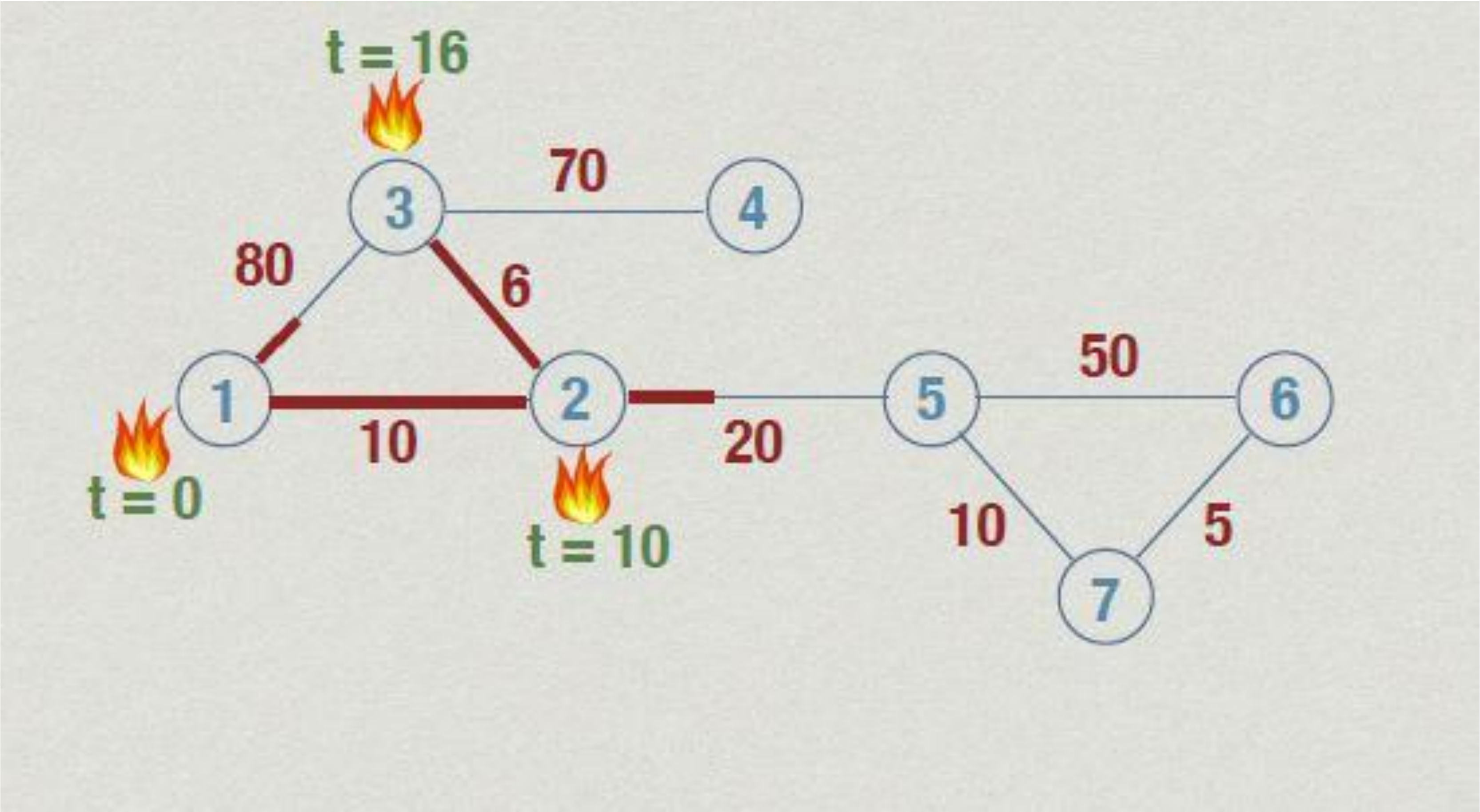
Single source shortest paths …



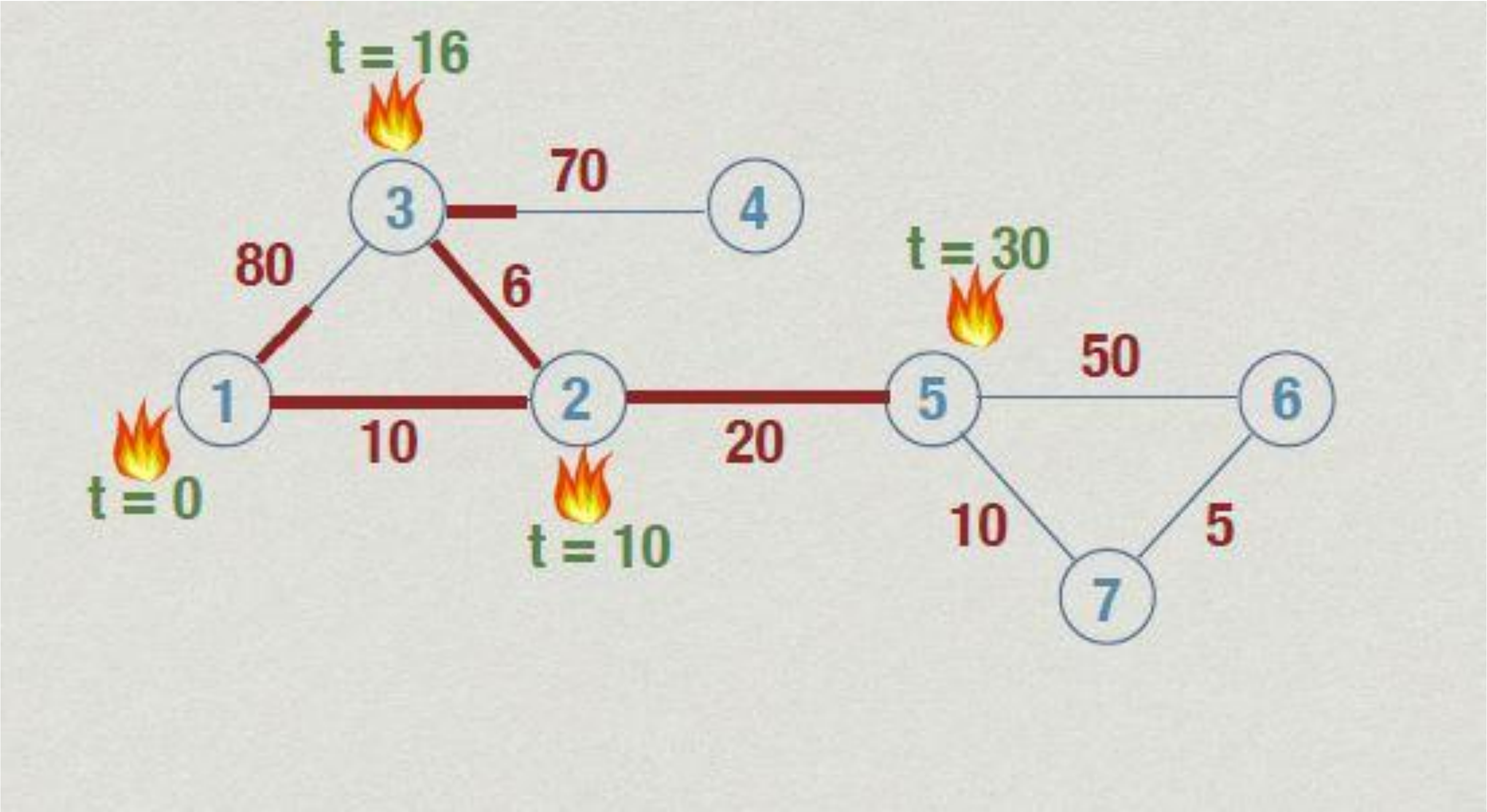
Single source shortest paths …



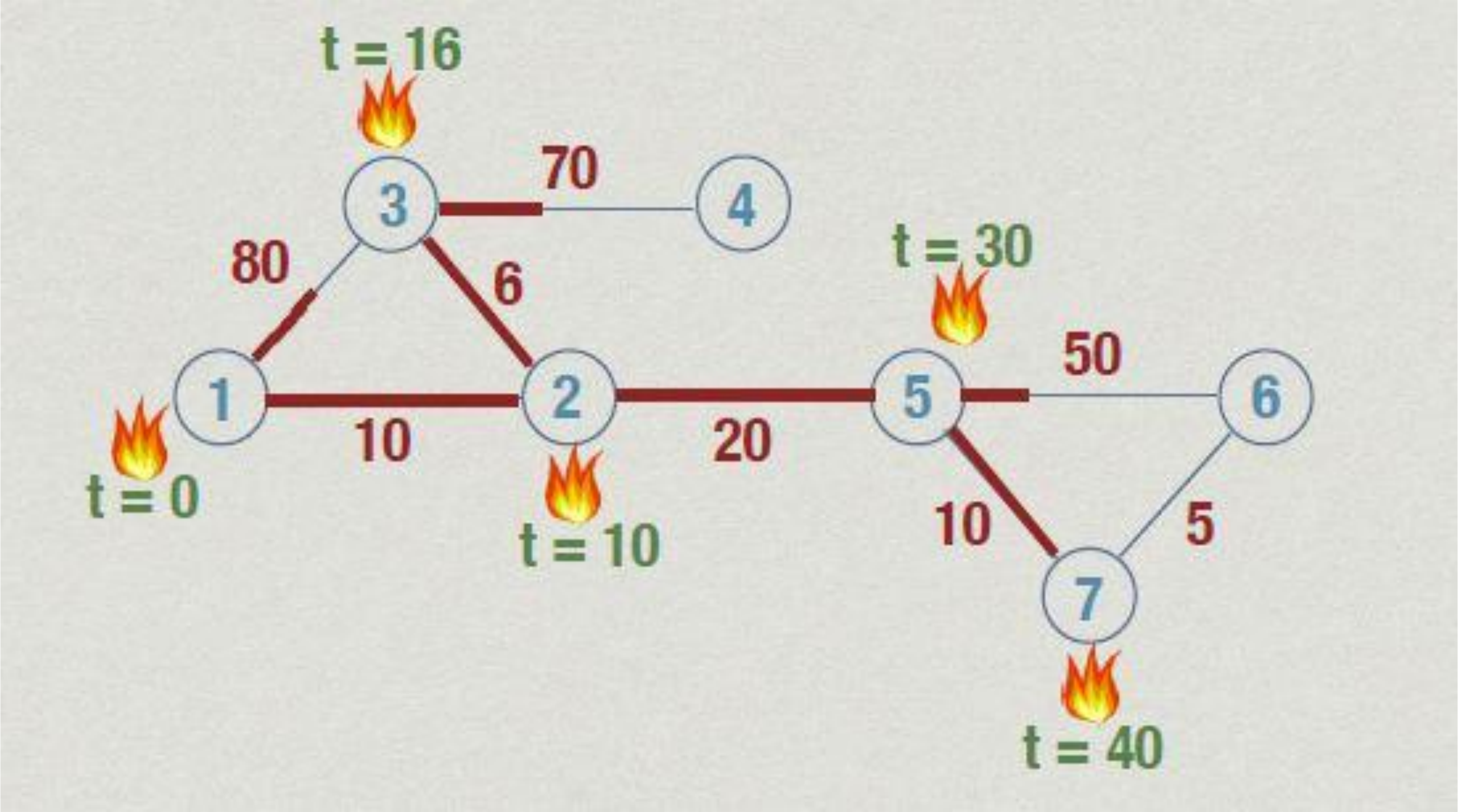
Single source shortest paths …



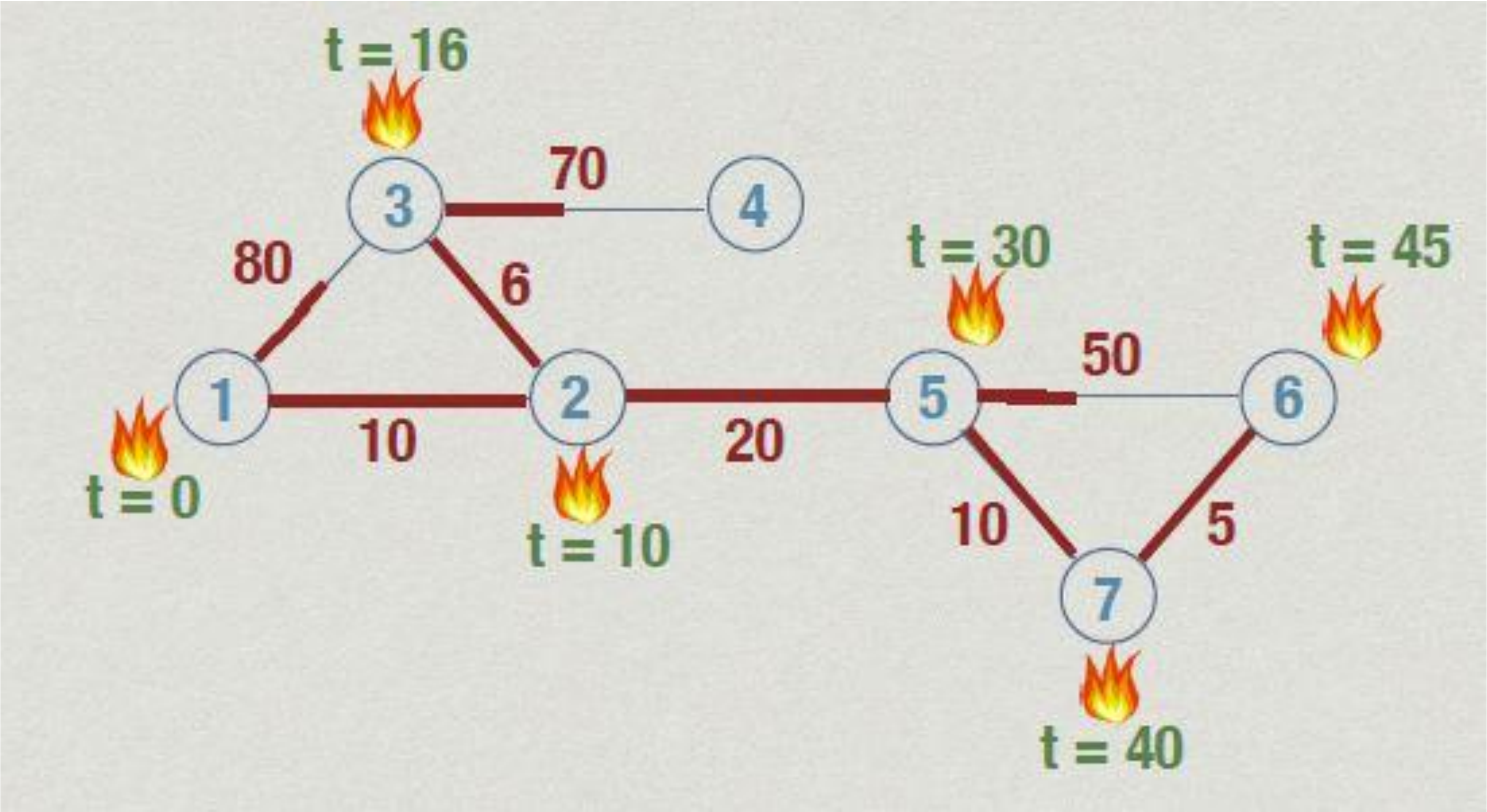
Single source shortest paths …



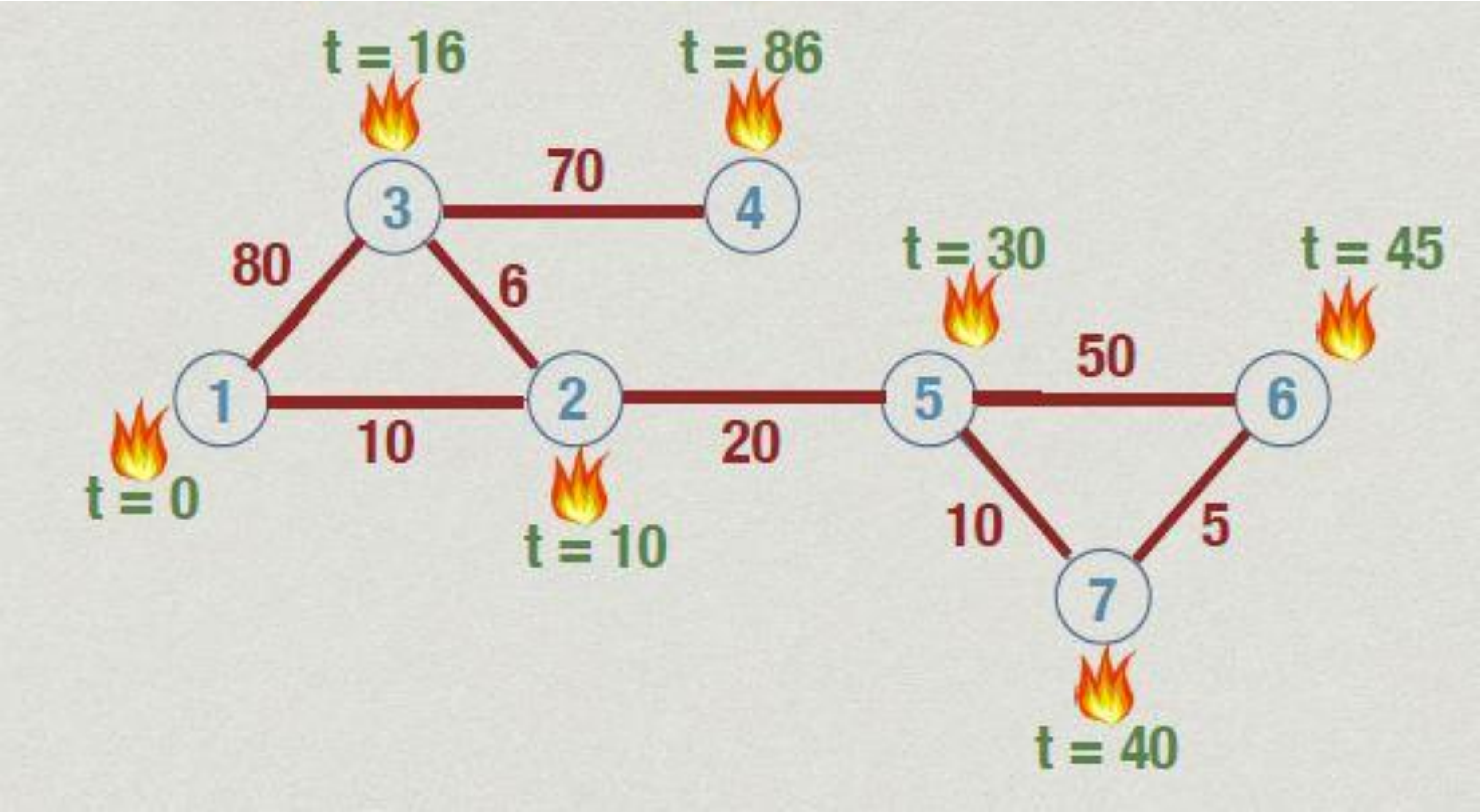
Single source shortest paths …



Single source shortest paths …



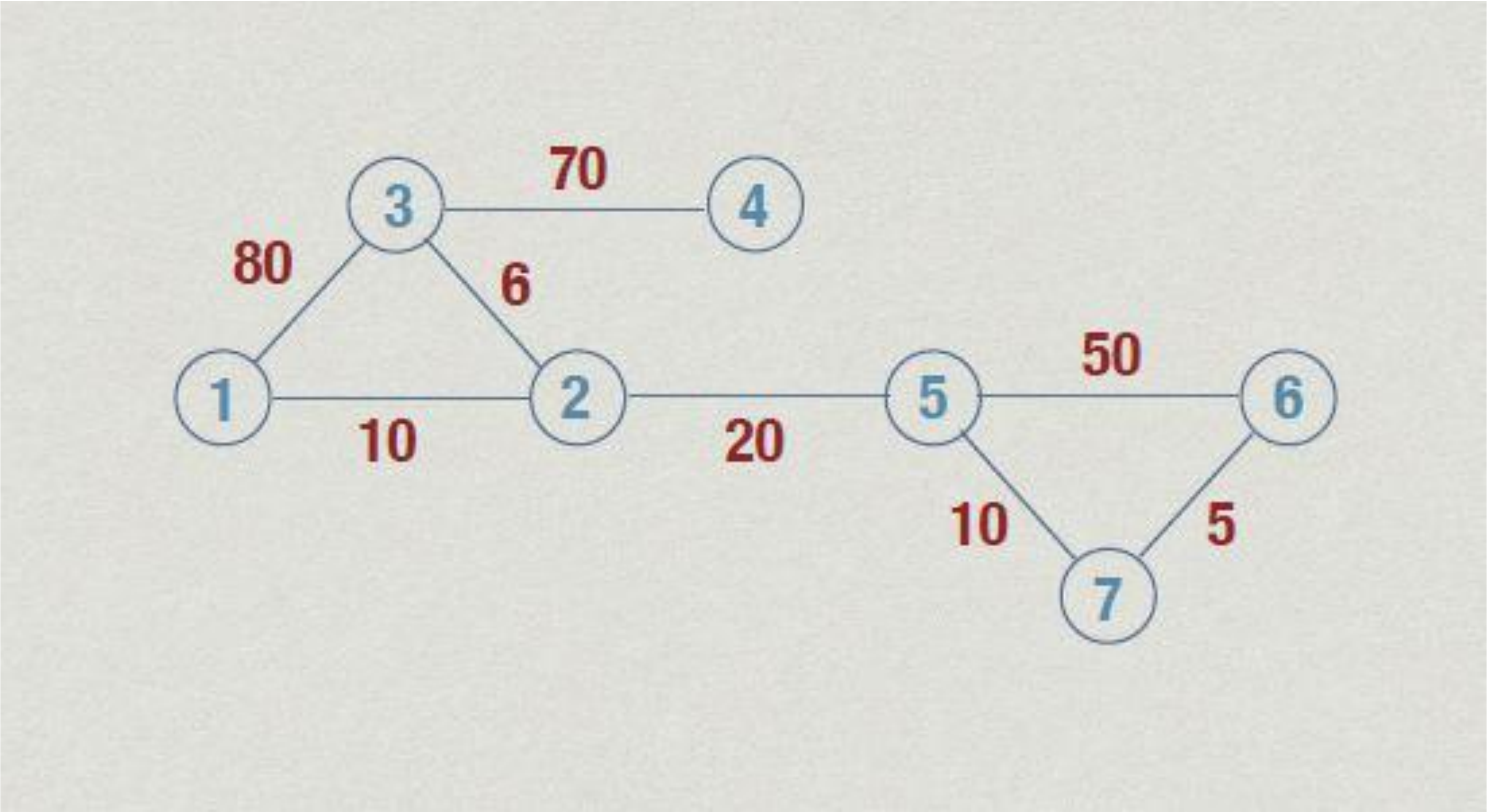
Single source shortest paths …



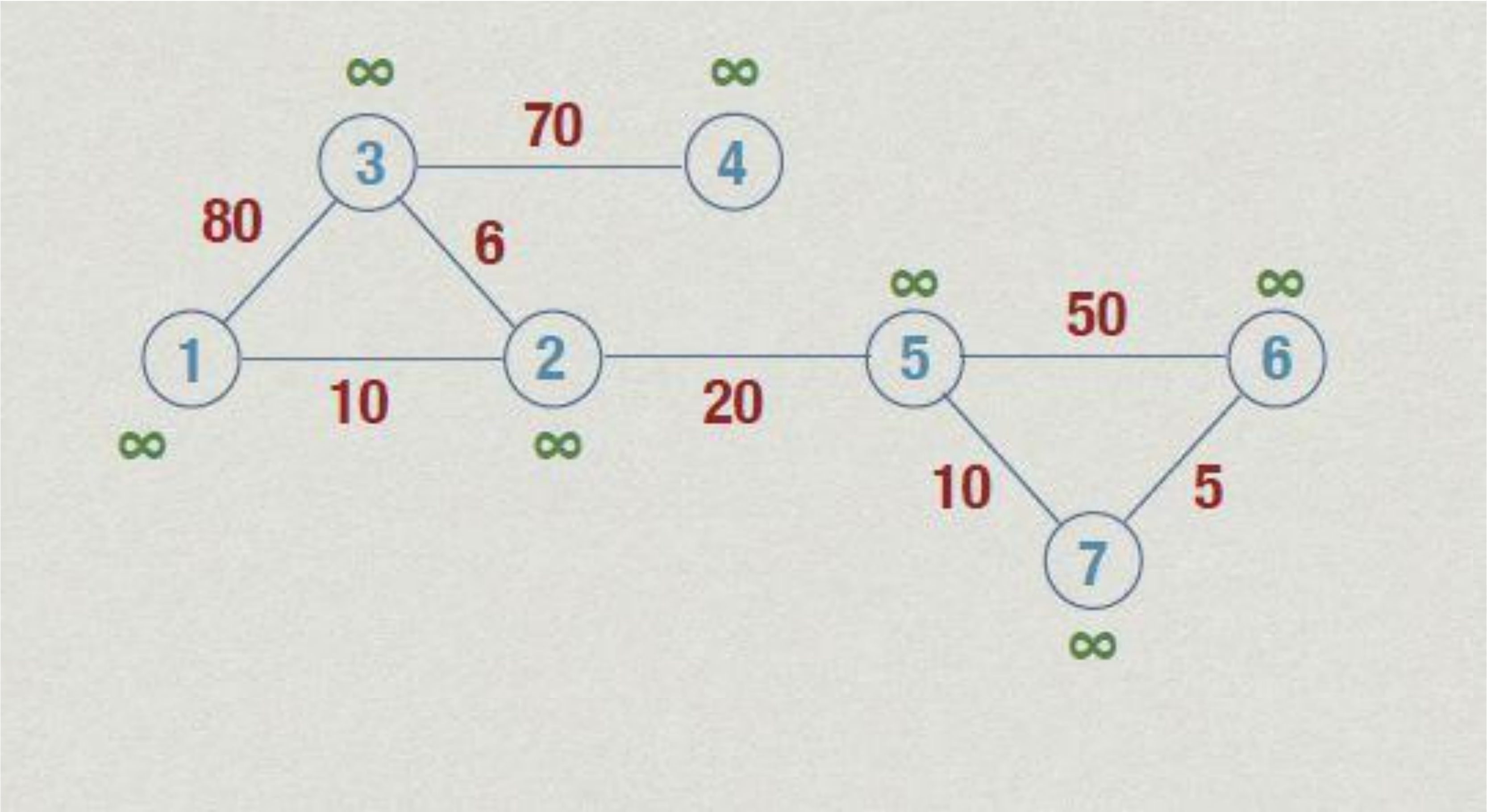
Single source shortest paths …

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

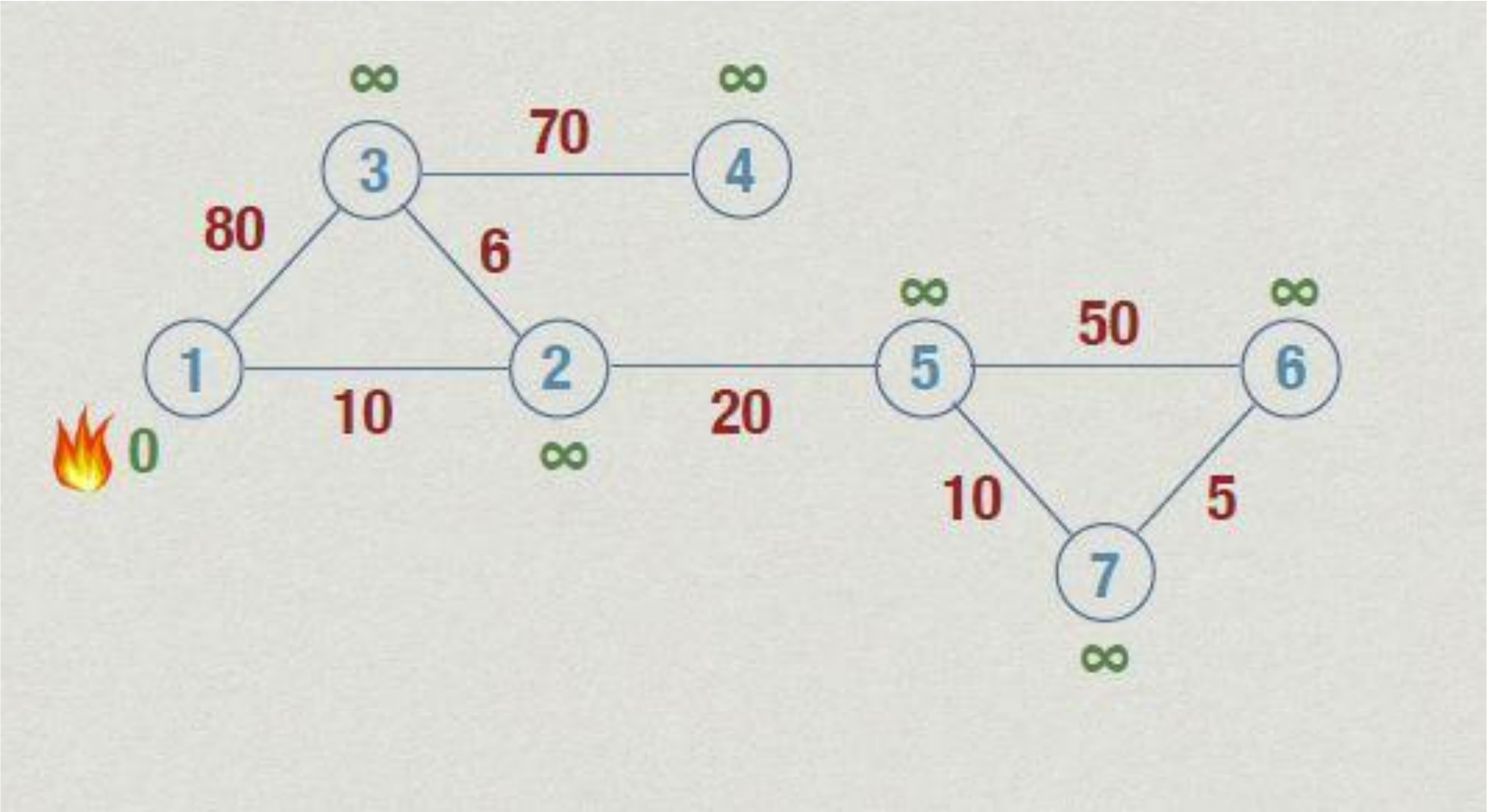
Single source shortest paths …



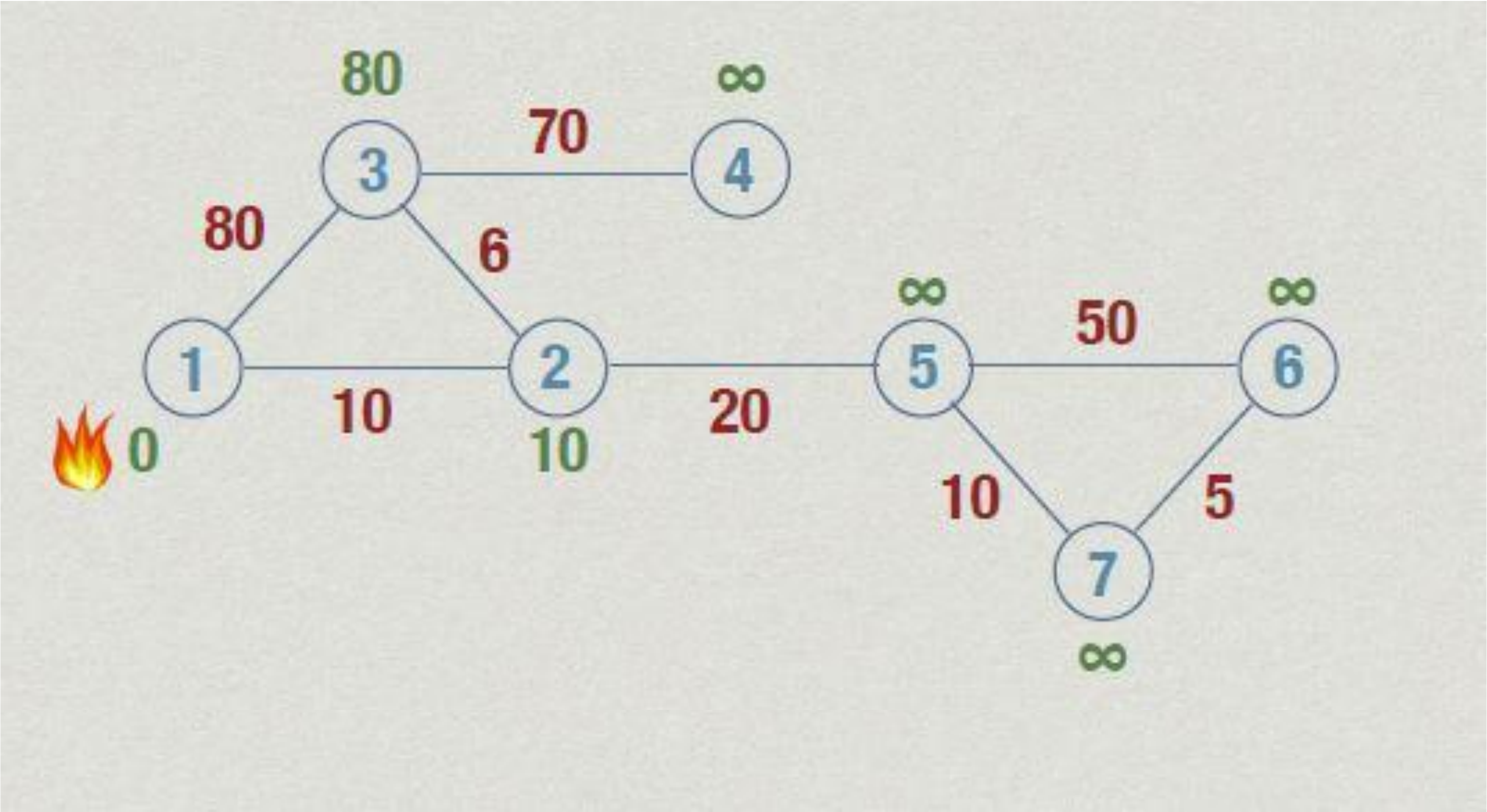
Single source shortest paths …



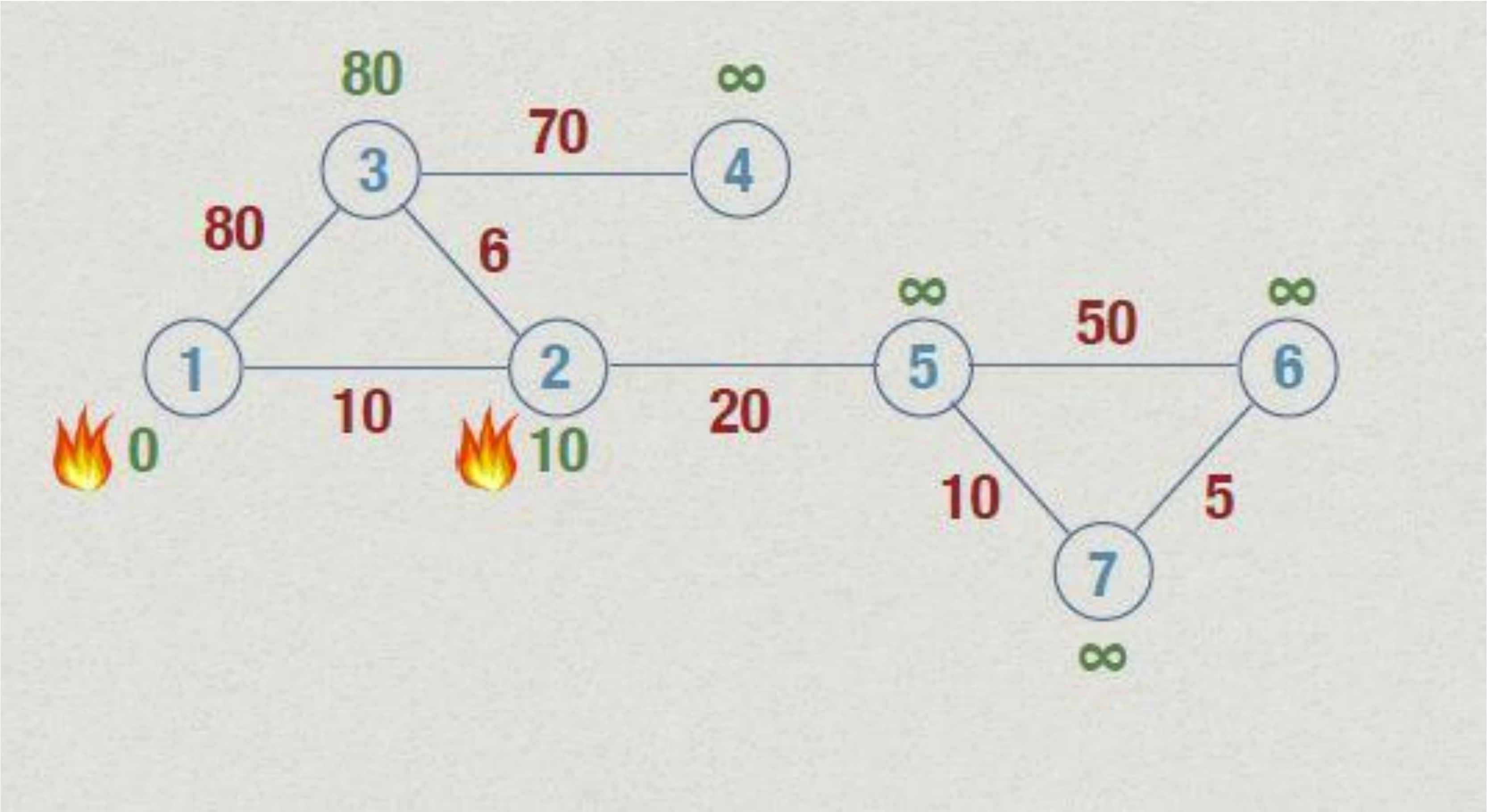
Single source shortest paths …



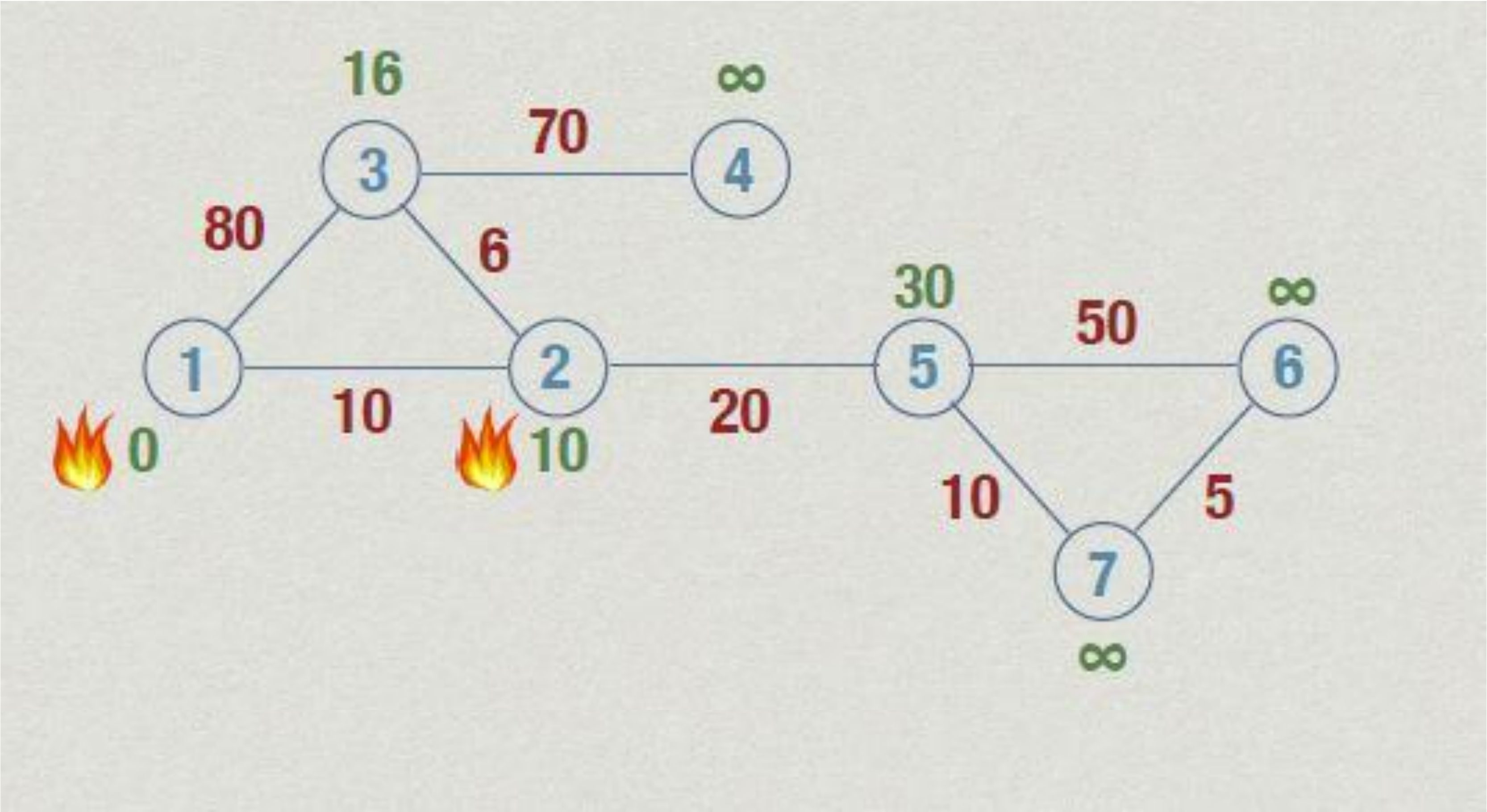
Single source shortest paths …



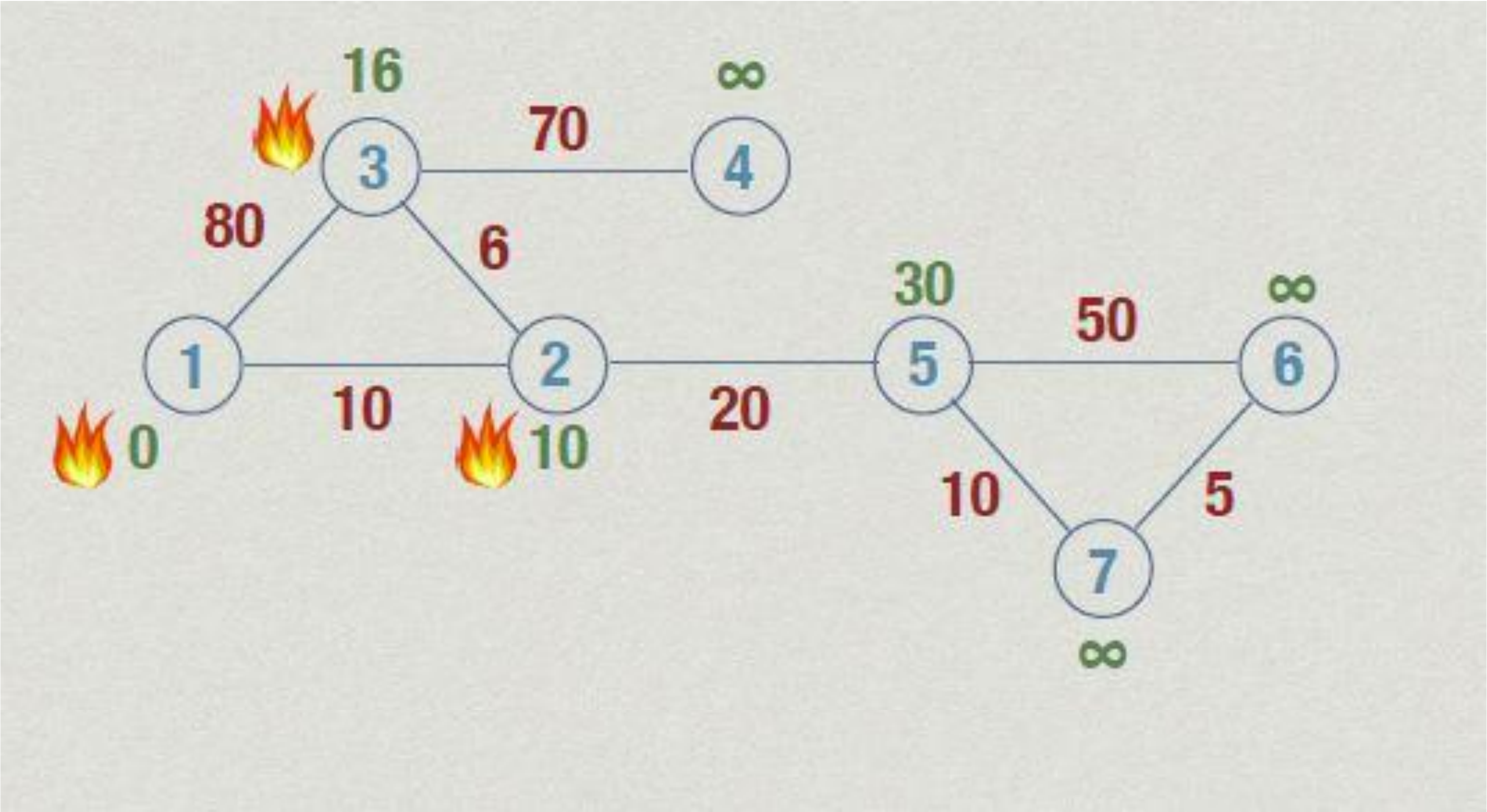
Single source shortest paths …



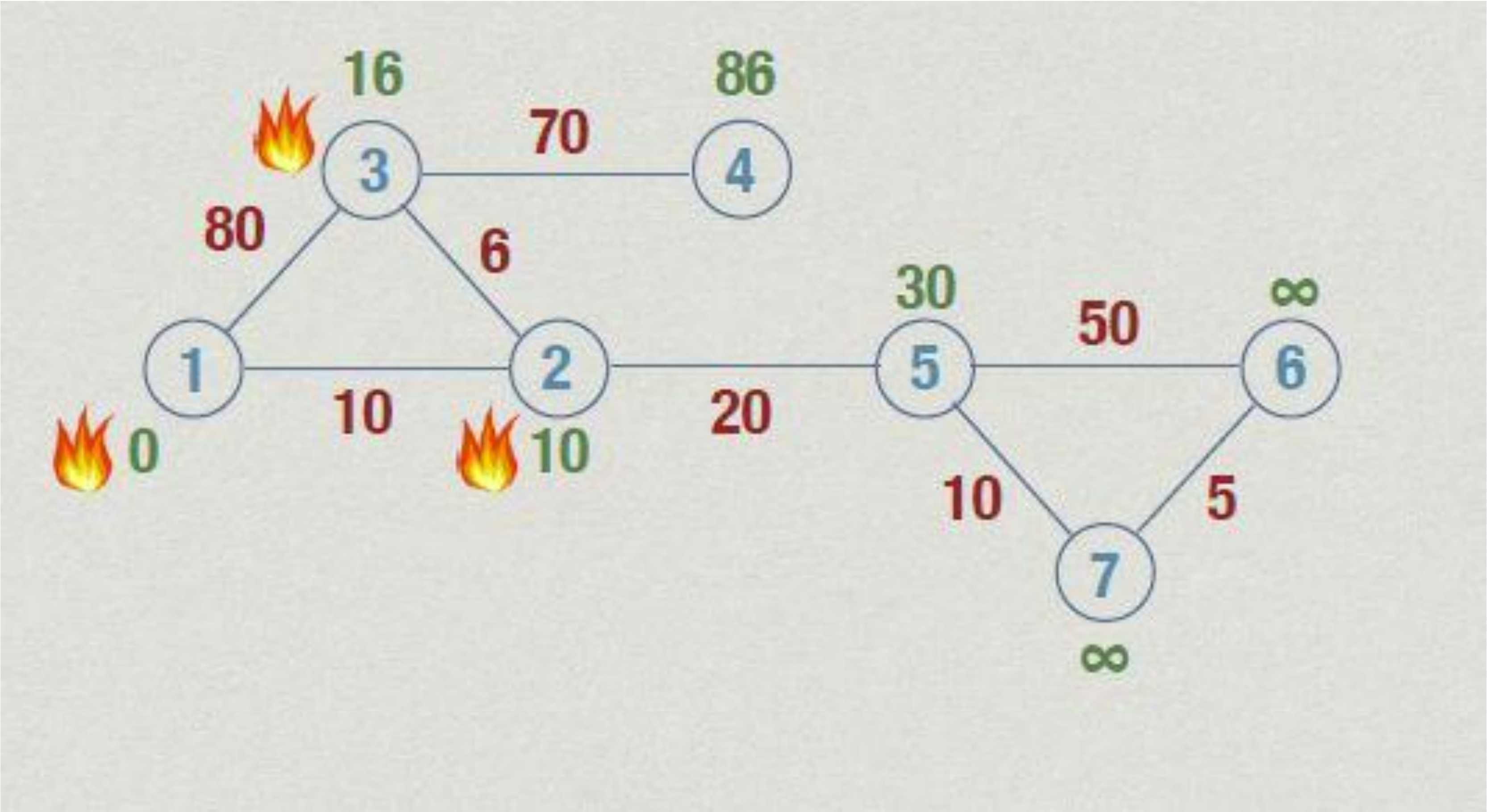
Single source shortest paths …



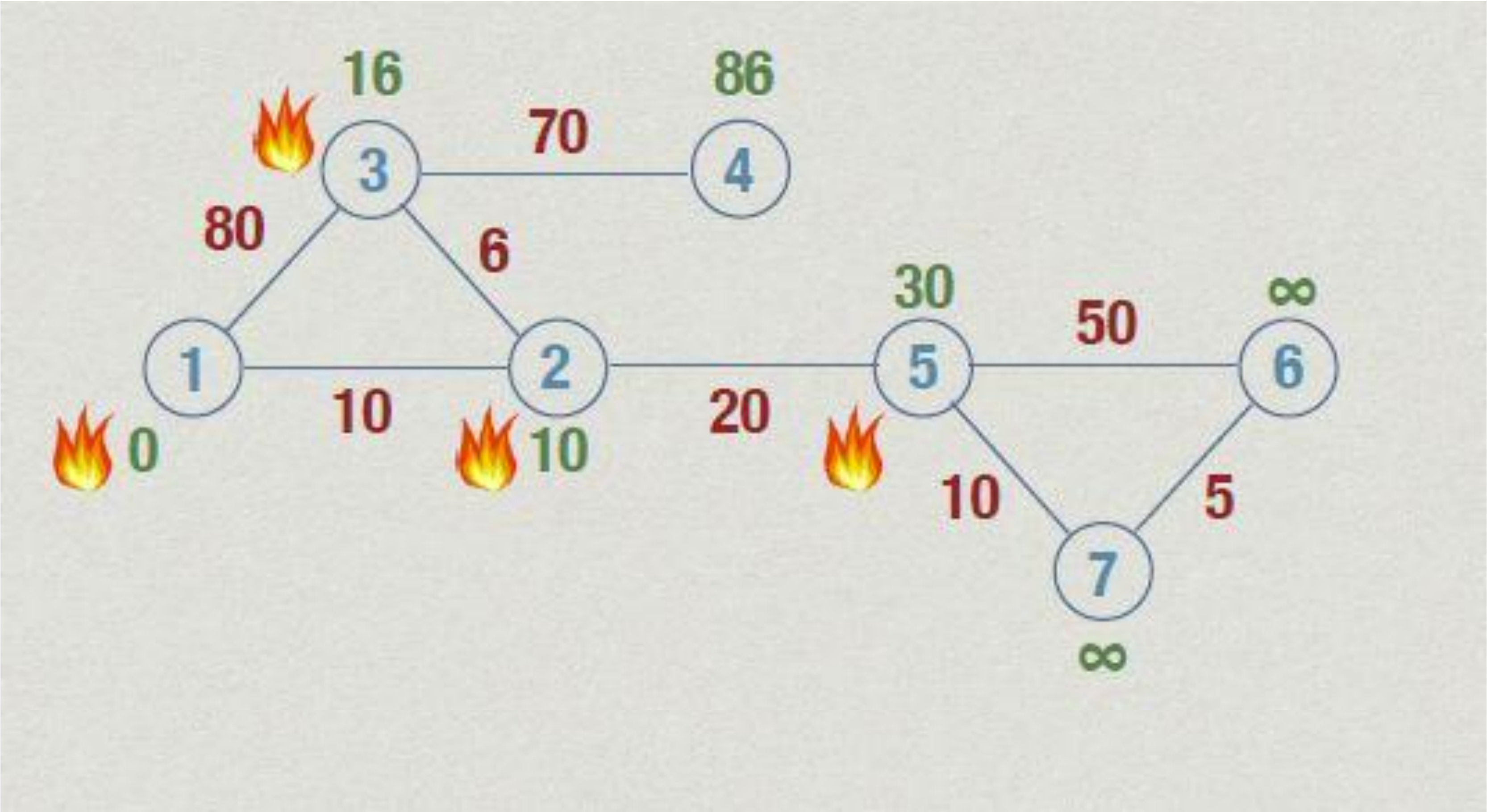
Single source shortest paths …



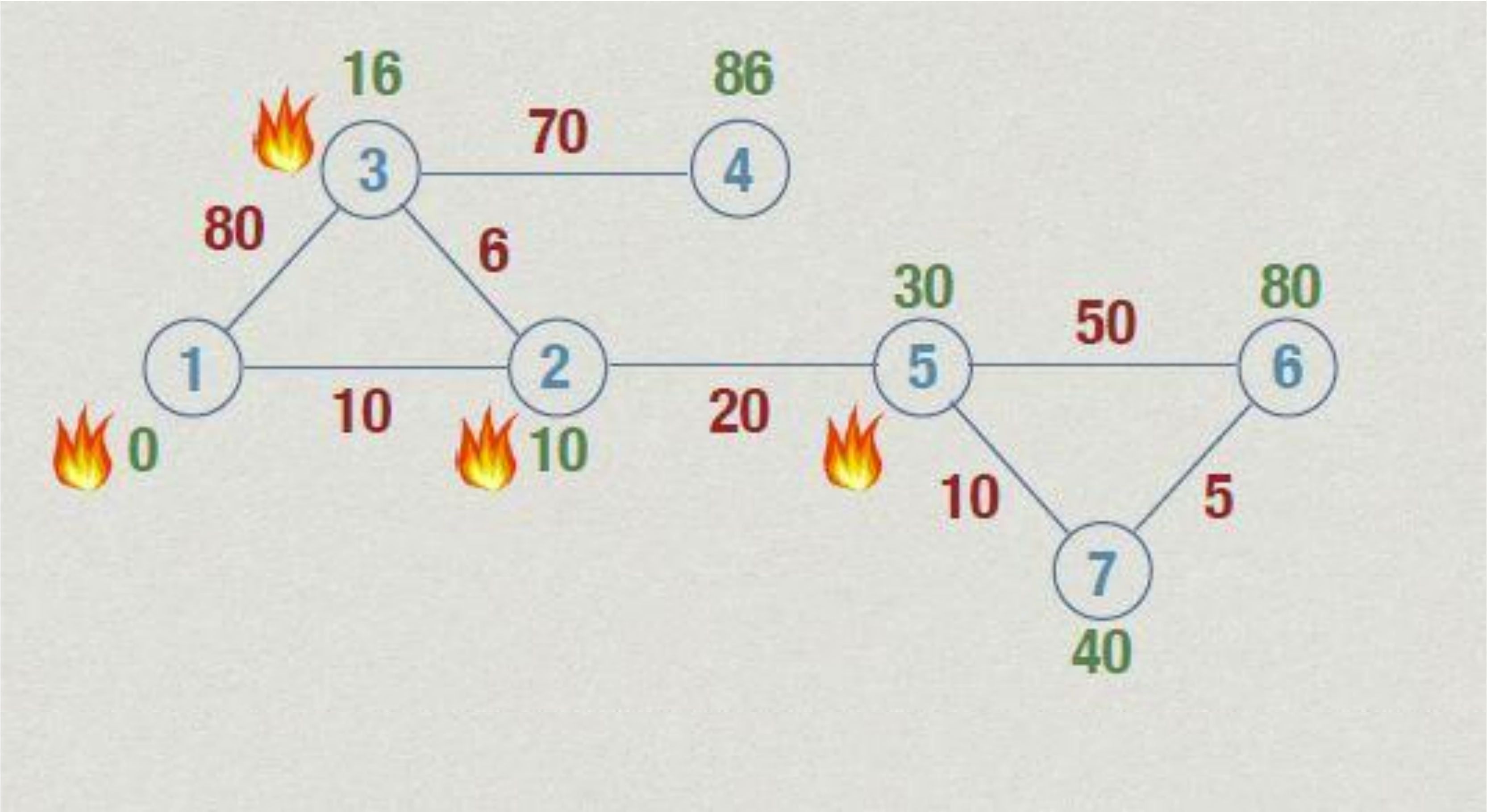
Single source shortest paths …



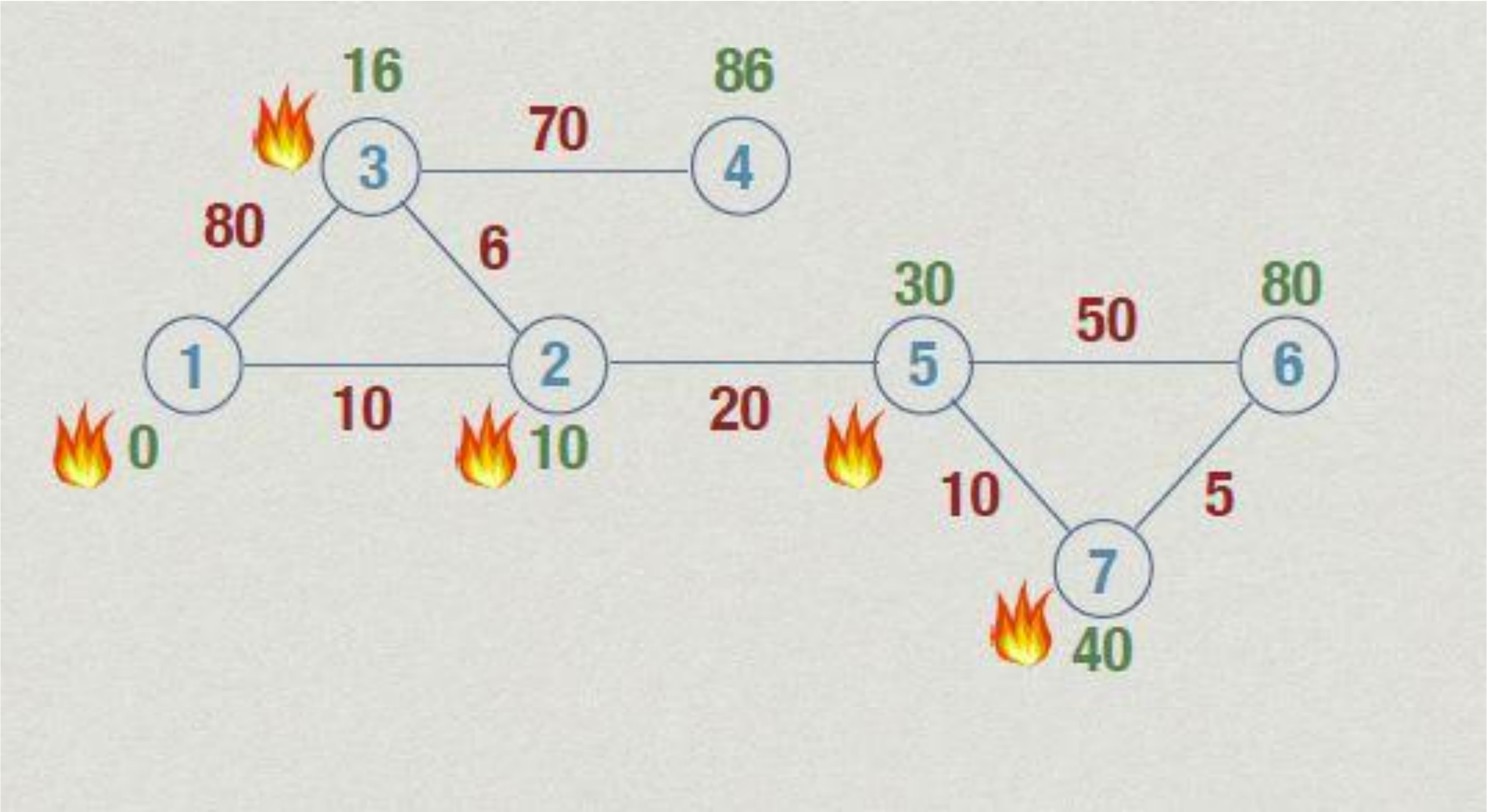
Single source shortest paths …



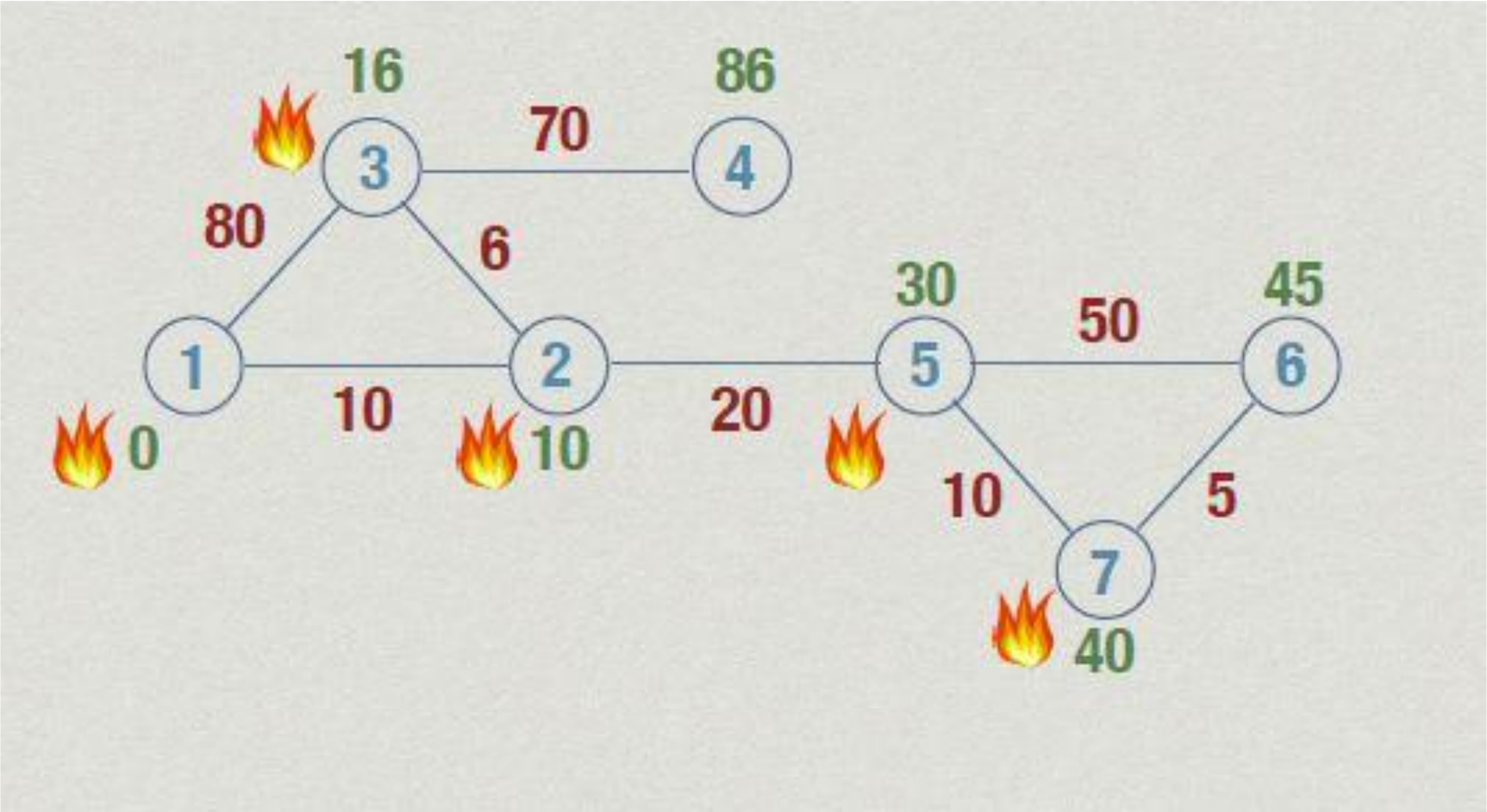
Single source shortest paths …



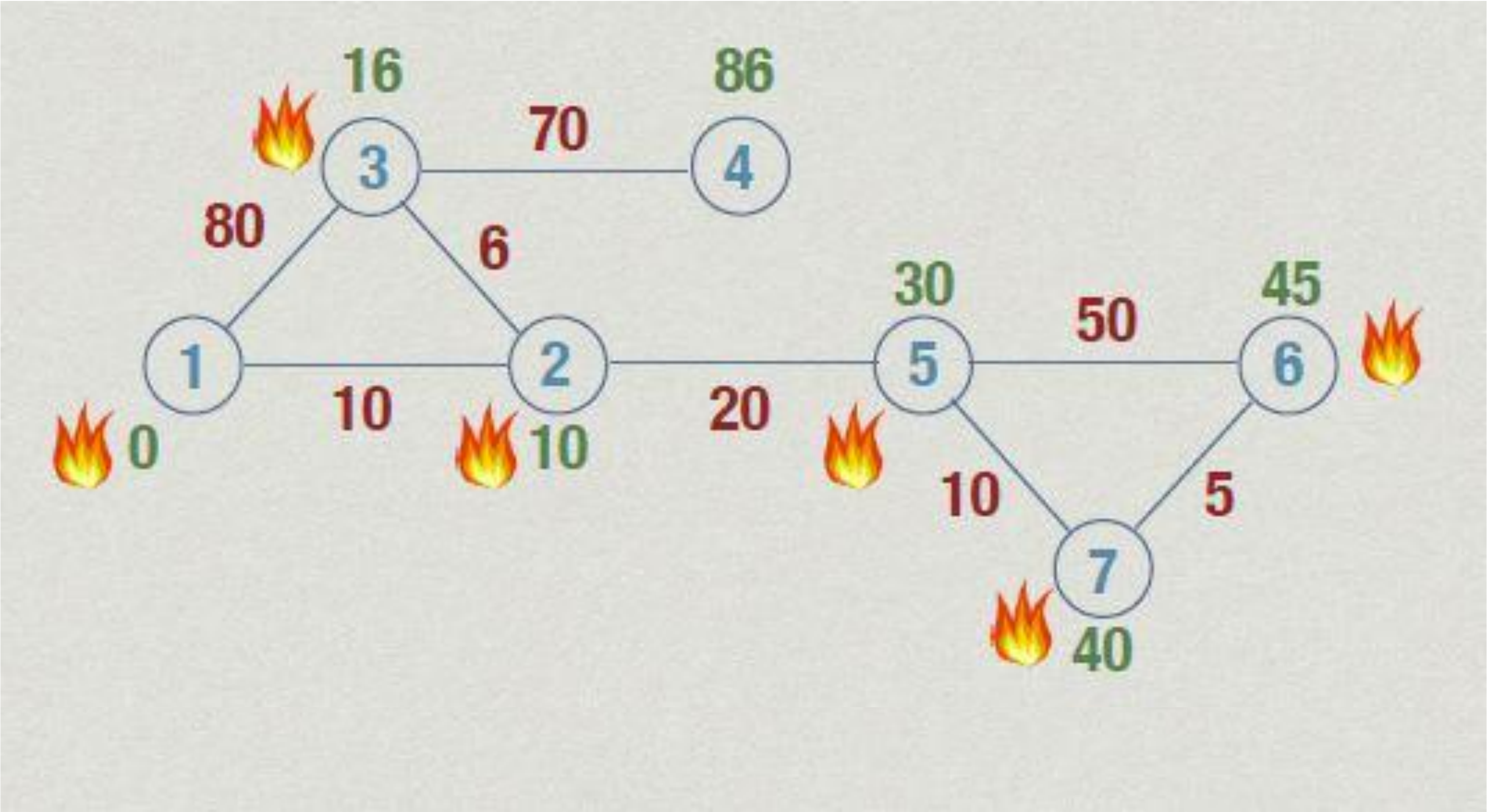
Single source shortest paths …



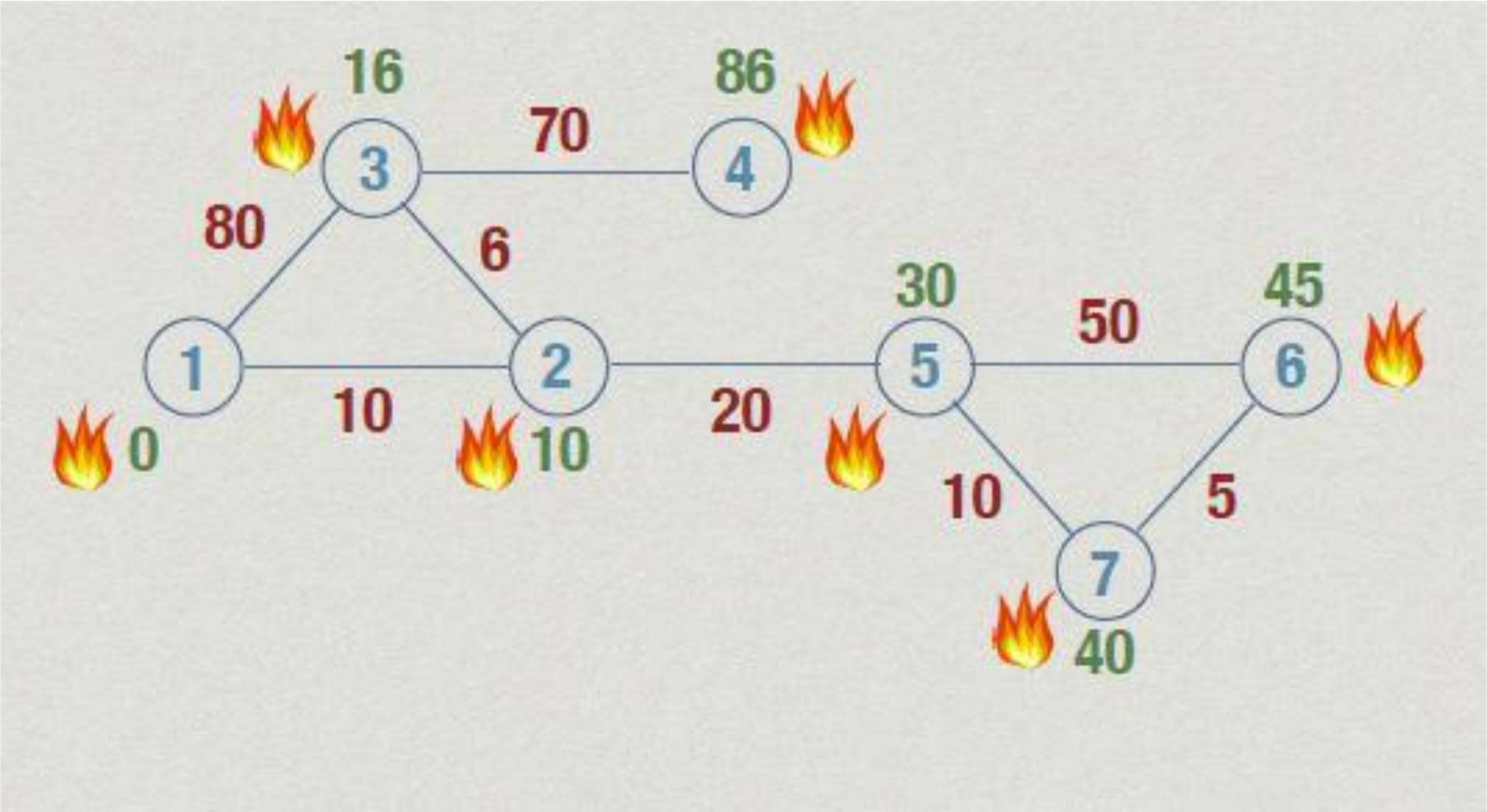
Single source shortest paths …



Single source shortest paths …



Single source shortest paths …



Single source shortest paths …

- Algorithmically
- Maintain two arrays
  - $BurnVertices[\ ]$, initially $False$ for all i
  - $ExpectedBurnTime[\ ]$, initially ∞ for all i

Single source shortest paths …

- Algorithmically
- Maintain two arrays
  - $BurnVertices[\,]$, initially $False$ for all i
  - $ExpectedBurnTime[\,]$, initially ∞ for all i
    - For ∞, use sum of all edge weights $+1$

Single source shortest paths …

- Algorithmically
- Maintain two arrays
  - $BurnVertices[\ ]$, 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 $BurnVertices[j] = True$
    - Recompute $ExpectedBurnTime[k]$ for each neighbor 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 $Distance[v]\ > Distance[u]\ +\ weight(u, v)$

$Distance[v]\ = Distance[u]\ +\ weight(u, v)$

Dijkstra’s algorithm: Analysis

Dijkstra’s algorithm: Analysis

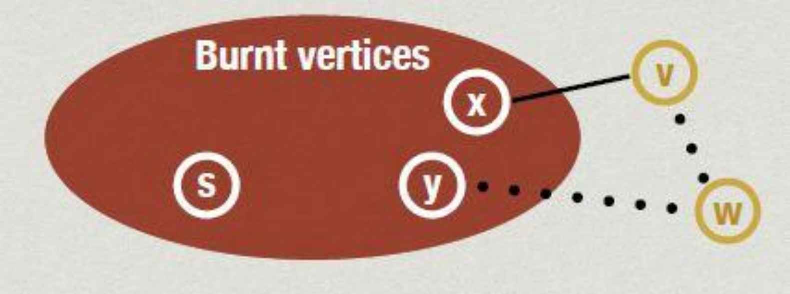
- Dijkstra’s algorithm is greedy
  - Select vertex with minimum expected burn time

Dijkstra’s algorithm: Analysis

- 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 may not be globally optimal

Dijkstra’s algorithm: Analysis …

- 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
- Can’t later find a shorter path from y to w to v

Dijkstra’s algorithm: Analysis …

- Complexity: ?

Dijkstra's algorithm: Analysis …

- 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 neighbors to update burn times
  - $O(n)$ scan of adjacency matrix to find all neighbors
- Overall $O(n^2)$

Dijkstra’s algorithm: Analysis …

- Complexity
- Does adjacency list help?

Dijkstra's algorithm: Analysis …

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

Dijkstra’s algorithm: Analysis …

- Complexity
- Can maintain ExpectedBurnTime in a more sophisticated data structure?

Dijkstra’s algorithm: Analysis …

- Complexity
- Can maintain ExpectedBurnTime in a more sophisticated data structure?
  - Different types of trees (heaps, red-black trees) allow both the following in $O(\log n)$ time
    - Find and delete minimum
    - Insert or update value

Dijkstra’s algorithm: Analysis …

- Complexity
- Can maintain ExpectedBurnTime in a more sophisticated data structure?
  - Different types of trees (heaps, red-black trees) allow both the following in $O(\log n)$ time
    - Find and delete minimum
    - Insert or update value
- 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 $O(n \log n + m \log n) = O((n + m) \log n)$

Dijkstra’s algorithm: Limitations

- What if edge weights can be negative?

Dijkstra’s algorithm: Limitations

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

Dijkstra’s algorithm: Limitations …

- Why negative weights?

Dijkstra’s algorithm: Limitations …

- Why negative weights?
- Weight represent money
  - Taxi driver earns money from airport to city, travels empty to next pick-up point
  - Some segment earn money. Some lose money
- Chemistry
  - Nodes are compounds, edges are reactions
  - Weights are energy absorbed/released by reaction

Negative weights …

- **Negative cycle:** loop with a negative weight
  - Problem is not well defined with negative cycles
  - Repeatedly traversing cycle pushes down cost
- With negative edges, but no negative cycles, shortest paths do exist

About shortest paths

- Shortest paths will never loop
  - Never visit the same vertex twice
  - At most length $n-1$

About shortest paths

- Shortest paths will never loop
  - Never visit the same vertex twice
  - At most length $n-1$
- Every prefix of a shortest path is itself a shortest path
  - Suppose the shortest path from s to t is

    $s \to v_1 \to v_2 \to v_3 \ldots \to v_m \to t$
  - Every prefix $s \to v_1 \to v_2 \to v_3 \ldots \to v_r$ is a shortest path to v_r

Updating Distance()

- When vertex j is “burnt”, for each (j, k) update
  - $Distance(k) = \min\left(Distance(k),\ Distance(j) + weight(j, k)\right)$
- Refer to this as $update(j, k)$
- Dijkstra’s algorithm
  - When we compute $update(j, k), Distance(j)$ is always guaranteed to be correct distance to j
- What we can say in general?

Properties of update(j, k)

- $update(j, k)$:
  - $Distance(k) = \min\big(Distance(k),\ Distance(j) + weight(j, k)\big)$
- $Distance(k)$ is no more than $Distance(j) + weight(j, k)$
- If $Distance(j)$ is correct and j is the second-last node on shortest path to k, $Distance(k)$ is correct
- Update is safe
  - $Distance(k)$ never becomes "too small"
  - Redundant updates cannot hurt

Updating distance() …

- $update(j, k)$
  - $Distance(k) = \min\big(Distance(k),\ Distance(j) + weight(j, k)\big)$
- Dijkstra’s algorithm performs a particular “greedy” sequence of updates
  - Computes shortest paths without negative weights
- With negative edges, this sequence does not work
- Is there some sequence that dose work?

Updating distance() …

- Suppose the shortest path from s to t is

$$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \ldots \rightarrow v_m \rightarrow t$$

- If our update sequence includes …, $update(s, v_1), \ldots,$ $update(v_1, v_2), \ldots, update(v_2, v_3), \ldots, update(v_m, t), \ldots,$ in that order, $Distance(t)$ will be computer correctly
- If $Distance(j)$ is correct and j is second-last node on shortest path to k, $Distance(k)$ is correct after $update(j, k)$

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1
...
update(s,v_1)
...
update(v_1,v_2)
...
update(v_2,v_3)
...
update(v_m,t)
...

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2
...	...
update(s,v_1)	update(s,v_1)
...	...
update(v_1,v_2)	update(v_1,v_2)
...	...
update(v_2,v_3)	update(v_2,v_3)
...	...
update(v_m,t)	update(v_m,t)
...	...

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2	...
...	...	...
update(s,v_1)	update(s,v_1)	...
...	...	...
update(v_1,v_2)	update(v_1,v_2)	...
...	...	...
update(v_2,v_3)	update(v_2,v_3)	...
...	...	...
update(v_m,t)	update(v_m,t)	...
...	...	...

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	update(v_1,v_2)	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	update(v_1,v_2)	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	**update(v_1,v_2)**	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	update(v_m,t)
…	…	…	…

Bellman-Ford algorithm

- Initialize $distance(s) = 0, distance(u) = \infty$ for all other vertices
- Update all edges $n - 1$ times!

Iteration 1	Iteration 2	…	Iteration n-1
…	…	…	…
update(s,v_1)	update(s,v_1)	…	update(s,v_1)
…	…	…	…
update(v_1,v_2)	**update(v_1,v_2)**	…	update(v_1,v_2)
…	…	…	…
update(v_2,v_3)	update(v_2,v_3)	…	update(v_2,v_3)
…	…	…	…
update(v_m,t)	update(v_m,t)	…	**update(v_m,t)**
…	…	…	…

Bellman-Ford algorithm

function $BellmanFord(s)$//source s, with -ve weights

for $i\ =\ 1\ to\ n$

$\quad Distance[i]\ =\ infinity$

$Distance[s]\ =\ 0$

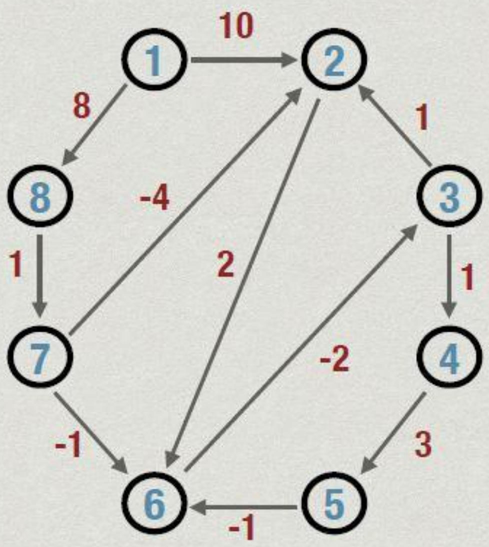
for $i\ =\ 1$ to $n-1$ //repeat n-1 times

$\quad$ for each $edge(j,k)$ in E

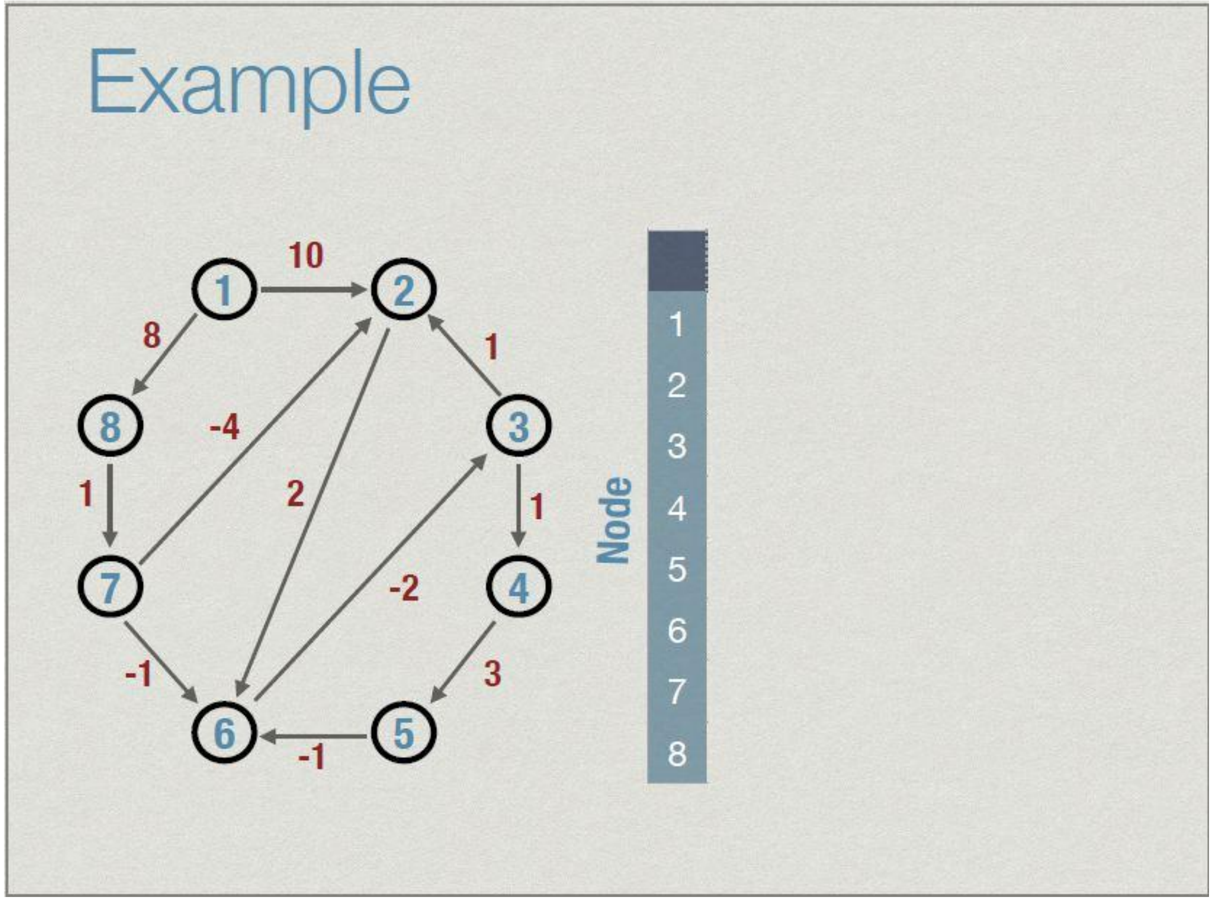
$\quad\quad Distance(k)\ =\ \min(Distance(k), Distance(j) + weight(j,k))$

Bellman-Ford algorithm

Example

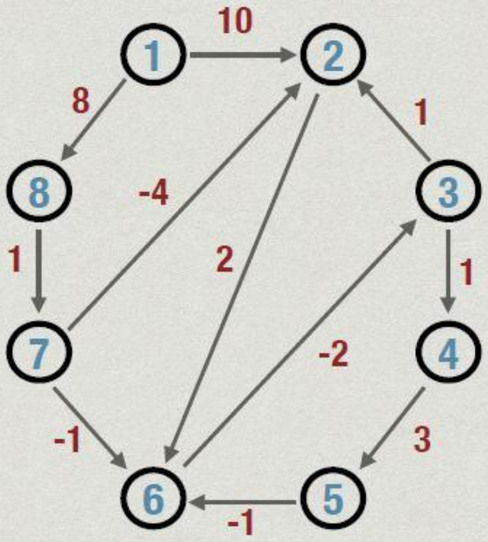


Bellman-Ford algorithm



Bellman-Ford algorithm

Example

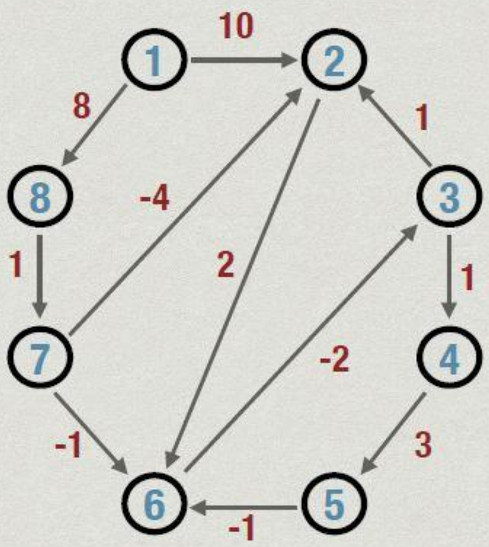


Iteration

Node	0
1	0
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞

Bellman-Ford algorithm

Example

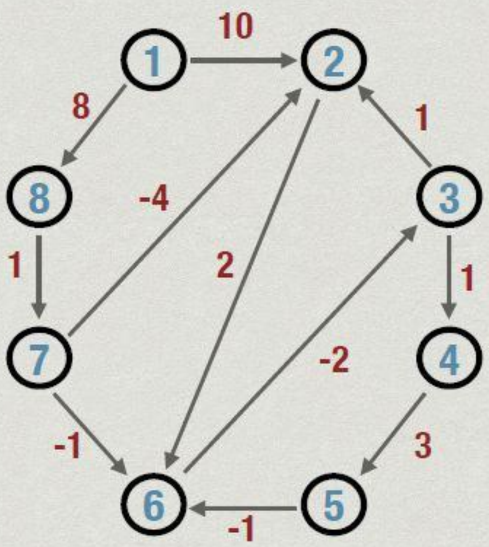


Iteration

Node	0	1
1	0	0
2	∞	10
3	∞	∞
4	∞	∞
5	∞	∞
6	∞	∞
7	∞	∞
8	∞	8

Bellman-Ford algorithm

Example

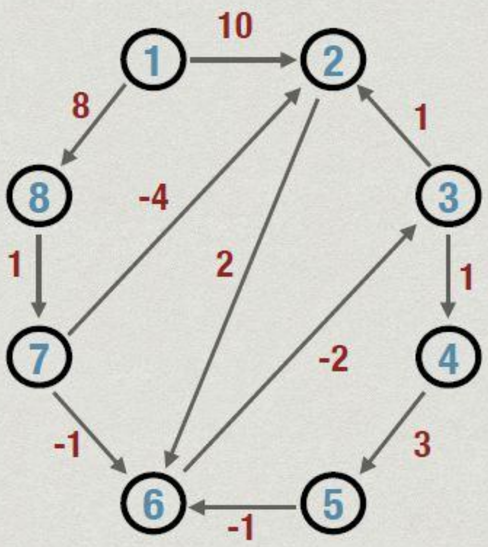


Iteration

Node	0	1	2
1	0	0	0
2	∞	10	10
3	∞	∞	∞
4	∞	∞	∞
5	∞	∞	∞
6	∞	∞	12
7	∞	∞	9
8	∞	8	8

Bellman-Ford algorithm

Example

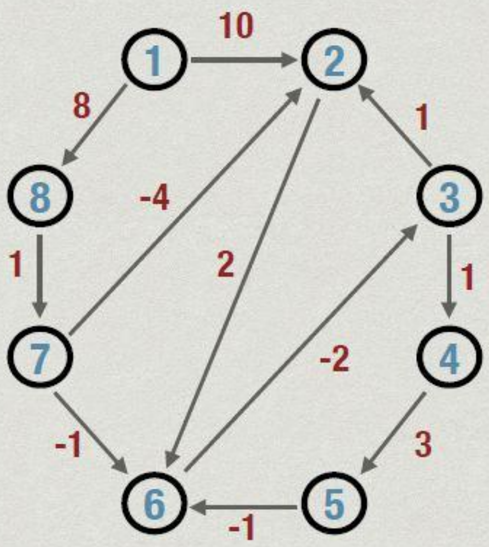


Iteration

Node	0	1	2	3
1	0	0	0	0
2	∞	10	10	5
3	∞	∞	∞	10
4	∞	∞	∞	∞
5	∞	∞	∞	∞
6	∞	∞	12	8
7	∞	∞	9	9
8	∞	8	8	8

Bellman-Ford algorithm

Example

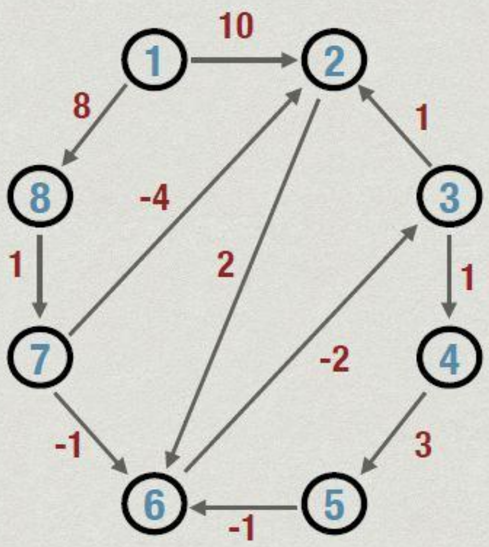


Iteration

Node	0	1	2	3	4
1	0	0	0	0	0
2	∞	10	10	5	5
3	∞	∞	∞	10	6
4	∞	∞	∞	∞	11
5	∞	∞	∞	∞	∞
6	∞	∞	12	8	7
7	∞	∞	9	9	9
8	∞	8	8	8	8

Bellman-Ford algorithm

Example

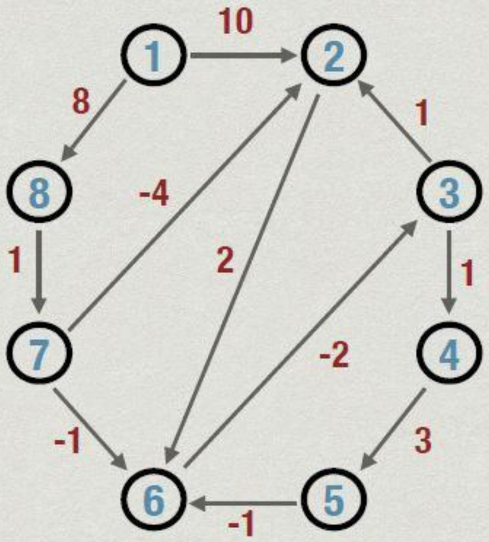


Iteration

Node	0	1	2	3	4	5
1	0	0	0	0	0	0
2	∞	10	10	5	5	5
3	∞	∞	∞	10	6	5
4	∞	∞	∞	∞	11	7
5	∞	∞	∞	∞	∞	14
6	∞	∞	12	8	7	7
7	∞	∞	9	9	9	9
8	∞	8	8	8	8	8

Bellman-Ford algorithm

Example

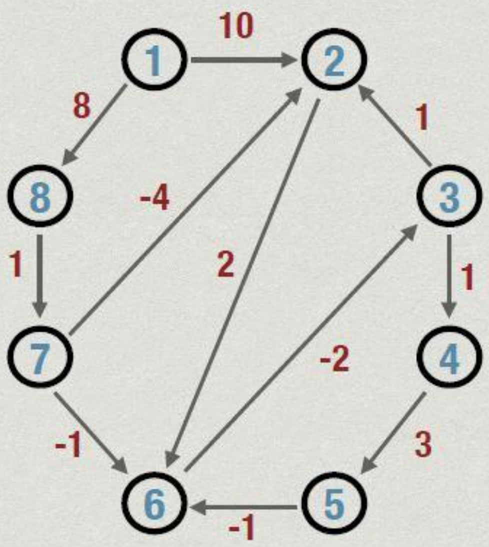


Iteration

Node	0	1	2	3	4	5	6
1	0	0	0	0	0	0	0
2	∞	10	10	5	5	5	5
3	∞	∞	∞	10	6	5	5
4	∞	∞	∞	∞	11	7	6
5	∞	∞	∞	∞	∞	14	10
6	∞	∞	12	8	7	7	7
7	∞	∞	9	9	9	9	9
8	∞	8	8	8	8	8	8

Bellman-Ford algorithm

Example



Iteration

Node	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	∞	10	10	5	5	5	5	5
3	∞	∞	∞	10	6	5	5	5
4	∞	∞	∞	∞	11	7	6	6
5	∞	∞	∞	∞	∞	14	10	9
6	∞	∞	12	8	7	7	7	7
7	∞	∞	9	9	9	9	9	9
8	∞	8	8	8	8	8	8	8

Bellman-Ford algorithm (Complexity)

- Outer loop runs n times
- In each loop, for each $edge\ (j,k)$, we run $update(j,k)$
  - Adjacency matrix-$O(n^2)$ to identify all edges
  - Adjacency list - $O(m)$
- Overall
  - Adjacency matrix-$O(n^3)$
  - Adjacency list - $O(mn)$

Weighted graphs

- Negative weights are allowed, but not negative cycles
- Shortest paths are still well defined
- Bellman-Ford algorithm computes single-source shortest paths
- Can we compute shortest paths between all pairs of vertices?

About shortest paths

- Shortest paths will never loop
    - Never visit the same vertex twice
    - At most length $n-1$
- Use this to inductively explore all possible shortest paths efficiently

Inductively exploring shortest paths

- Simplest shortest path from i to j is a direct edge(i, j)
- General case:
  - $i \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \dots \rightarrow v_m \rightarrow j$
  - All of $\{v_1, v_2, v_3 \dots, v_m\}$ are distinct, and different from i and j
  - Restrict what vertices can appear in this set

Inductively exploring shortest paths …

- Recall that $V = \{1,2,\ldots,n\}$
- $W^k(i,j)$: weight of shortest path from i to j among paths that only go via $\{1,2,\ldots,k\}$
  - {k+1,…,n} cannot appear on the path
  - i, j themselves need not be in $\{1,2,\ldots,k\}$
- $W^0(i,j)$: direct edges
  - $\{1,2,\ldots,n\}$ cannot appear between i and j

Inductively exploring shortest paths …

- From $W^{k-1}(i,j)$ to $W^{k}(i,j)$
- $Case$ 1: Shortest path via $\{1,2,\ldots,k\}$ does not use vertex k
  - $W^{k}(i,j) = W^{k-1}(i,j)$
- $Case$ 2: Shortest path via $\{1,2,\ldots,k\}$ does go via k
  - k can appear only once along this path
  - Break up as paths i to k and k to j, each via $\{1,2,\ldots,k-1\}$
  - $W^{k}(i,j) = W^{k-1}(i,k) + W^{k-1}(k,j)$
- Conclusion: $W^{k}(i,j) = \min(W^{k}(i,j)\ , W^{k-1}(i,k) + W^{k-1}(k,j))$

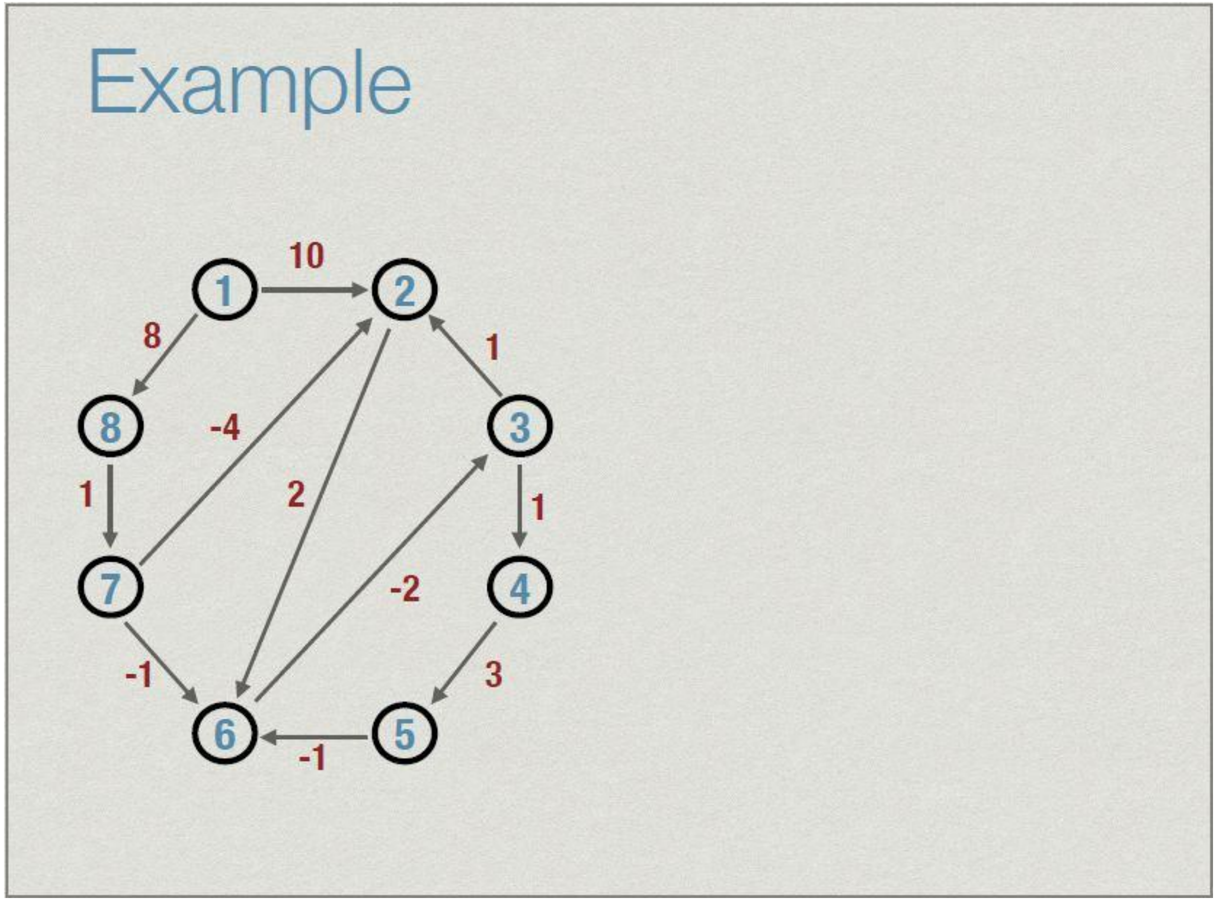
Floyd-Warshall algorithm

- W^0 is adjacency matrix with edge weights
  - $W^0[i][j]$ = weight (i, j) if there is an edge (i, j),
    $= \infty$, otherwise
- For k in $1,2, \dots, n$
  - Compute $W^k(i, j)$ from $W^{k-1}(i, j)$ using
    $W^k(i, j)$ = $\min(W^k(i, j)\ , W^{k-1}(i, k)$ + $W^{k-1}(k, j))$
- W^n contains weights of shortest paths for all pairs

Floyd-Warshall algorithm …

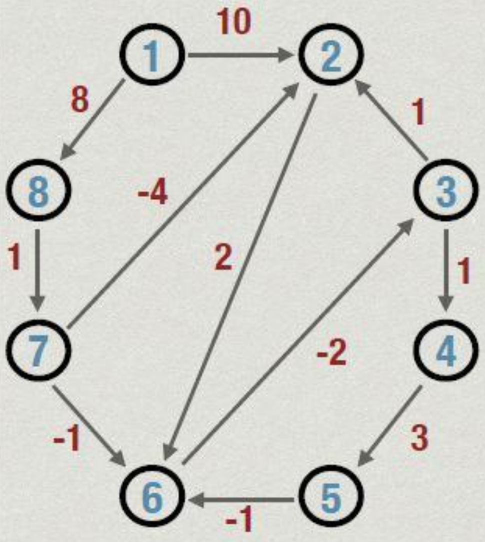
$function$ FloydWarshall
for $i \ = \ 1$ to n
    for $j \ = \ 1$ to n
        $W[i][j][0]$ = infinity
for each edge (i, j) in E
    $W[i][j][0]$ = $weight(i, j)$
for $k \ = \ 1$ to n
    for $i \ = \ 1$ to n
        for $j \ = \ 1$ to n
            $W[i][j][k]$ = $\min(W[i][j][k-1], W[i][k][k-1] \ + \ W[k][j][k-1])$

Floyd-Warshall algorithm …



Floyd-Warshall algorithm …

Example

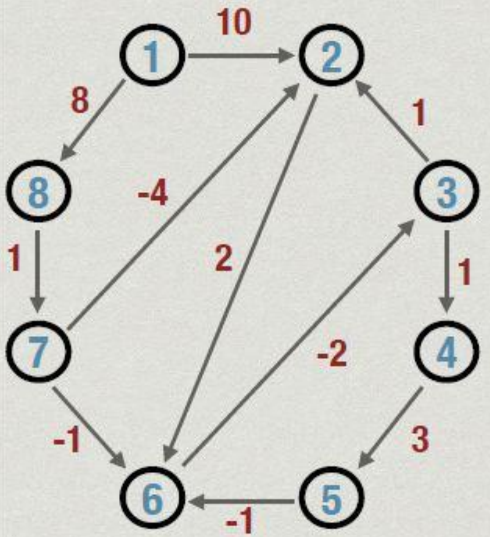


W^0

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Floyd-Warshall algorithm …

Example



W^0

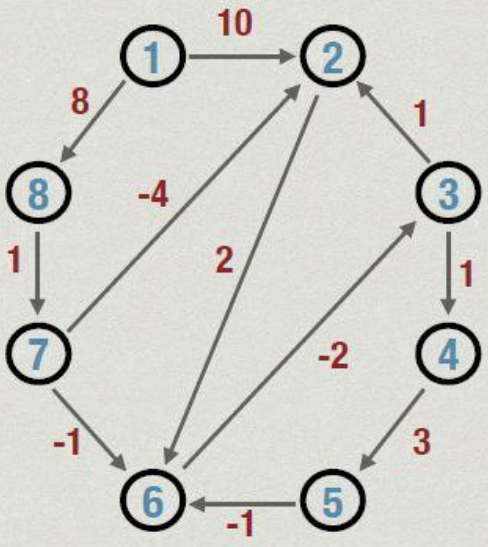
	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Floyd-Warshall algorithm …

Example

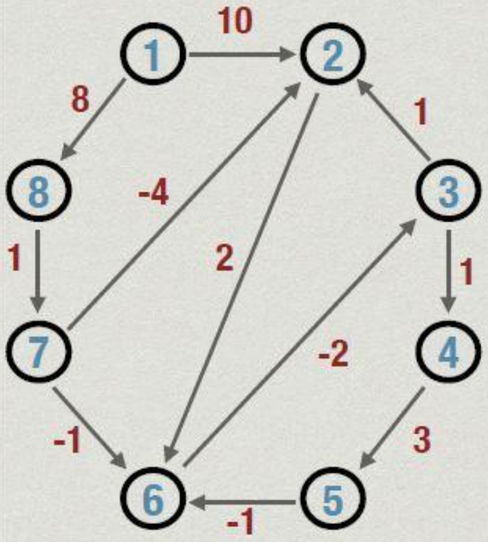


W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Floyd-Warshall algorithm …

Example



W^2

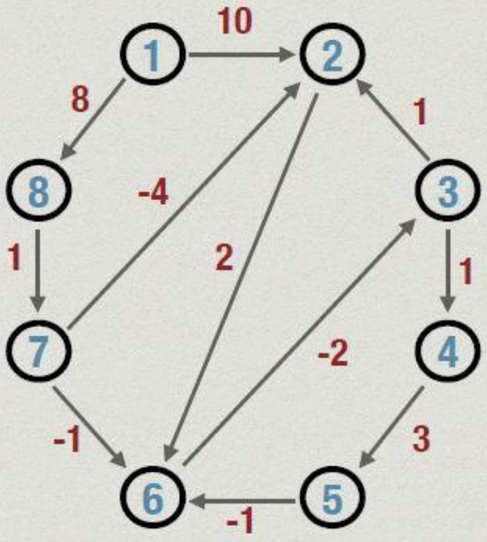
	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Floyd-Warshall algorithm …

Example

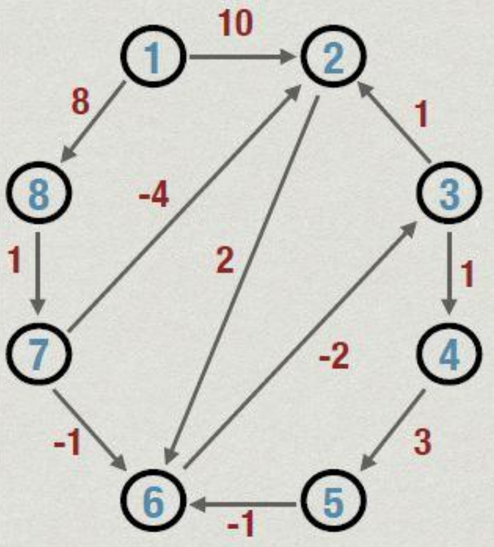


W^2

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Floyd-Warshall algorithm …

Example



W^2

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^3

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	∞	∞	3	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	-1	-2	-1	∞	1	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Complexity

Complexity

- Easy to see that the complexity is $O(n^3)$
  - n iterations
  - In each iteration, we update n^2entries