QEEE module

Graphs
Part 2

G Venkatesh
IIT Madras

Introduction:

How graphs fits into the study of algorithms and data structures

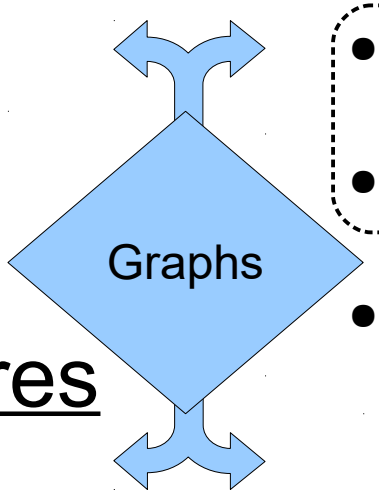
Algorithms and Data Structures

What is the problem?

- Prototype problems
- Variations

Algorithmic techniques

- Divide and conquer
- Greedy
- Dynamic Programming
- Network Flow



Data structures

- Arrays, Lists
- Queues, Stacks
- Heaps, Binary Trees

Analysis

- Parameters:
  - No of nodes – n
  - No of edges - m

Definition of graphs

- Graph G = (V,E)
- V is a set of Vertices (or nodes)
- E is a set of edges, $E \subseteq V \times V$
- Each edge is a pair of vertices (v,v')
- If graph is undirected, then (v',v) and (v,v') are the same edge and both should be in E
- A *path from u to v* is a sequences of vertices $u=v_1,v_2,v_3,...v_n=v$ such that $(v_i,v_{i+1}) \in E$.
- Graph can be represented using *Adjacency matrix* or *Adjacency list*

Exploring a graph

- *Breadth First Search (BFS):* explore level by level starting from any vertex.
  - BFS finds paths with least no of edges to all nodes from a single source
  - Can be used to find (odd) cycles in a graph – which can be used to find if the graph is bipartite
- *Depth First Search (DFS):* explore by going deep down one path starting from any vertex.
  - DFS exposes the structure of the graph
  - Can be used to find cut edges and vertices, for finding strongly connected components in directed graphs and for topological sorting of acyclic directed graphs

Single source shortest paths

Graphs with weighted edges

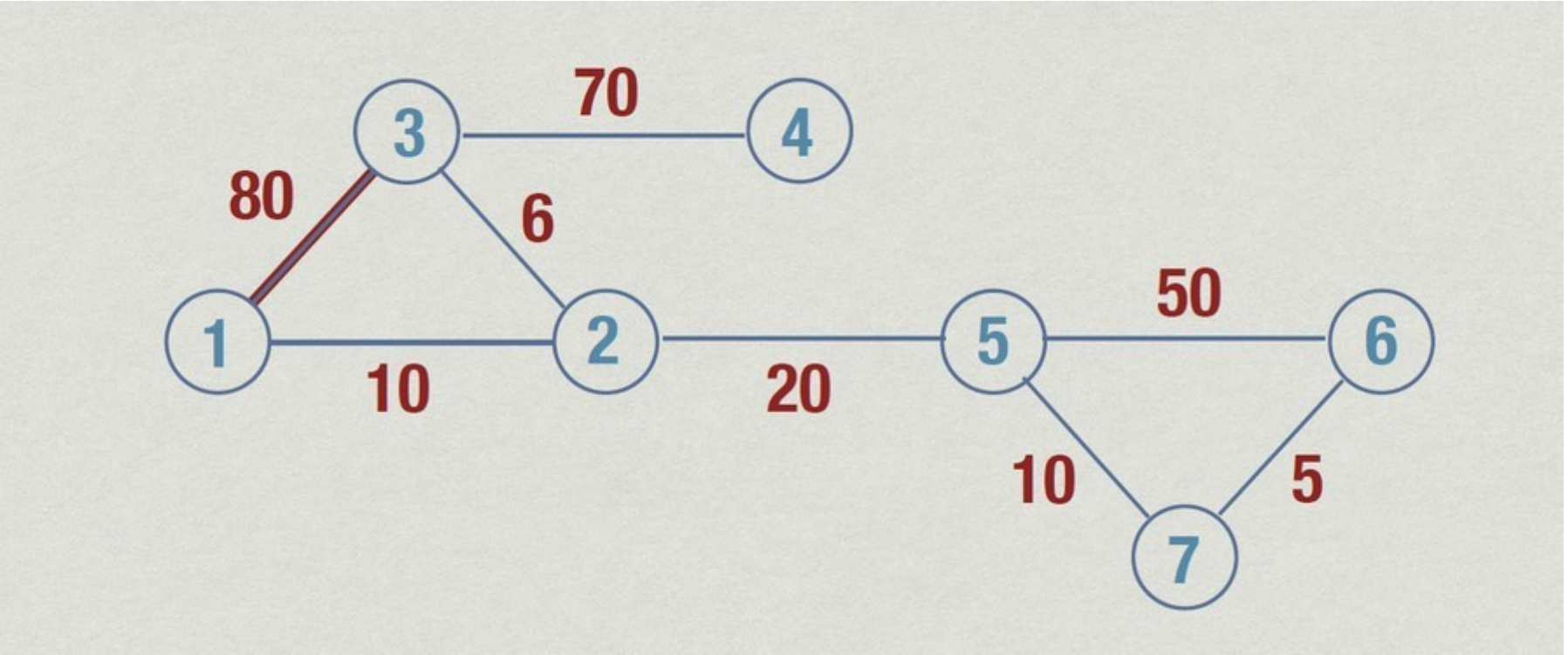
- A weighted graph is:
  - A graph G = (V,E) together with a ***weight function***
  - w: E → Reals
- Let $e_1 = (v_1,v_2)$, $e_2 = (v_2,v_3)$, ..., $e_n = (v_n,v_{n+1})$ be a path from v_1 to v_{n+1}
- *Cost of the path* is $w(e_1) + w(e_2) + ... + w(e_n)$.
- *Shortest path* from v_1 to v_{n+1} is the path with minimum cost

Does BFS find the shortest path?

- We are interested in finding the shortest paths to all nodes from a single source
- BFS finds path with the least number of edges
- May not be the shortest path in a weighted graph

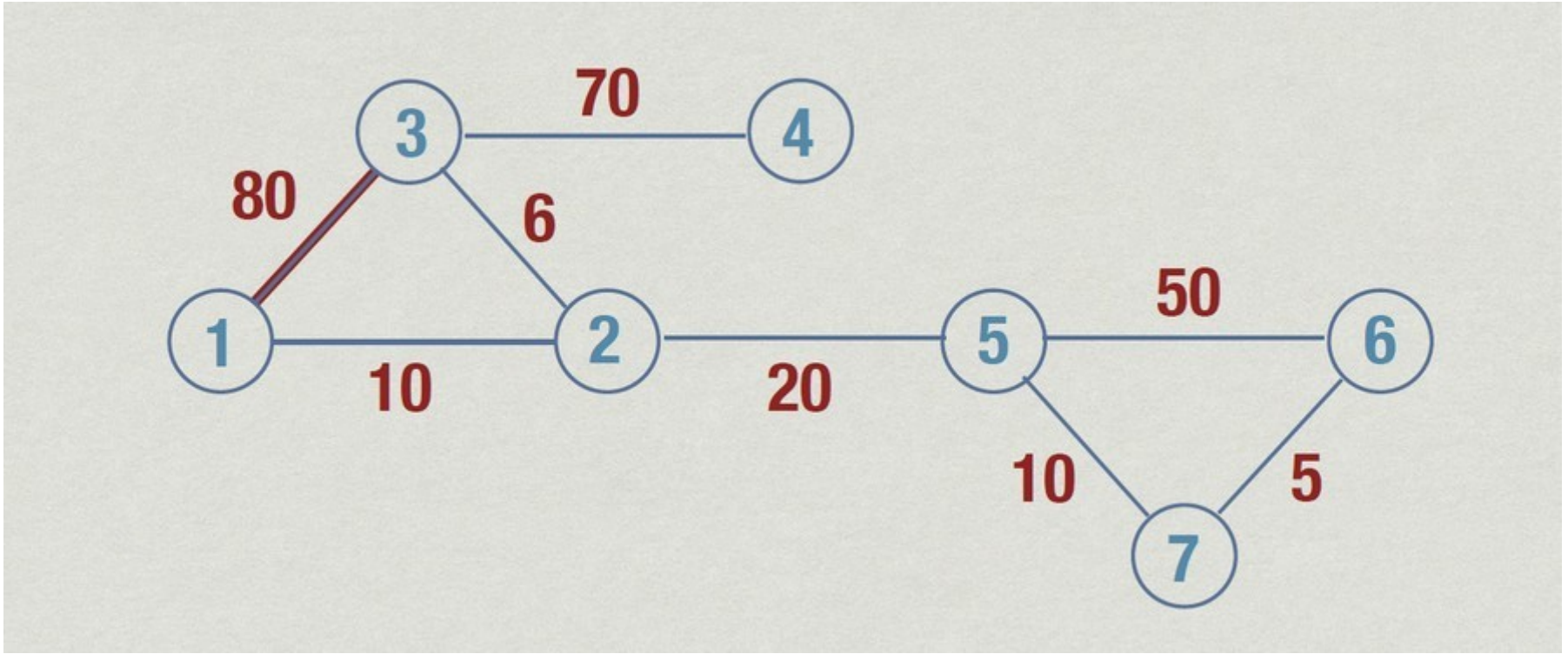
Does BFS find the shortest path?

- We are interested in finding the shortest paths to all nodes from a single source
- BFS finds path with the least number of edges
- May not be the shortest path in a weighted graph



Does BFS find the shortest path?

- BFS finds path with the least number of edges
- May not be the shortest path in a weighted graph



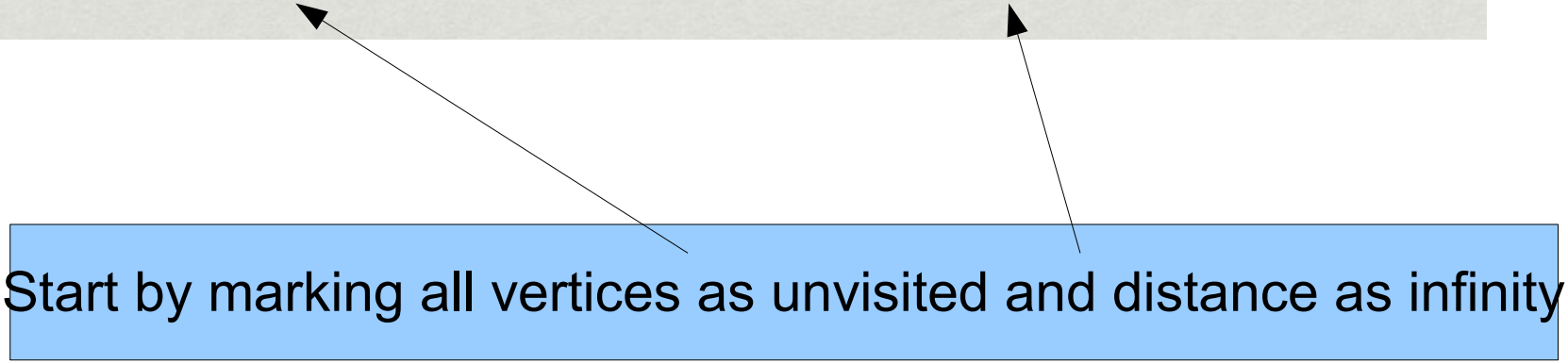
Shortest path from 1 to 3 is 1 → 2 → 3 (w = 16),
whereas BFS will pick 1 → 3 (w = 80)

Can we modify BFS to take care of weights?

- BFS picks all vertices at the next level from the current level (i.e. one edge away)
- We need to pick vertices that are the *shortest distance* away from the set of visited vertices
- Maintain a vector called distance[i] that records the weight of the shortest path found to i so far
- At each step we update the value of distance[i] by looking at neighbours of the visited vertices

Greedy algorithm

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

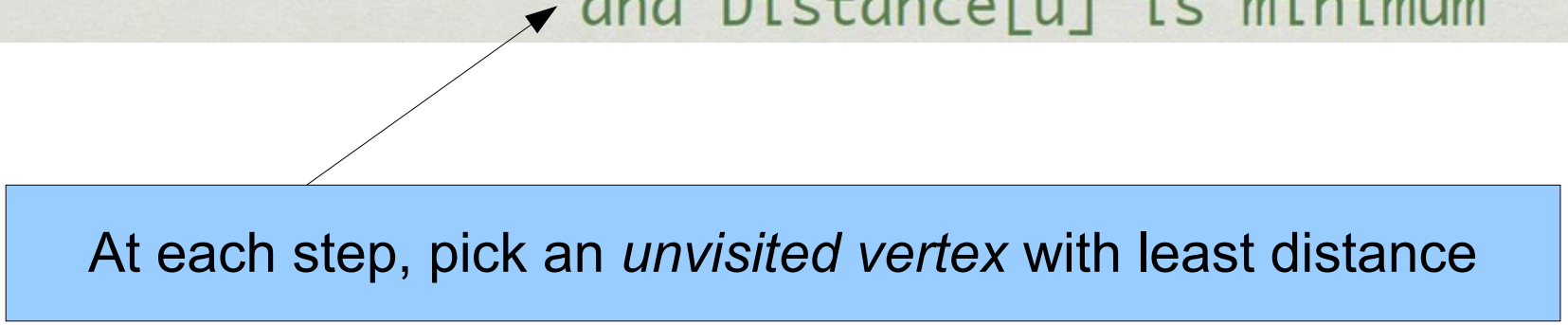


Greedy algorithm

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

  Distance[s] = 0
```

Greedy 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
```



Greedy 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
```

Include the vertex in the list of visited vertices

Greedy 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)
```

Update the distance value for all neighbouring vertices

Greedy 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)
```

Repeat this process n times

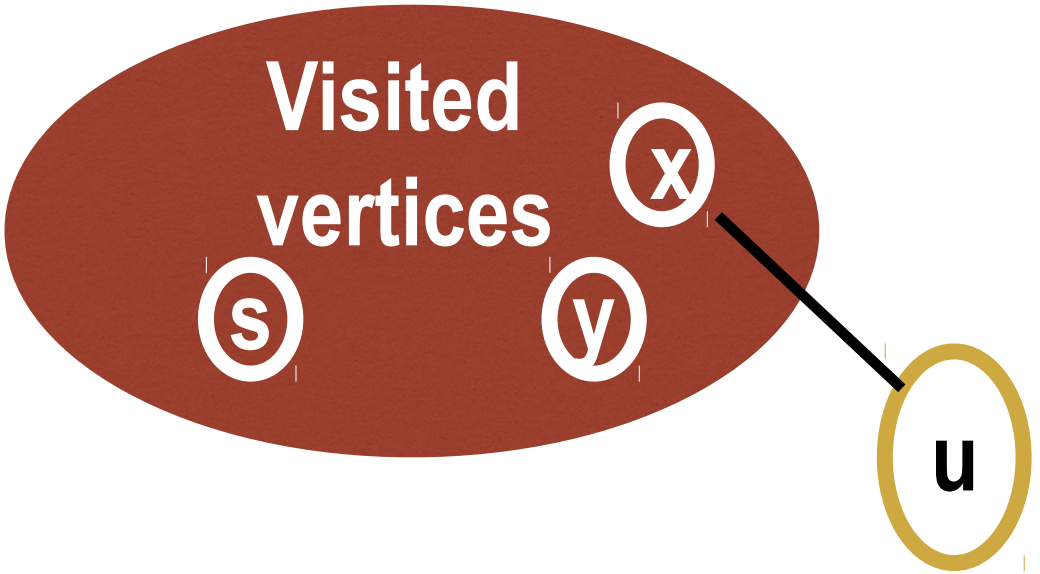
Dijkstra's algorithm

```
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)
```

How the greedy algorithm works

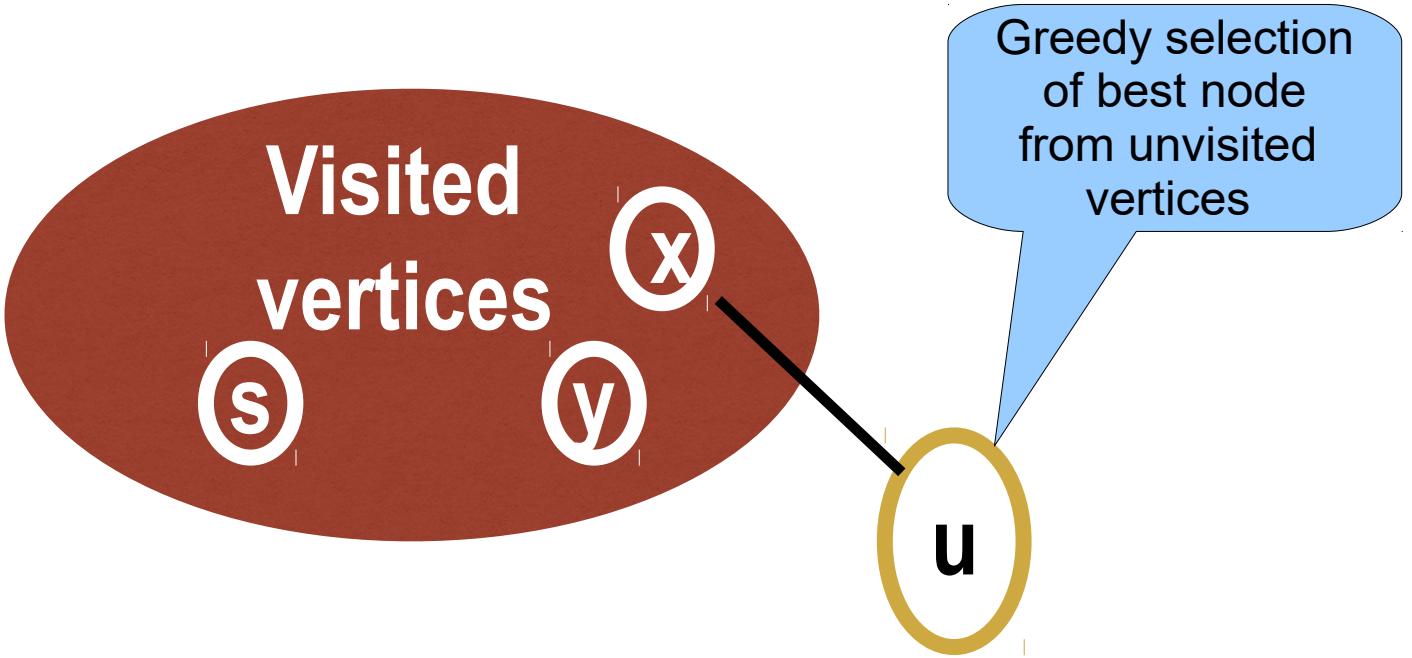
At any given point in time, the set of vertices is partitioned into two sets:
The set of visited vertices and the set of unvisited vertices.

How the greedy algorithm works



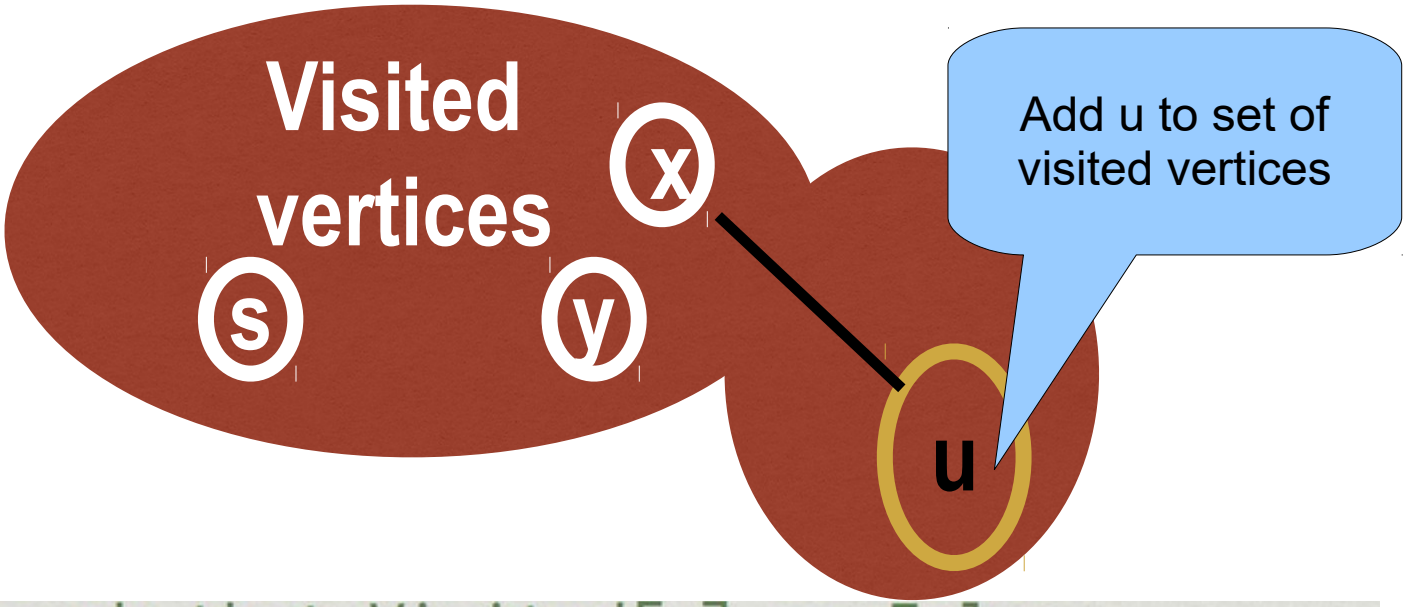

```
Choose u such that Visited[u] == False
                   and Distance[u] is minimum
```

How the greedy algorithm works




```
Choose u such that Visited[u] == False
                   and Distance[u] is minimum
```

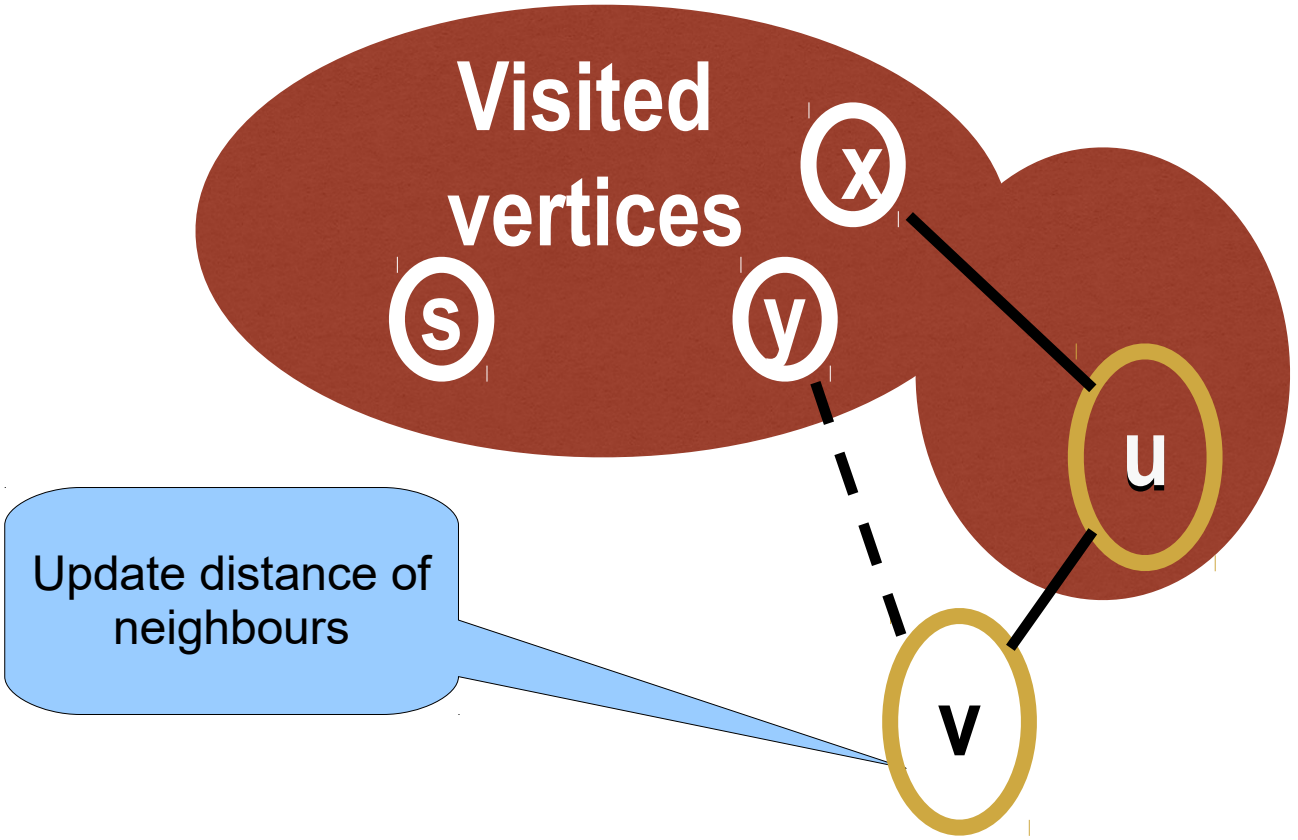
How the greedy algorithm works




```
Choose u such that Visited[u] == False
                   and Distance[u] is minimum
Visited[u] = True
```

Note that once a node u is added to the set of visited vertices, we will not come back to it again

How the greedy algorithm works




```
for each edge (u,v) with Visited[v] == False
  if Distance[v] > Distance[u] + weight(u,v)
    Distance[v] = Distance[u] + weight(u,v)
```

Complexity of Dijkstra's algorithm

Key steps in the algorithm

- Selecting an unvisited vertex u with least distance value
- Removing u from list of unvisited vertices
- Access all neighbours of u and update distance values. In particular, this will impact the first (selection) step

Complexity of these steps?

Key steps in the algorithm

- Selecting an unvisited vertex u with least distance value ... O(n) time for adjacency array or list without the use of any other data structure ... for n steps – this gives $O(n^2)$

Key steps in the algorithm

- Selecting an unvisited vertex u with least distance value ... O(n) time for adjacency array or list without the use of any other data structure ... for n steps – this gives $O(n^2)$
- Removing u from list of unvisited vertices ... O(1), just update visited array entry. For n steps, this is O(n)

Key steps in the algorithm

- Selecting an unvisited vertex u with least distance value ... O(n) time for adjacency array or list without the use of any other data structure ... for n steps – this gives $O(n^2)$
- Removing u from list of unvisited vertices ... O(1), just update visited array entry. For n steps, this is O(n)
- Access all neighbours of u and update distance values. In particular, this will impact the first (selection) step ... Each edge is visited twice (as in BFS and DFS), so O(m) across all steps

Key steps in the algorithm

- Selecting an unvisited vertex u with least distance value ... O(n) time for adjacency array or list without the use of any other data structure ... for n steps – this gives $O(n^2)$
- Removing u from list of unvisited vertices ... O(1), just update visited array entry. For n steps, this is O(n)
- Access all neighbours of u and update distance values. In particular, this will impact the first (selection) step ... Each edge is visited twice (as in BFS and DFS), so O(m) across all steps
- Overall $O(m + n^2) = O(n^2)$

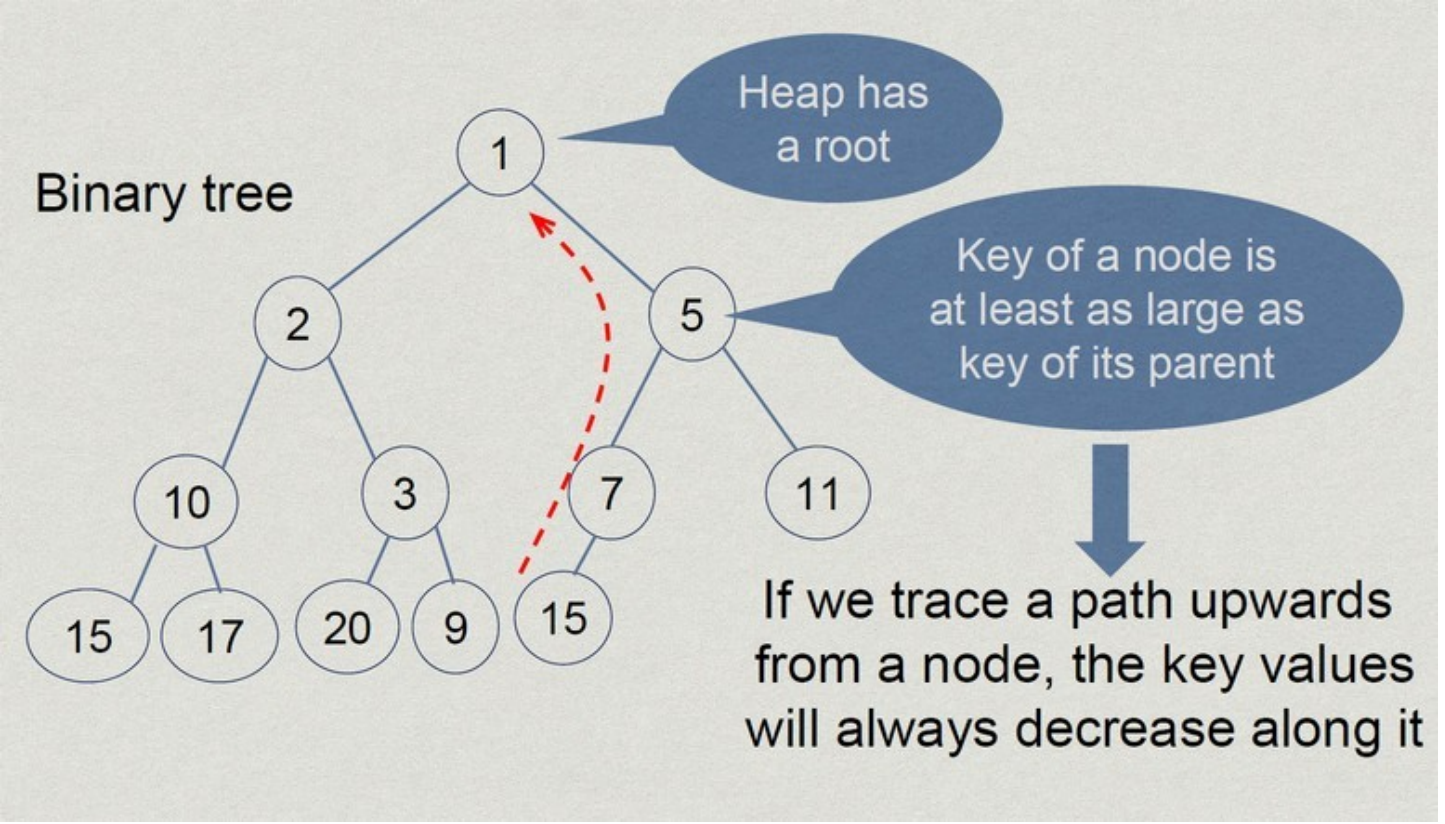
Using heaps to reduce complexity

We can use heaps to make it O(n log n)

- Selecting an unvisited vertex u with least distance value ... for n steps – this gives $O(n^2)$
  - Can be done in O(n) steps if we use priority queues (heaps). In heaps, finding min is O(1) time.

We can use heaps to make it O(n log n)

- Selecting an unvisited vertex u with least distance value ... for n steps – this gives $O(n^2)$
  - Can be done in O(n) steps if we use priority queues (heaps). In a minimum heap, finding min is O(1) time since the root will have the minimum value.



We can use heaps to make it O(n log n)

- Selecting an unvisited vertex u with least distance value ... for n steps – this gives $O(n^2)$
  - Can be done in O(n) steps if we use priority queues (heaps). In heaps, finding min is O(1) time.
- Removing u from the unvisited vertices – O(n)
  - If we use a heap, this will take O(log n) per step – so O(n log n) across all steps

We can use heaps to make it O(n log n)

- Selecting an unvisited vertex u with least distance value ... for n steps – this gives $O(n^2)$
  - Can be done in O(n) steps if we use priority queues (heaps). In heaps, finding min is O(1) time.
- Removing u from the unvisited vertices – O(n)
  - If we use a heap, this will take O(log n) per step – so O(n log n) across all steps
- Update distance values of neighbours of u:O(m)
  - Since distance value is updated, reconstruct the heap for each update – so O(m log n)

We can use heaps to make it O(n log n)

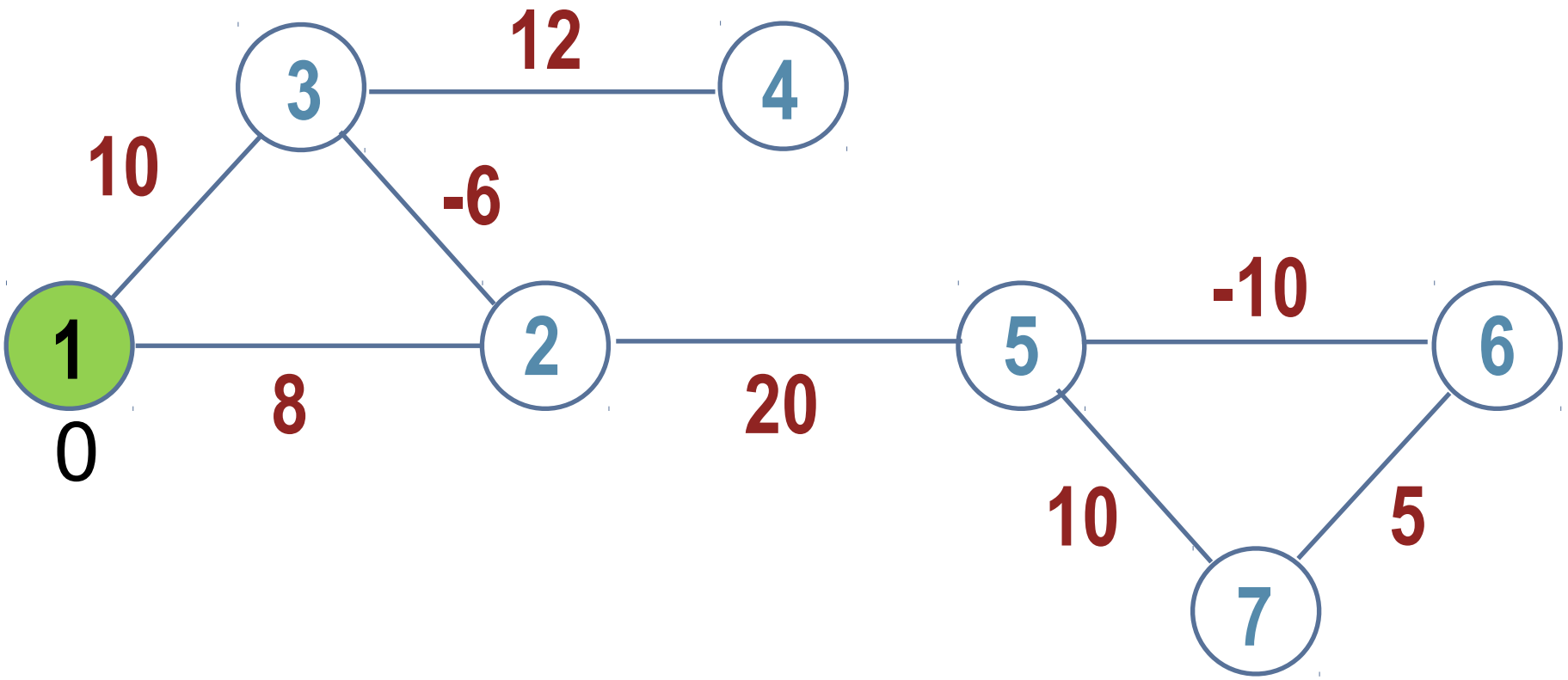
- Selecting an unvisited vertex u with least distance value ... for n steps – this gives $O(n^2)$
  - Can be done in O(n) steps if we use priority queues (heaps). In heaps, finding min is O(1) time.
- Removing u from the unvisited vertices – O(n)
  - If we use a heap, this will take O(log n) per step – so O(n log n) across all steps
- Update distance values of neighbours of u:O(m)
  - Since distance value is updated, reconstruct the heap for each update – so O(m log n)
- Overall O((m + n) log n).
  - If m is small (e.g. O(n)), then this is O(n log n).

Graphs with negative edge weights

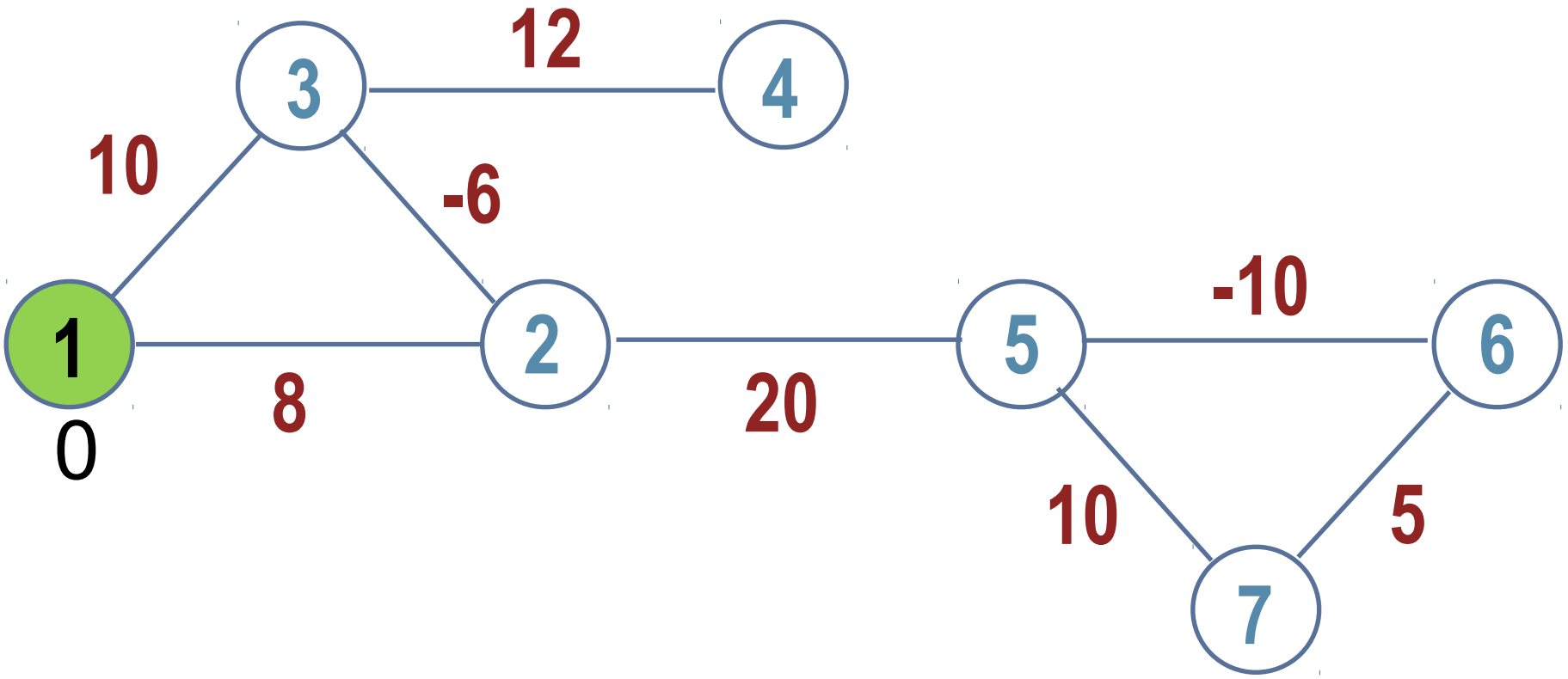
Graphs with negative edge weights

- Revisit ... A weighted graph is:
  - A graph G = (V,E) together with a ***weight function***
  - w: E → Reals
  - Which means that the weight w could be negative !
- An application of graphs with negative weights:
  - We want to model alternative financial plans that a company can implement. Nodes represent business stages. Edges represent state changes that arise from company's annual operations which could result in profits (positive edges) or losses (negative edges).

Graphs with negative edge weights

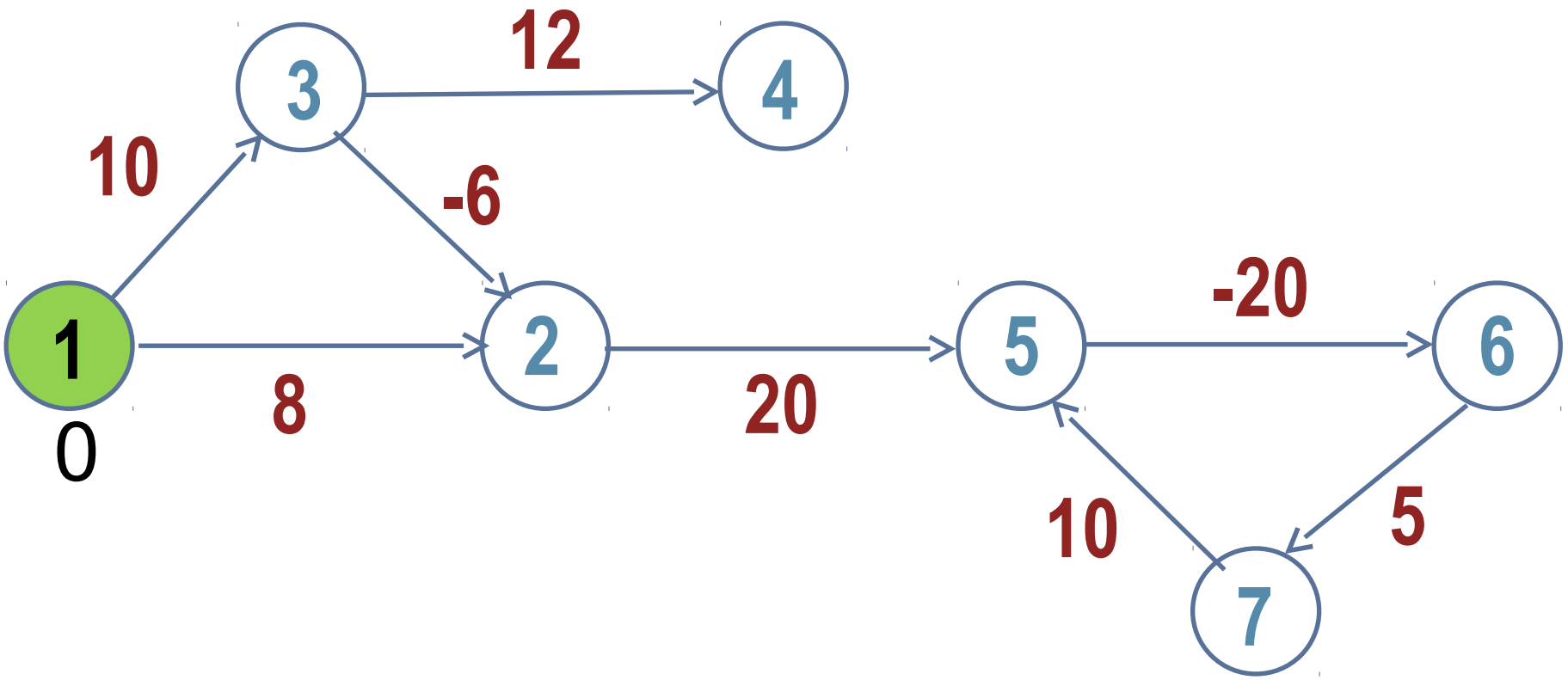


Graphs with negative edge weights



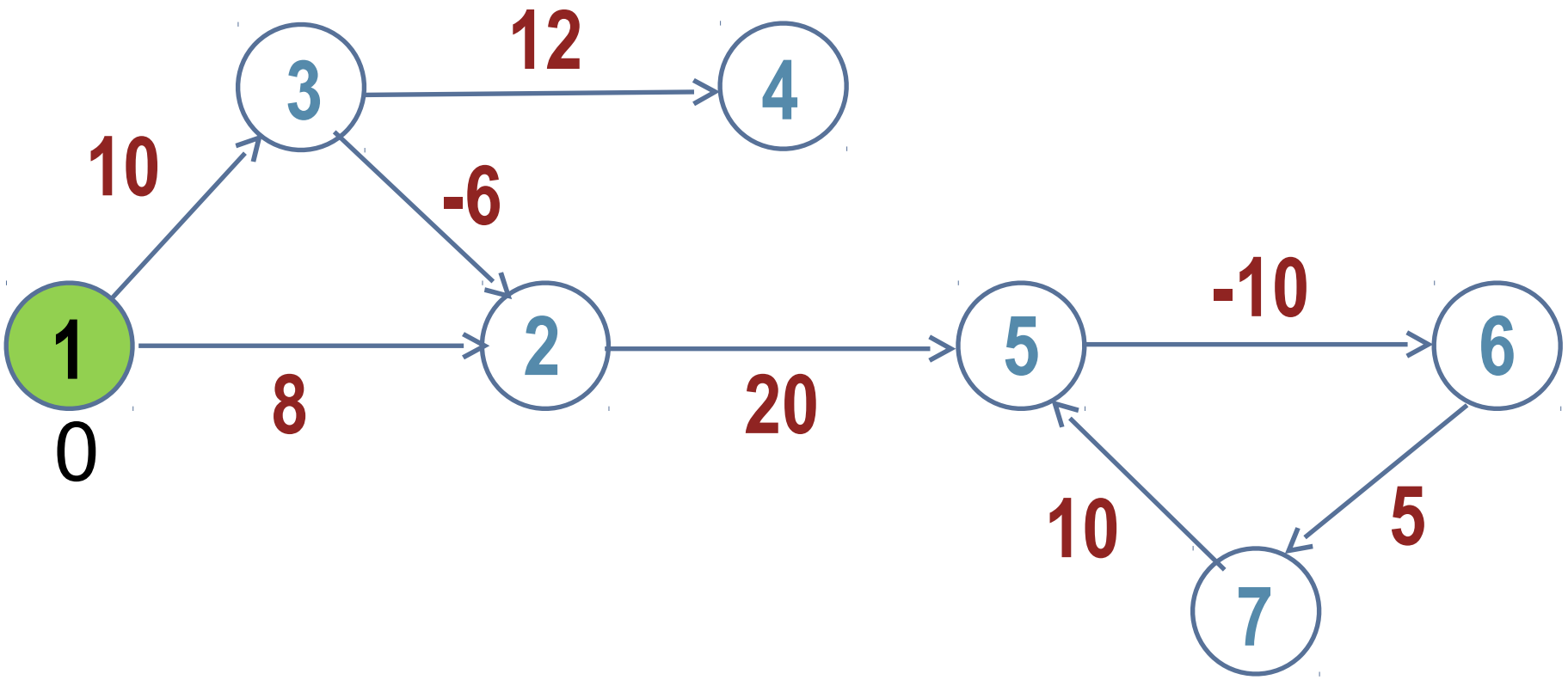
Negative edge weights do not make sense in an undirected graph:
We can go from 3 to 2 to 3 and this will reduce the weight of the path by 12
So any path that passes through 3 can be made smaller and smaller like this !!

What if the graph has negative cycles

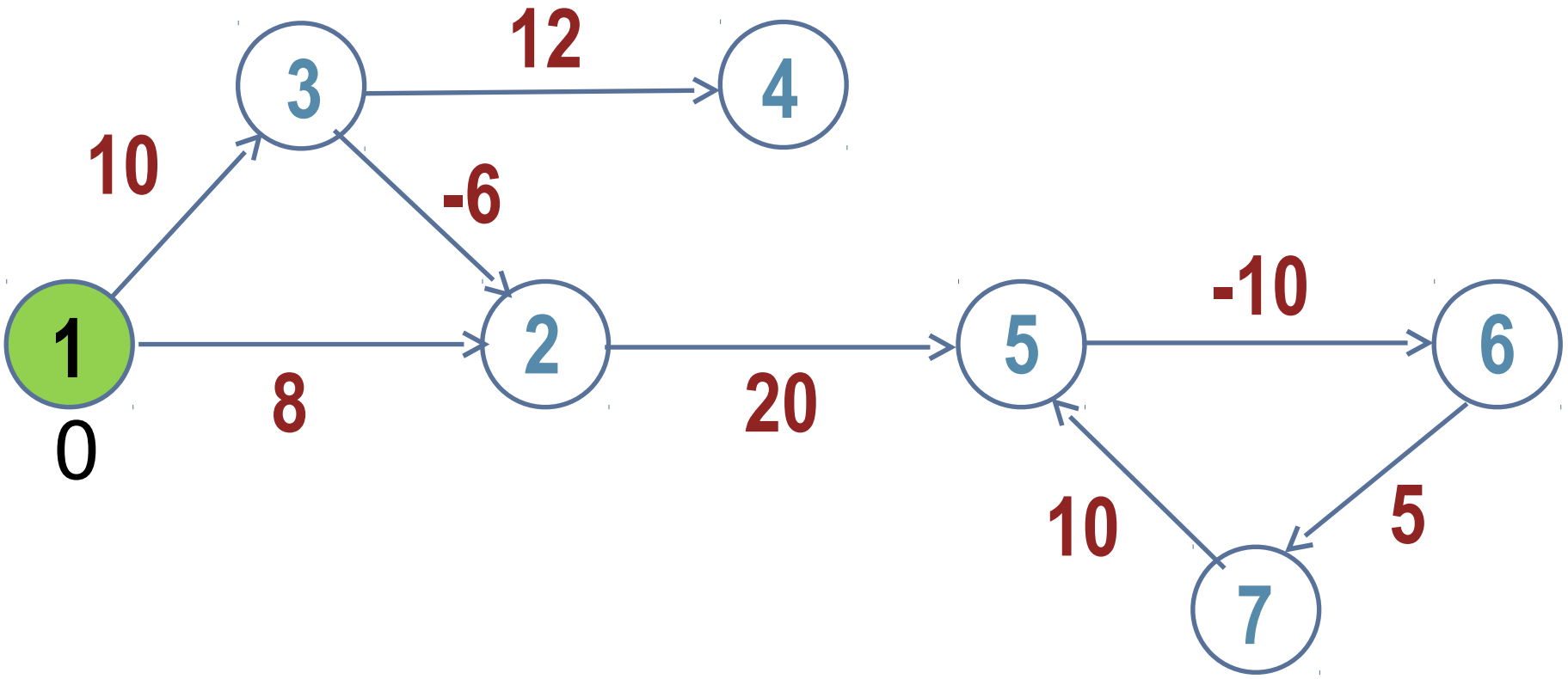


We can go around the cycle 5,6,7 and the weight will reduce by 5 !
Do it as many times as we want to keep getting
lower and lower weight paths to 5,6,7

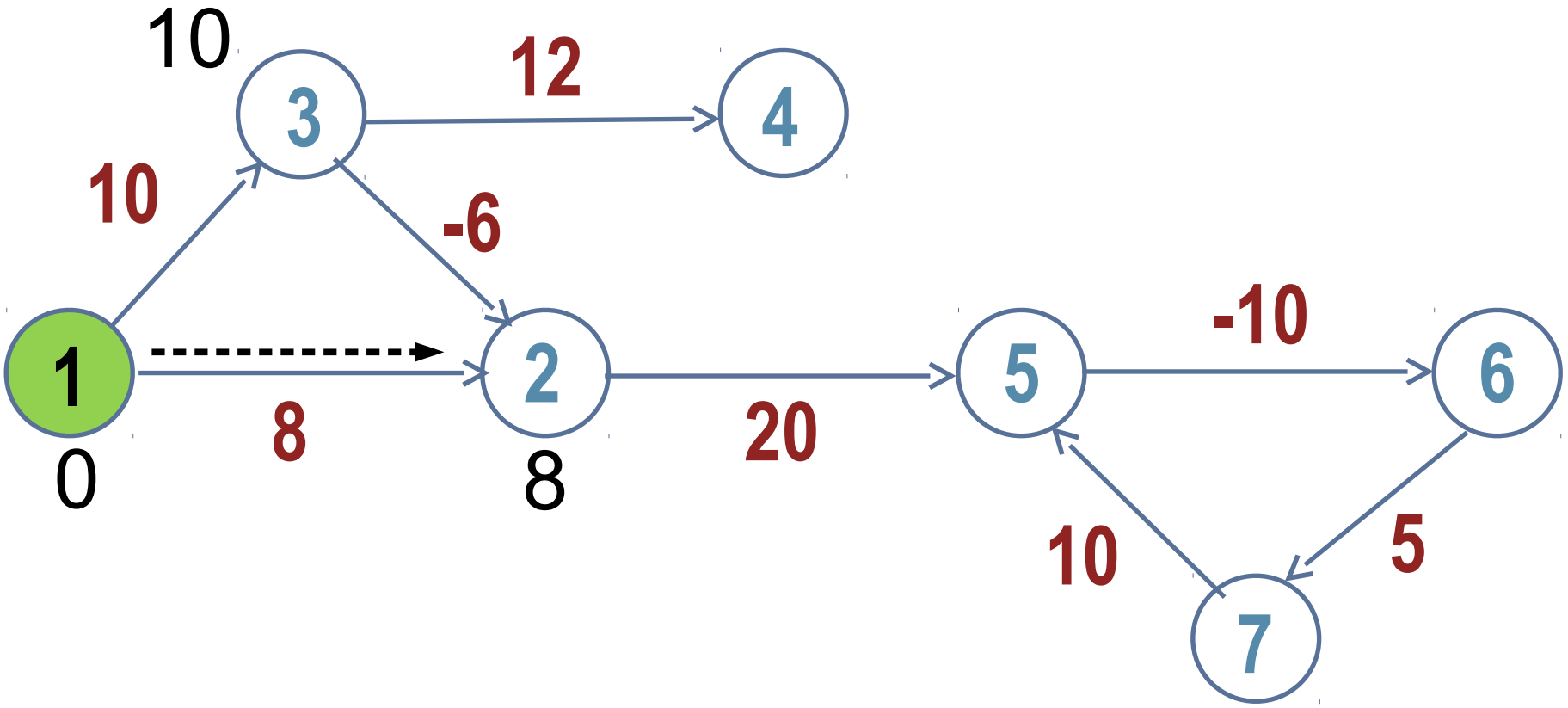
Meaningful only for directed graphs which have no negative cycles



Will Dijkstra's algorithm work?

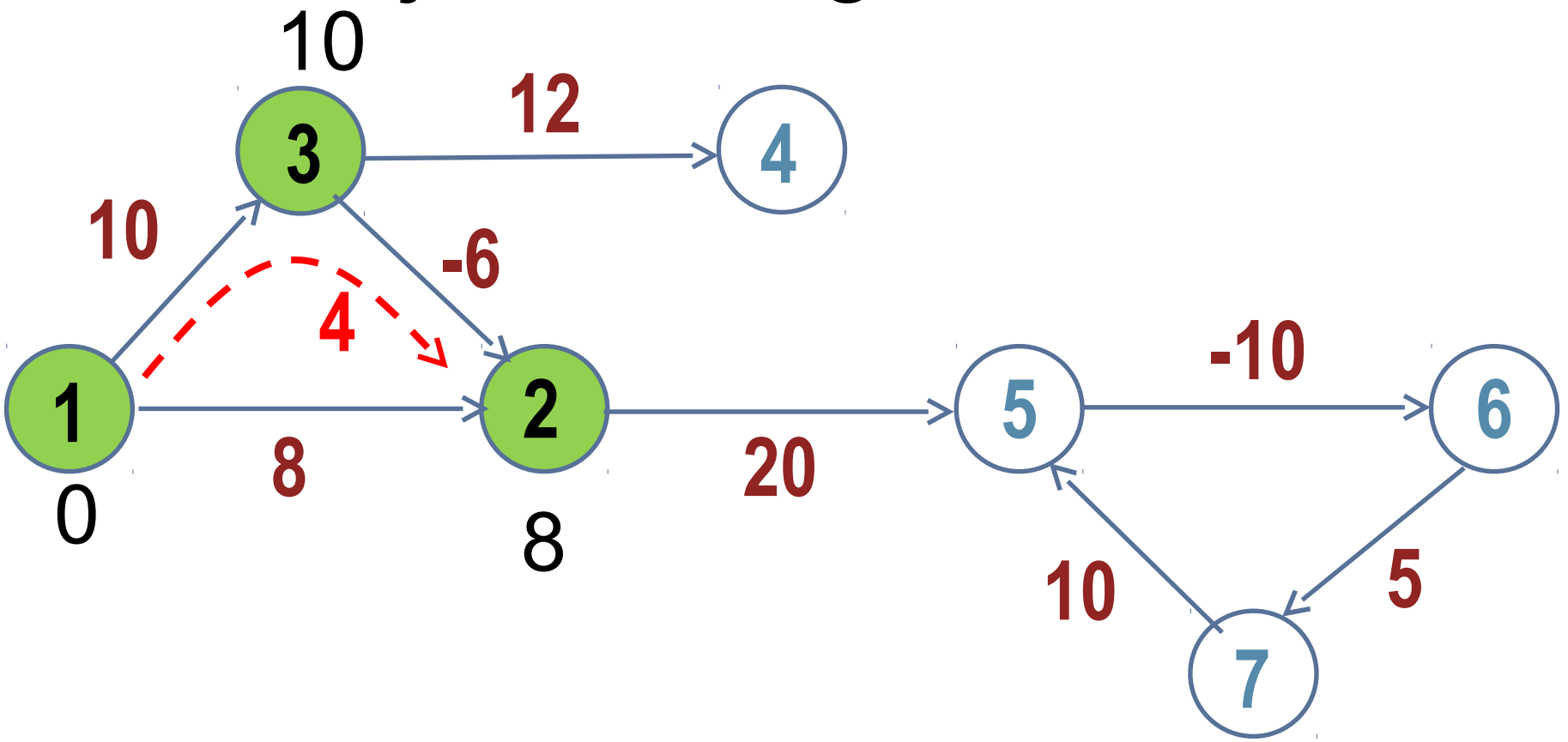


Will Dijkstra's algorithm work?



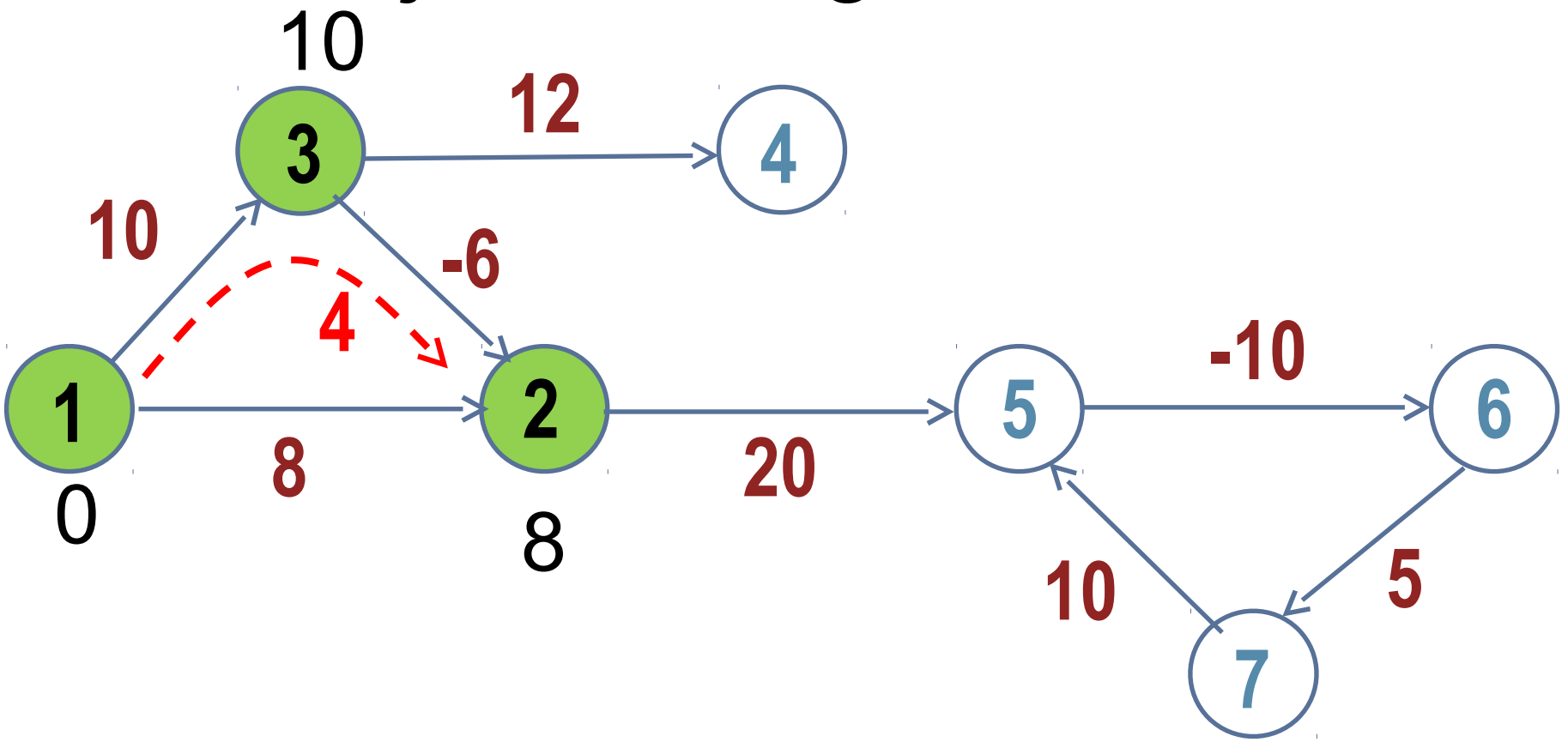
Dijkstra's algorithm picks node 2 first, since it has the shortest distance after distance updates from vertex 1 have been completed

Will Dijkstra's algorithm work?



If we had picked 3 first, we could have found a shorter path to 2 of weight 4

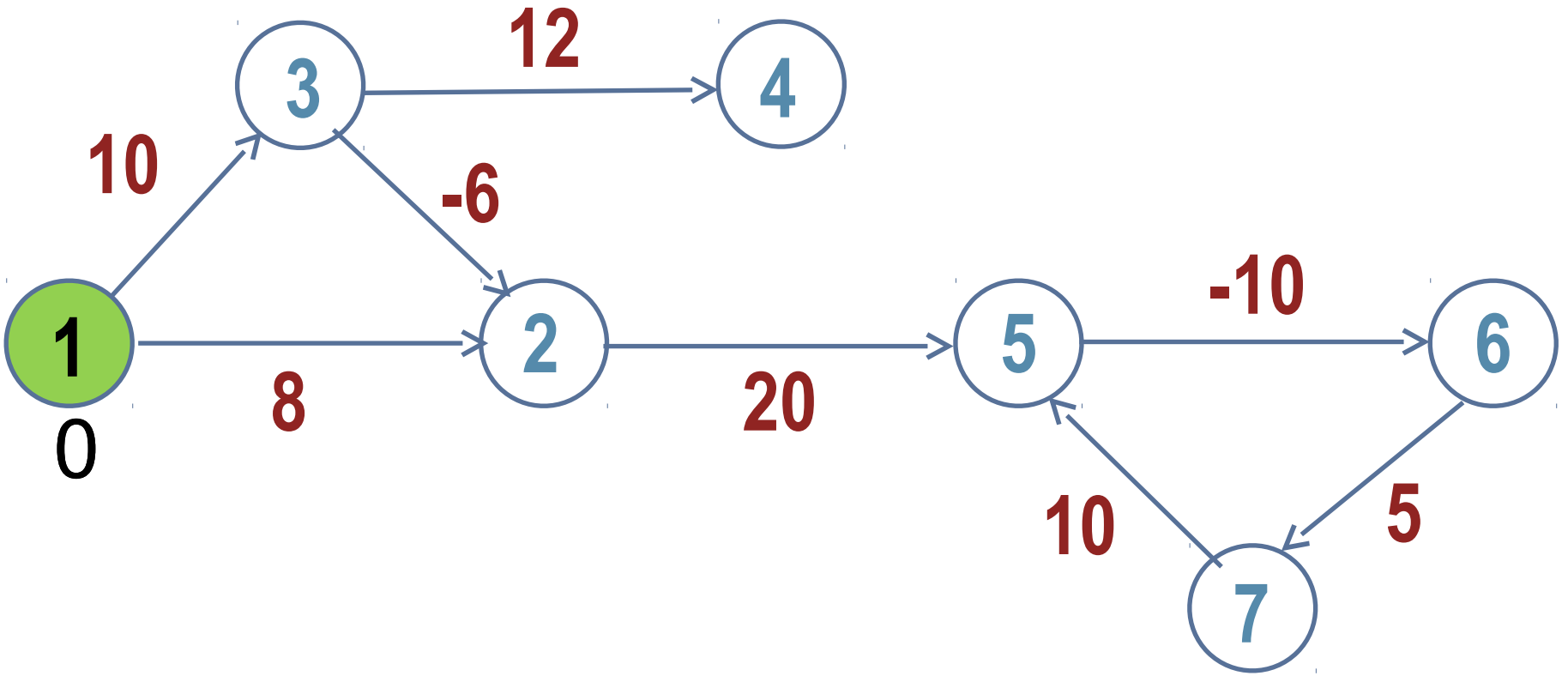
Will Dijkstra's algorithm work?



If we had picked 3 first, we could have found a shorter path to 2 of weight 4

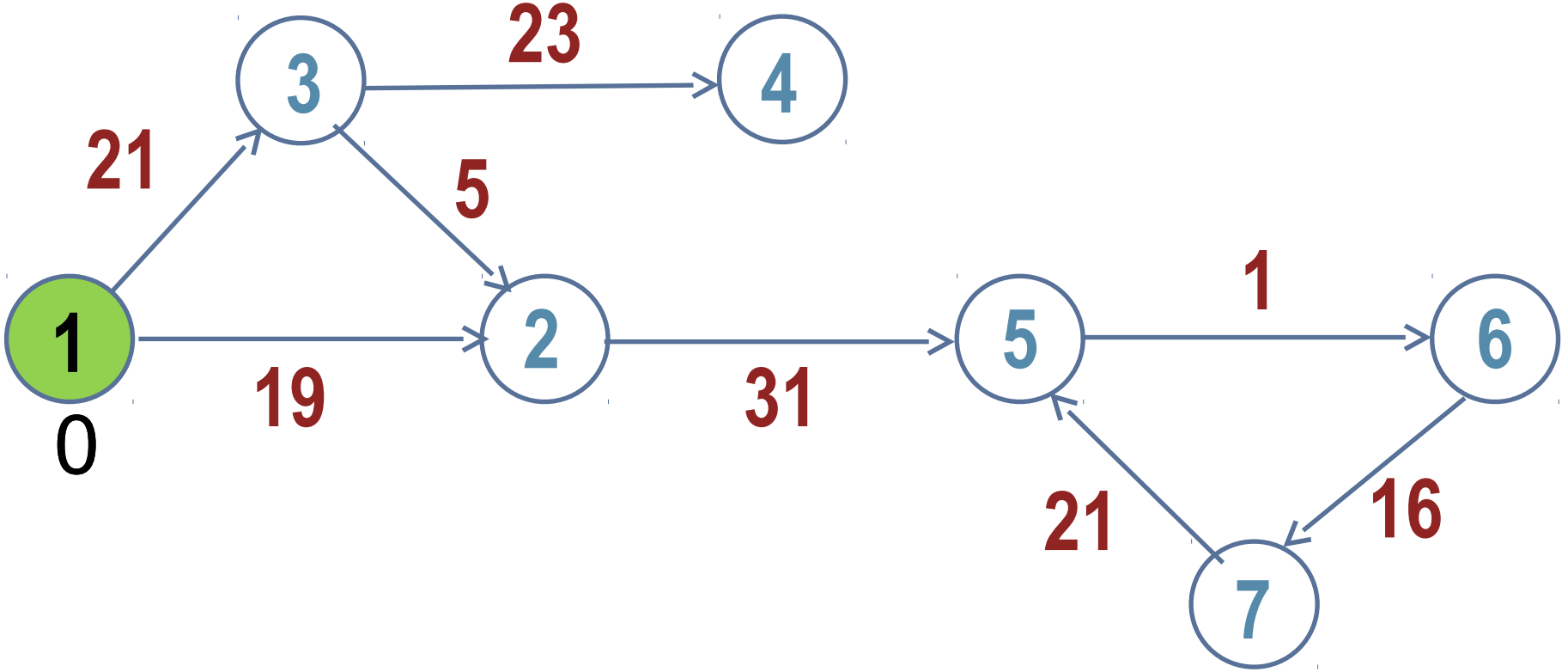
Too late, since Dijkstra's algorithm is not going to visit vertex 2 again !

Can we increase weights to make it positive? In this case add 11.



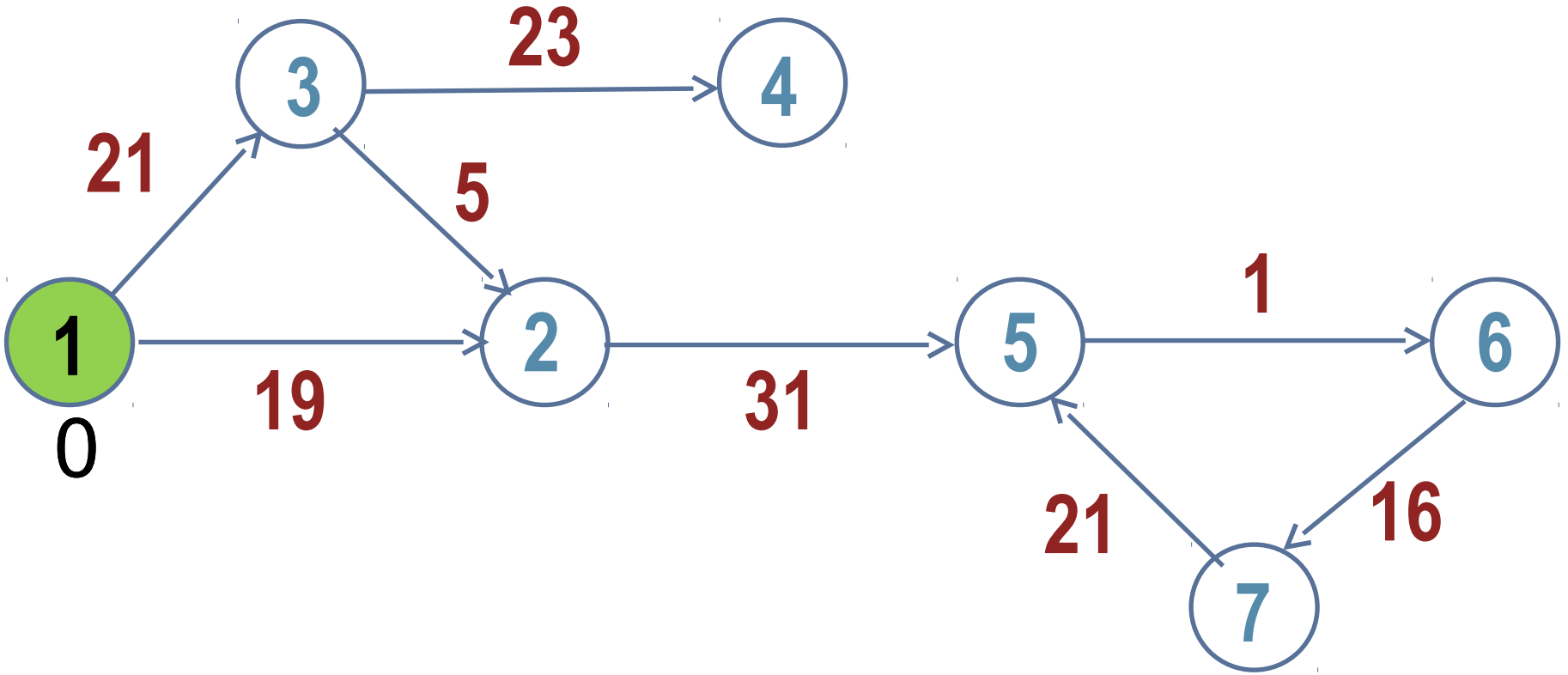
Adding the same value to all edges should not affect the problem?

Can we increase weights to make it positive? In this case add 11.



No negative edges – this should work now?

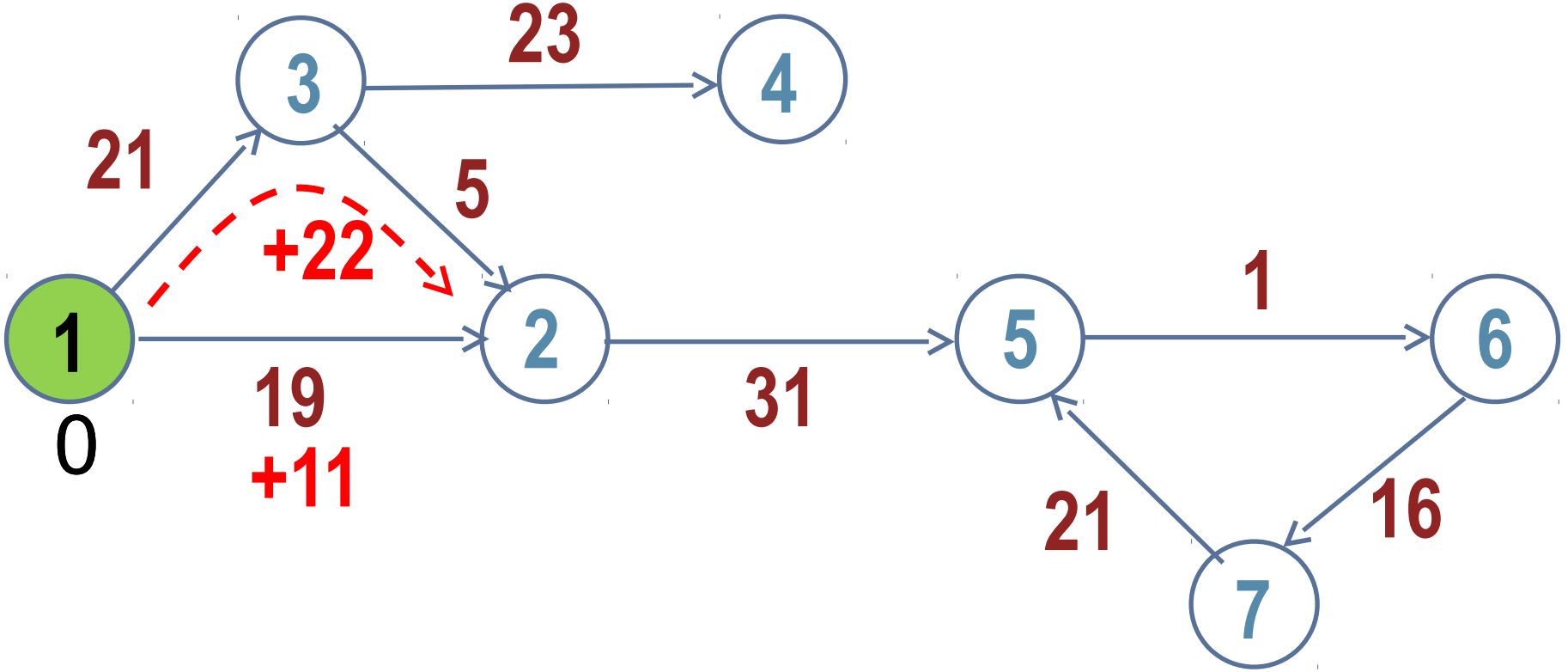
Can we increase weights to make it positive? In this case add 11.



No negative edges – this should work now?

But it doesn't – shortest path to 2 is 1,2 and not 1,3,2

Can we increase weights to make it positive? In this case add 11.



Path through 3 got penalised twice .. +22 while the one to 2 only 11

But it doesn't – shortest path to 2 is 1,2 and not 1,3,2

Bellman-Ford algorithm

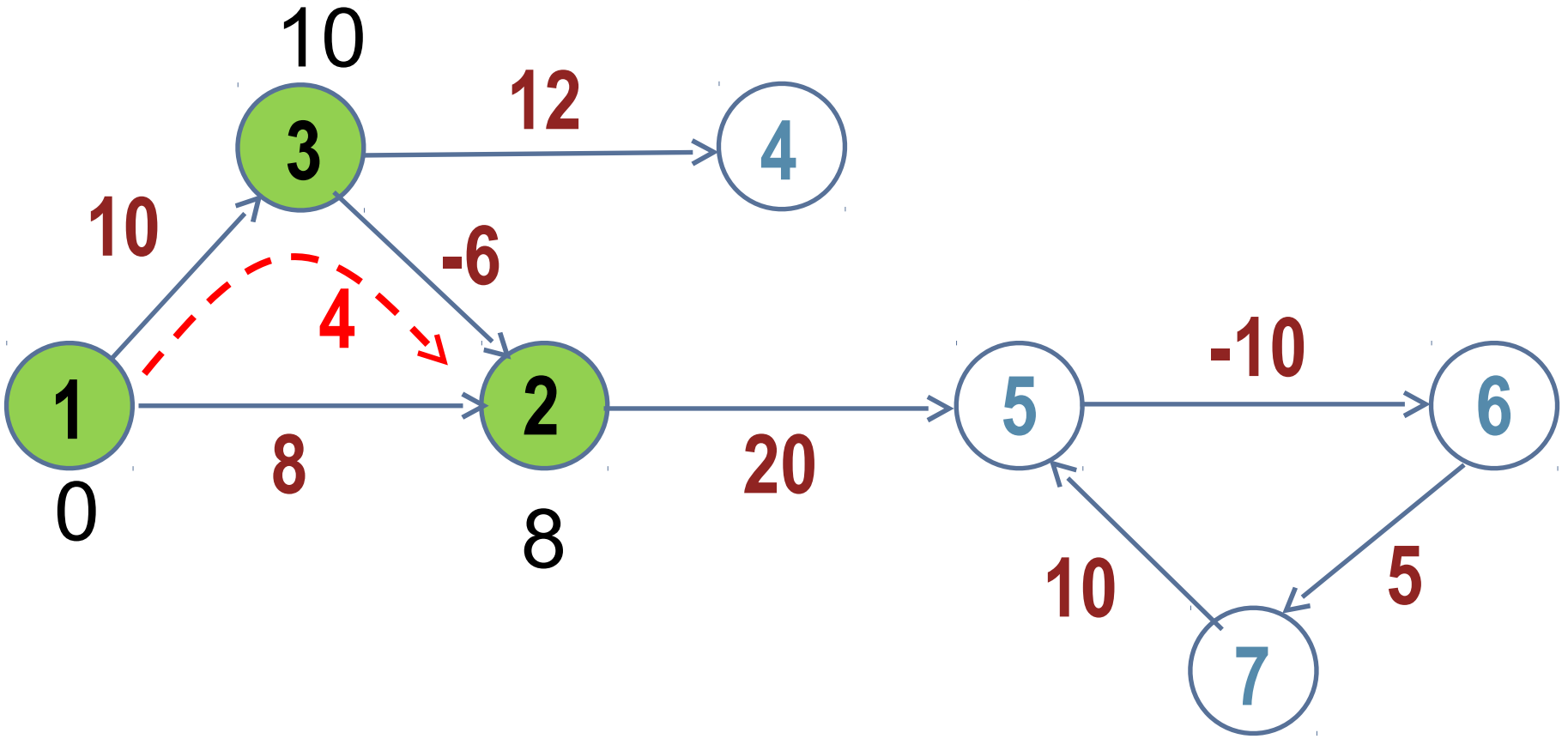
If we need to revisit nodes, can we try dynamic programming

- Dynamic programming
  - Breaks a problem into subproblems and writes the solution of the problem in terms of the solution of subproblems
  - It remembers the subproblems whose solutions have been found, so that they do not have to repeated again

Trying dynamic programming

- What is the problem?
  - Find the shortest path from source s to vertex t
- What are the sub-problems?
  - Sub problem k: Find the shortest path from source s to vertex t *which has at most k edges.*
  - We need only consider values of k upto n – so there are only n sub-problems
- How is the problem related to the sub-problems?
  - Shortest path from s to t can be obtained by taking the min of all the n sub-problems

Trying Dynamic Programming



Sub-problem 1: Shortest path from node 2 to 3 with 1 edge has weight 8

Sub-problem 2: Shortest path from node 2 to 3 with 2 edges has weight 4

Min of sub-problems yields 4 which is the right answer

Recursive formulation

- What happens when we go from k-1 to k edges?
- Either the path with k-1 edges from s to t was already optimal
- Or else we found a better path with k edges from s to t
  - If so, then take the vertex preceding t in this new path – say it is v.
  - Path from s to v is has k-1 edges and so should have been explored when we did the k-1 subproblem

Bellman-Ford algorithm

```
SP(s,t,k) {
    if k == 0:
        if s == t return 0 else return infinity;
    else:
       m = SP(s,t,k-1)
       for all edges (w,t) do:
          m = min(m, SP(s,w,k-1) + weight(w,t))
       end for
       return m
     end if
}
```

Bellman-Ford algorithm

```
SP(s,t,k) {
    if k == 0:
        if s == t return 0 else return infinity;
    else:
       m = SP(s,t,k-1)
       for all edges (w,t) do:
          m = min(m, SP(s,w,k-1) + weight(w,t))
       end for
       return m
     end if
}
```

Base case: Only path of length 0 is when t=s

Bellman-Ford algorithm

```
SP(s,t,k) {
    if k == 0:
        if s == t return 0 else return infinity;
    else:
        m = SP(s,t,k-1)
        for all edges (w,t) do:
            m = min(m, SP(s,w,k-1) + weight(w,t))
        end for
        return m
     end if
}
```

Start with the best solution found with k-1 edges

Bellman-Ford algorithm

```
SP(s,t,k) {
    if k == 0:
        if s == t return 0 else return infinity;
    else:
       m = SP(s,t,k-1)
       for all edges (w,t) do:
          m = min(m, SP(s,v,k-1) + weight(v,t))
       end for
       return m
     end if
}
```

If possible improve this solution
by looking at k edge paths
whose first k-1 edges is a path from s to v
and v is a neighbour of t

Bottom-up (non recursive) formulation of Bellman-Ford

```
// Maintain the optimal value in an array O[t,k]
// O[s,k] = 0 for all k and O[t,0] =  for all t

SP() {
 for k = 1 to n do:
    for all edges (v,t) do:
       O[t, k] = min (O[t,k-1], O[v,k-1] + weight(v,t) )
    end for
 end for
}
```

Complexity of Bellman-Ford

```
// Maintain the optimal value in an array O[t,k]
// O[s,k] = 0 for all k and O[t,0] =  for all t

SP() {
 for k = 1 to n do:
    for all edges (v,t) do:
       O[t, k] = min (O[t,k-1], O[v,k-1] + weight(v,t) )
    end for
 end for
}
```

Complexity is O(m n), since outer loop iterates
n times and inner for all could take O(m) worst case time

We can also detect negative cycles

```
// Maintain the optimal value in an array O[t,k]
// O[s,k] = 0 for all k and O[t,0] =  for all t

SP() {
 for k = 1 to n do:
    for all edges (v,t) do:
       O[t, k] = min (O[t,k-1], O[v,k-1] + weight(v,t) )
    end for
 end for
}
```

After n steps, the best path to a vertex should always be found.
If it has not been found, it means that there are negative cycles !

If after algorithm stops, if there is a vertex t and its neighbour v
such that $O[v,n] + weight(v,t) < O[t,n]$ then the graph has negative cycles

Constraint satisfaction

Bellman-Ford can be used for constraint satisfaction

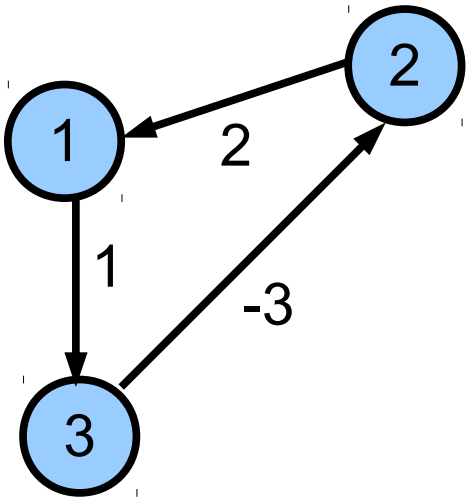
- Given a set of variables $x_1, x_2, \ldots, x_n$, a difference constraint is an equation of the form $x_i - x_j \leq c_{ij}$, where c_{ij} can be any real (can be negative)
- Difference constraints occur in many places – for instance in compilers, during VLSI architecture design etc
- Is there an easy way to check if a set of constraints has a solution and if so to find one?

Constraint satisfaction example

- Set of constraints:
  - $x_1 \leq x_2 + 2$
  - $x_2 + 3 \leq x_3$
  - $x_3 \leq x_1 + 1$
- Writing it in standard form:
  - $x_1 - x_2 \leq 2$
  - $x_2 - x_3 \leq -3$
  - $x_3 - x_1 \leq 1$

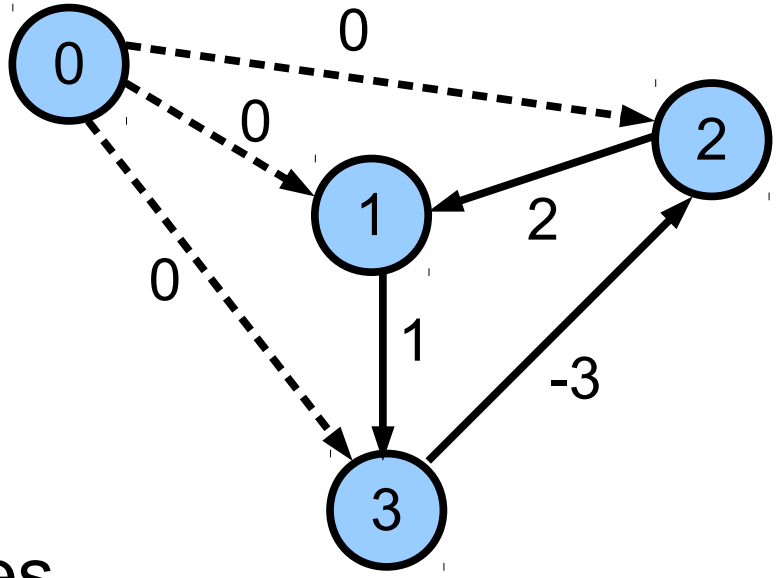
Constraint satisfaction example

- Set of constraints:
  - $x_1 - x_2 \leq 2$
  - $x_2 - x_3 \leq -3$
  - $x_3 - x_1 \leq 1$
- Construct a graph
  - vertices i
  - Edge from j to i if there is a constraint $x_i - x_j \leq c_{ij}$
  - weight c_{ij} on the edge



Constraint satisfaction example

- Set of constraints:
  - $x_1 - x_2 \leq 2$
  - $x_2 - x_3 \leq -3$
  - $x_3 - x_1 \leq 1$
- Construct a graph
  - Add a vertex 0 and edges from 0 to all vertices with weight 0
  - Interpret x_i as shortest path from 0 to i

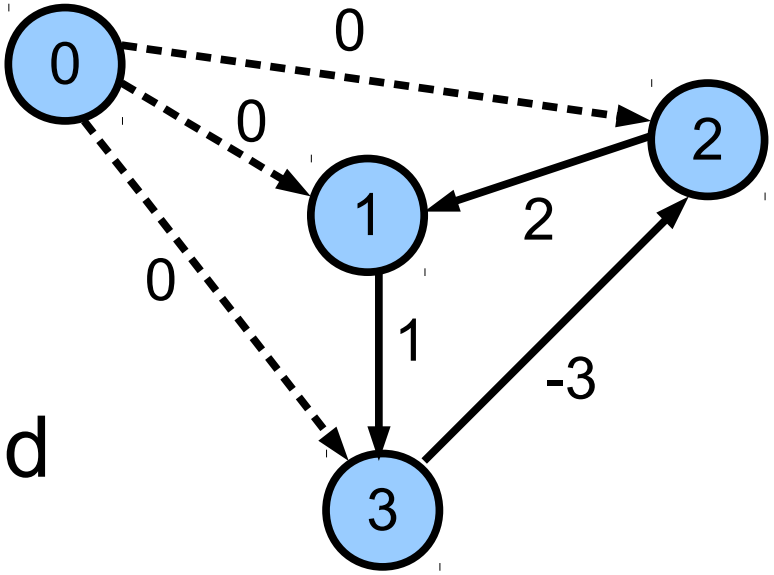


- $x_1 \leq x_2 + 2$

(Dist of shortest path to 1) ≤
(Dist of shortest path to 2) + 2

Constraint satisfaction example

- Set of constraints:
  - $x_1 - x_2 \leq 2$
  - $x_2 - x_3 \leq -3$
  - $x_3 - x_1 \leq 1$
- Use Bellman-Ford to find shortest paths
  - $x_1 = -1$
  - $x_2 = -3$
  - $x_3 = 0$



Problem: Solve

- $x_1 \leq x_2$
- $x_2 \leq x_5 + 1$
- $x_4 \leq x_1 + 4$
- $x_5 \leq x_3 - 3$
- $x_1 \leq x_5 - 1$
- $x_3 \leq x_1 + 5$
- $x_4 \leq x_3 - 1$
- $x_5 \leq x_4 - 3$

Minimum Spanning Trees (MST)

Minimum Spanning Trees

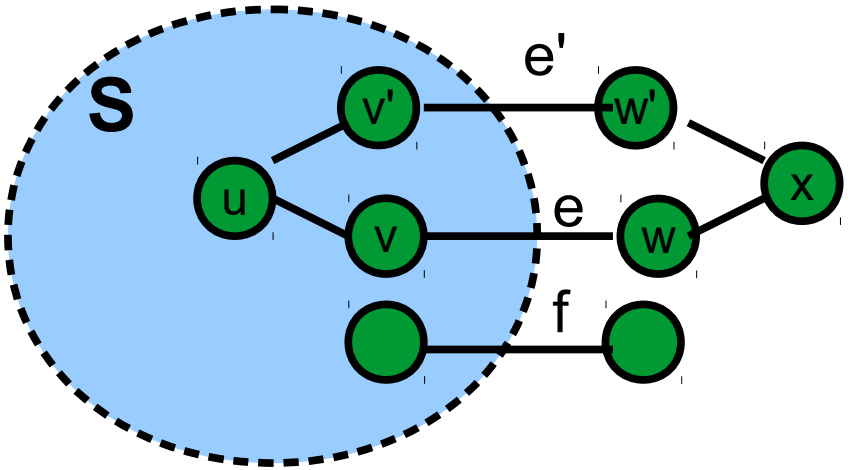
- Given a set of locations V = $\{v_1, v_2, v_3, \ldots v_n\}$, we want to connect all these locations with a network.
  - $c(v_i, v_j)$ is the cost of laying a link between v_i and v_j
  - We could make a complete graph G with vertices V (E = VxV), with $c(v_i, v_j) > 0$ being the weight of (v_i, v_j).
- A ***MST*** is a subset $T \subseteq E$ such that:
  - (V,T) is connected
  - $\sum_{e \in E} c(e)$ is minimum
- Note that T is a tree due to the second condition

Kruskal and Prim algorithms

- Kruskal:
  - Start with an empty set of edges
  - Add edges one by one in increasing order of cost
  - Discard an edge if it results in a cycle
- Prim:
  - Start with a root node s, S = {s}
  - Use greedy (Dijkstra's), but pick vertex j with least *attachment* cost (c_{ij}, $i \in S$, $j \notin S$ is minimum)
- Backward Kruskal:
  - Start with all the edges
  - Remove edges in decreasing order of cost
  - Put the edge back if it disconnects the graph

Why does Kruskal work?

- Cut property
  - Let S $\subset$ V and
  - X = { (v,w) | v $\in$S and w $\in$V – S}
  - Let e be a min cost edge in X
  - Then every MST will contain e



Say MST does not contain e
But contains e' and f.
We cannot replace f by e
(since it will create a cycle)
But we can replace e' by e

Such an e' will always exist, since MST is connected and should have a path from u to x

The End