CS2020
Data Structures and Algorithms

Shortest Paths

Roadmap

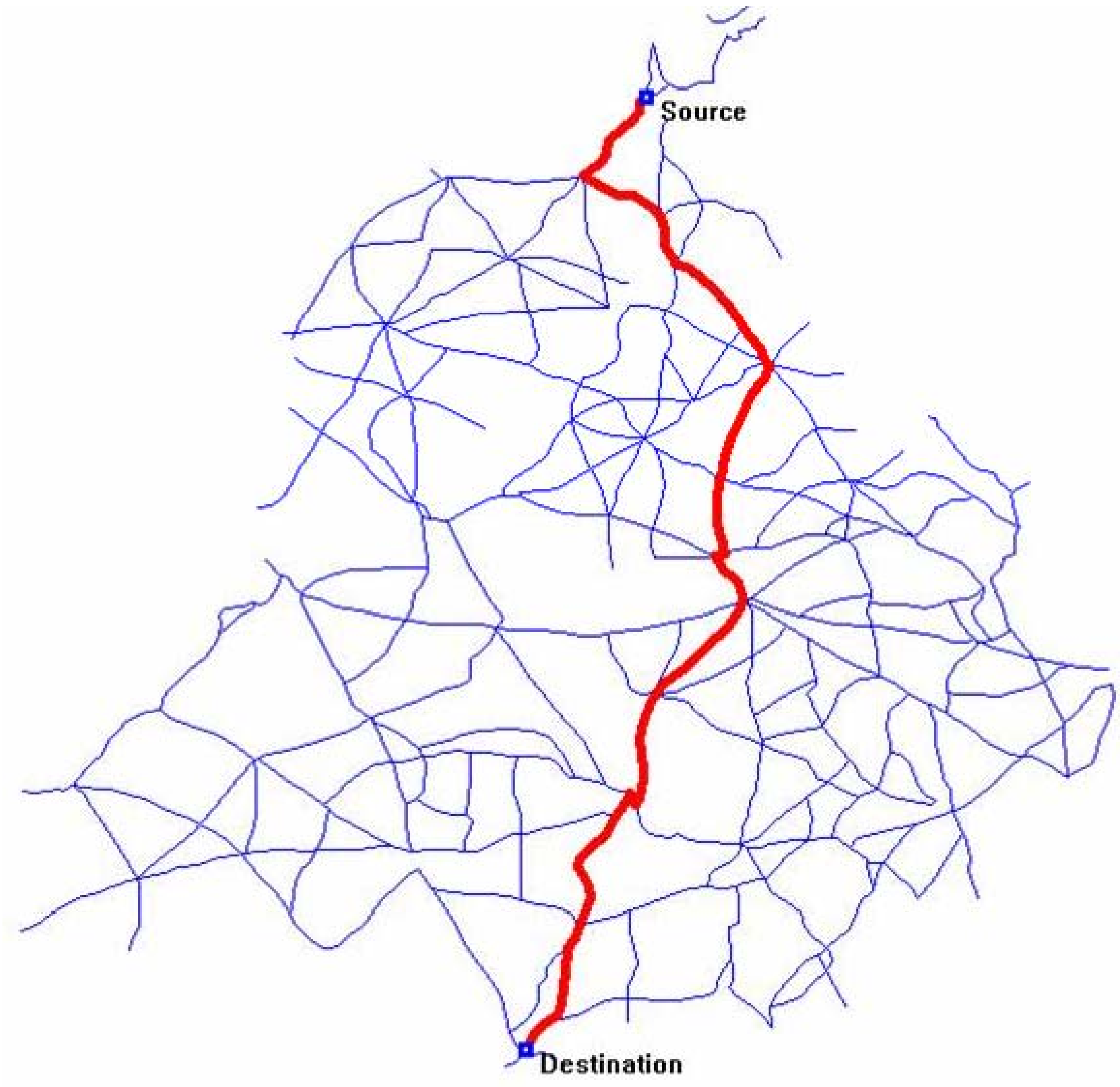
Part I: Shortest Paths

- Special Case: Tree
- Special Case: Non-negative weights (Dijkstra’s)
- Special Case: Directed Acyclic Graphs

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

SHORTEST PATHS
(ON WEIGHTED GRAPHS)



Shortest Path Problem

Basic question: find the shortest path!

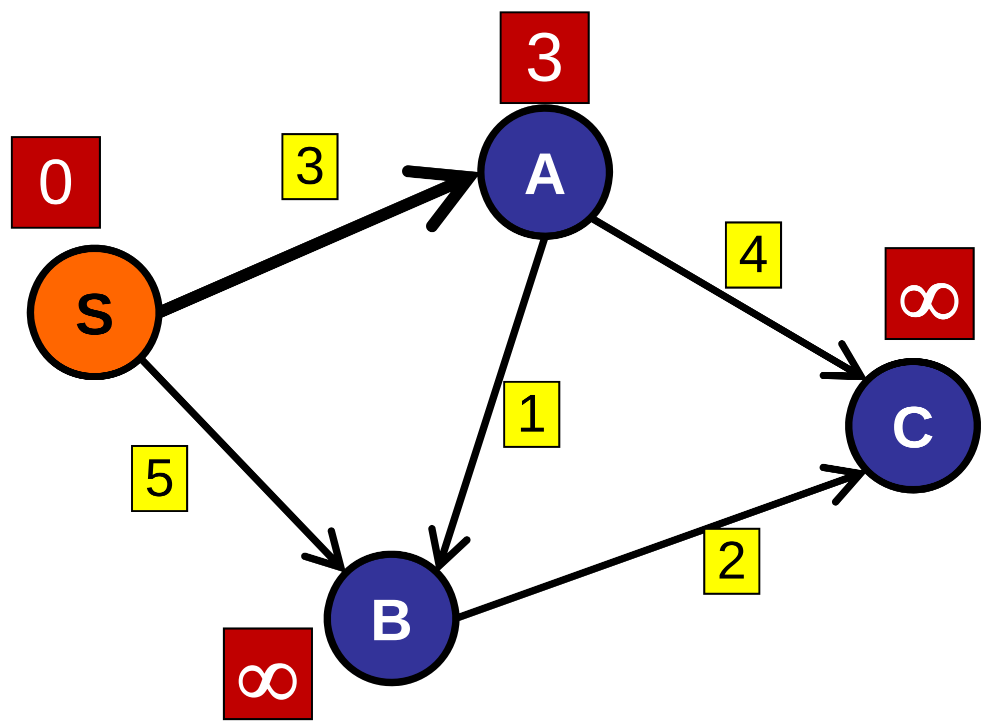
- Source-to-destination: one vertex to another
- Single source: one vertex to every other
- All pairs: between all pairs of vertices

Variants:

- Edge weights: non-negative, arbitrary, Euclidean, ...
- Cycles: cyclic, acyclic, no negative cycles

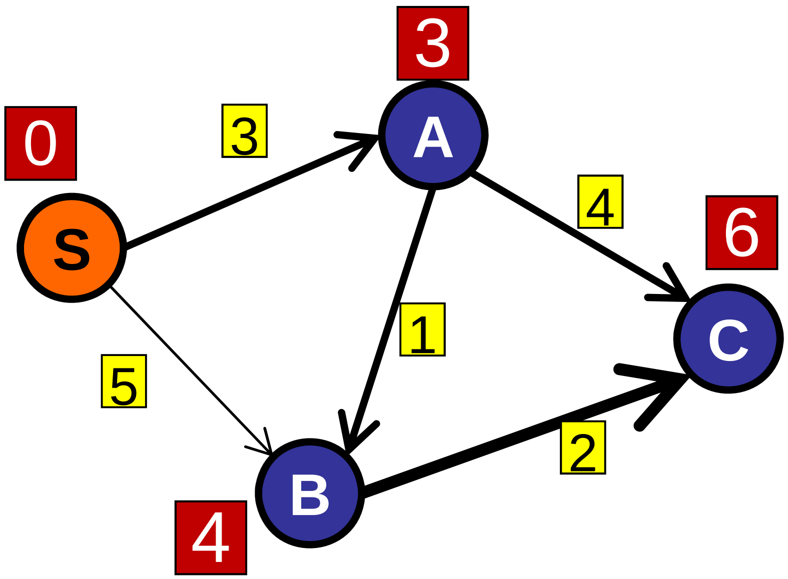
Shortest Paths

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```



Bellman-Ford

```
n = V.length;
for (i=0; i<n; i++)
    for (Edge e : graph)
            relax(e)
```

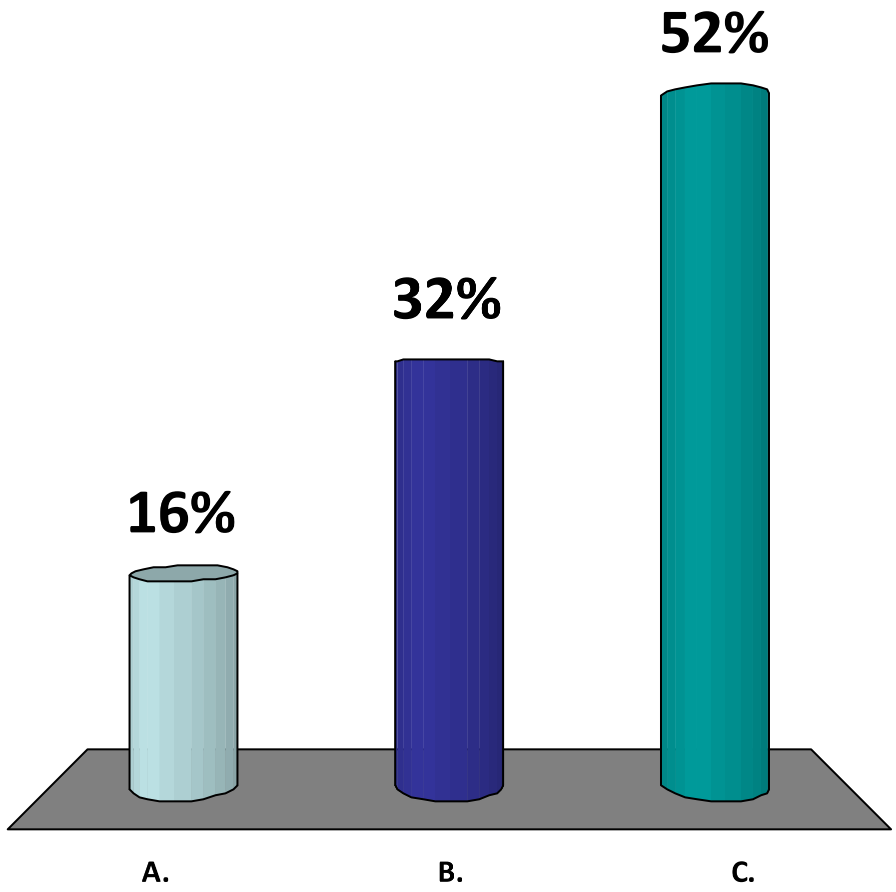


What is the meaning of negative cycles in SSSP?

A. Use Bellman-Ford in the last lecture

B. We will talk about how to solve today

C. The meaning of negative cycles will make the SSSP meaningless



Bellman-Ford Summary

Basic idea:

- Repeat |V| times: relax every edge
- Stop when “converges”.
- O(VE) time.

Special issues:

- If negative weight-cycle: impossible.
- Use Bellman-Ford to **detect** negative weight cycle.
- If all weights are the same, use BFS.

Today

Key idea:

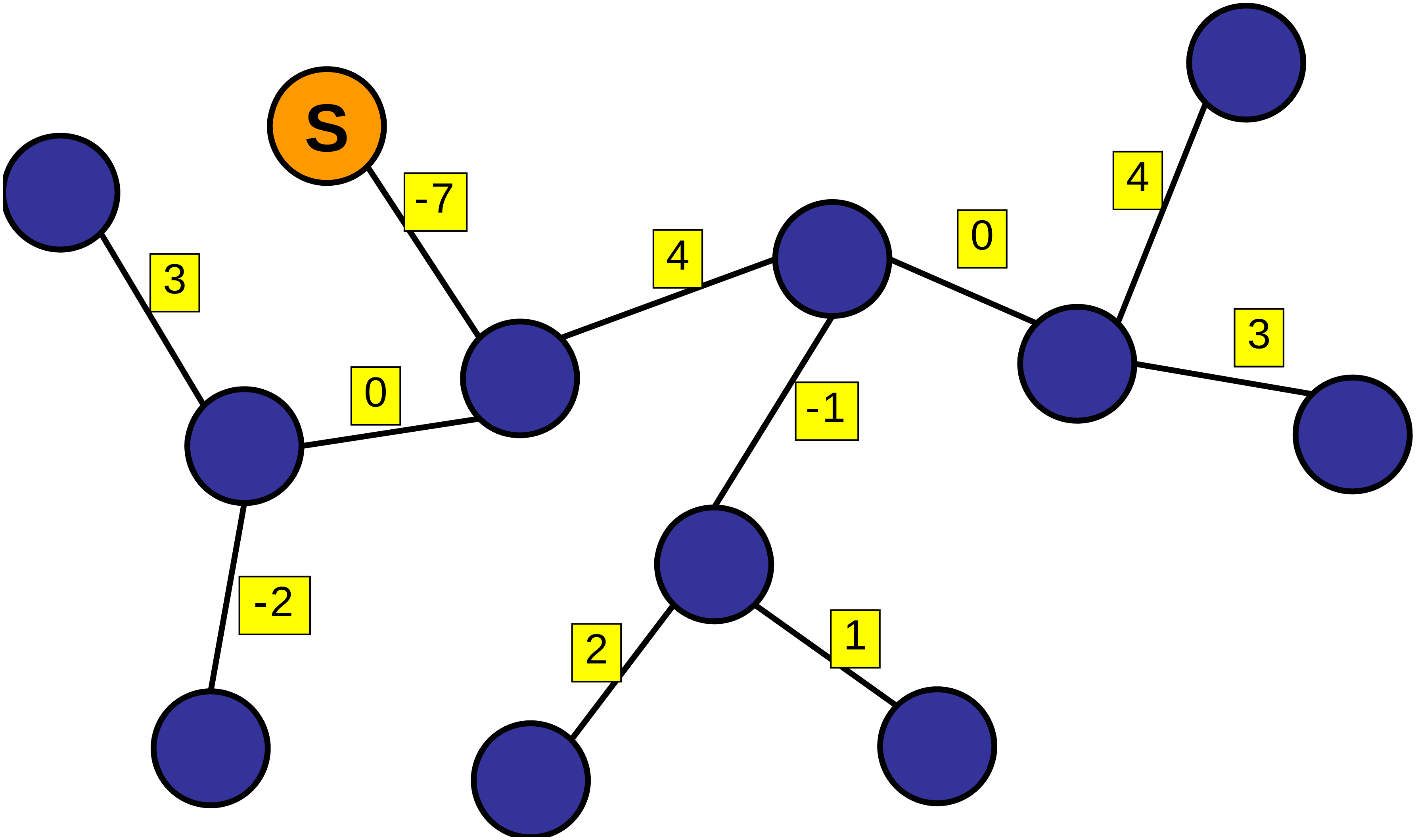
Relax the edges in the "right" order.

Only relax each edge once:

– O(E) cost (for relaxation step).

Special Case: Tree

Undirected, weighted



Aside: Trees, Redefined

What is an (undirected) tree?

– A graph with no cycles is an (undirected) tree.

What is a *rooted* tree?

– A tree with a special designated root note.

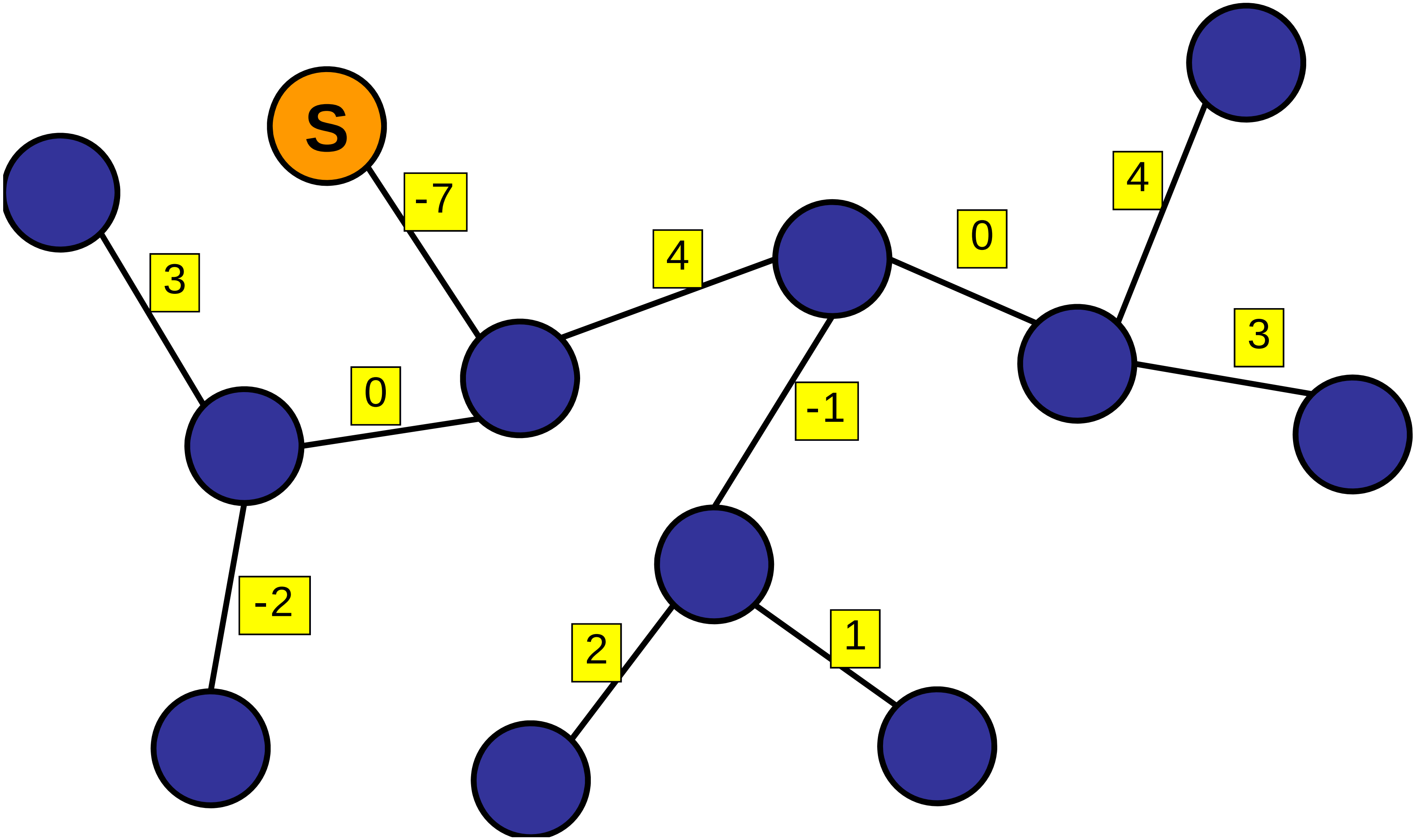
Our previous (recursive) definition of a *tree*:

– A node with zero, one, or more sub-trees.

– I.e., a *rooted* tree.

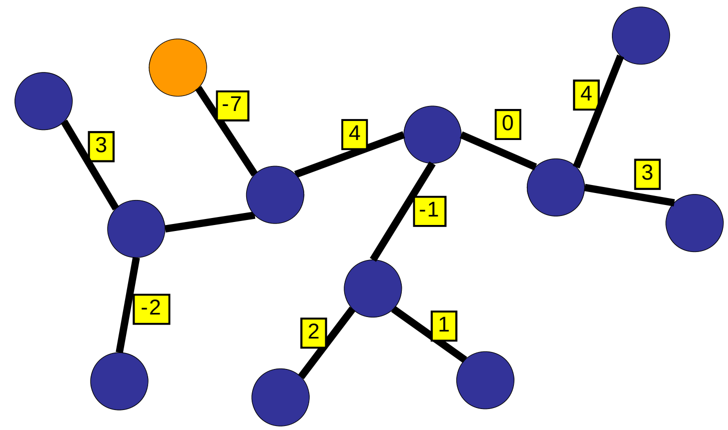
Special Case: Tree

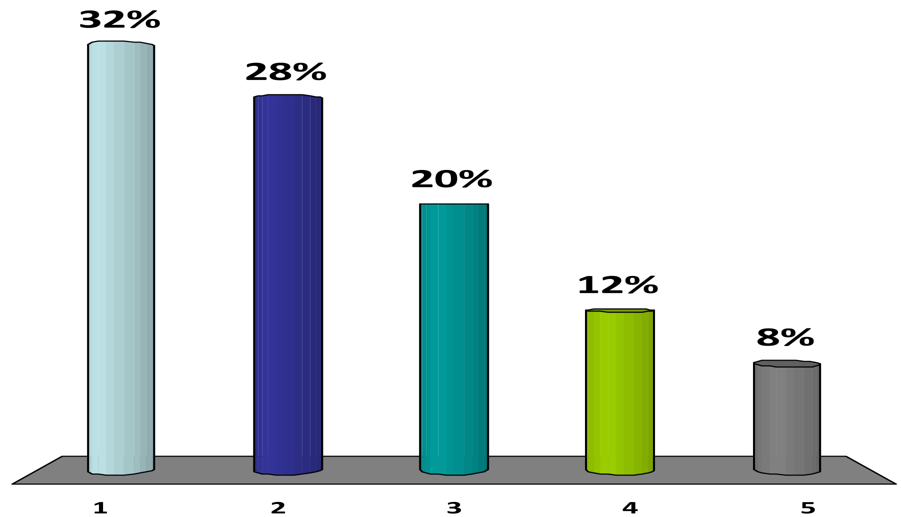
Undirected, weighted



Which algorithm is best for checking if a graph is a tree?

1. BFS
2. DFS
3. Bellman Ford
4. Topological Sort
5. Dijkstra's Algorithm





Aside: Tree Checking

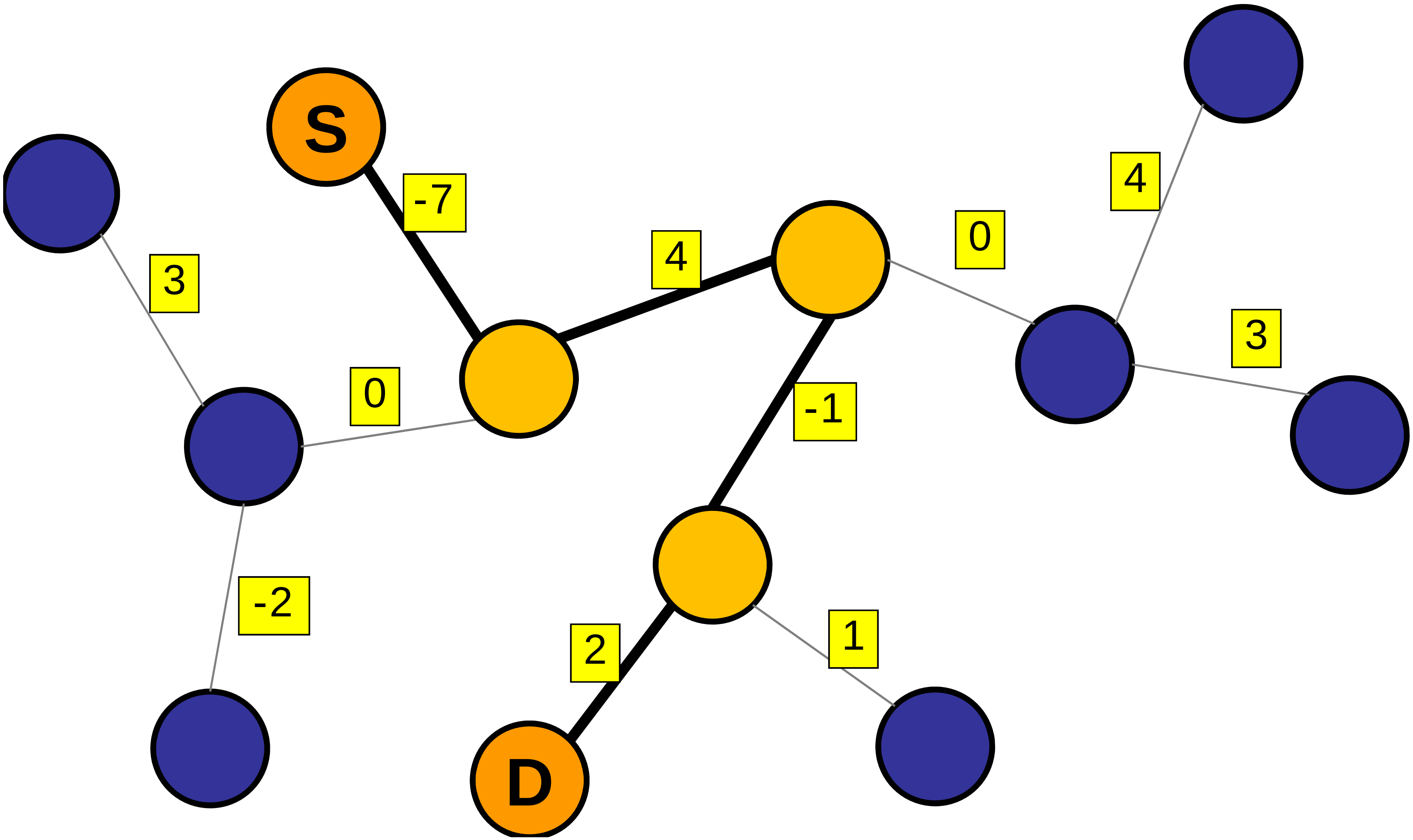
If it is connected...

If it is disconnected...

If it is directed...

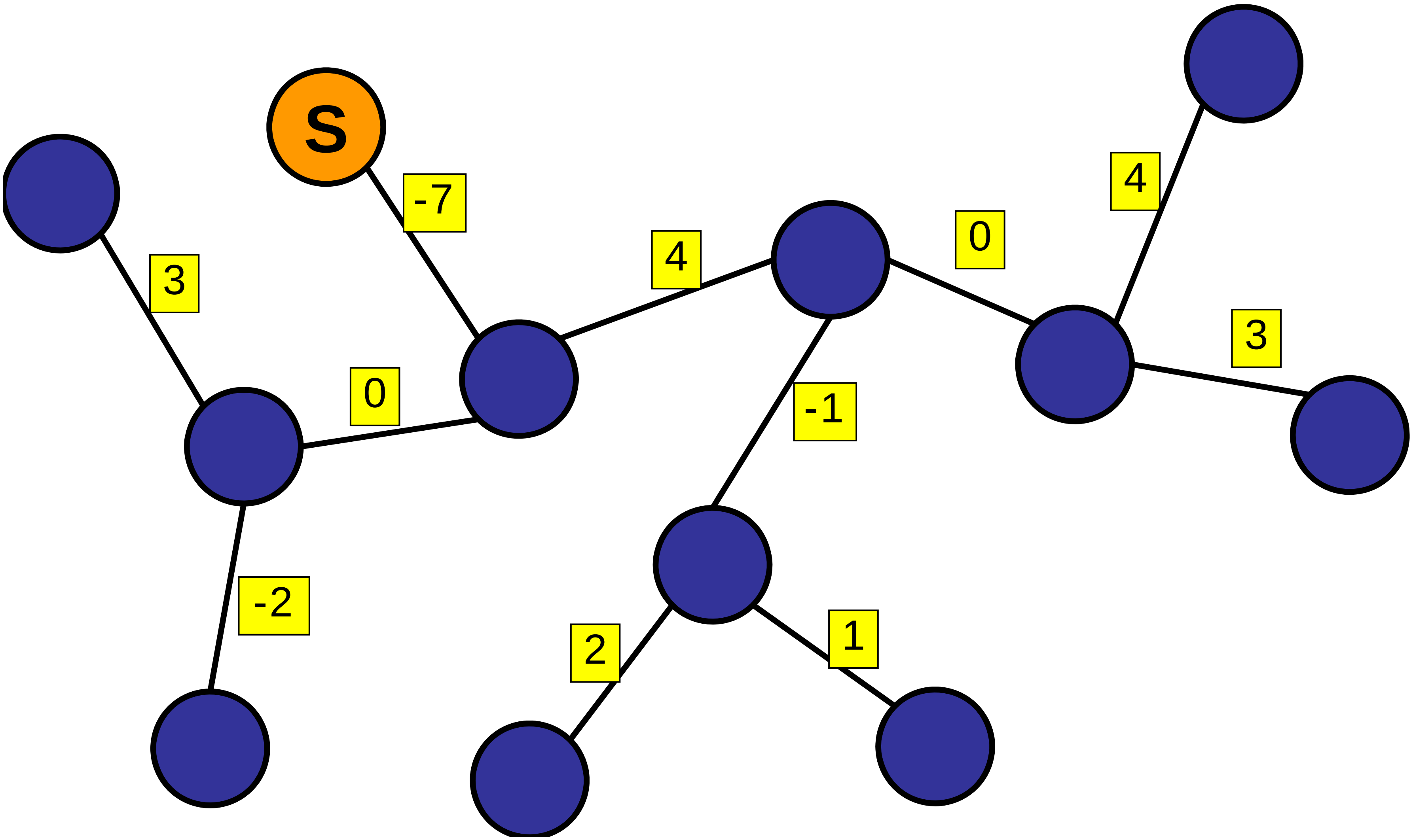
Special Case: Tree

source-to-destination: only one possible path!



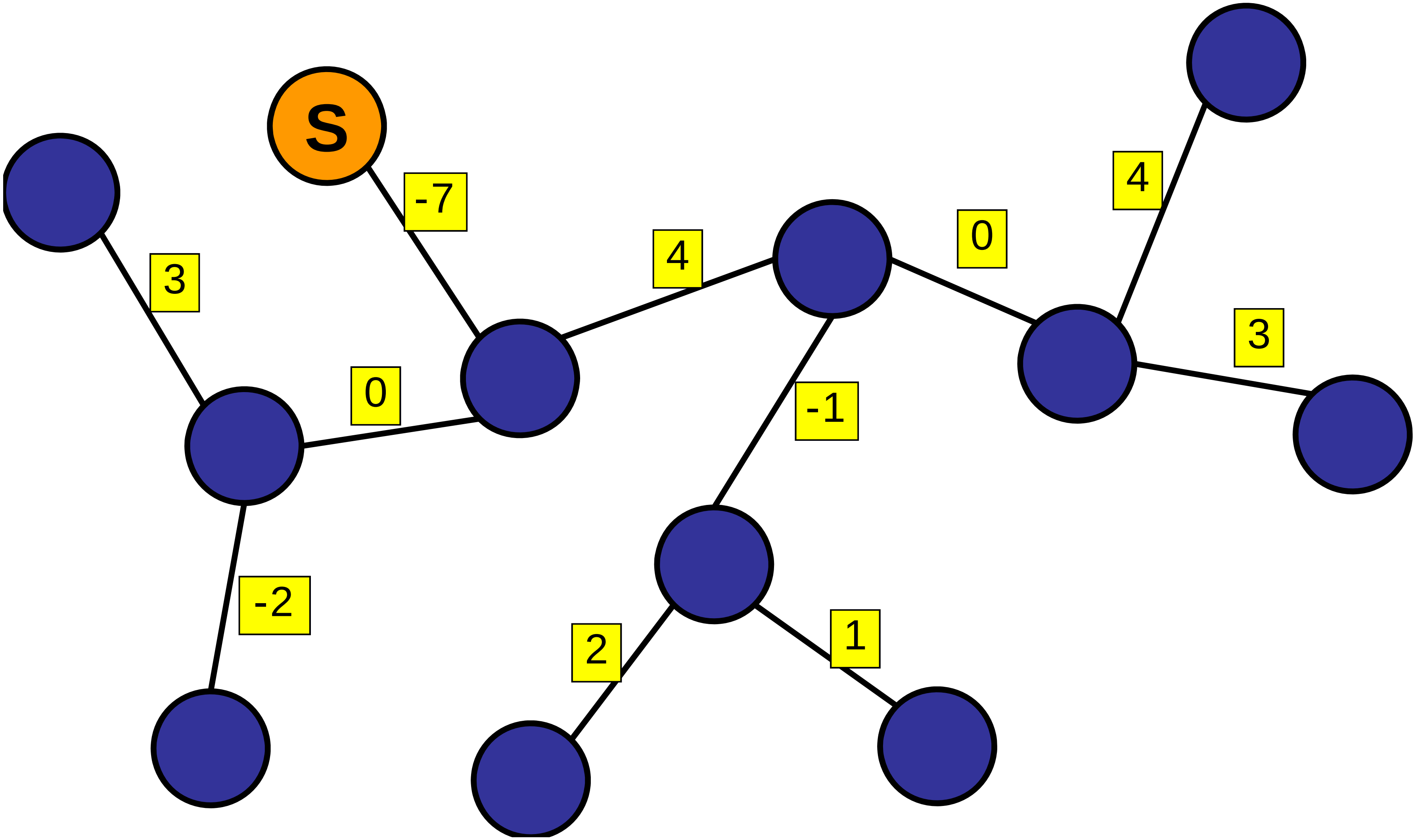
Special Case: Tree

source-to-all: what order to relax?



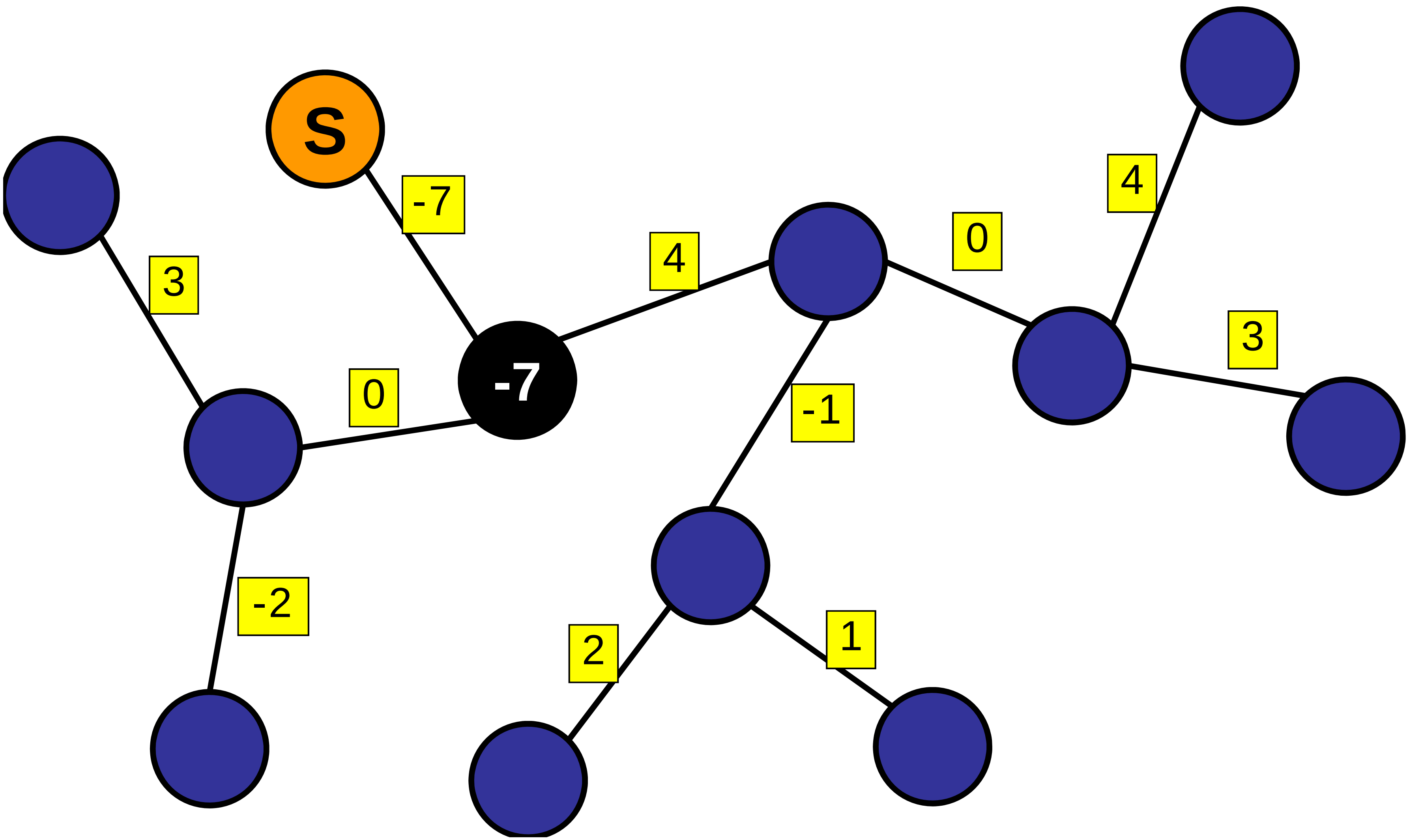
Special Case: Tree

Relax edges in (BFS or DFS order).



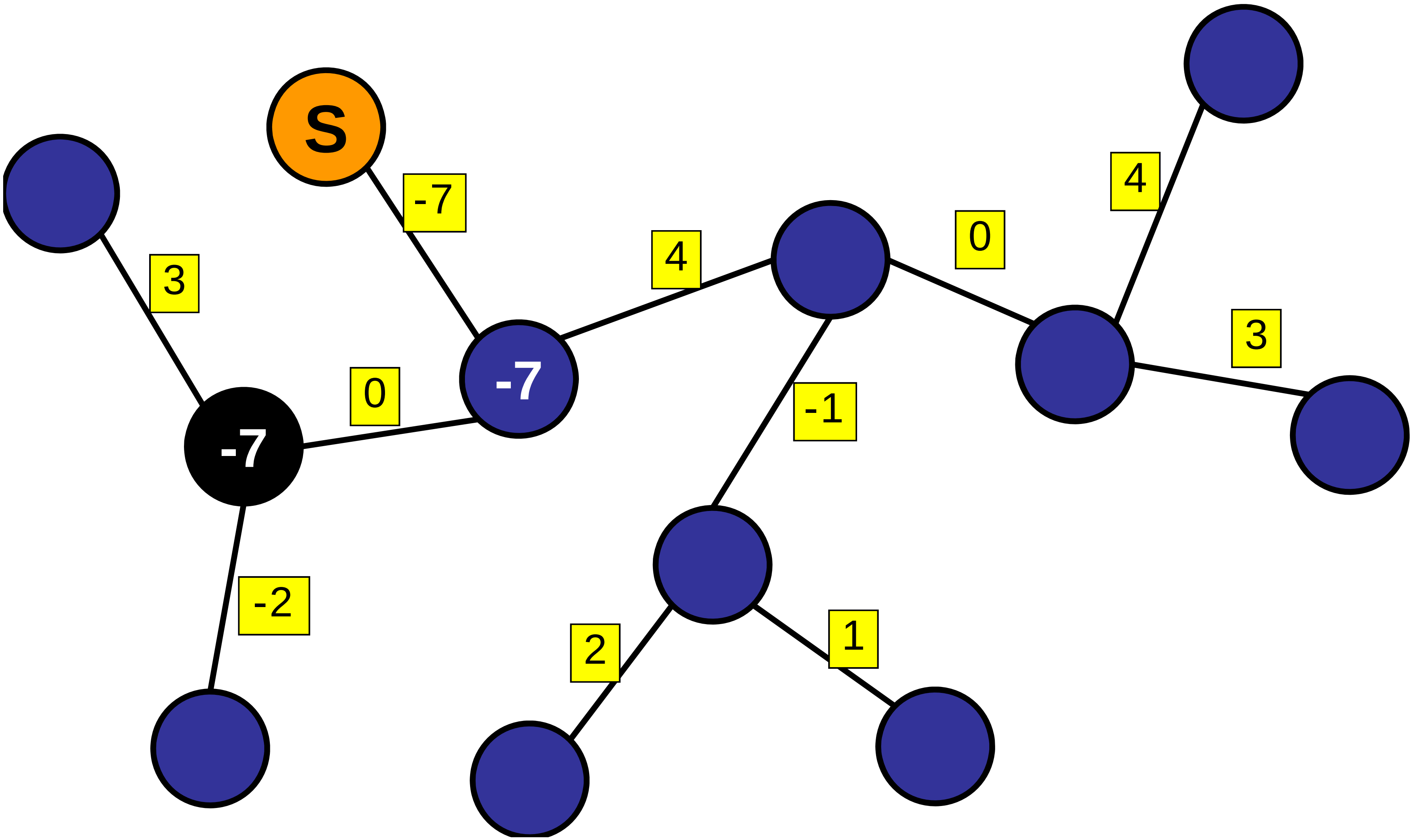
Special Case: Tree

Relax edges in DFS order.



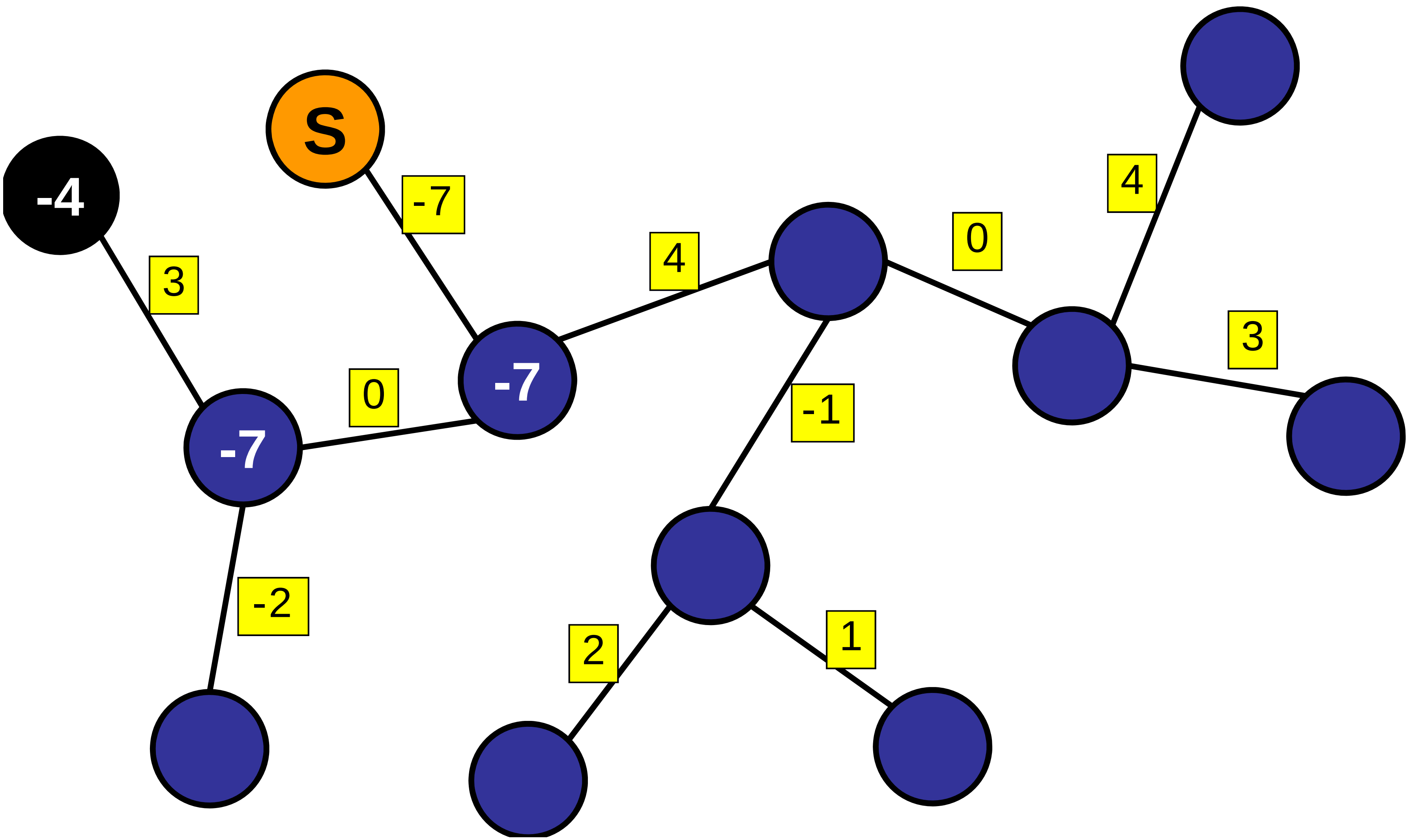
Special Case: Tree

Relax edges in DFS order.



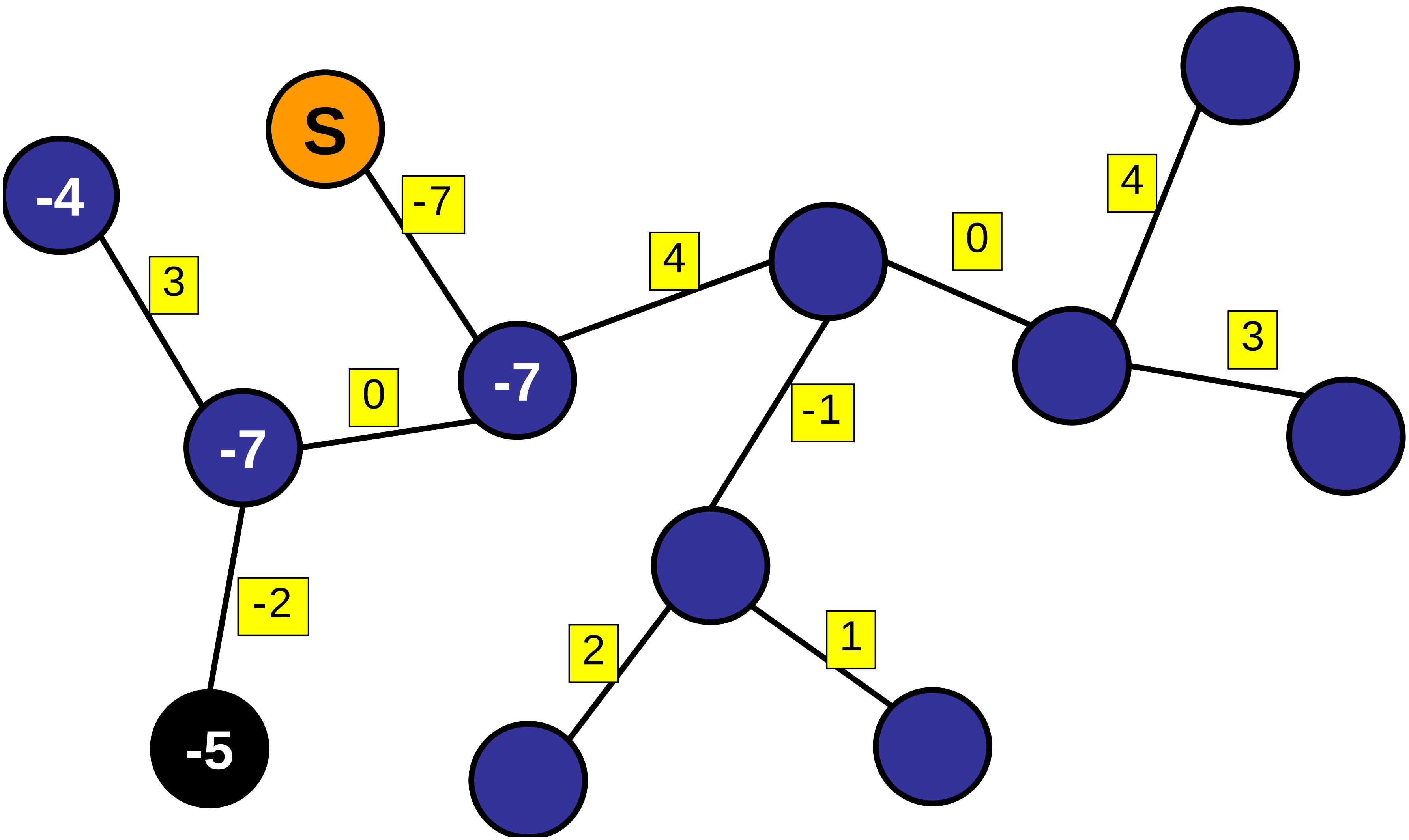
Special Case: Tree

Relax edges in DFS order.



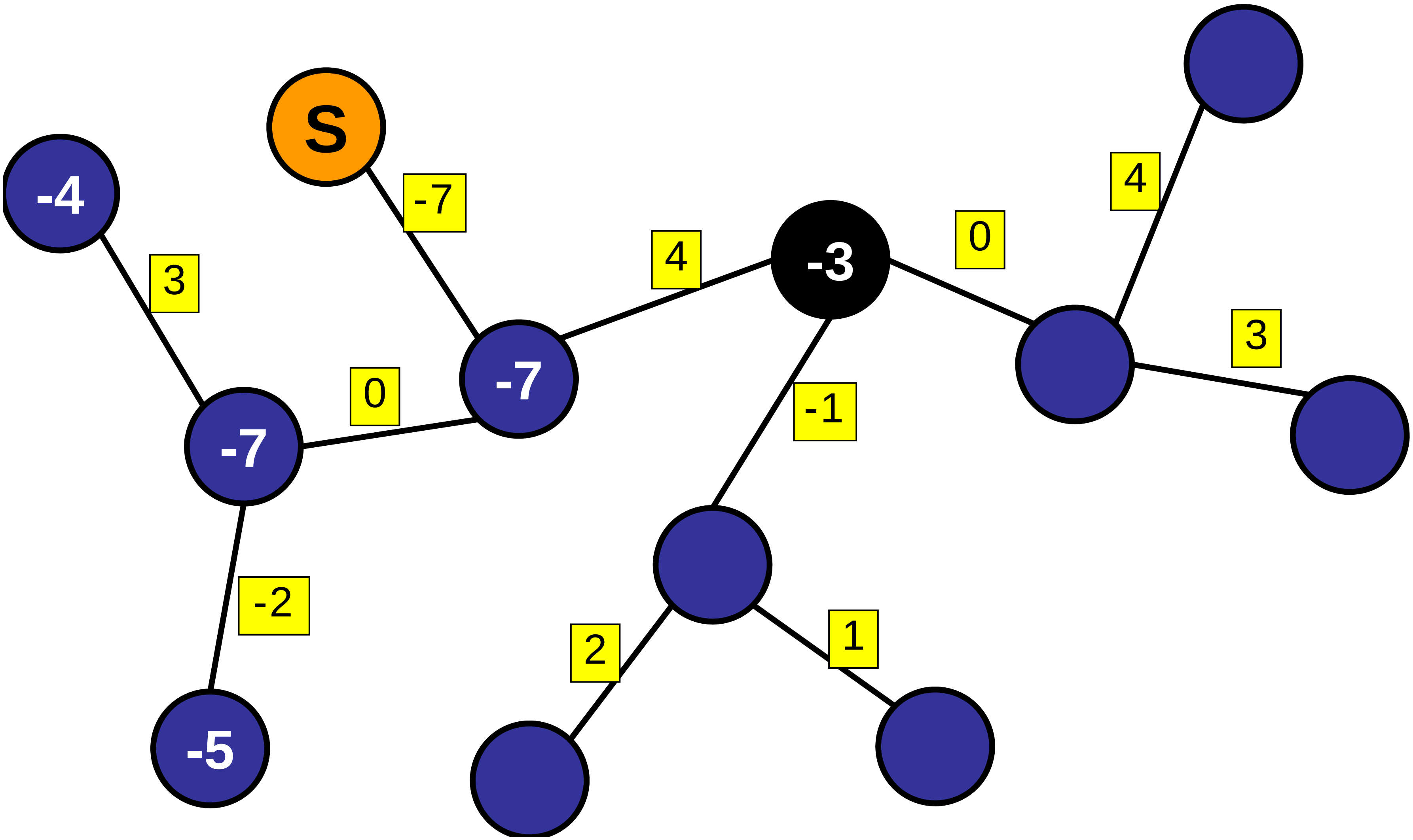
Special Case: Tree

Relax edges in DFS order.



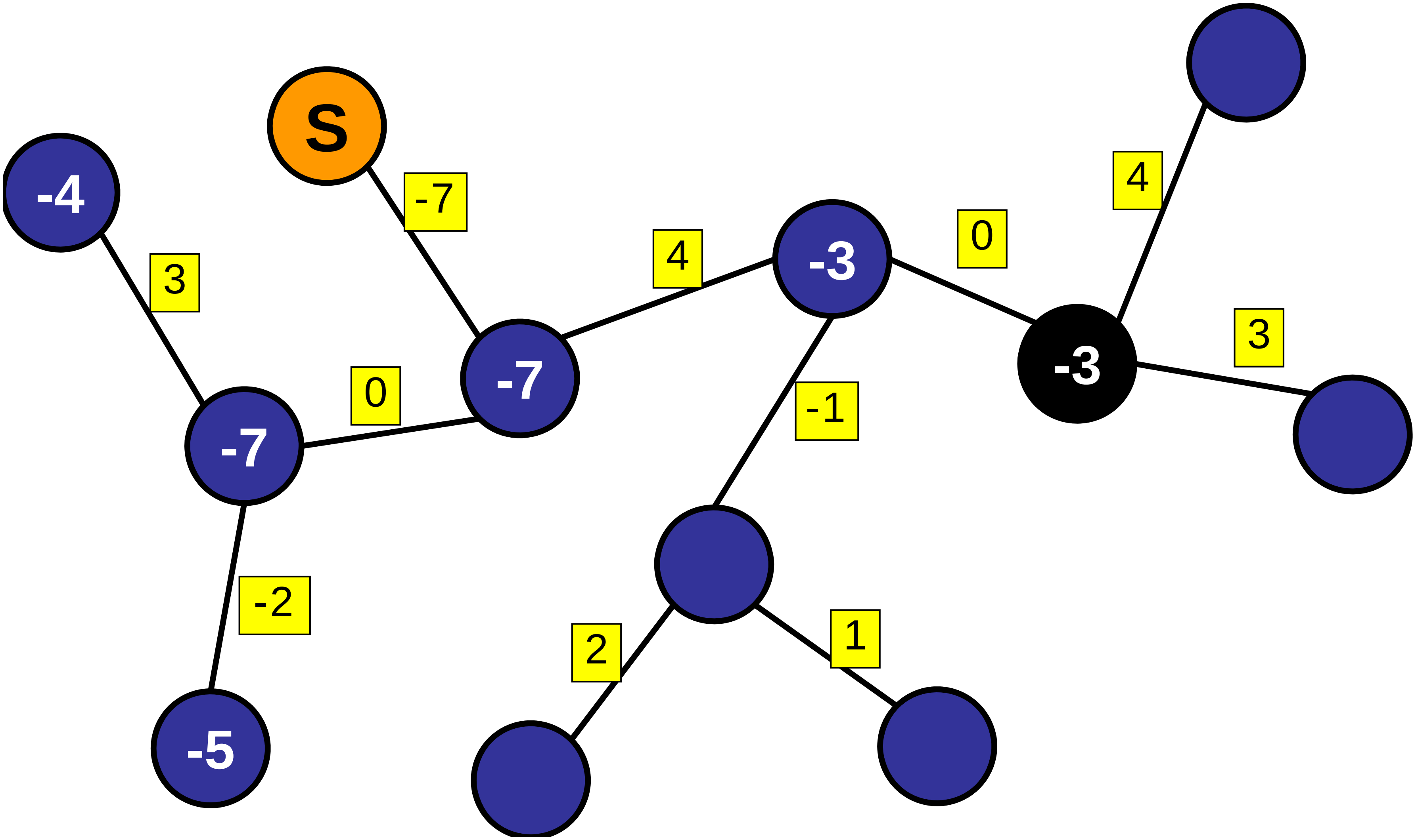
Special Case: Tree

Relax edges in DFS order.



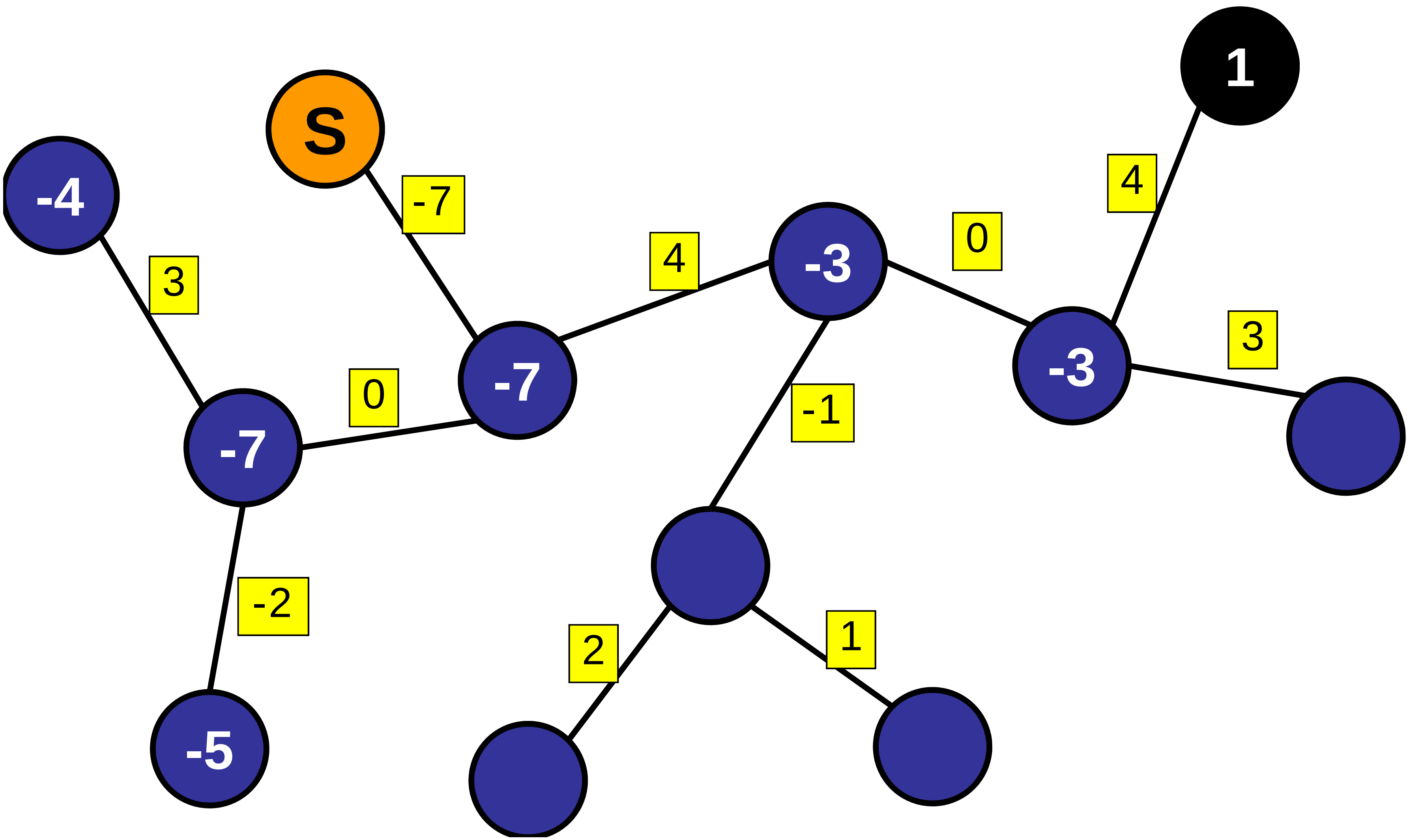
Special Case: Tree

Relax edges in DFS order.



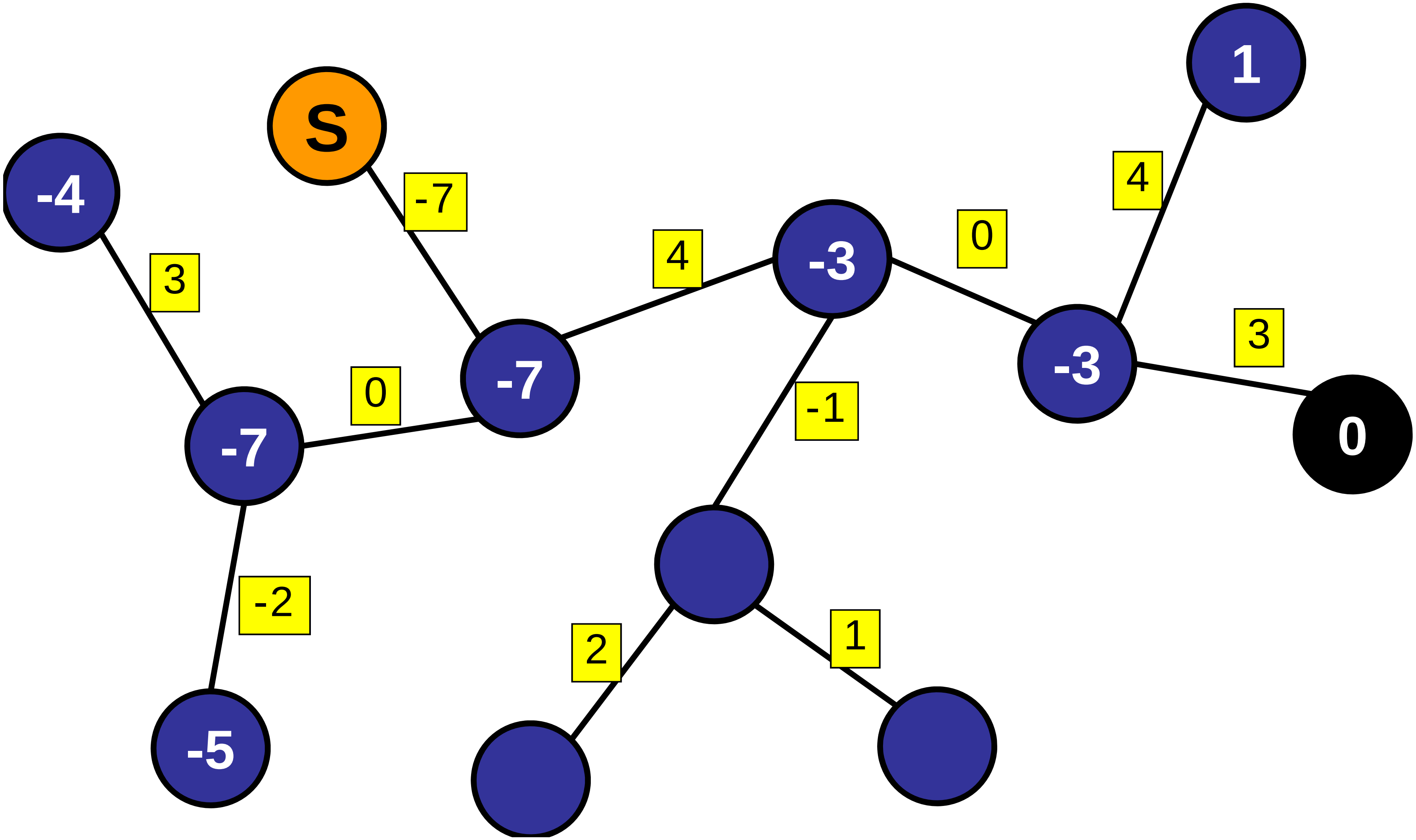
Special Case: Tree

Relax edges in DFS order.



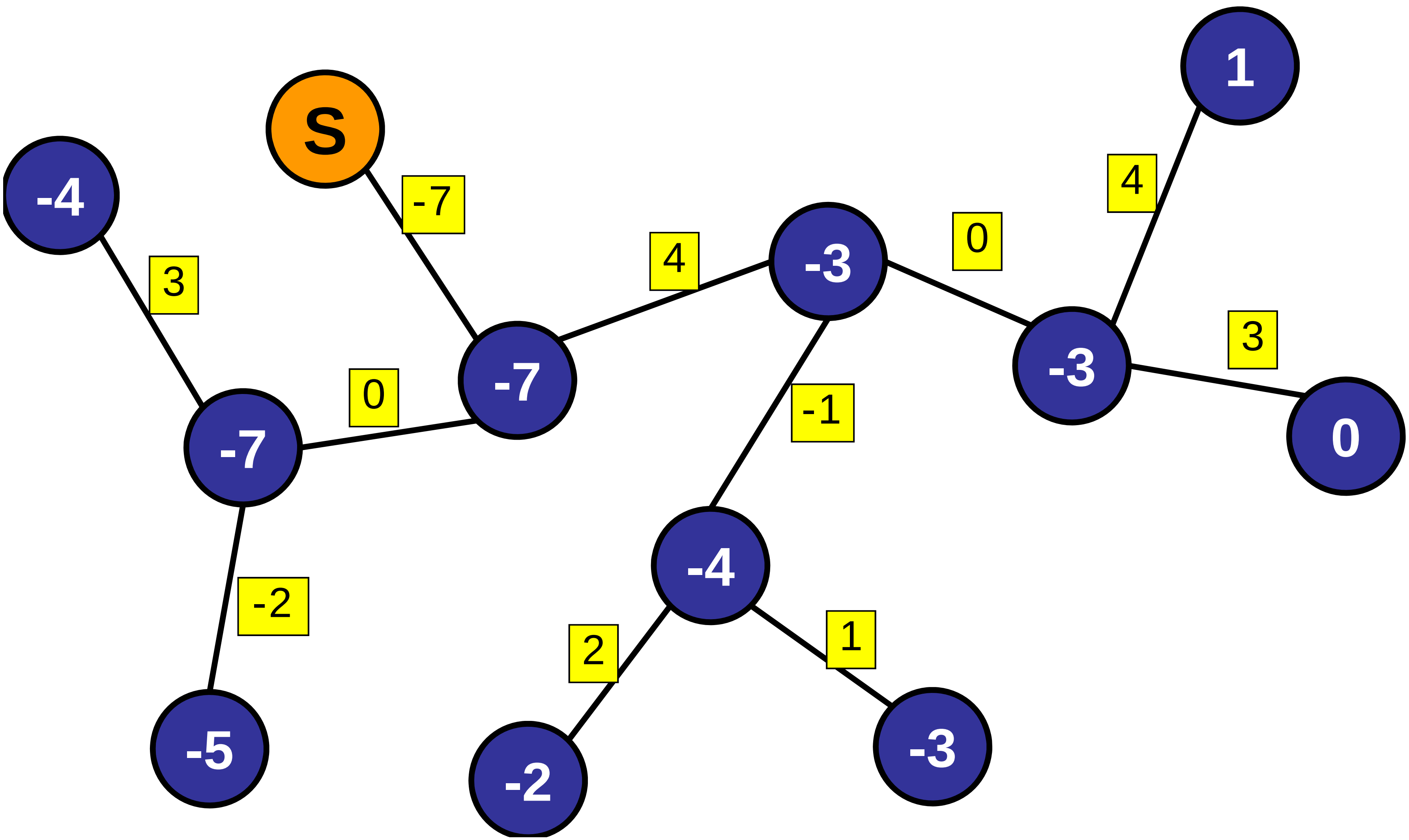
Special Case: Tree

Relax edges in DFS order.



Special Case: Tree

Relax edges in DFS order.



Special Case: Tree

Basic idea:

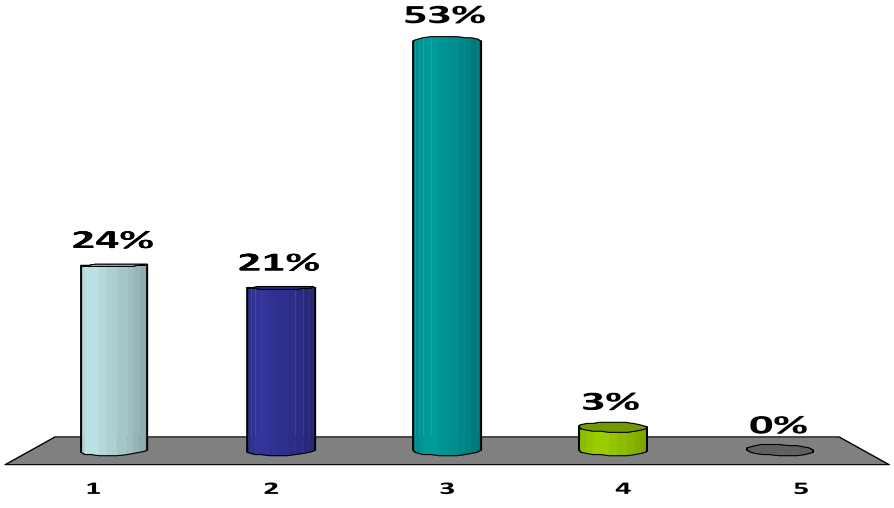
- Perform DFS or BFS
- Relax each edge the first time you see it.
- $O(V)$ time.

Assumptions:

- Weighted edges
- Positive or negative weights
- Undirected tree

Why is the running time O(V)?

1. You only need to explore 1 outgoing edge for each vertex.
2. DFS/BFS run in O(V) time on a graph.
3. There are only O(V) edges in a tree.
4. It is not O(V): you need to explore every edge!
5. I'm confused.



Special Case: Tree

Basic idea:

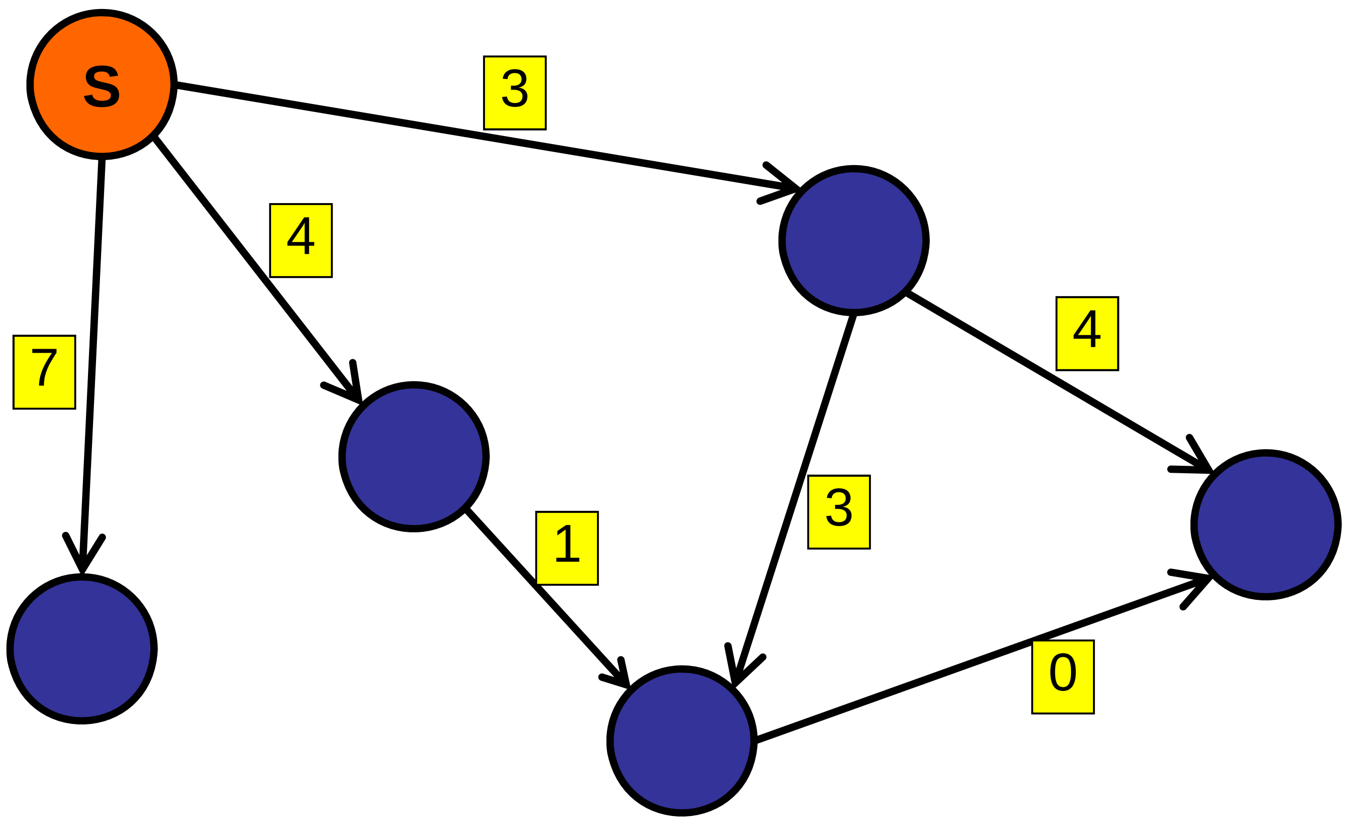
- Perform DFS or BFS
- Relax each edge the first time you see it.
- $O(V)$ time.

Assumptions:

- Weighted edges
- Positive or negative weights
- Undirected tree

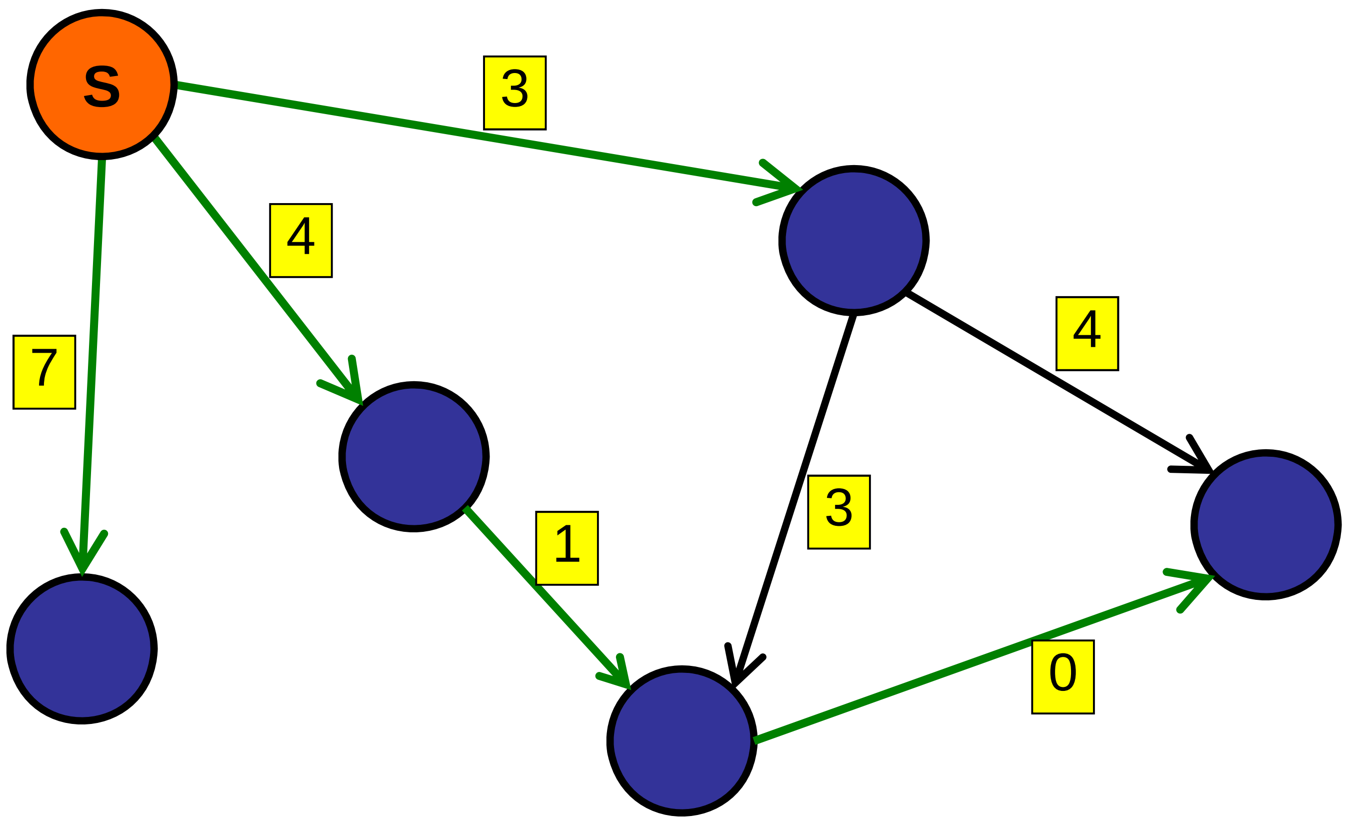
General Graph

Non-negative edges



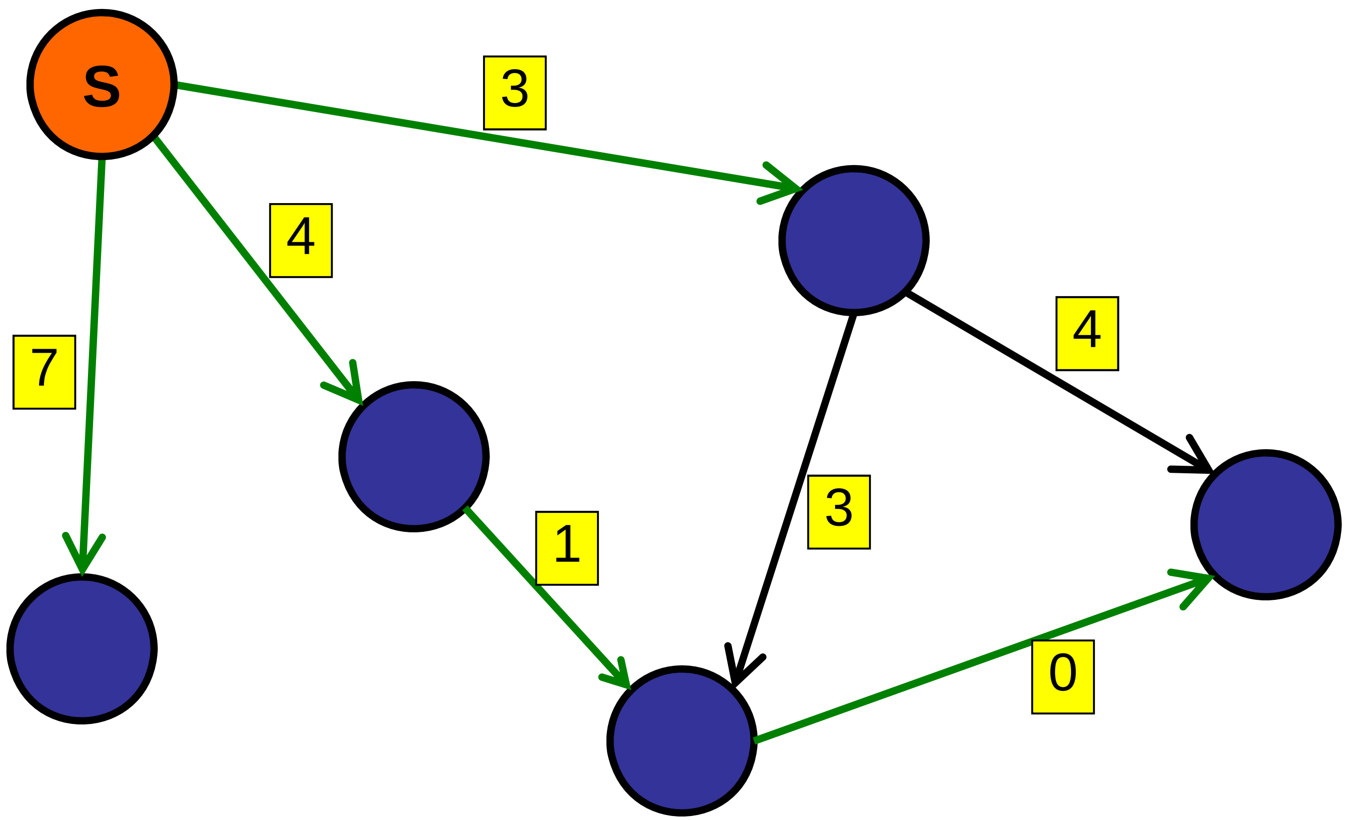
Shortest Path Tree

For every node: add 1 shortest path to the tree.



Shortest Path Tree

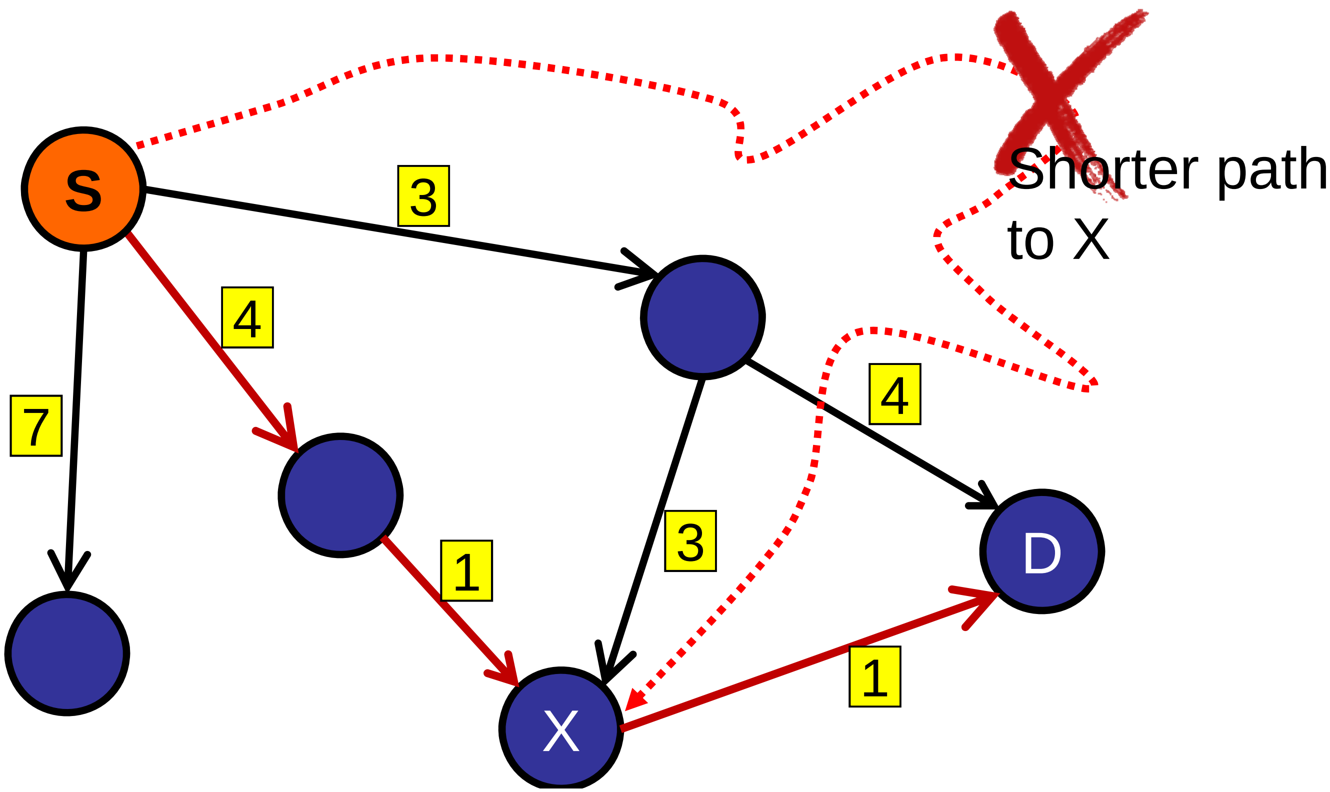
Why are there no cycles?



Shortest Path Tree

Key property:

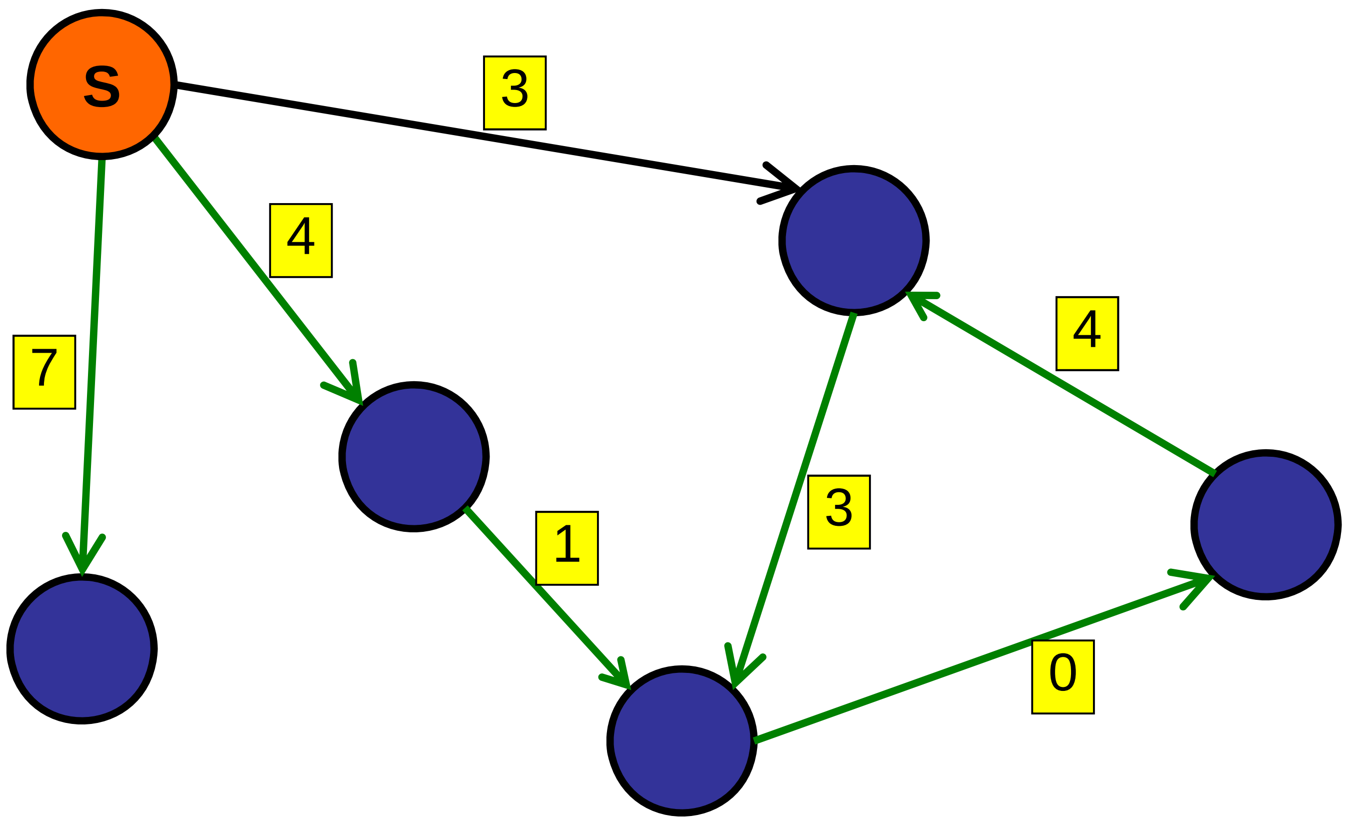
If P is the shortest path from S to D, and if P goes through X, then P is also the shortest path from S to X (and from X to D).



Shortest Path Tree

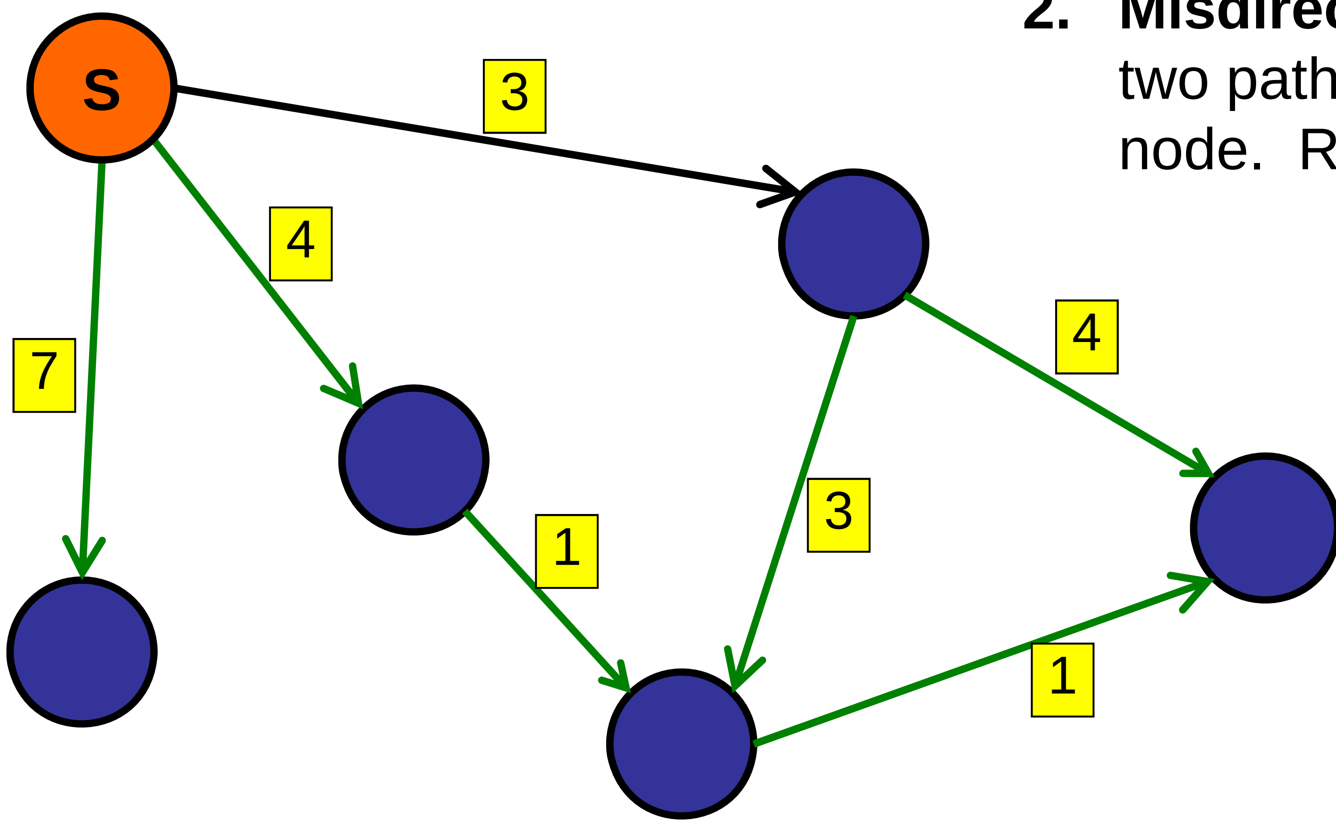
Why are there no cycles?

1. **Directed cycle**: remove one edge to get shorter paths.



Shortest Path Tree

Why are there no cycles?

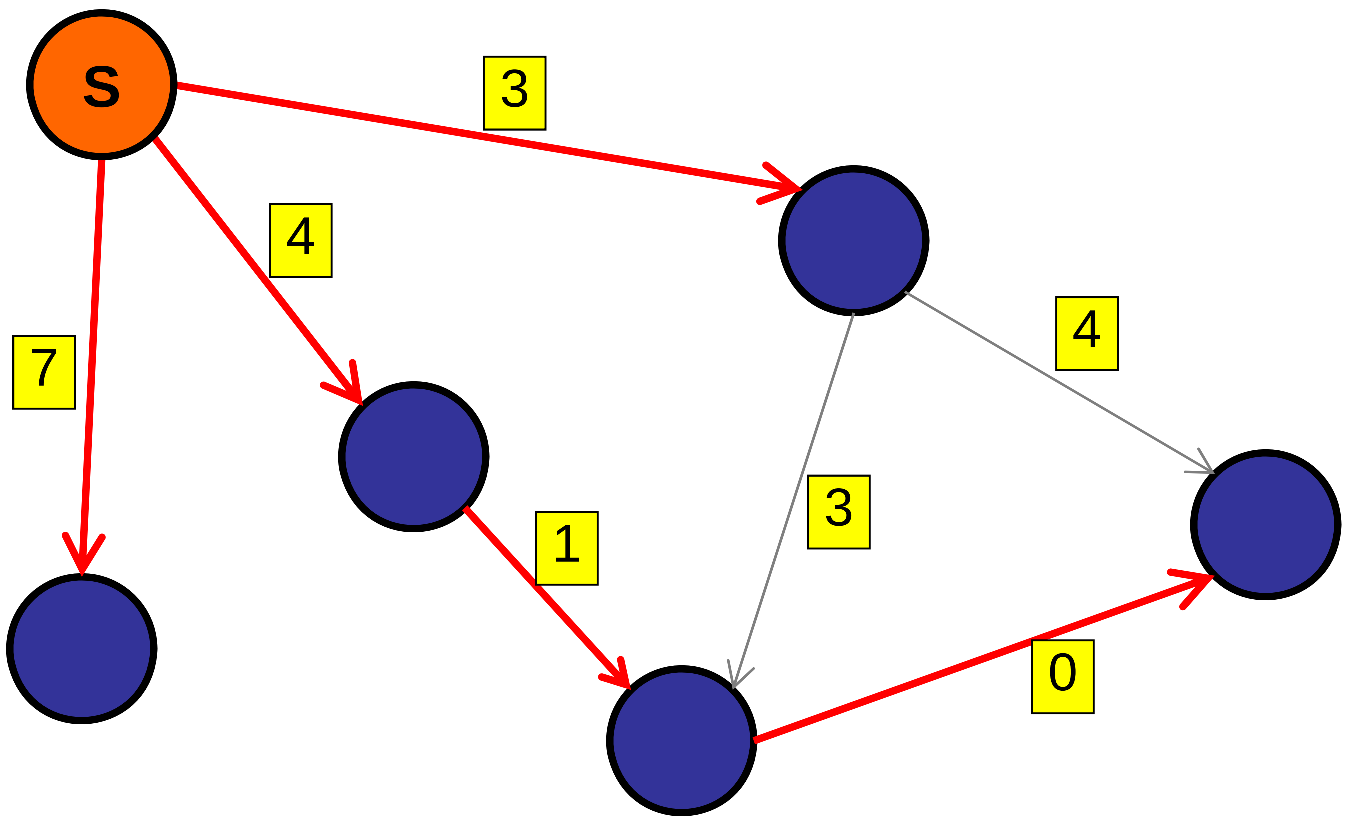


1. **Directed cycle**: remove one edge to get shorter paths.

2. **Misdirected cycle**: two paths to some node. Remove one.

Shortest Path Tree

No cycles in the shortest path tree.



Today

Key idea:

Relax the edges in the “right” order.

Only relax each edge once:

– O(E) cost (for relaxation step).

Edsger W. Dijkstra

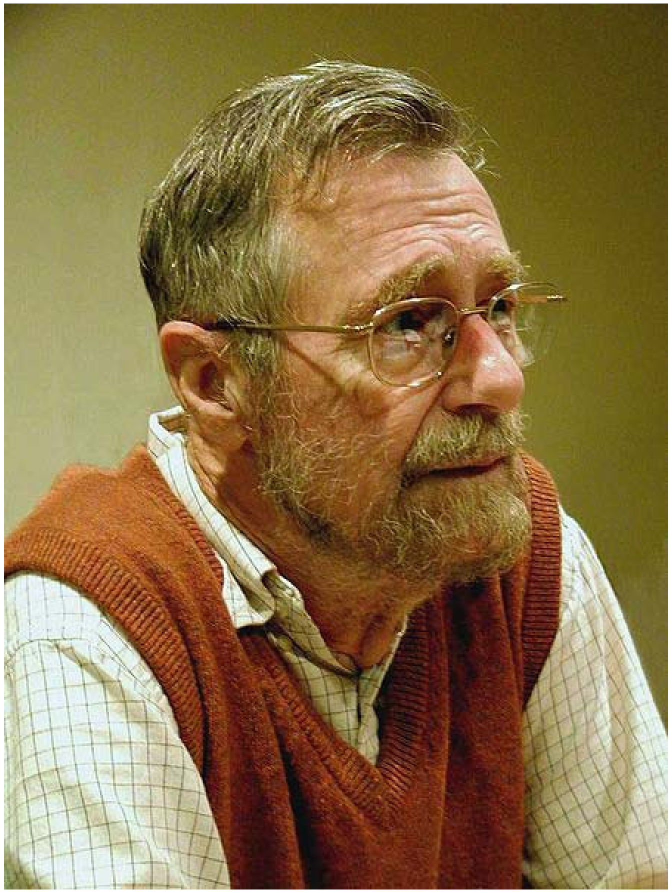
“Computer science is no more about computers than astronomy is about telescopes.”

“The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.”

“There should be no such thing as boring mathematics.”

“Elegance is not a dispensable luxury but a factor that decides between success and failure.”

“Simplicity is prerequisite for reliability.”



1930-2002

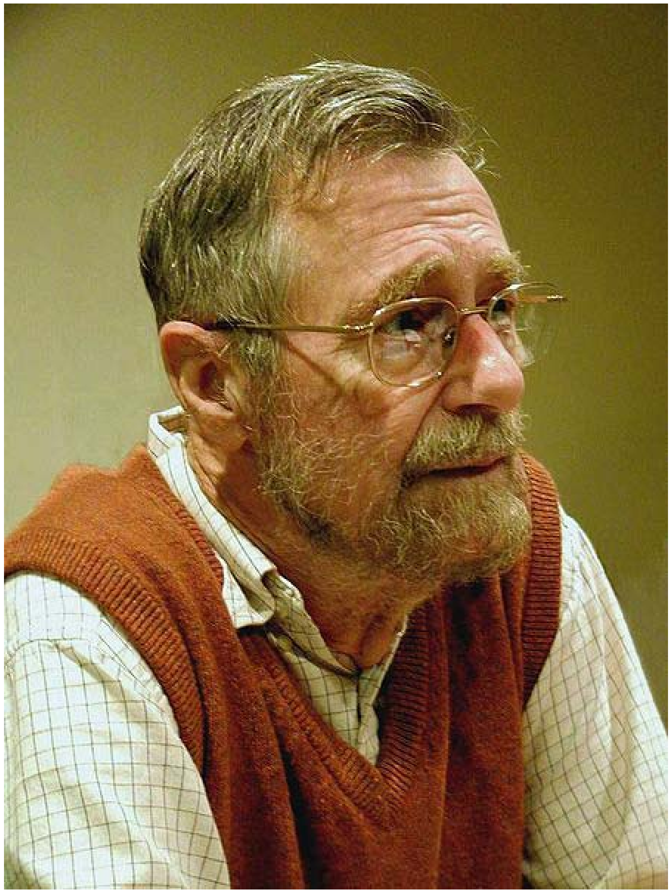
Edsger W. Dijkstra

"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

"The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."

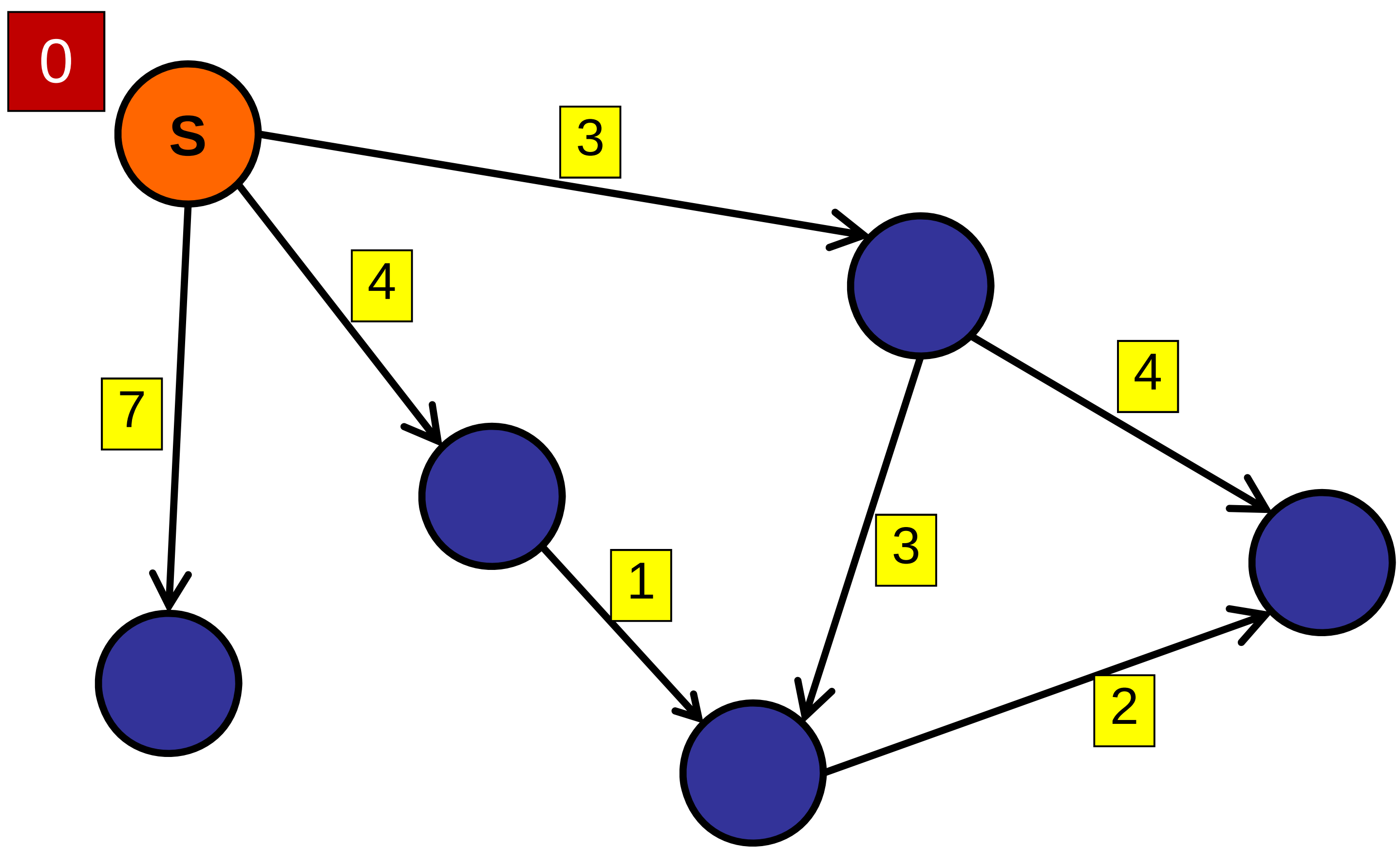
"APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums."

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."



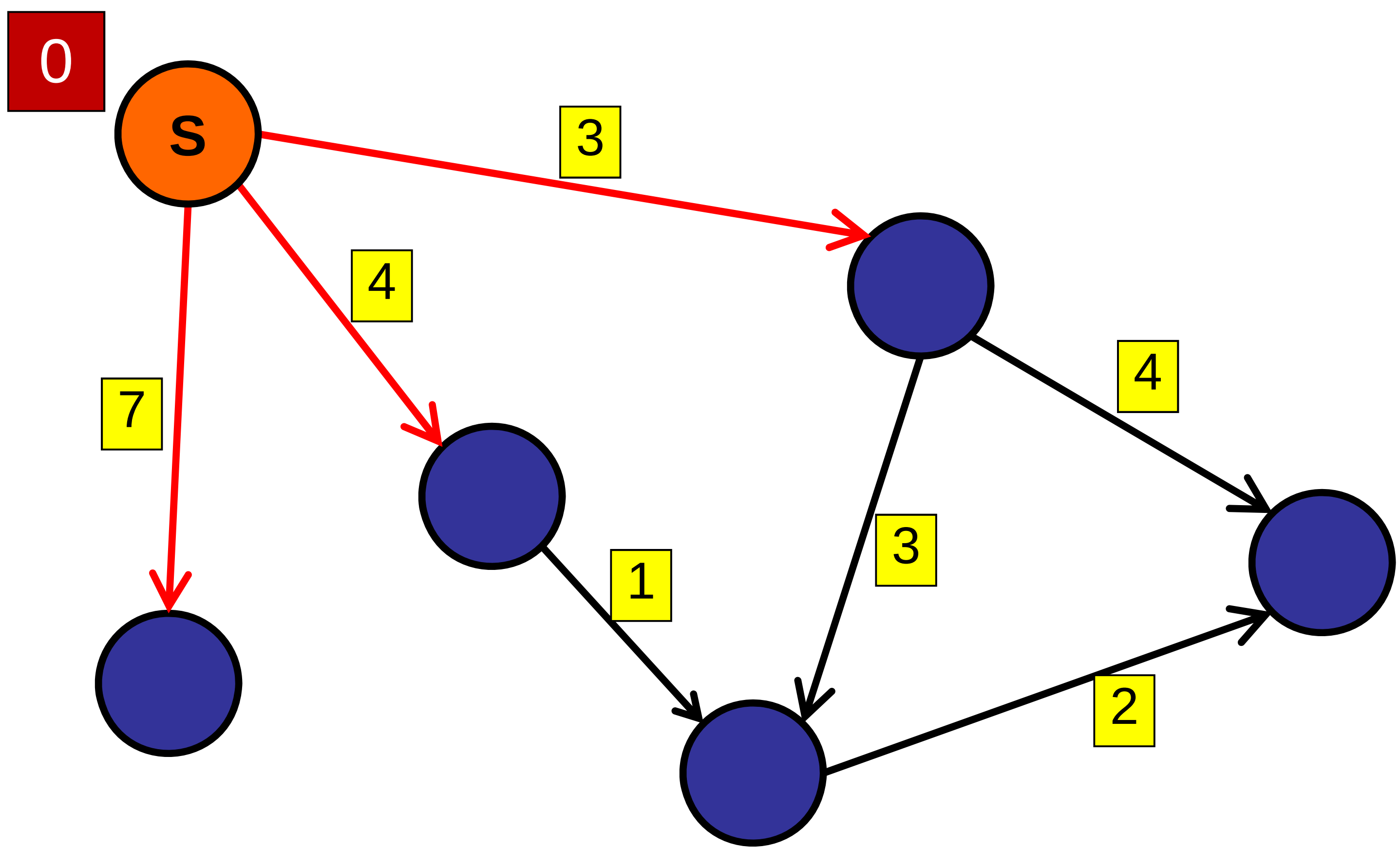
Dijkstra's Algorithm (First Try)

Relax shortest edge first



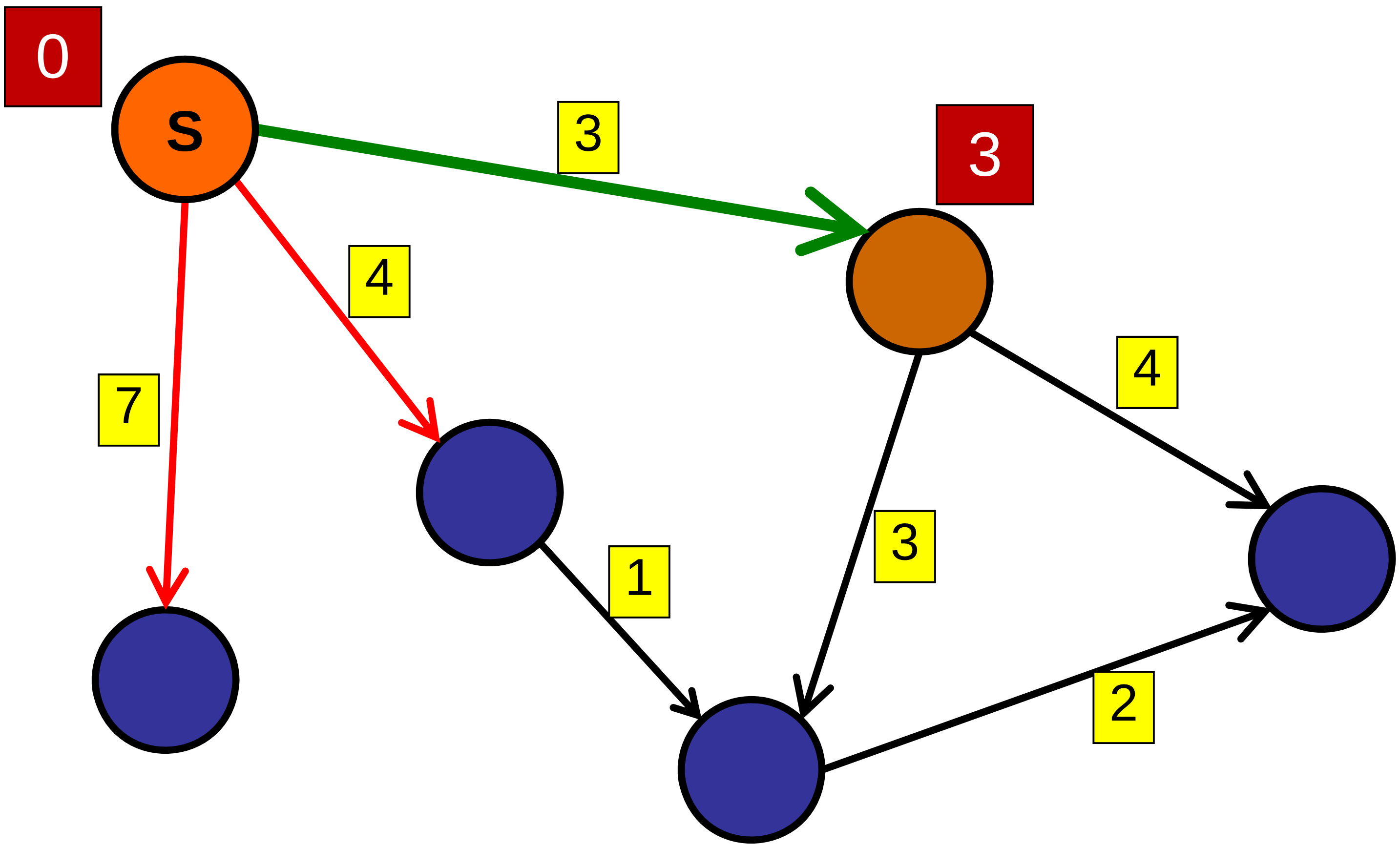
Dijkstra's Algorithm (First Try)

Relax shortest edge first



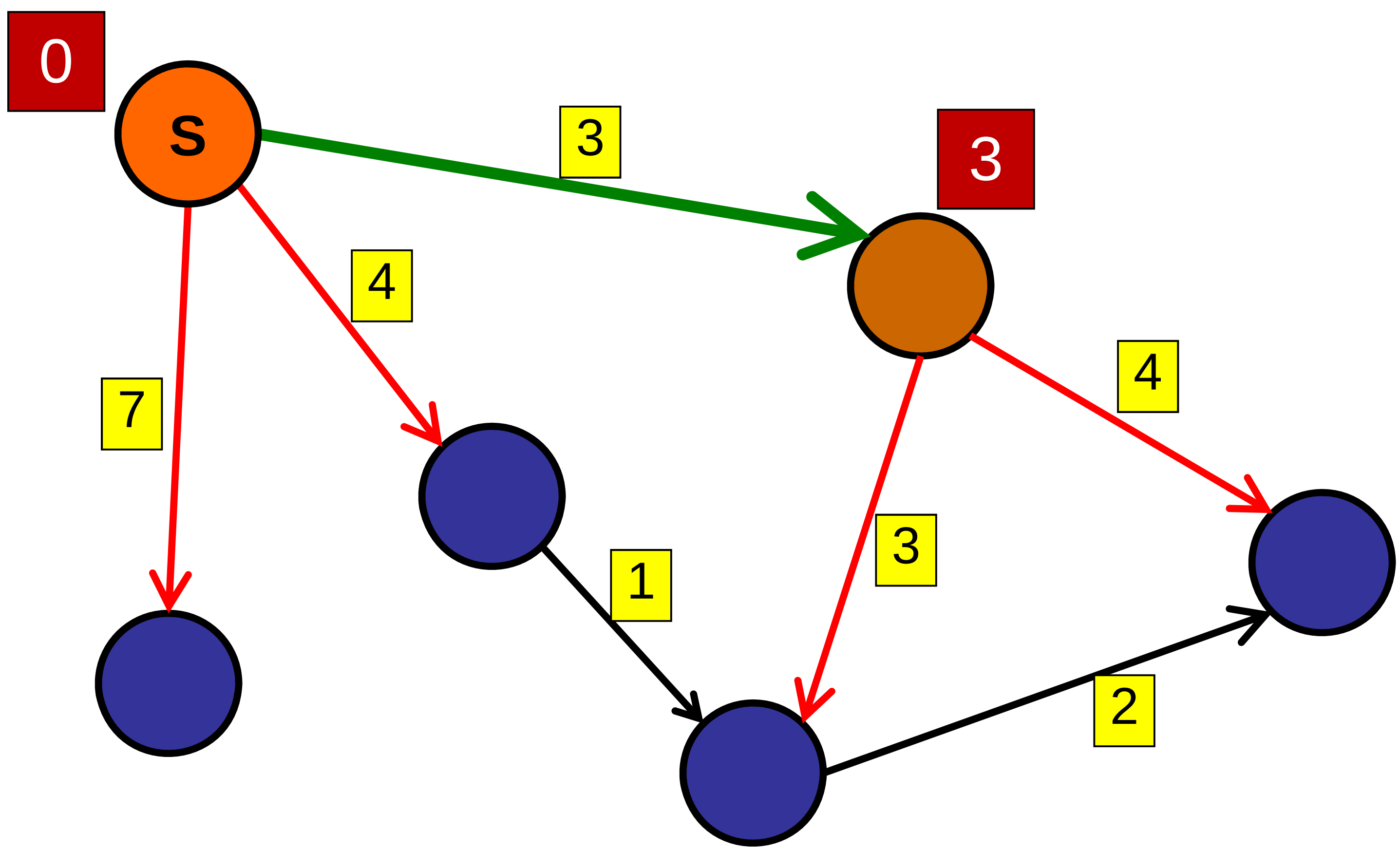
Dijkstra's Algorithm (First Try)

Relax shortest edge first



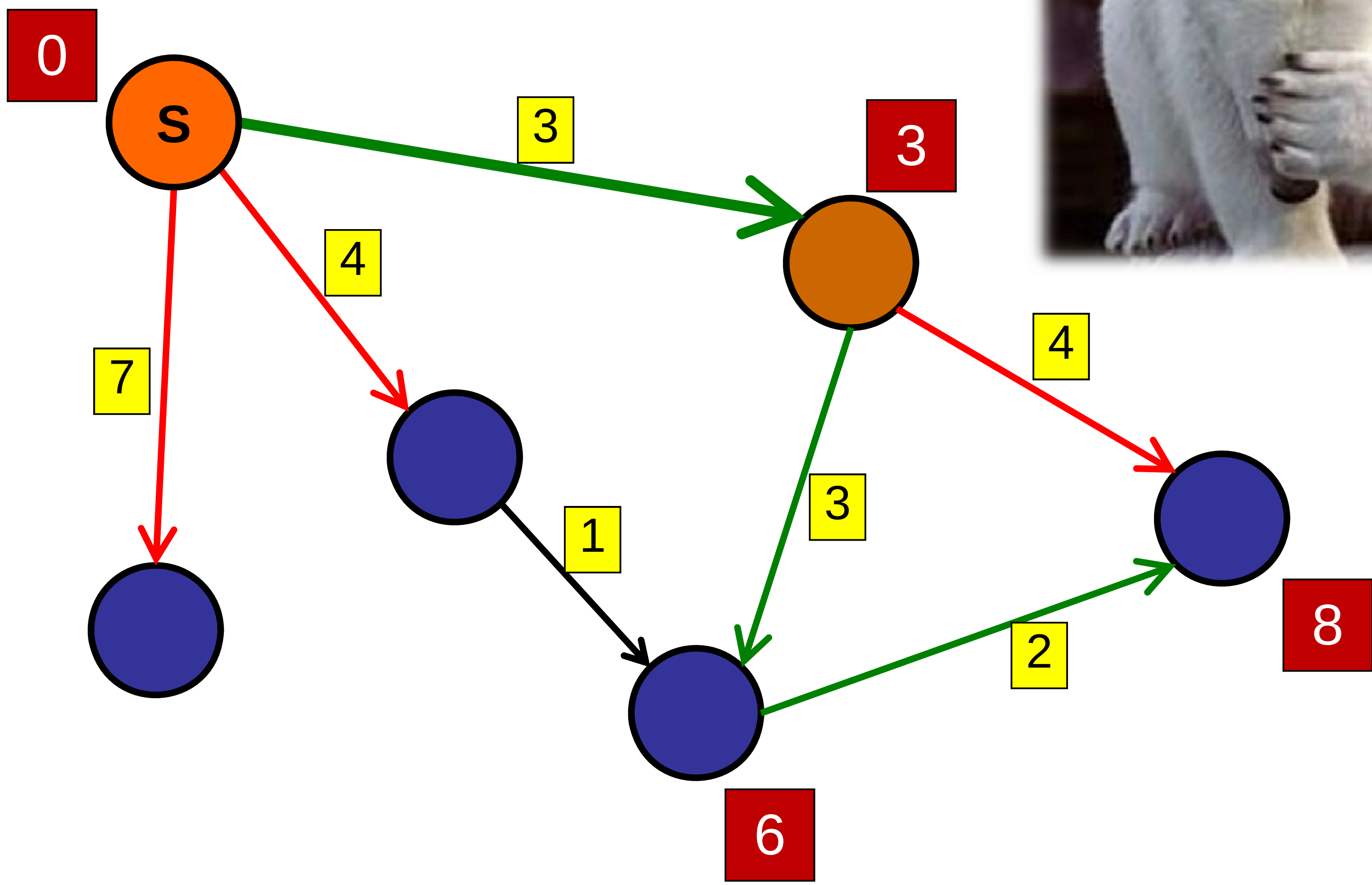
Dijkstra's Algorithm (First Try)

Relax shortest edge first



Dijkstra's Algorithm (Failed Try)

Oops....

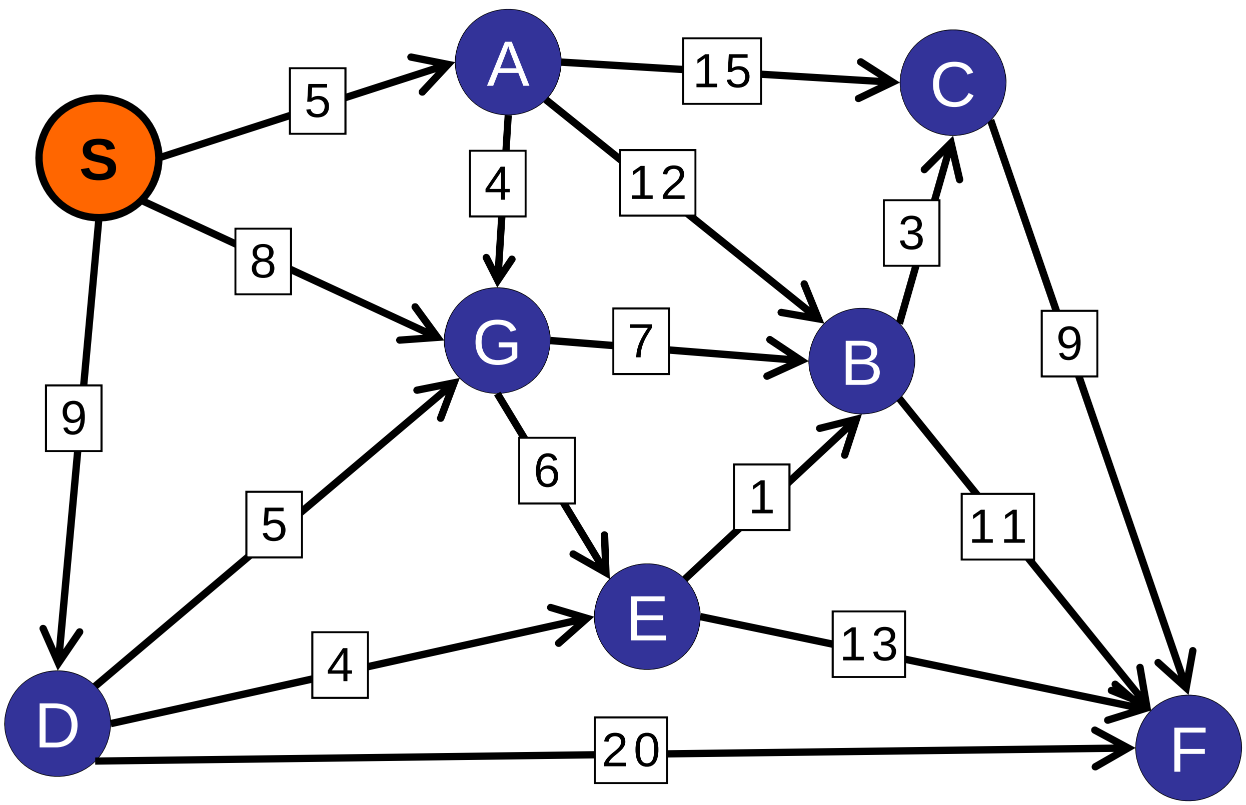


Dijkstra’s Algorithm

Basic idea:

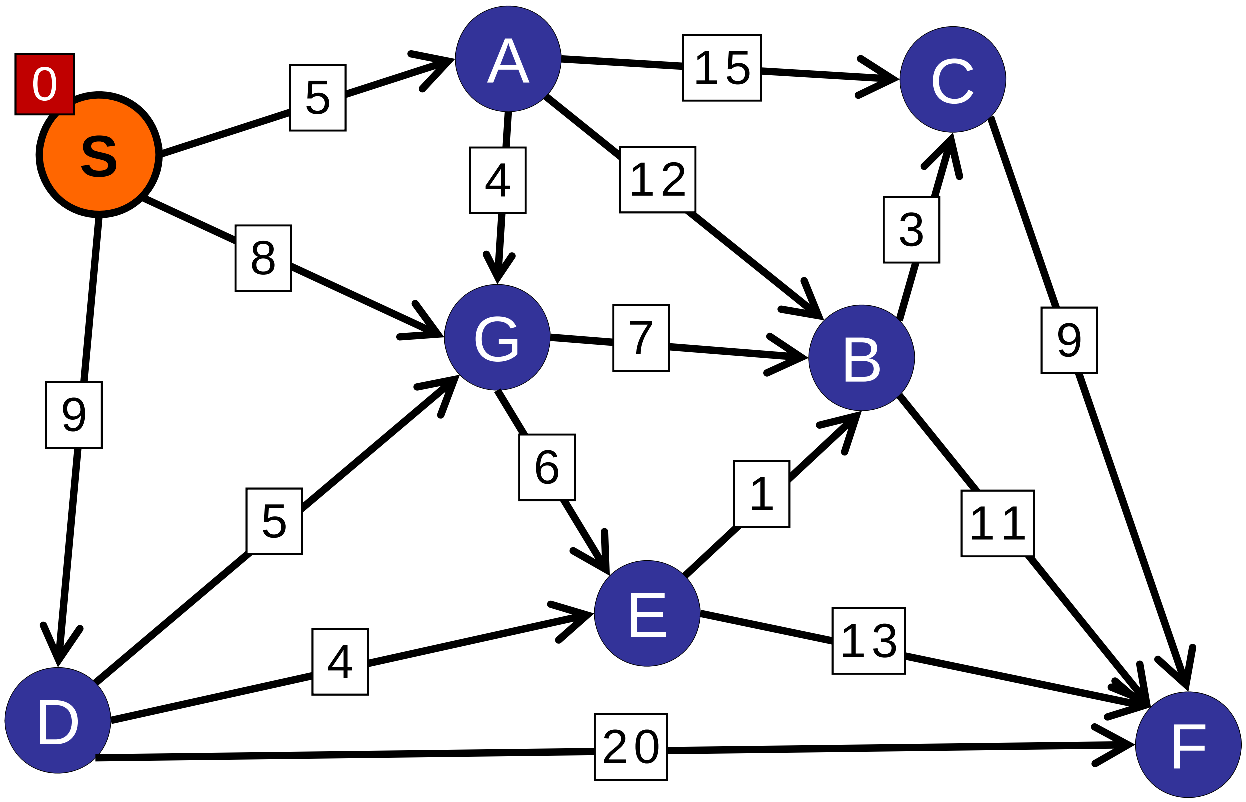
- Maintain distance **estimate** for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Consider vertex with minimum **estimate**.
  - Add vertex to shortest-path-tree.
  - Relax all outgoing edges.

Shortest Paths



Dijkstra's Algorithm

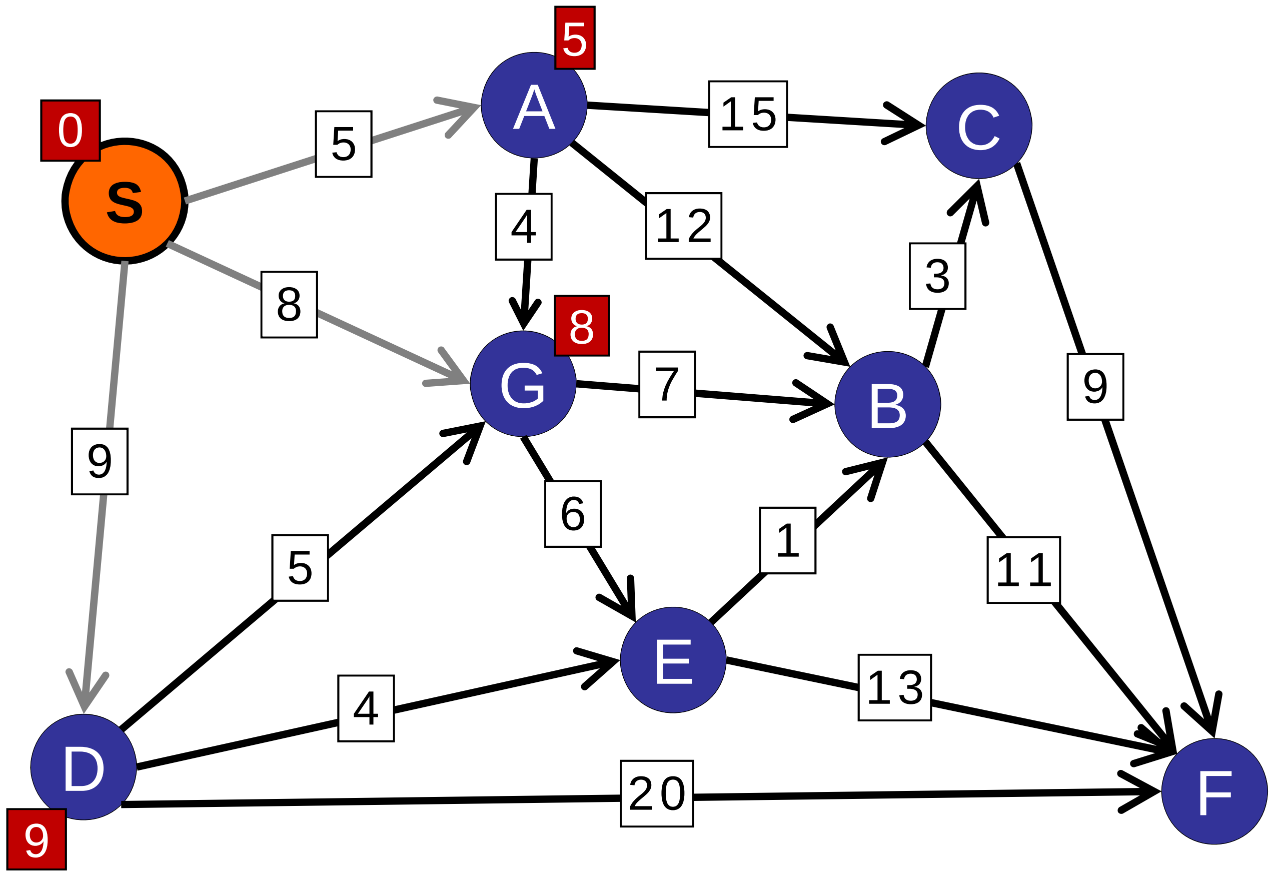
Step 1: Add source



Vertex	Dist.
S	0

Dijkstra's Algorithm

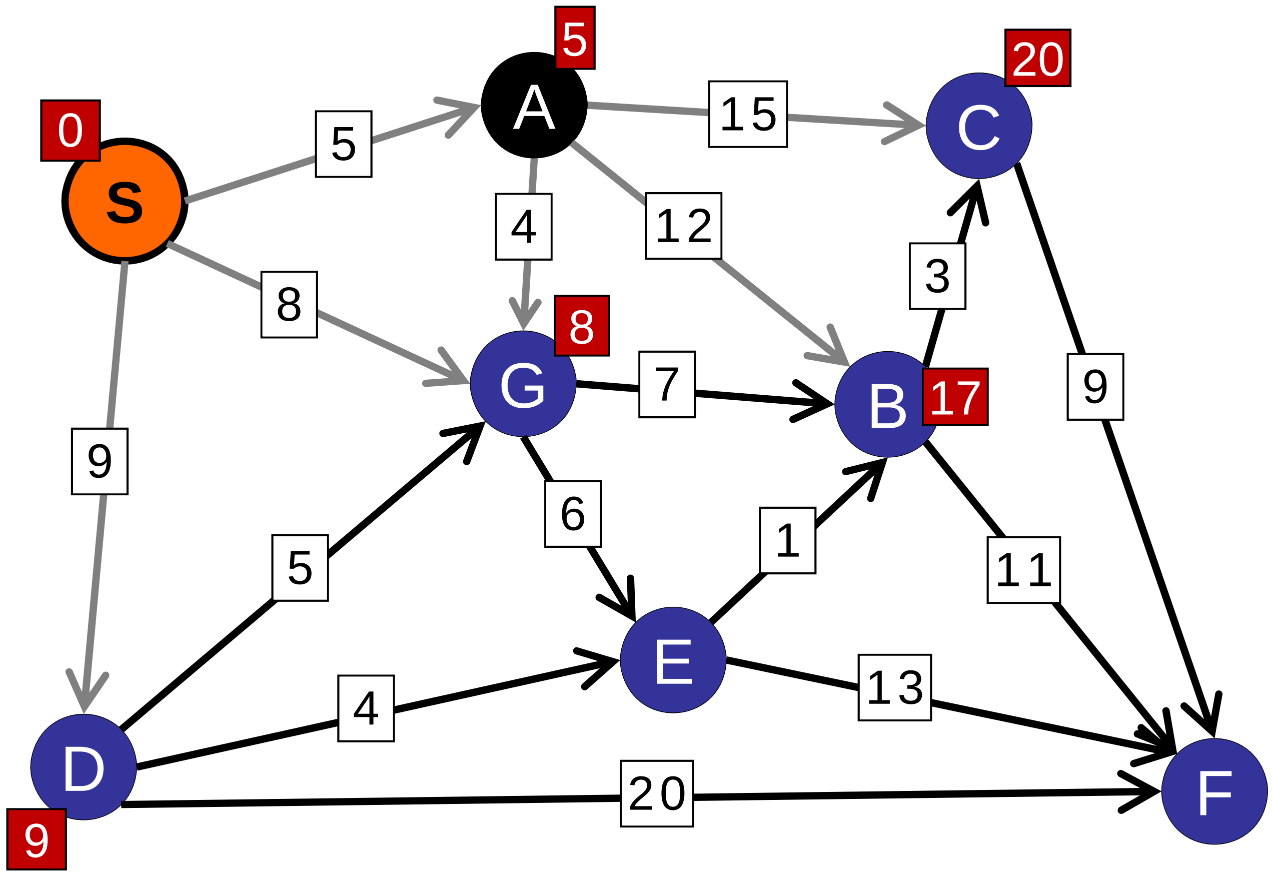
Step 2: Remove S and relax.



Vertex	Dist.
A	5
G	8
D	9

Dijkstra's Algorithm

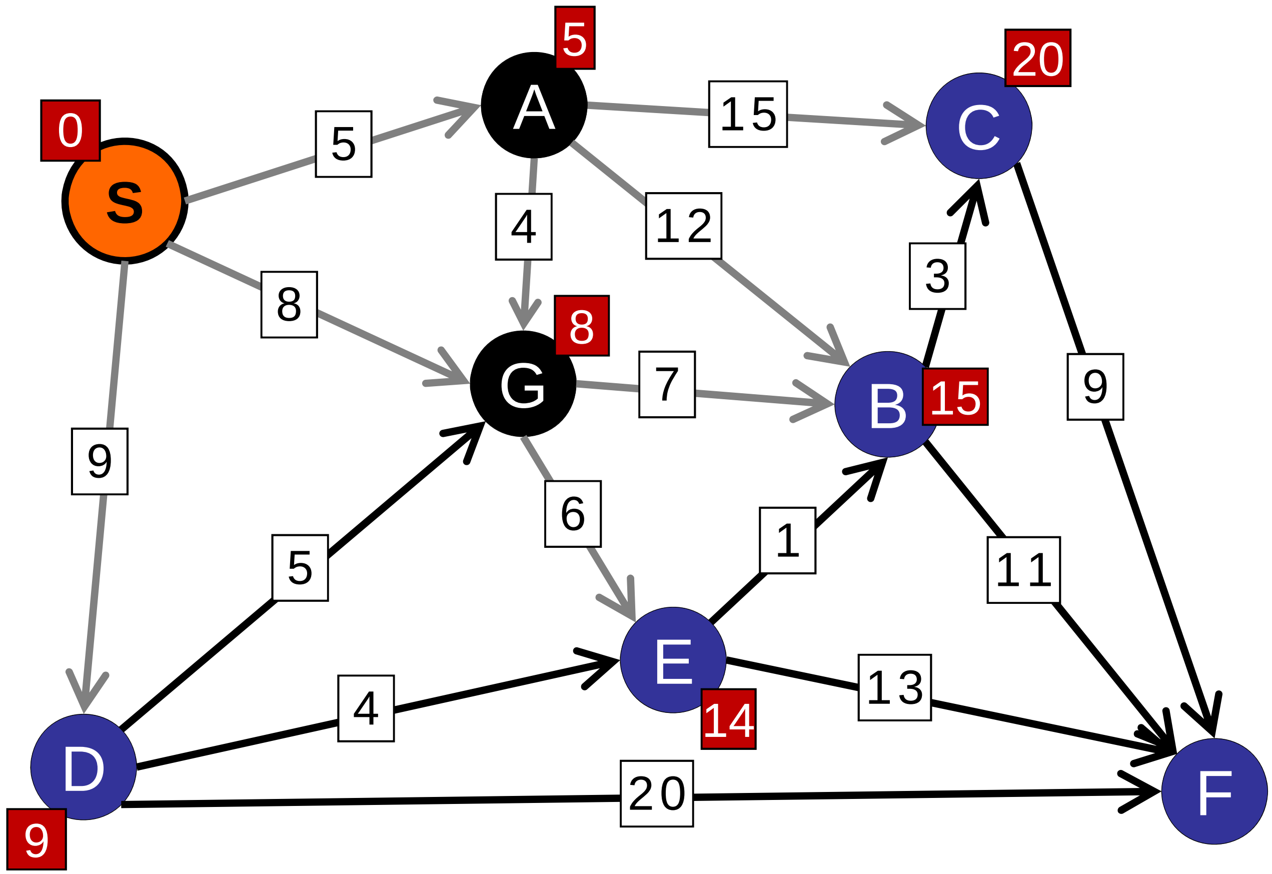
Step 3: Remove A and relax.



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

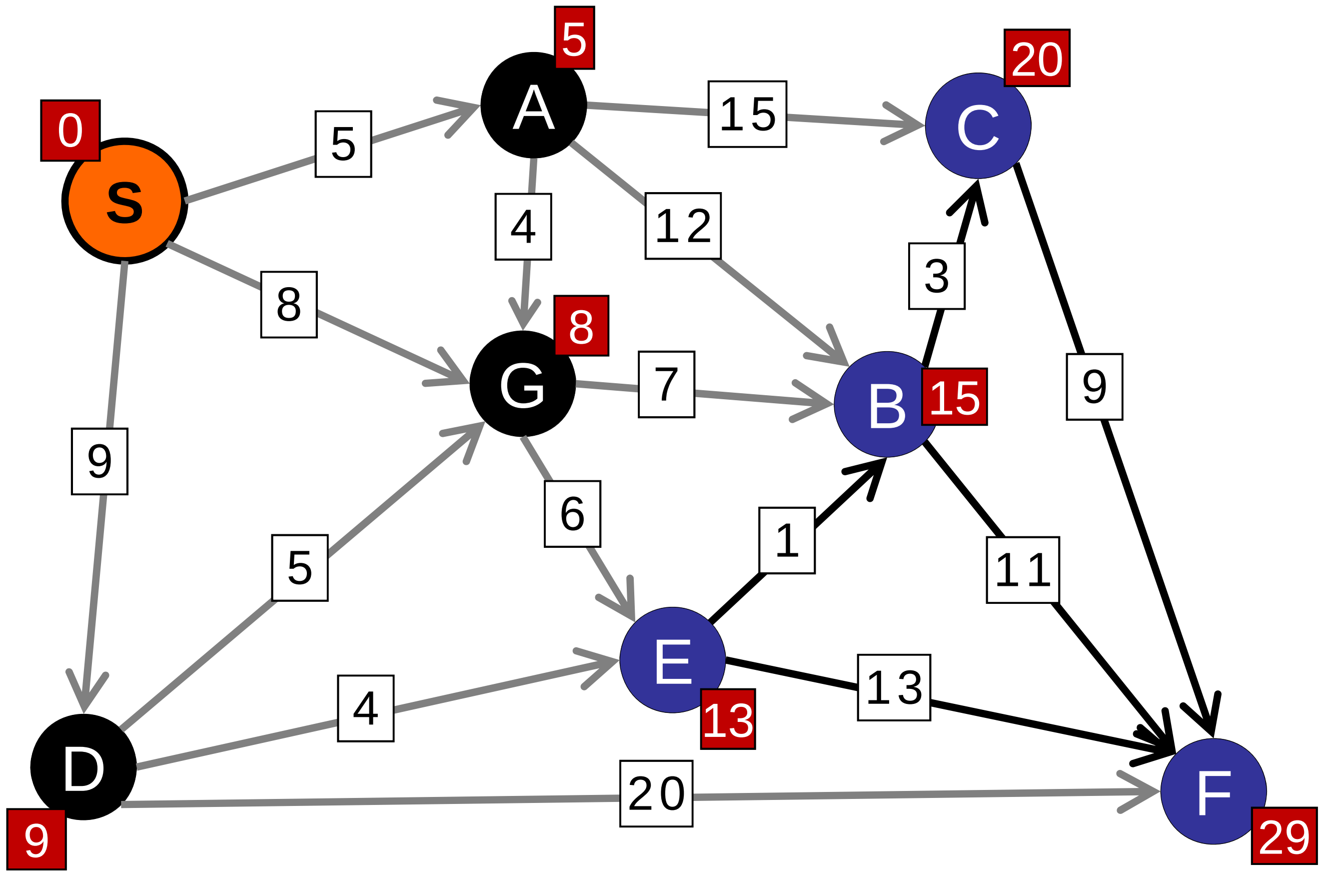
Step 4: Remove G and relax.



Vertex	Dist.
D	9
E	14
B	15
C	20

Dijkstra's Algorithm

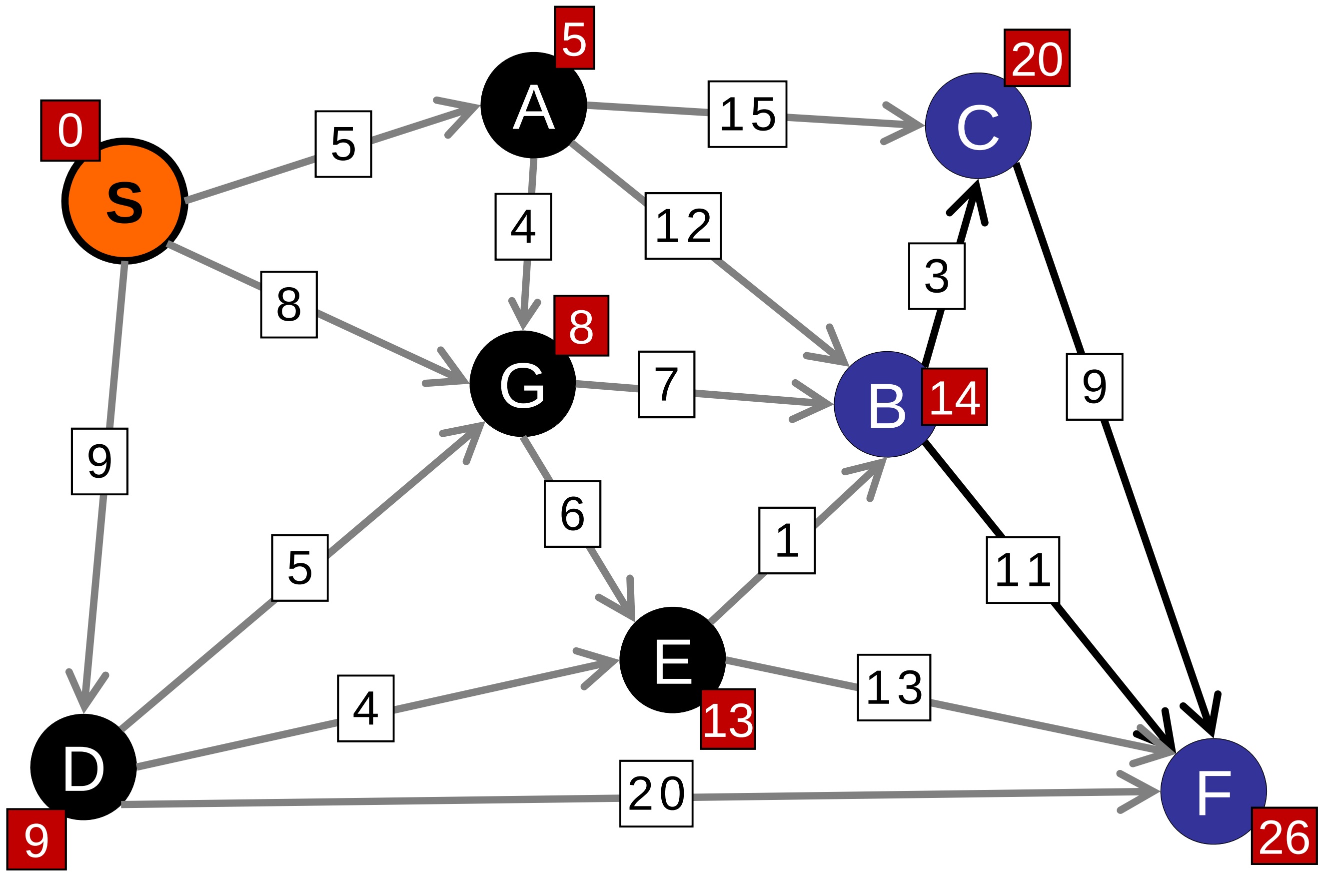
Step 5: Remove D and relax.



Vertex	Dist.
E	**13**
B	15
C	20
F	**29**

Dijkstra's Algorithm

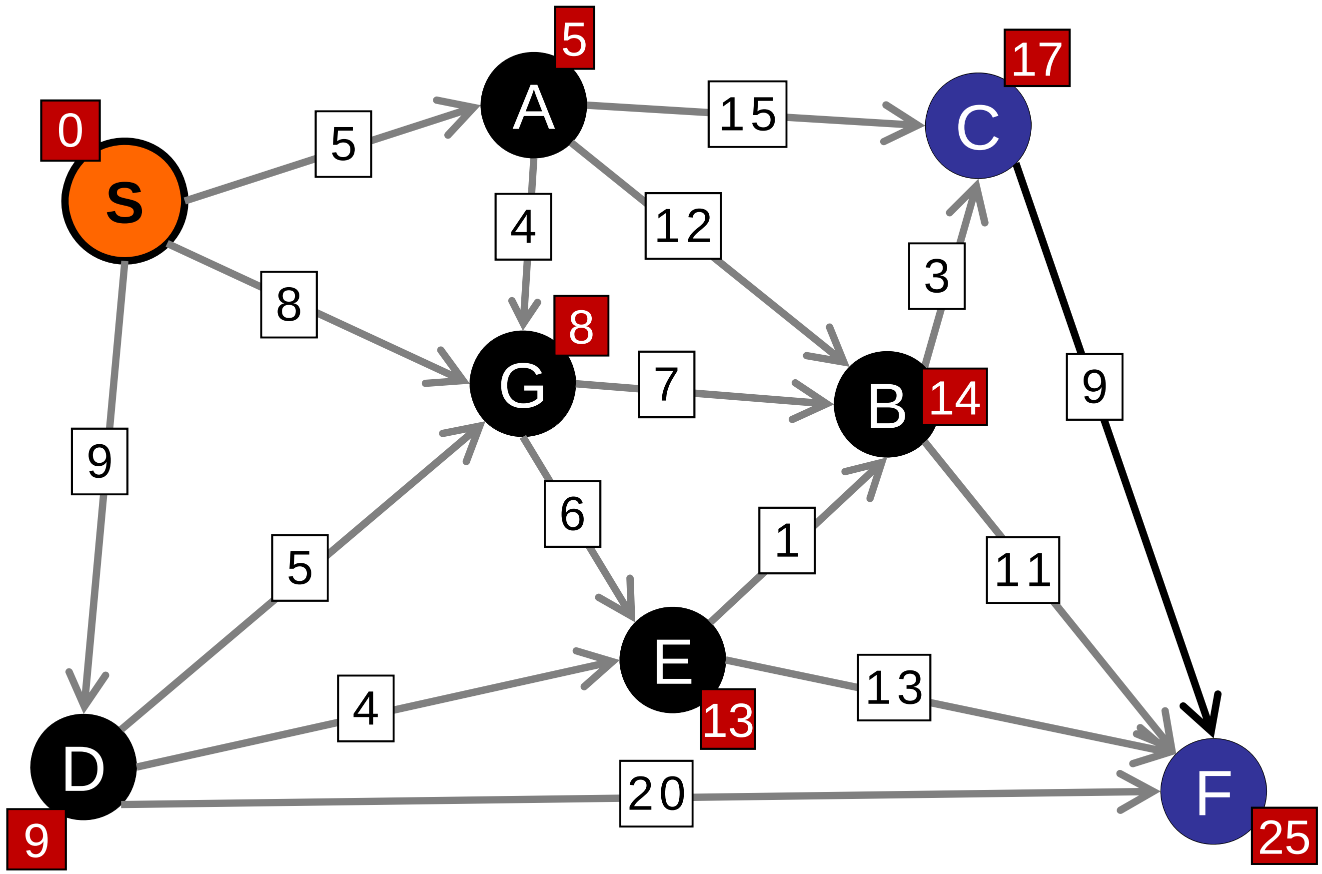
Step 5: Remove E and relax.



Vertex	Dist.
B	**14**
C	20
F	**26**

Dijkstra's Algorithm

Step 5: Remove B and relax.

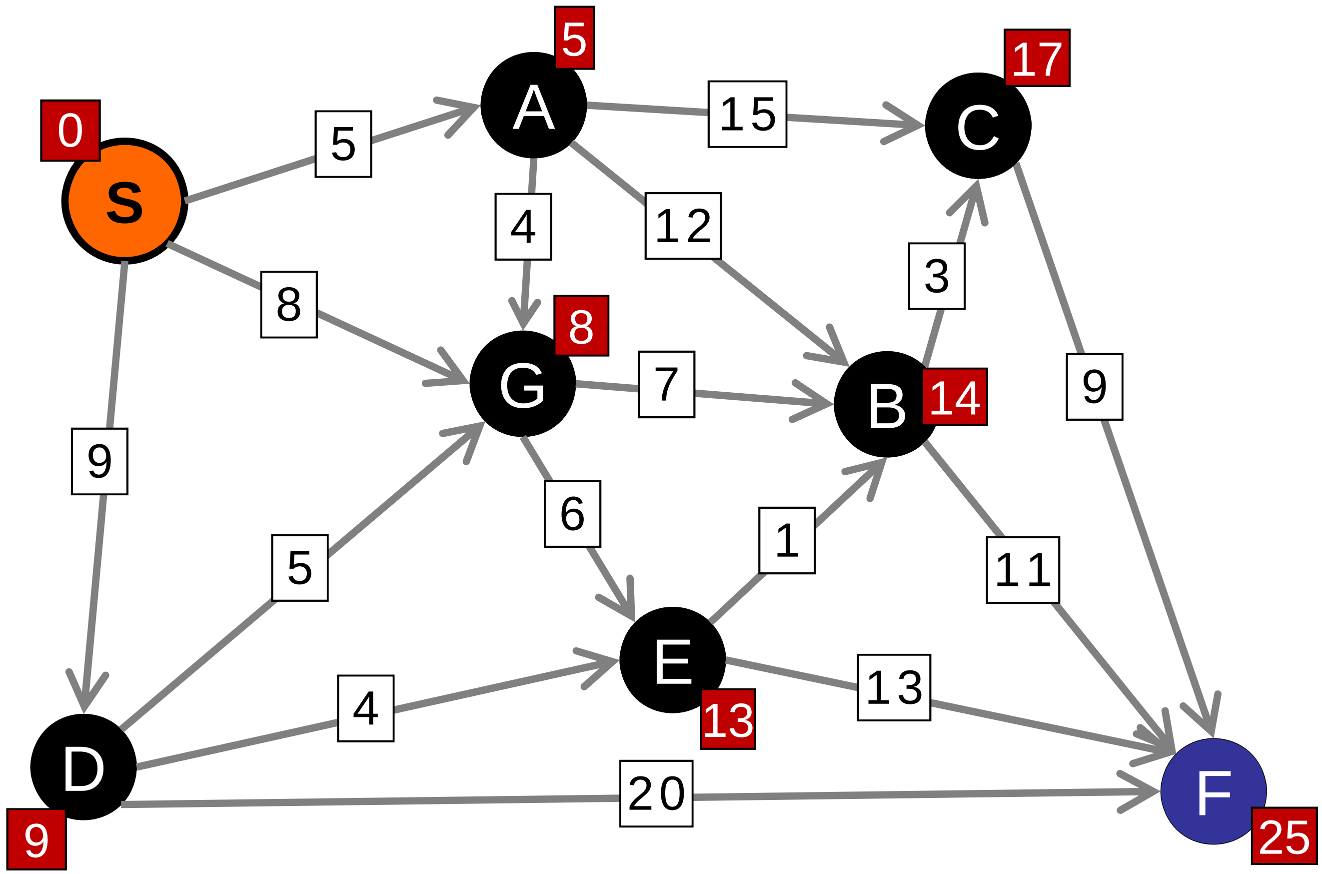


Vertex	Dist.
C	20
F	25

Dijkstra's Algorithm

Step 5: Remove C and relax.

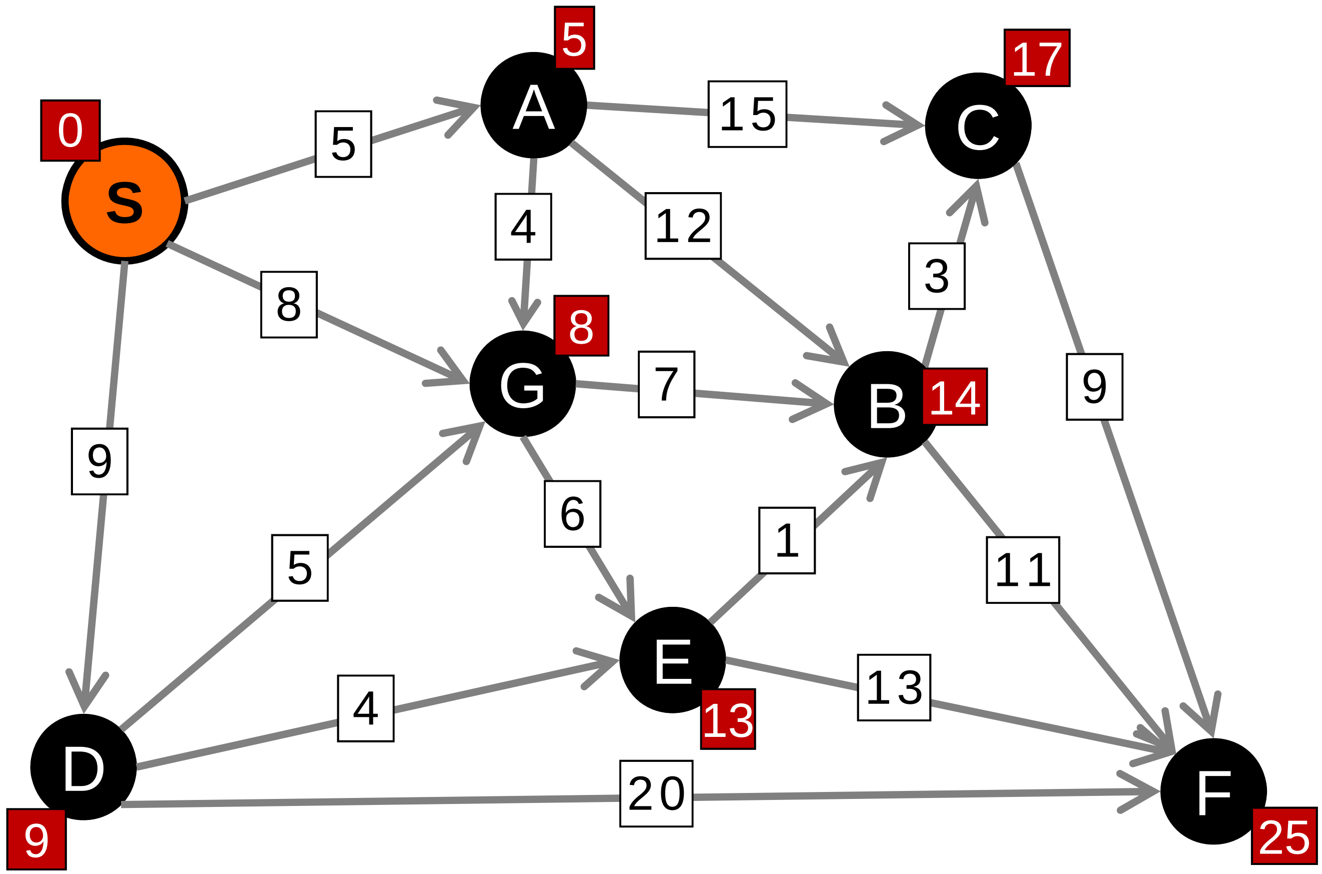
Vertex	Dist.
F	25



Dijkstra's Algorithm

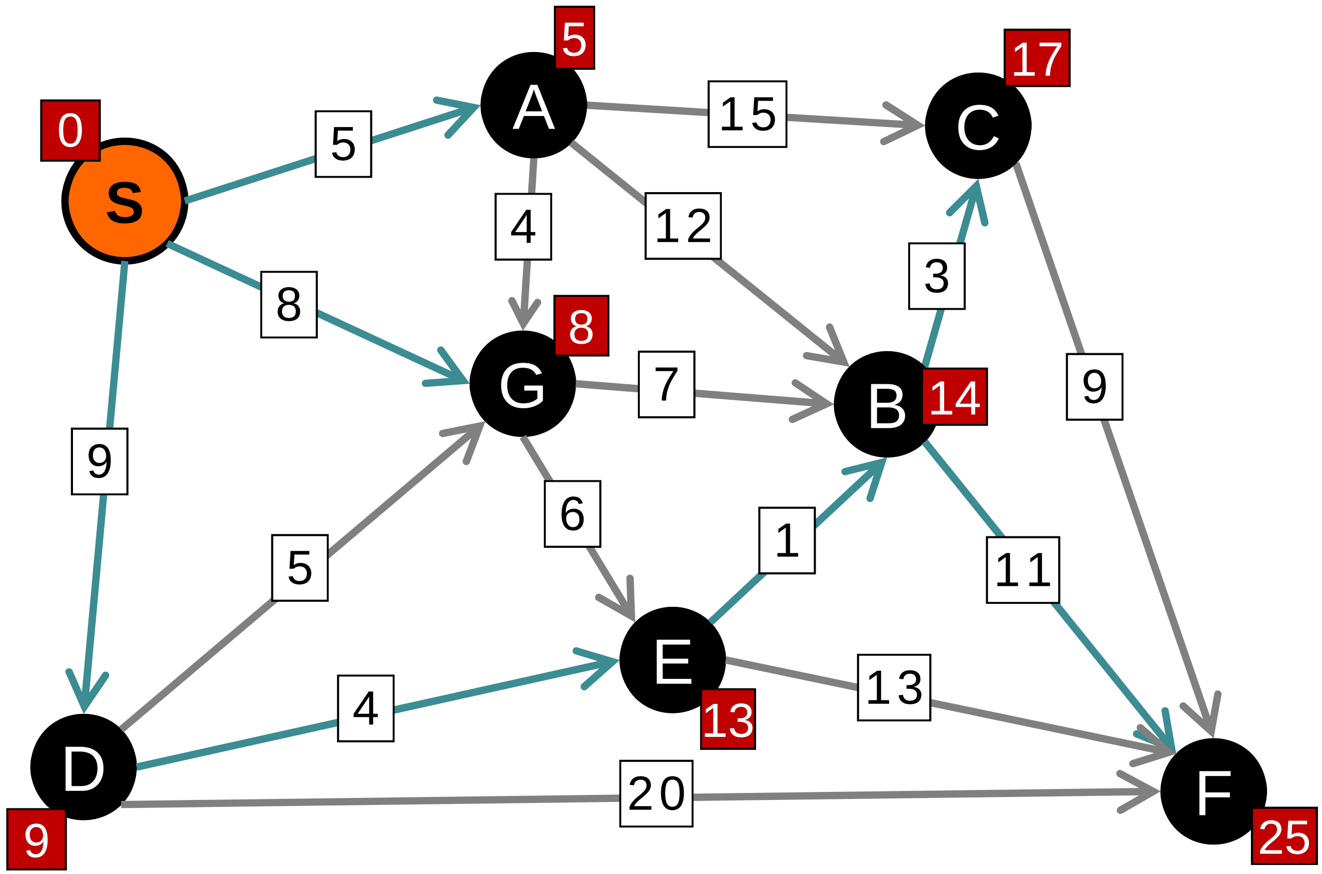
Step 5: Remove F and relax.

Vertex	Dist.



Dijkstra's Algorithm

Done

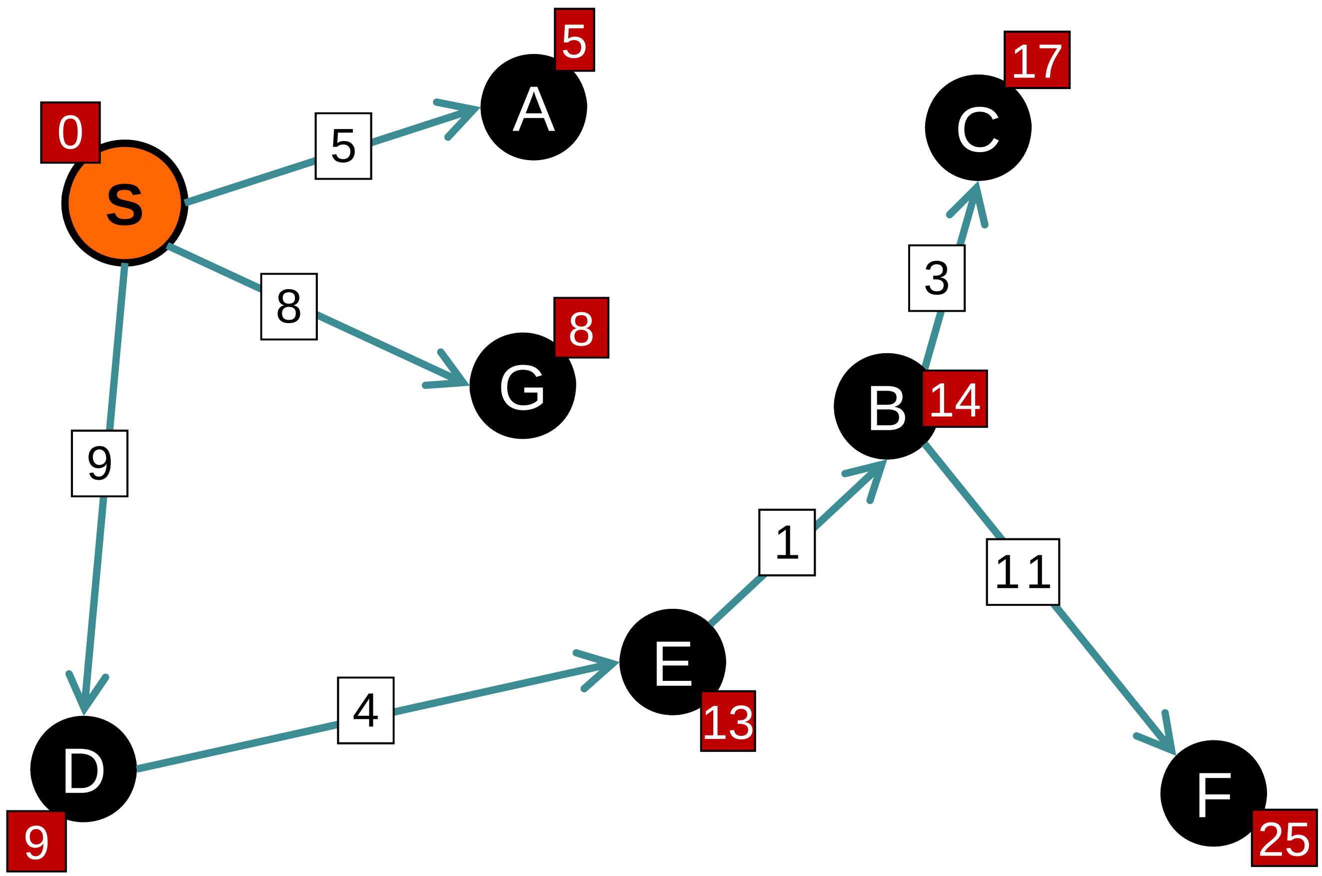


Vertex	Dist.

Dijkstra's Algorithm

Shortest Path Tree

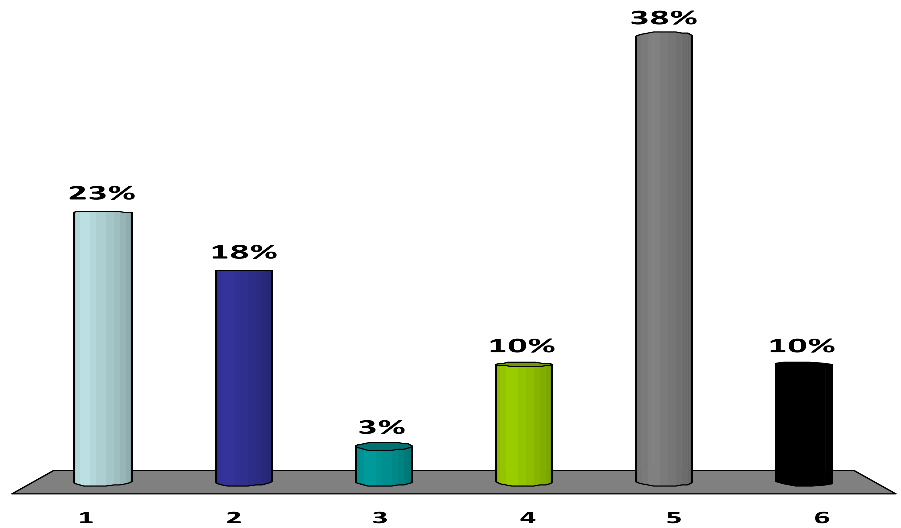
Vertex	Dist.



What data structure to store vertices/distances?

1. Array
2. Linked list
3. Stack
4. Queue
5. ✓ AVL Tree
6. Huh?

Vertex	Dist.
B	**14**
C	20
F	**26**



Abstract Data Type

Priority Queue

```
interface  IPriorityQueue<Key, Priority>
```

`void`	`insert(Key k, Priority p)`	*insert k with priority p*
`Data`	`extractMin()`	*remove key with minimum priority*
`void`	`decreaseKey(Key k, Priority p)`	*reduce the priority of key k to priority p*
`boolean`	`contains(Key k)`	*does the priority queue contain key k?*
`boolean`	`isEmpty()`	*is the priority queue empty?*

Notes:

Assume data items are unique.

```
public Dijkstra{
     private Graph G;
     private IPriorityQueue pq = new PriQueue();
     private double[] distTo;

     searchPath(int start) {
          pq.insert(start, 0.0);
          distTo = new double[G.size()];
          Arrays.fill(distTo, INFTY);
          distTo[start] = 0;
          while (!pq.isEmpty()) {
               int w = pq.deleteMin();
               for (Edge e : G[w].nbrList)
                    relax(e);
          }
     }
```

Dijkstra’s Algorithm

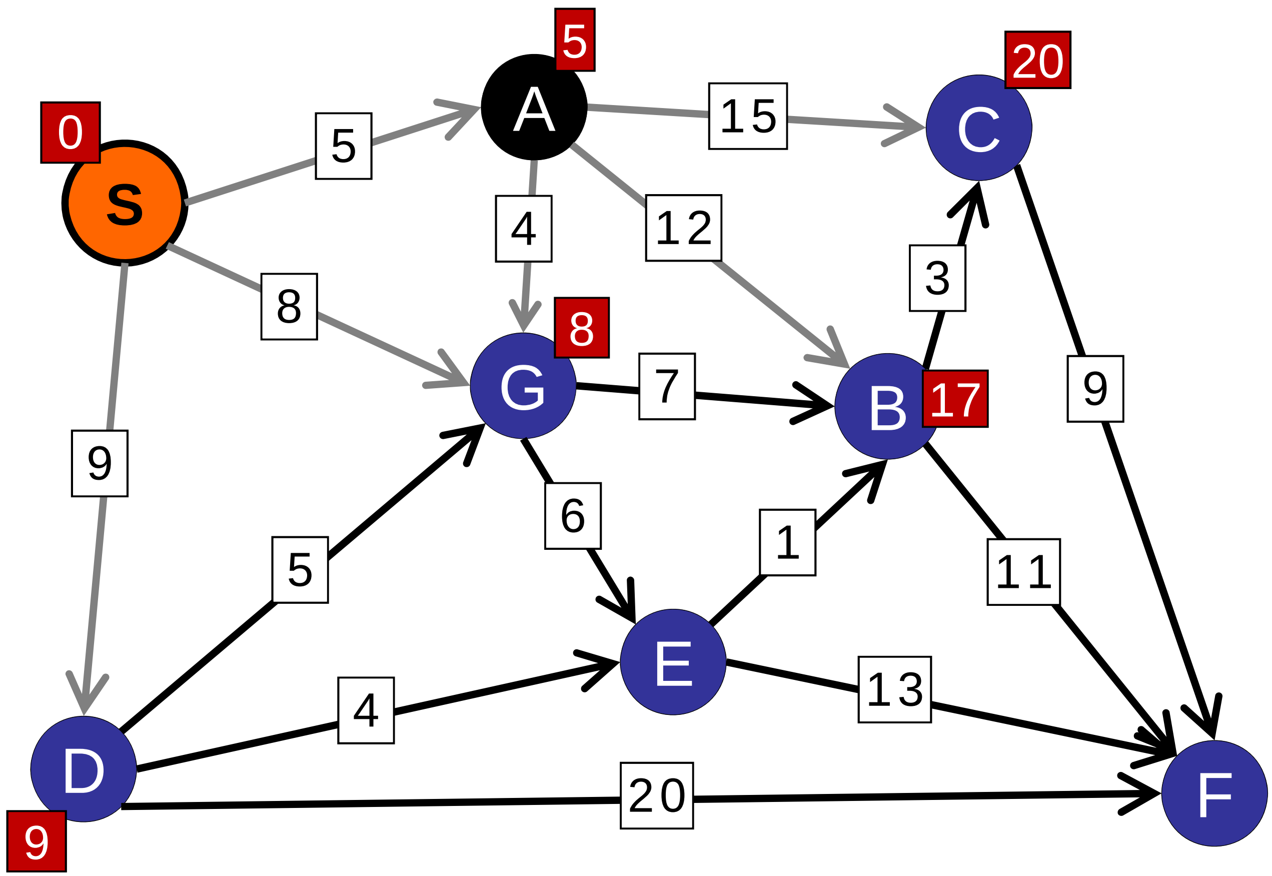
```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Dijkstra's Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Dijkstra’s Algorithm

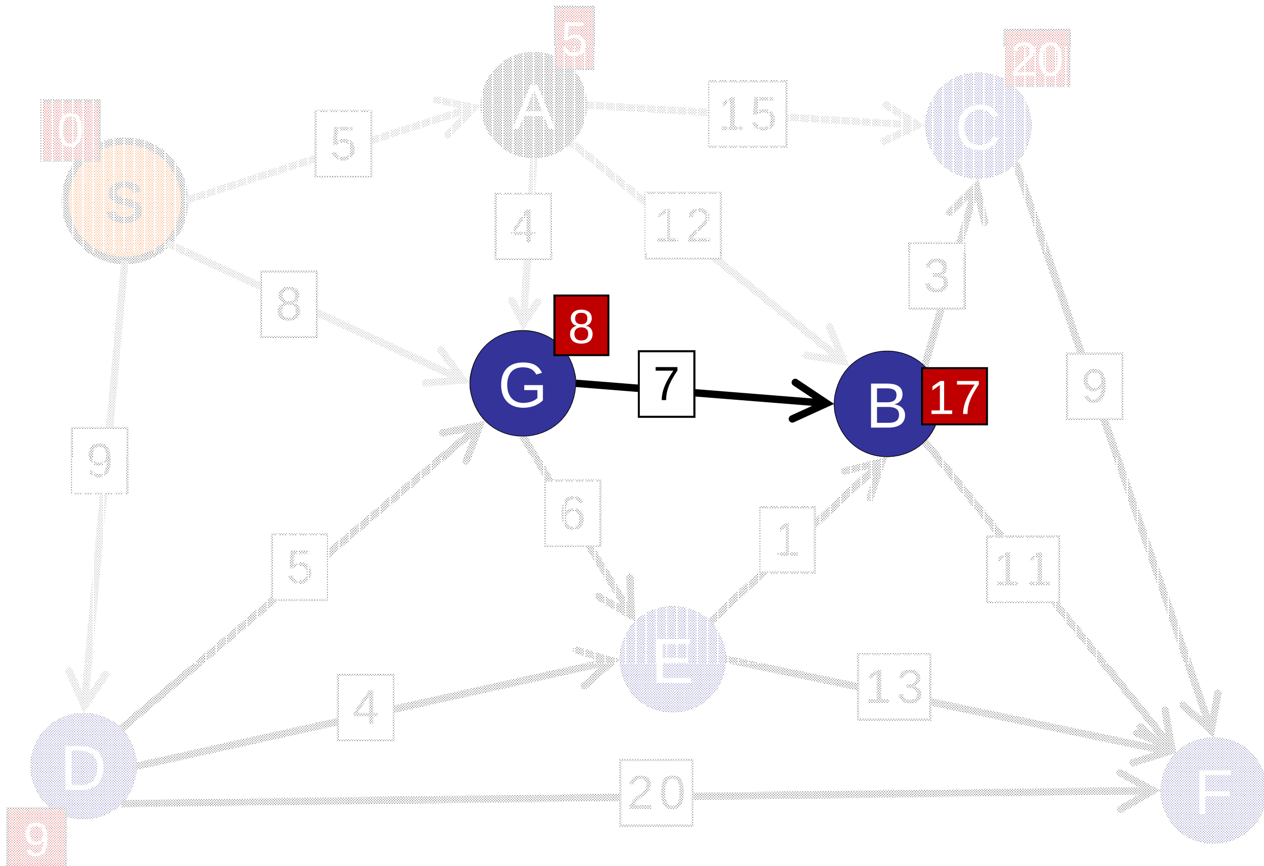
Step 3: Remove A and relax.



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

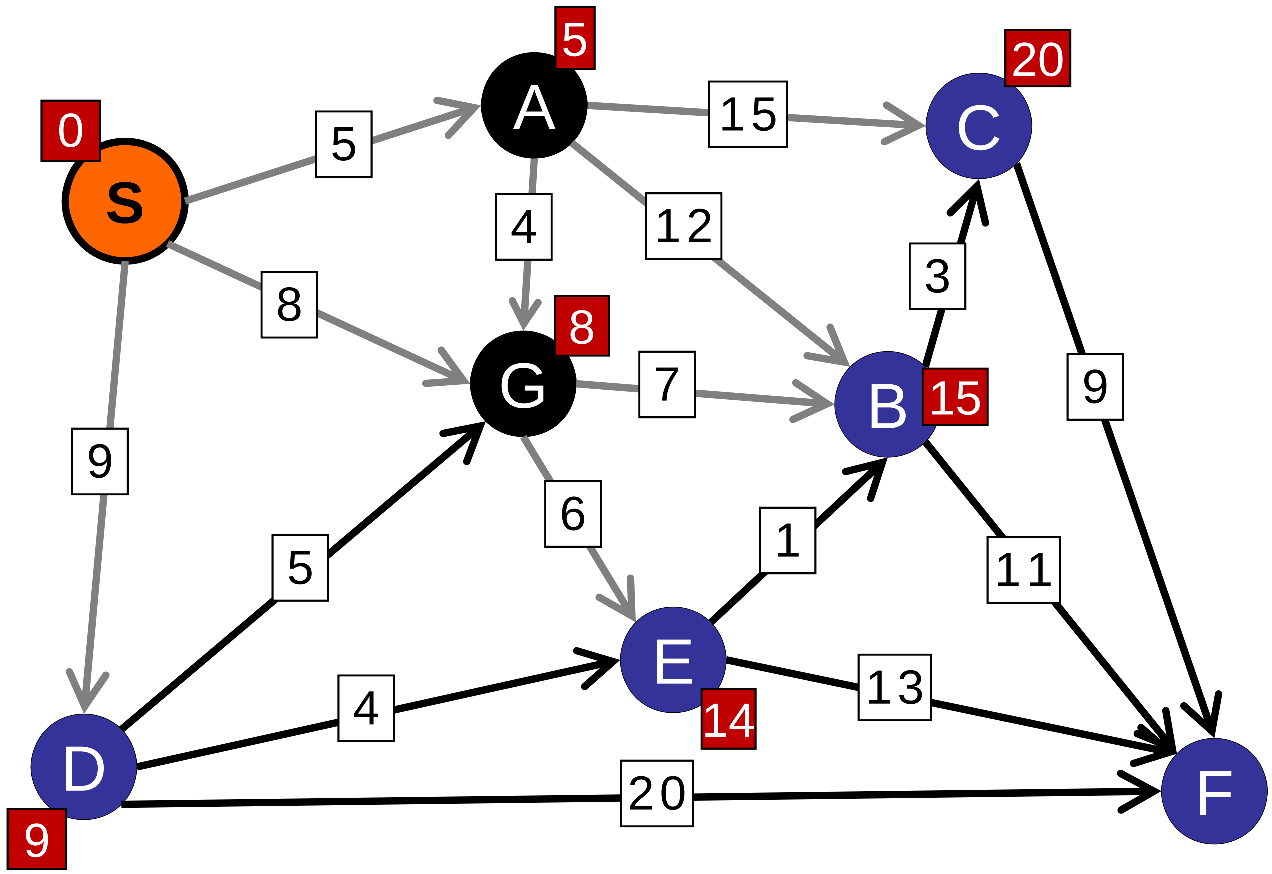
Step 3: Remove A and relax.



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

Step 4: Remove G and relax.



Vertex	Dist.
D	9
E	14
B	15
C	20

Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Abstract Data Type

Priority Queue

```
interface  IPriorityQueue<Key, Priority>
```

`void`	`insert(Key k, Priority p)`	*insert k with priority p*
`Data`	`extractMin()`	*remove key with minimum priority*
`void`	`decreaseKey(Key k, Priority p)`	*reduce the priority of key k to priority p*
`boolean`	`contains(Key k)`	*does the priority queue contain key k?*
`boolean`	`isEmpty()`	*is the priority queue empty?*

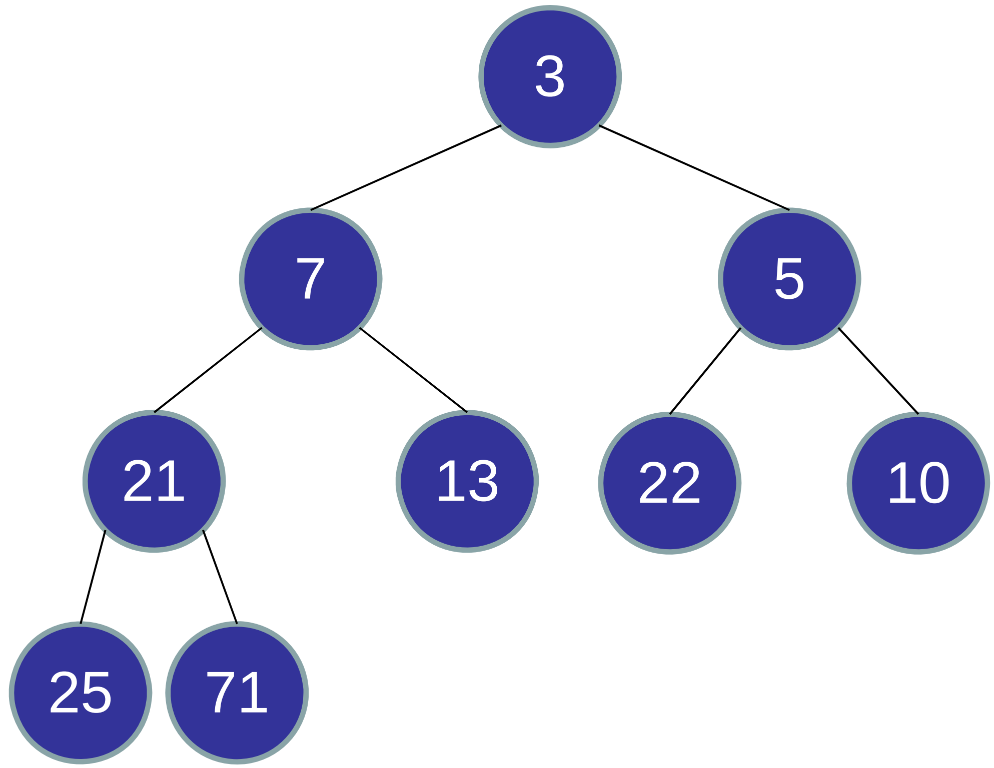
Notes:

Assume data items are unique.

Priority Queue

Binary Heap

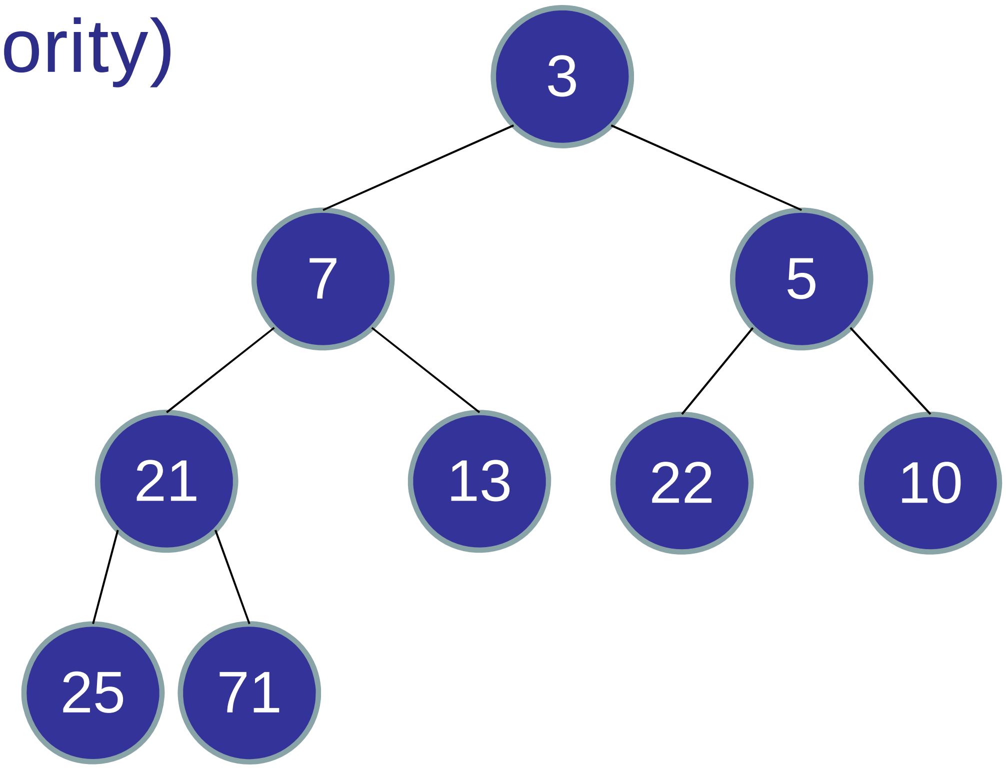
- Complete binary tree
- deleteMin: O(log n)
  - remove root
  - swap leaf to root
  - bubble down
- insert: O(log n)
  - add new leaf
  - bubble up



Priority Queue

Binary Heap

- How do we find a key? **(Hint: not a search tree!)**
- contains(key)
- decreaseKey(key, priority)



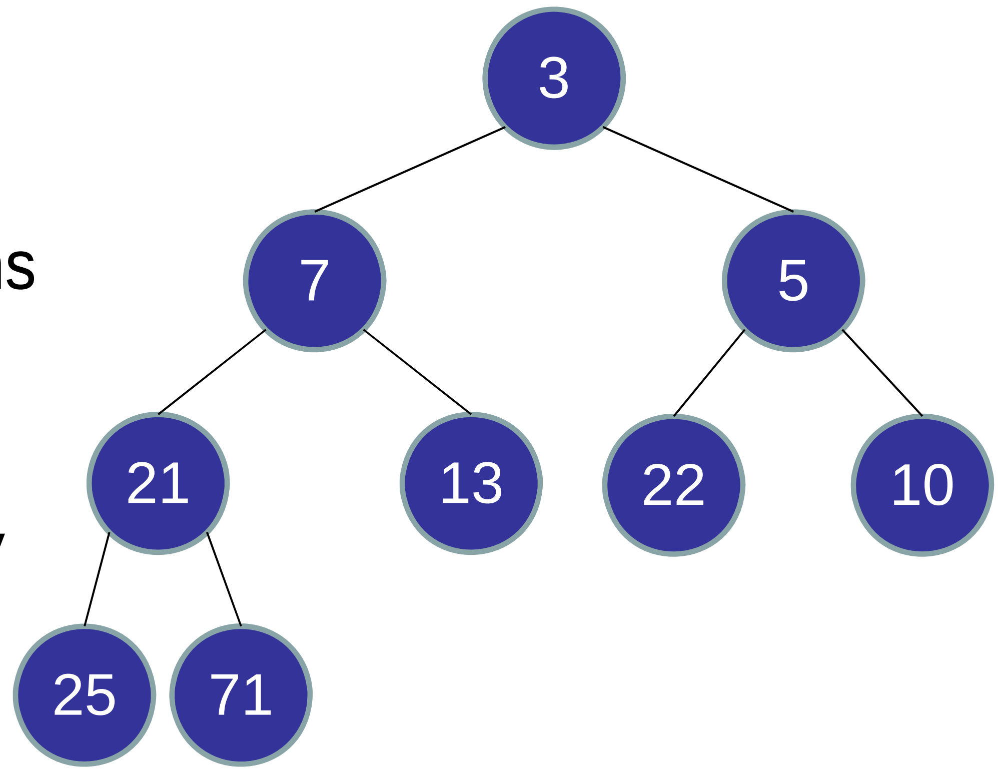
Priority Queue

Binary Heap

– decreaseKey(key, priority): O(logn)

– Hash Table:

- Map keys to locations in the binary tree.
- Update hash table whenever the binary tree changes.

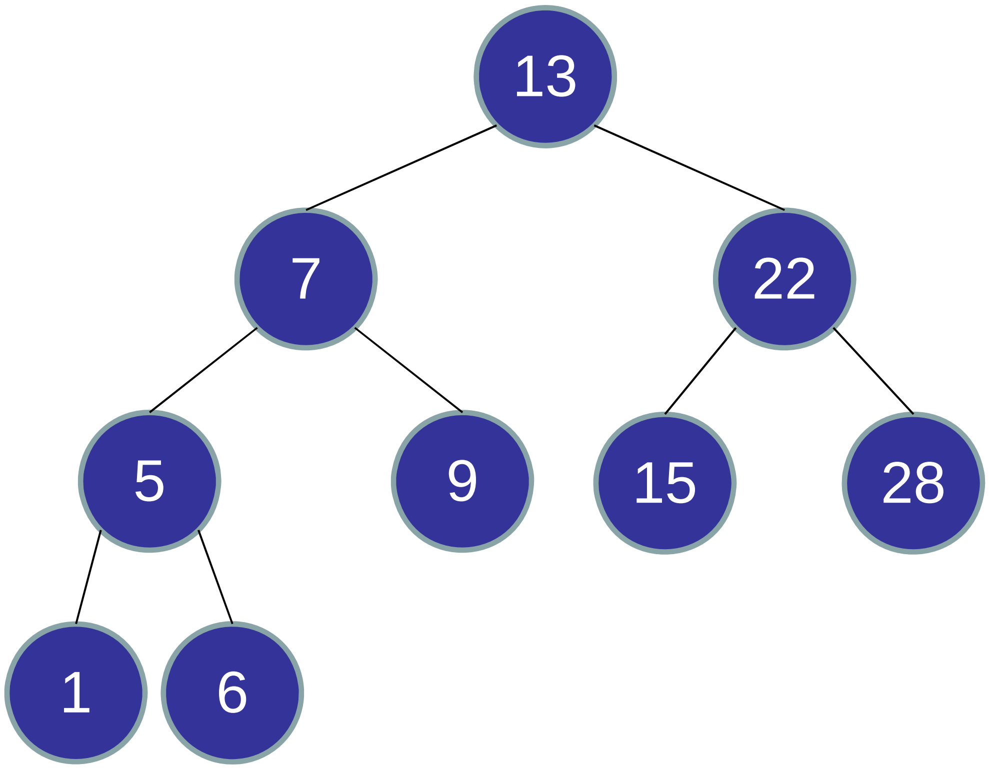


0	1	2	3	4	5	6	7	8	[9]	10	11
	3	7	6	21	13	22	10	25	71		

Priority Queue

AVL Tree

- Indexed by: priority
- Existing operations:
  - deleteMin()
  - insert(key, priority)

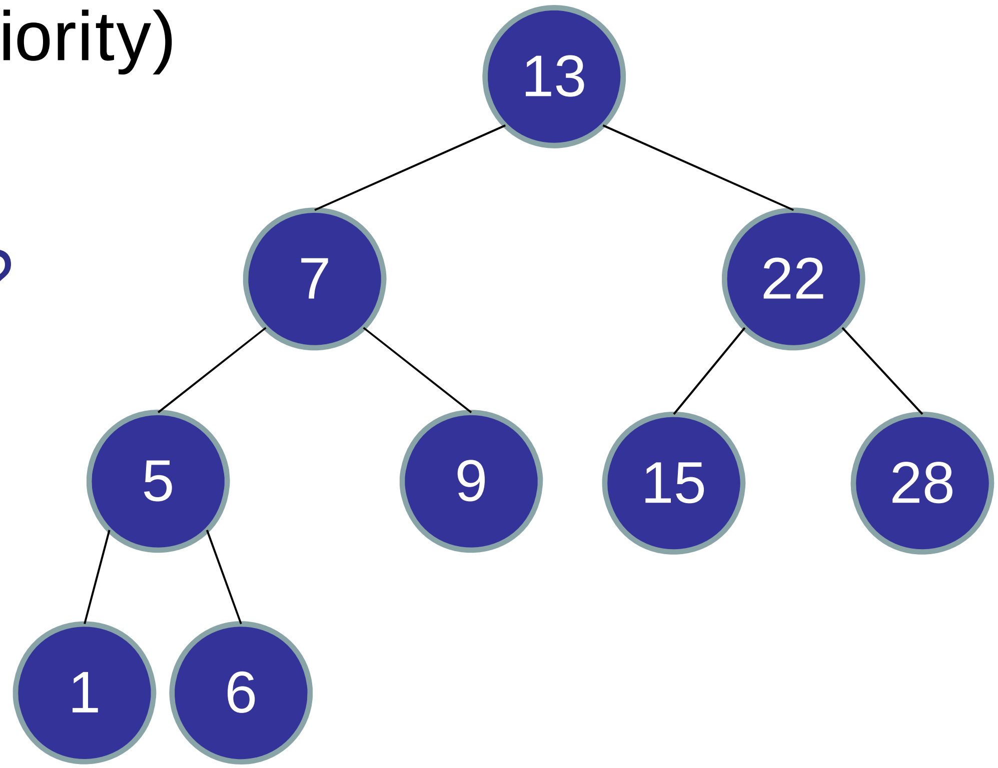


Priority Queue

AVL Tree

- Other operations:
  - contains()
  - decreaseKey(key, priority)

- How to find a vertex?

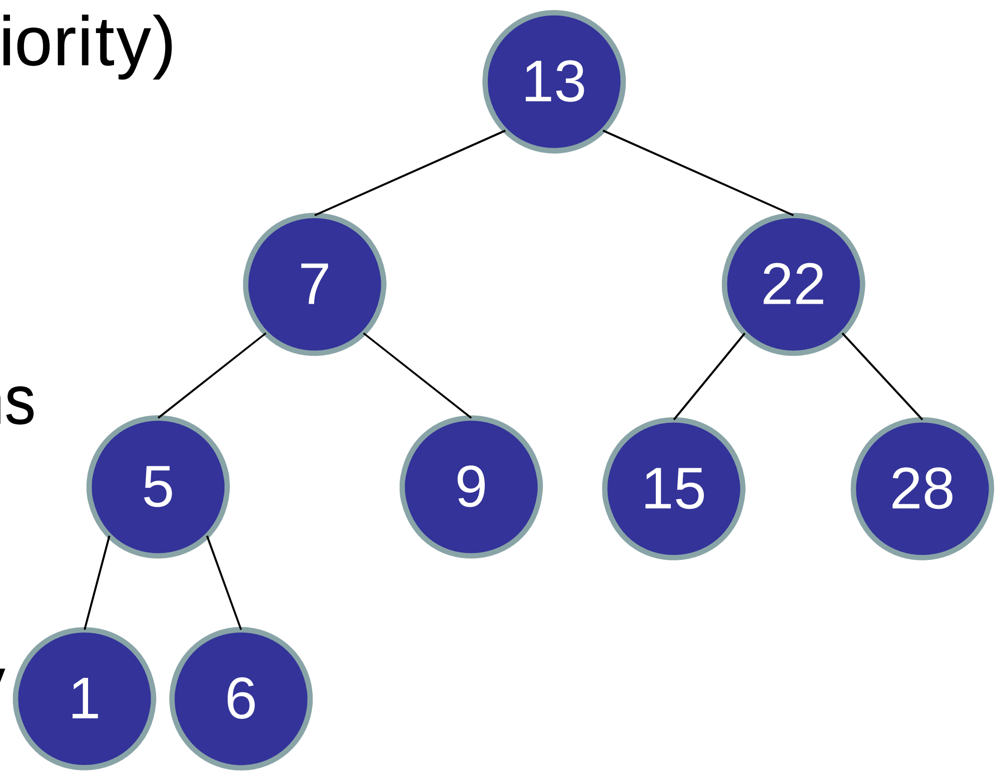


Priority Queue

AVL Tree

– Other operations:
  - contains()
  - decreaseKey(key, priority)

– Hash Table:
  - Map keys to locations in the binary tree.
  - Update hash table whenever the binary tree changes.



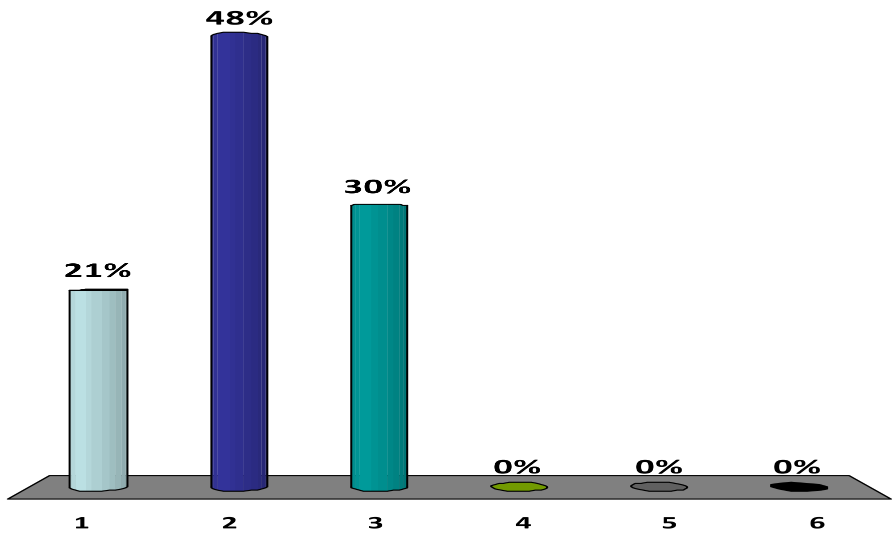
Dijkstra's Algorithm

Priority Queue by AVL tree:

- insert(key, priority): O(log n)
- deleteMin(): O(log n)
- decreaseKey(key, priority): O(log n)
- contains(key): O(1)

What is the running time of Dijkstra's Algorithm, using an AVL tree Priority Queue?

1. $O(V + E)$
2. $O(E \log V)$
3. $O(V \log V)$
4. $O(V^2)$
5. $O(VE)$
6. None of the above



```
public Dijkstra{
    private Graph G;
    private MinPriQueue pq = new MinPriQueue();
    private double[] distTo;

    searchPath(int start) {
        pq.insert(start, 0.0);
        distTo = new double[G.size()];
        Arrays.fill(distTo, INFTY);
        distTo[start] = 0;
        while (!pq.isEmpty()) {
            int w = pq.deleteMin();
            for (Edge e : G[w].nbrList)
                relax(e);
        }
    }
```

How many times?

How many times?

Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Dijkstra’s Algorithm

Analysis:

- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- relax / decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

Dijkstra’s Algorithm

Why does it work?

Dijkstra's Algorithm

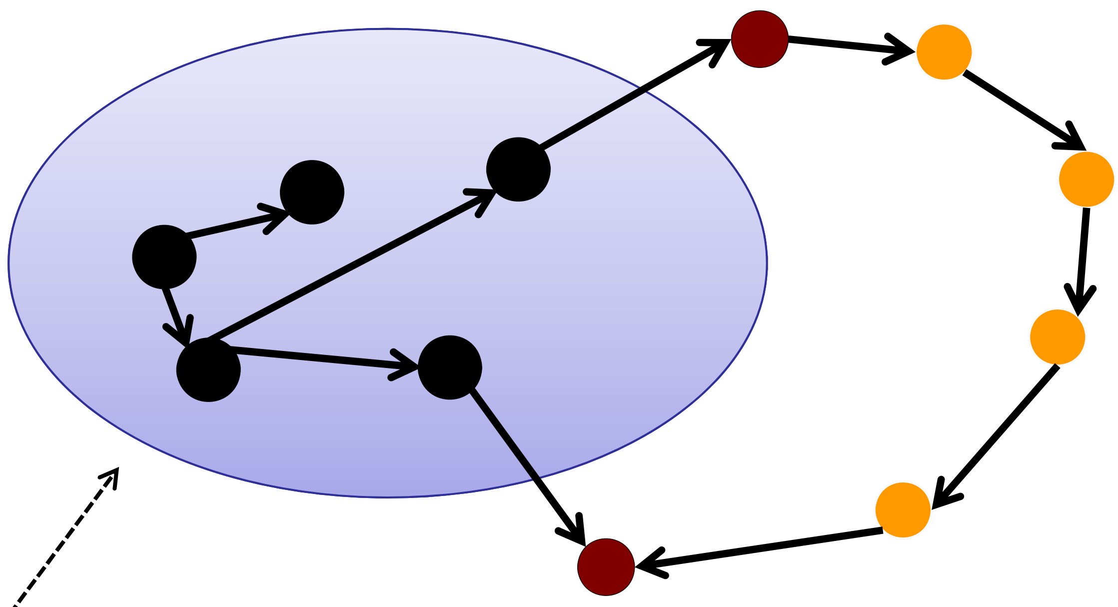
Proof by induction:

- Every "finished" (dequeued) vertex has a correct estimate.
  - Namely, shortest path is found for that vertex
- Initially: only "finished" vertex is start.

Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.

fringe vertices:
neighbor of a
finished vertex.



other vertices:
no known
estimate

finished vertices:
distance is accurate.

fringe vertices: in priority queue
neighbor of a finished vertex.

Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

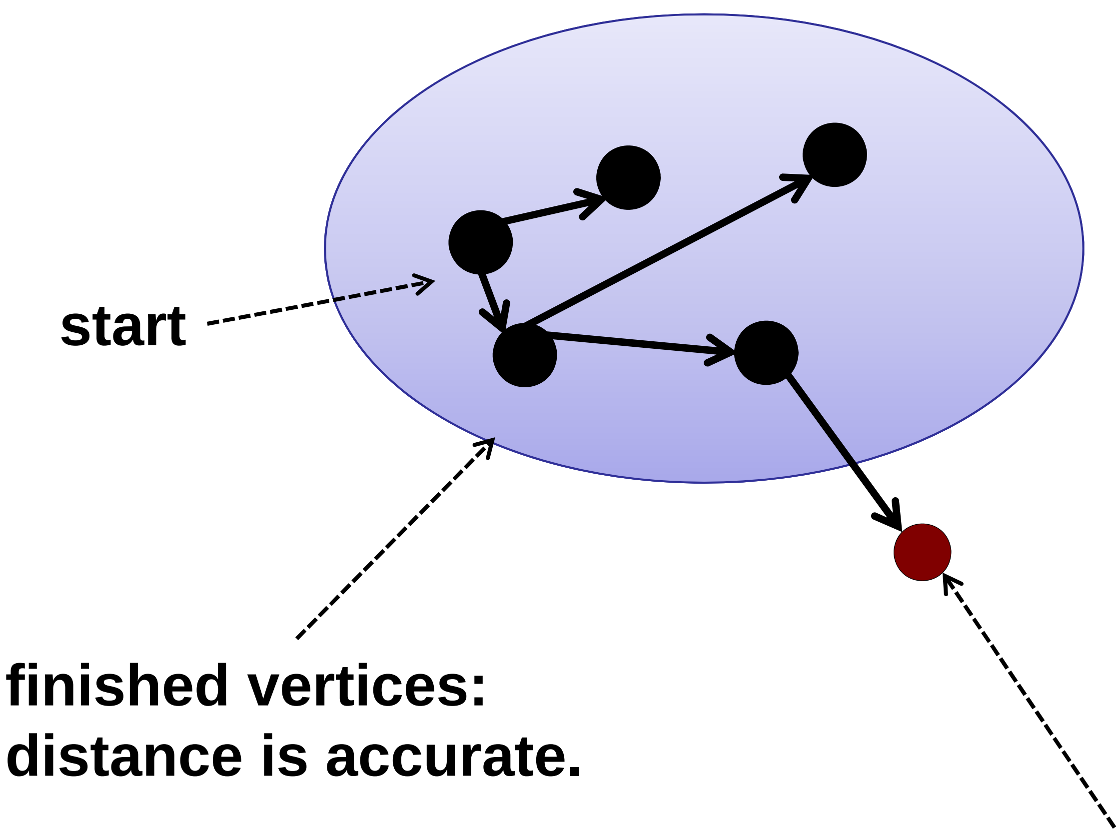
Dijkstra's Algorithm

Proof by induction:

- Every "finished" vertex has correct estimate.
- Initially: only "finished" vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

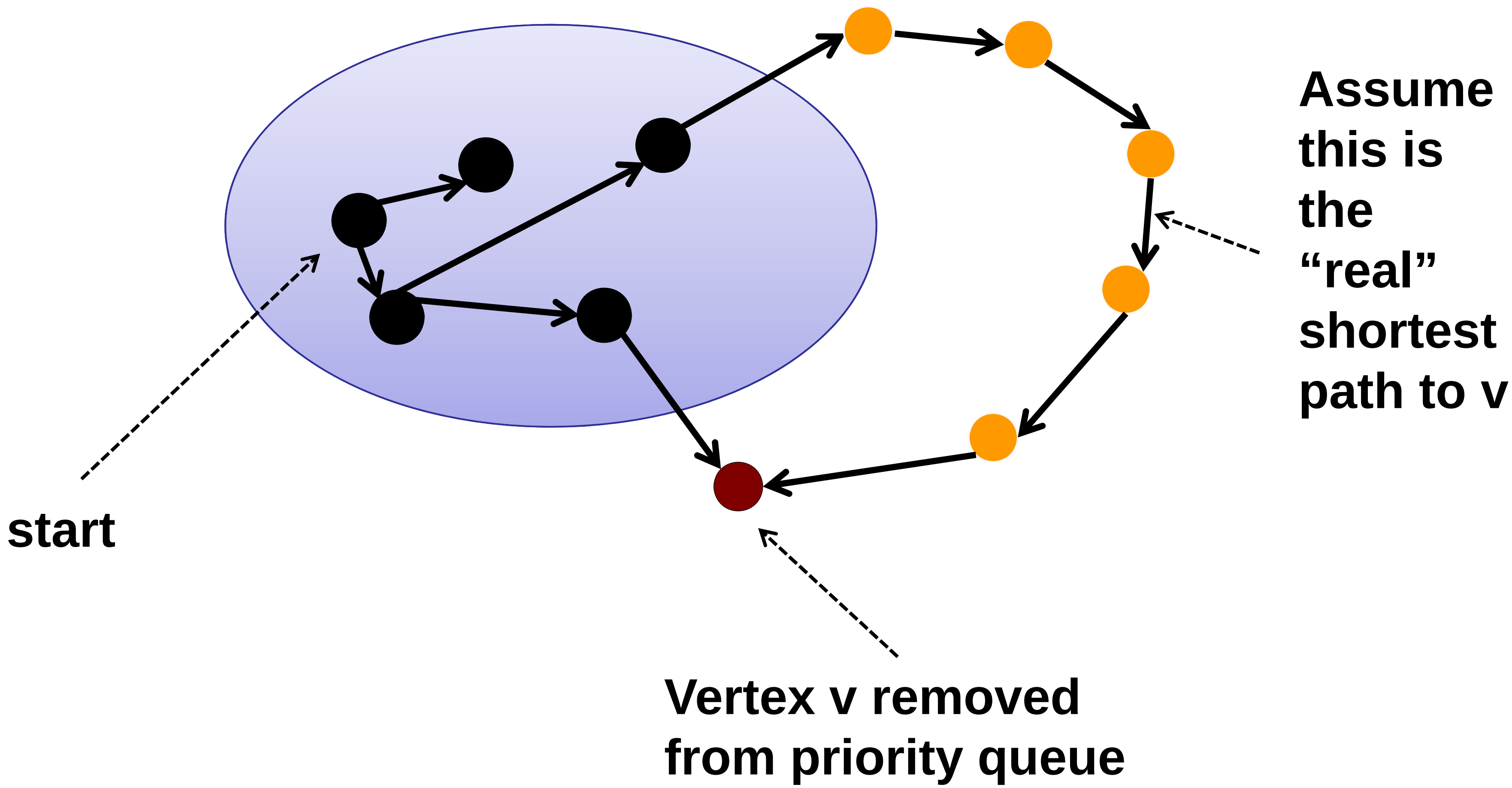
Dijkstra's Algorithm



Vertex v going to be removed from priority queue next. Thus, with minimum distance amount the unfinished

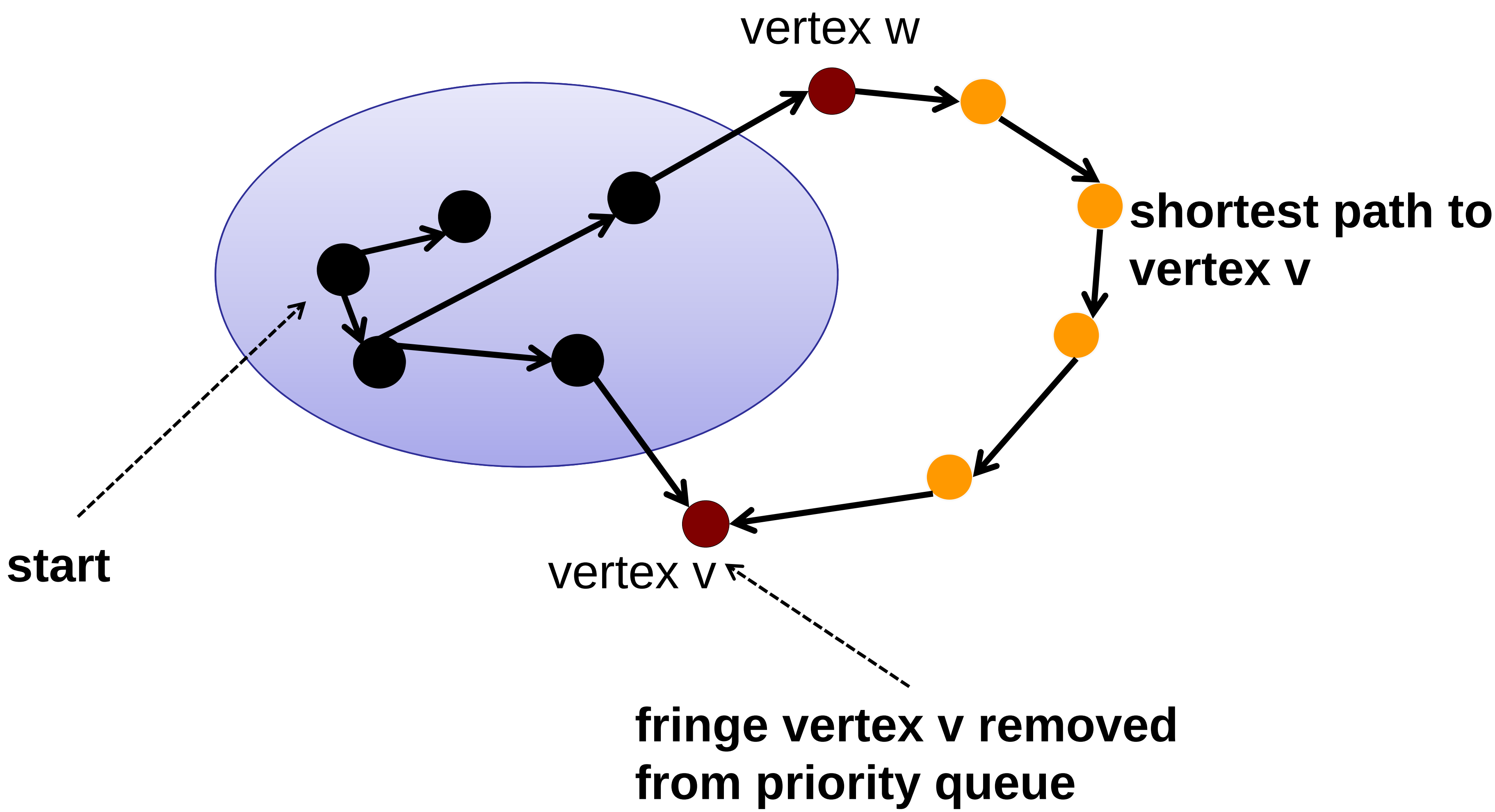
Dijkstra's Algorithm

Assume NOT. The current estimate is not the shortest path.



Dijkstra’s Algorithm

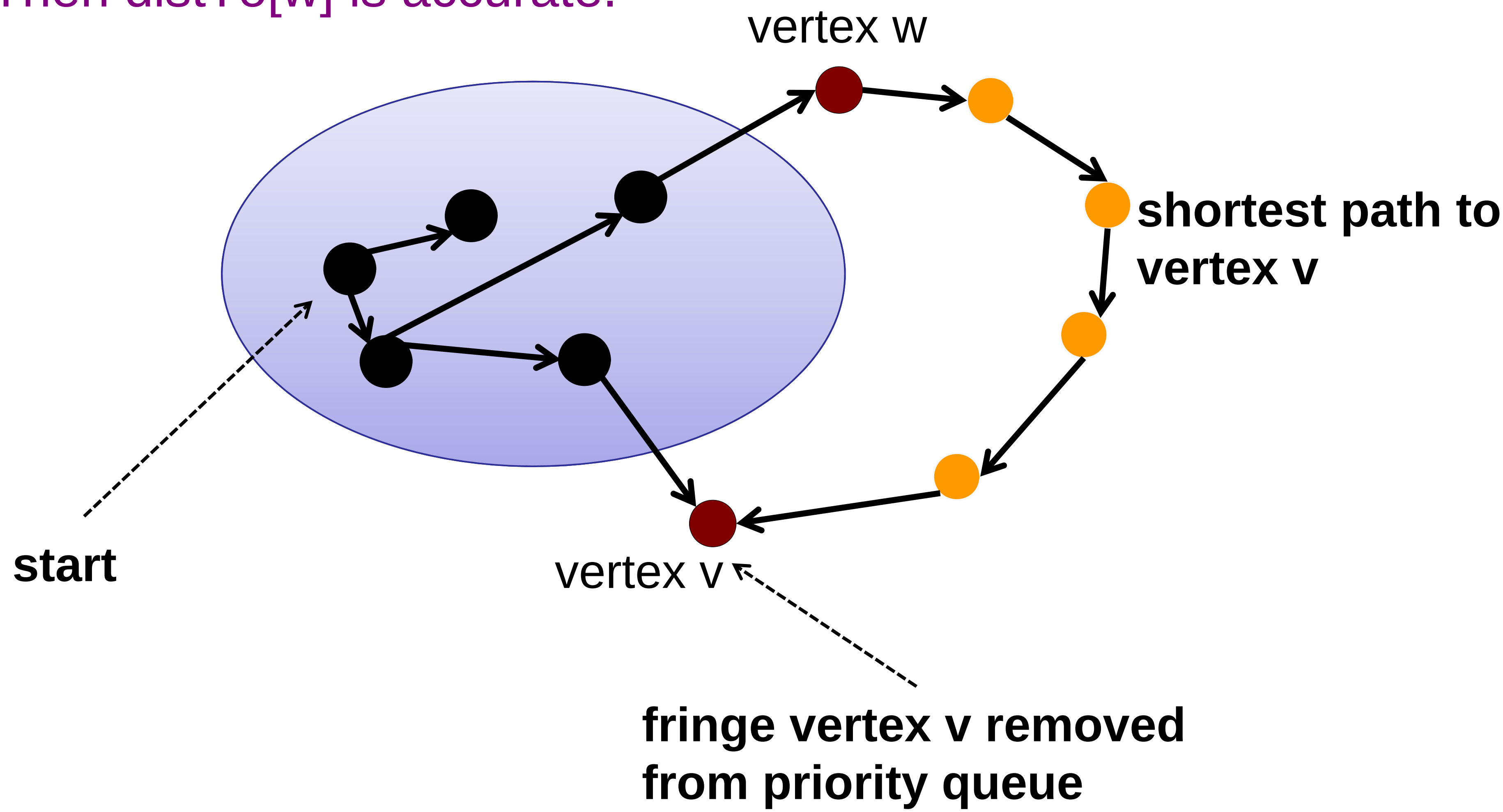
There must be a vertex w in the current PQ on this “real” path.



Dijkstra’s Algorithm

If P is shortest path to v, then prefix of P is shortest path to w.

Then distTo[w] is accurate.

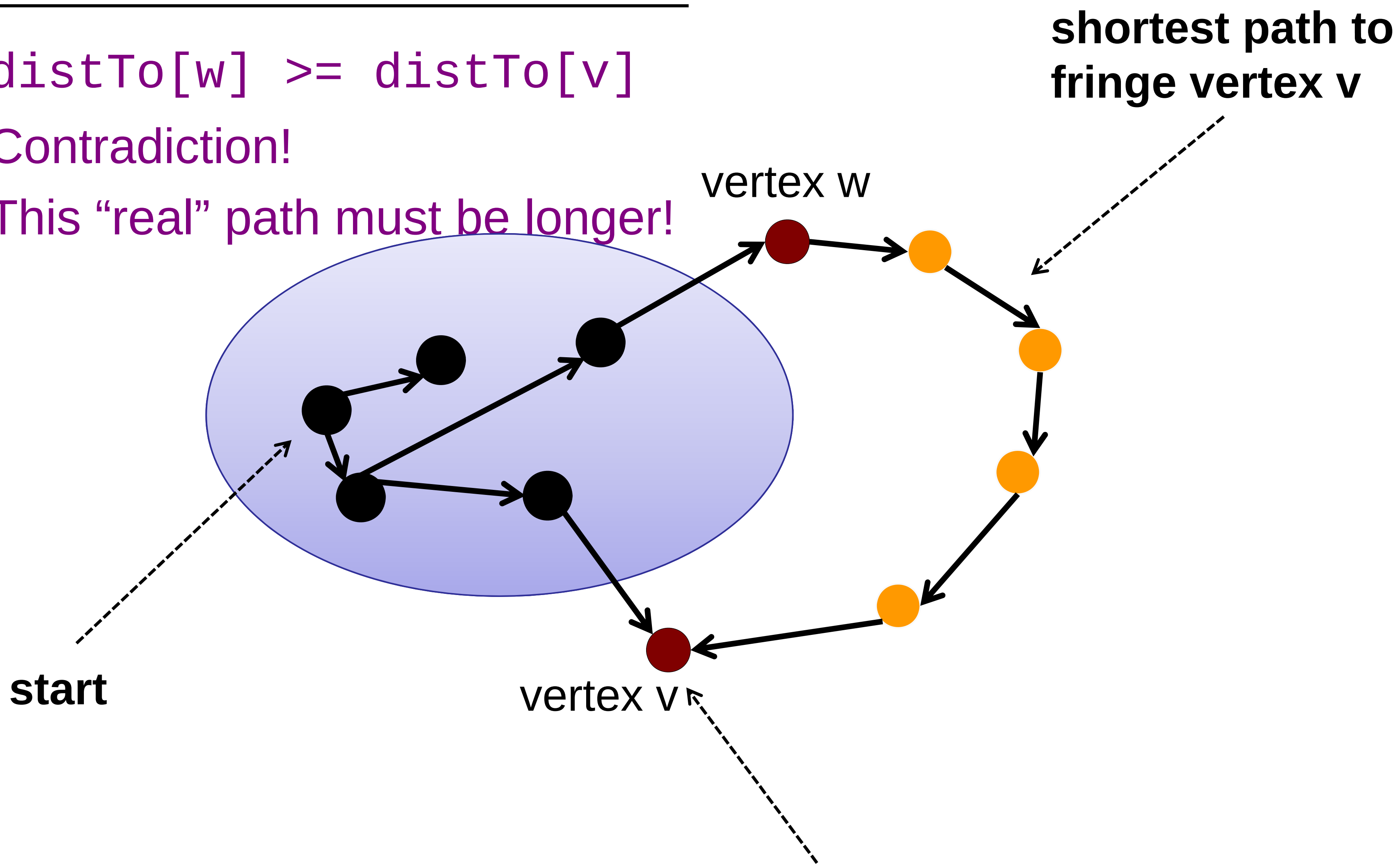


Dijkstra's Algorithm

`distTo[w] >= distTo[v]`

Contradiction!

This "real" path must be longer!



Vertex v going to be removed from priority queue next. Thus, with minimum distance amount the unfinished

Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

Dijkstra's Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Dijkstra's Algorithm

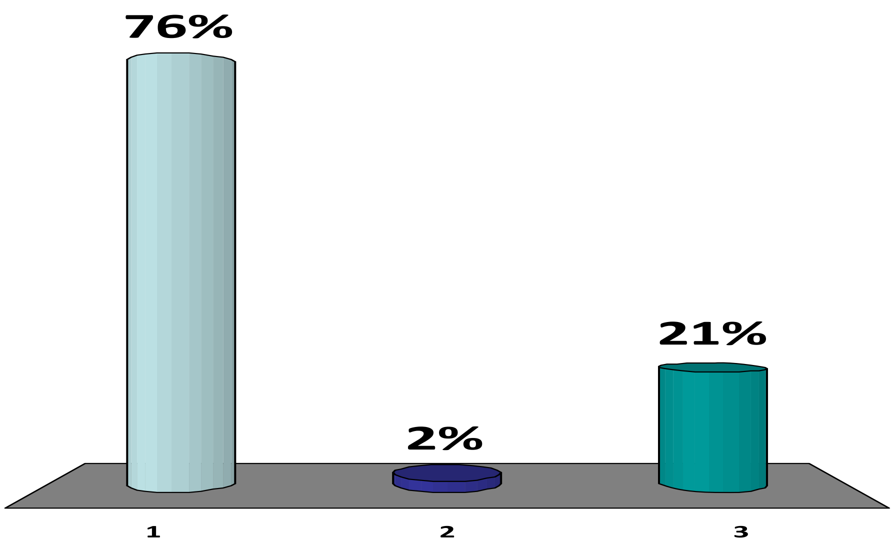
Analysis:

- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

Source-to-Destination Dijkstra

Can we stop as soon as we dequeue the destination?

1. Yes.
2. Only if the graph is sparse.
3. No.



Dijkstra’s Algorithm

Source-to-Destination:

- What if you stop the first time you dequeue the destination?

- Recall:
  - a vertex is “finished” when it is dequeued
  - if the destination is finished, then stop

Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.

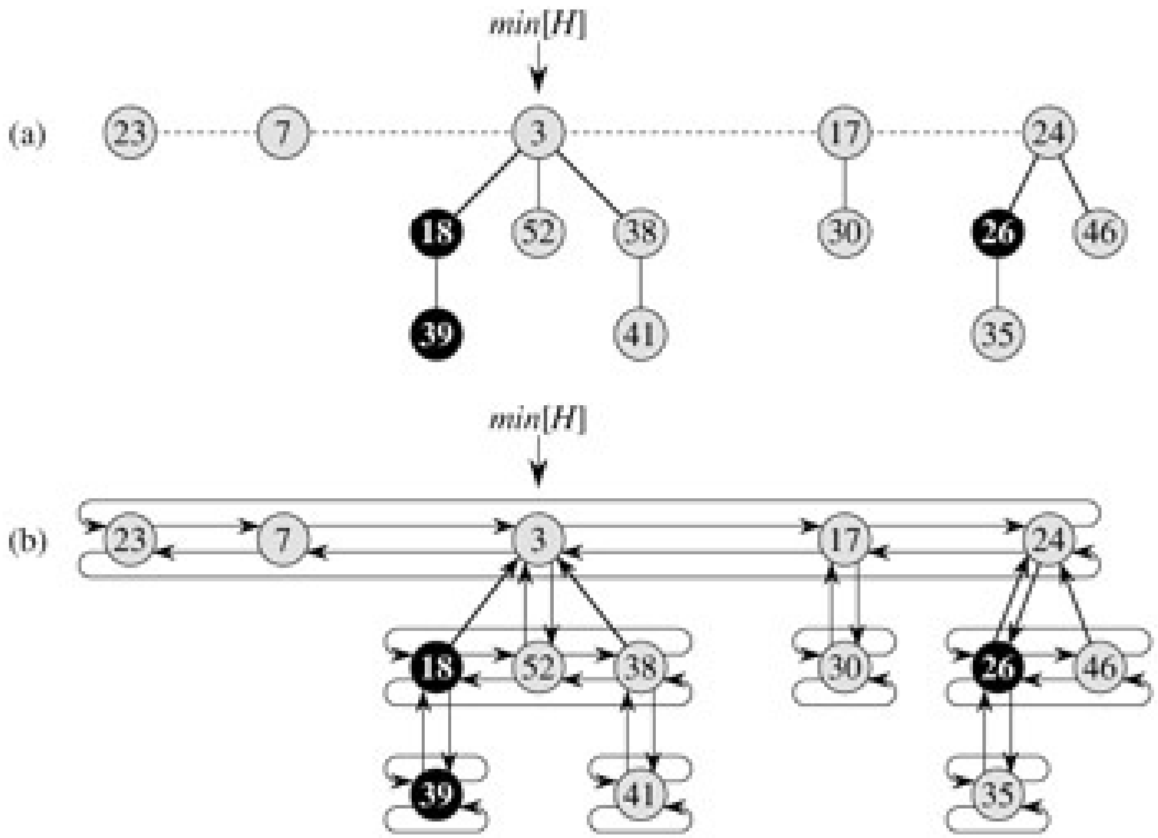
- O(E log V) time (with AVL tree).

Dijkstra's Performance

PQ Implementation	insert	deleteMin	decreaseKey	Total
Array	1	V	1	**$O(V^2)$**
AVL Tree	log V	log V	log V	**$O(E \log V)$**
d-way Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	**$O(E\log_{E/V} V)$**
Fibonacci Heap	1	log V	1	**$O(E + V \log V)$**

Fibonacci Heap

- Not in CS2020

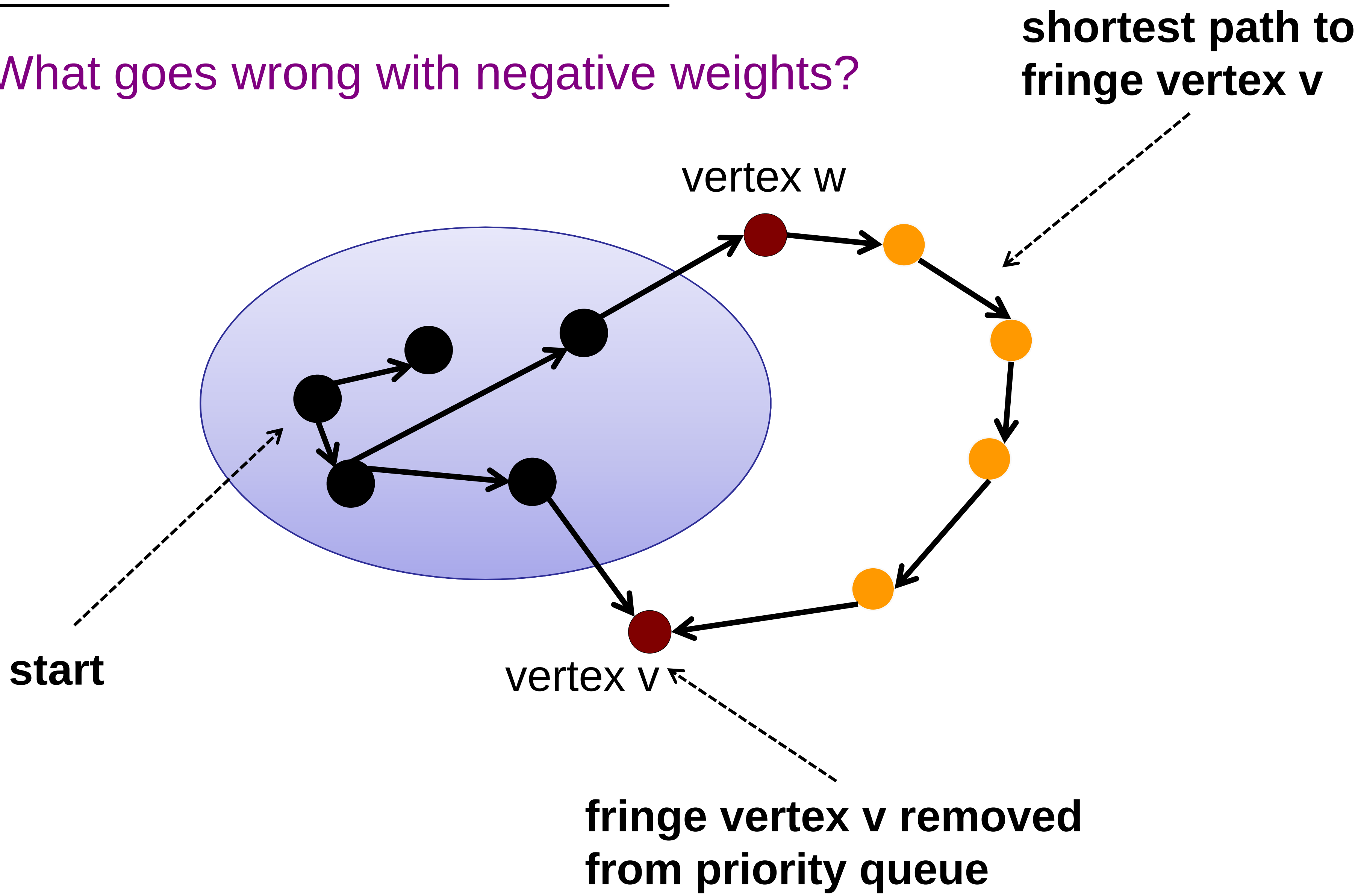


Dijkstra Summary

Edges with negative weights?

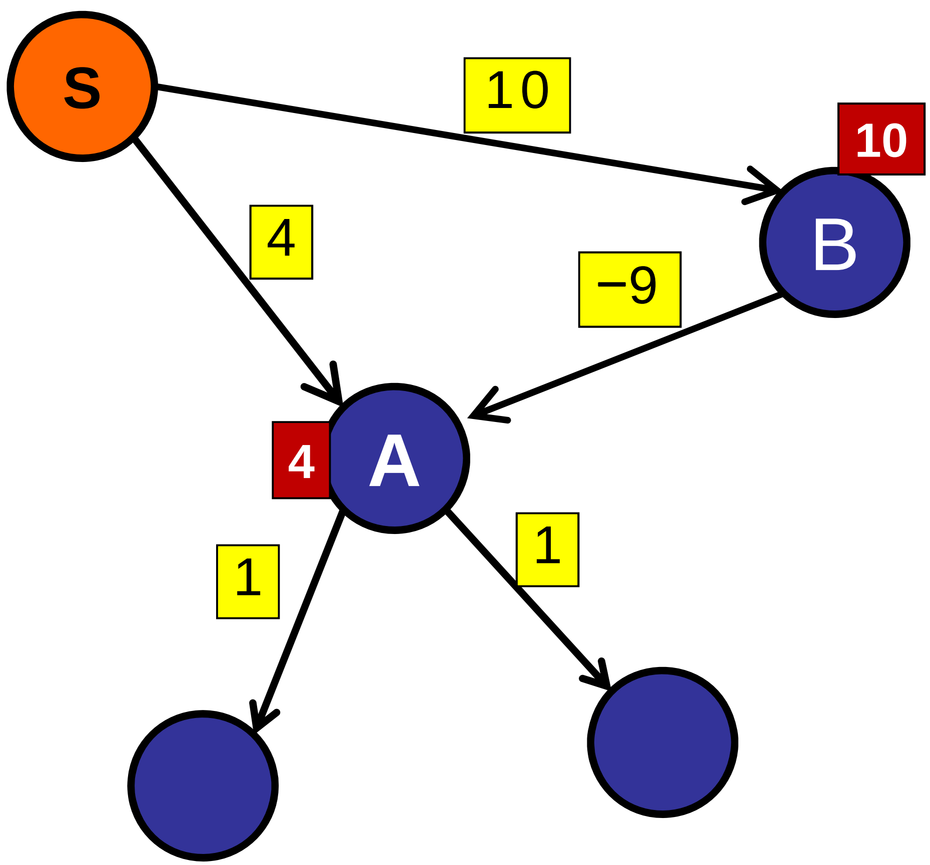
Dijkstra's Algorithm

What goes wrong with negative weights?



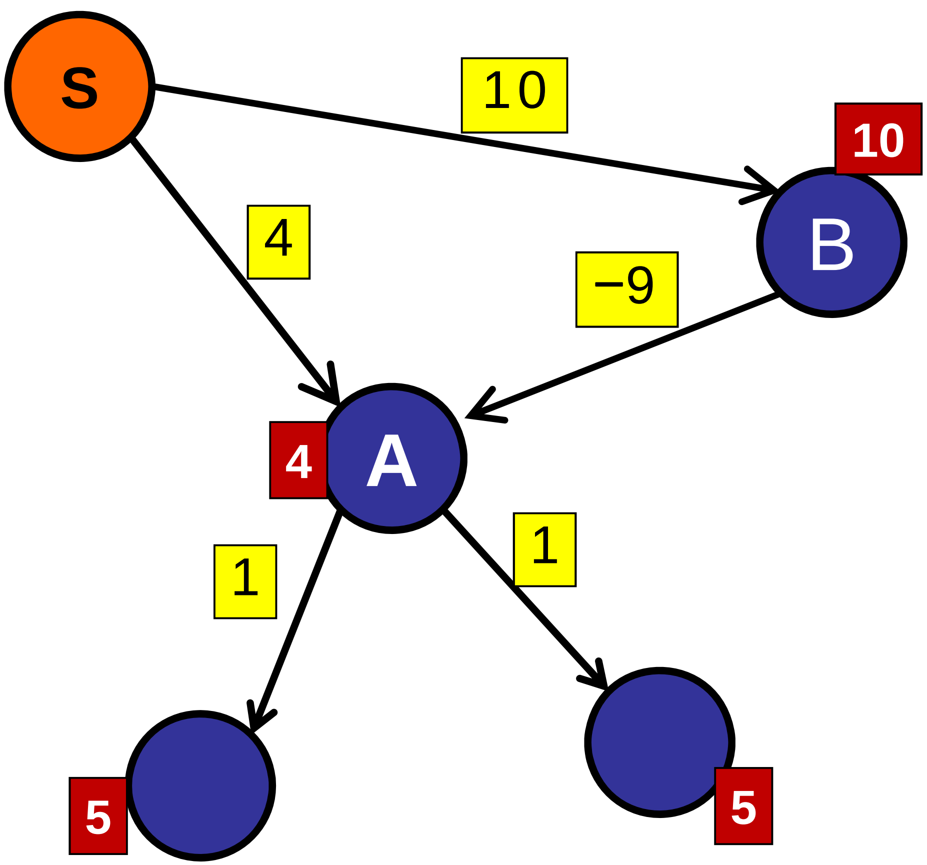
Dijkstra's Algorithm

Edges with negative weights?



Dijkstra's Algorithm

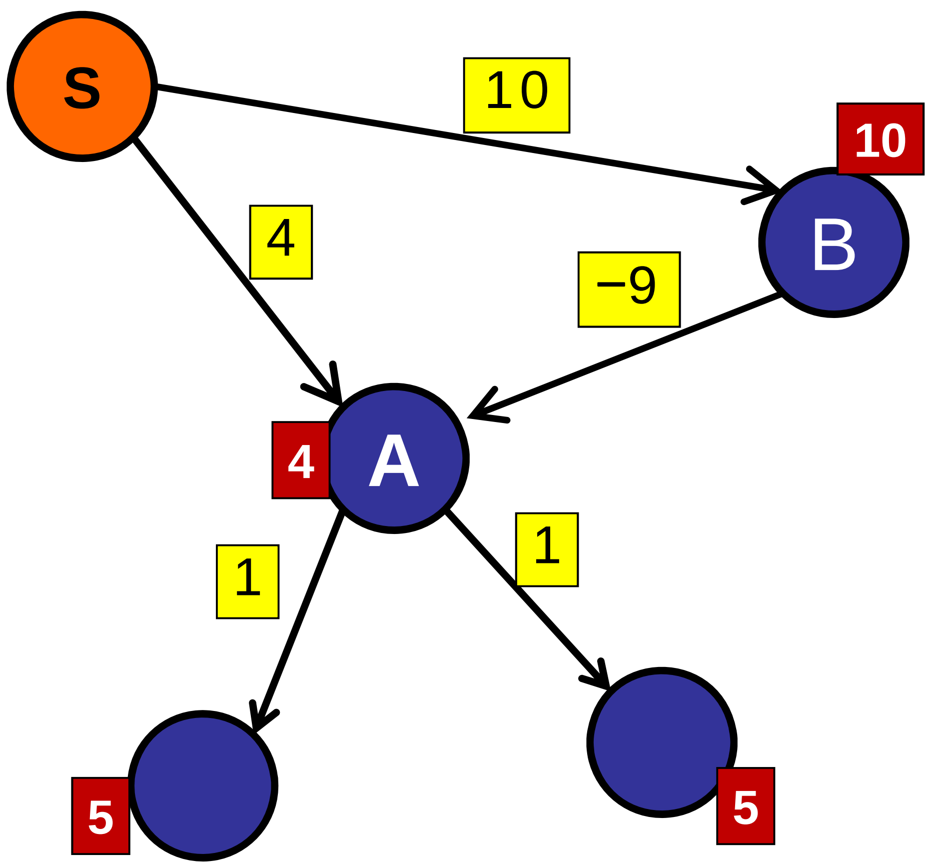
Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

Dijkstra's Algorithm

Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

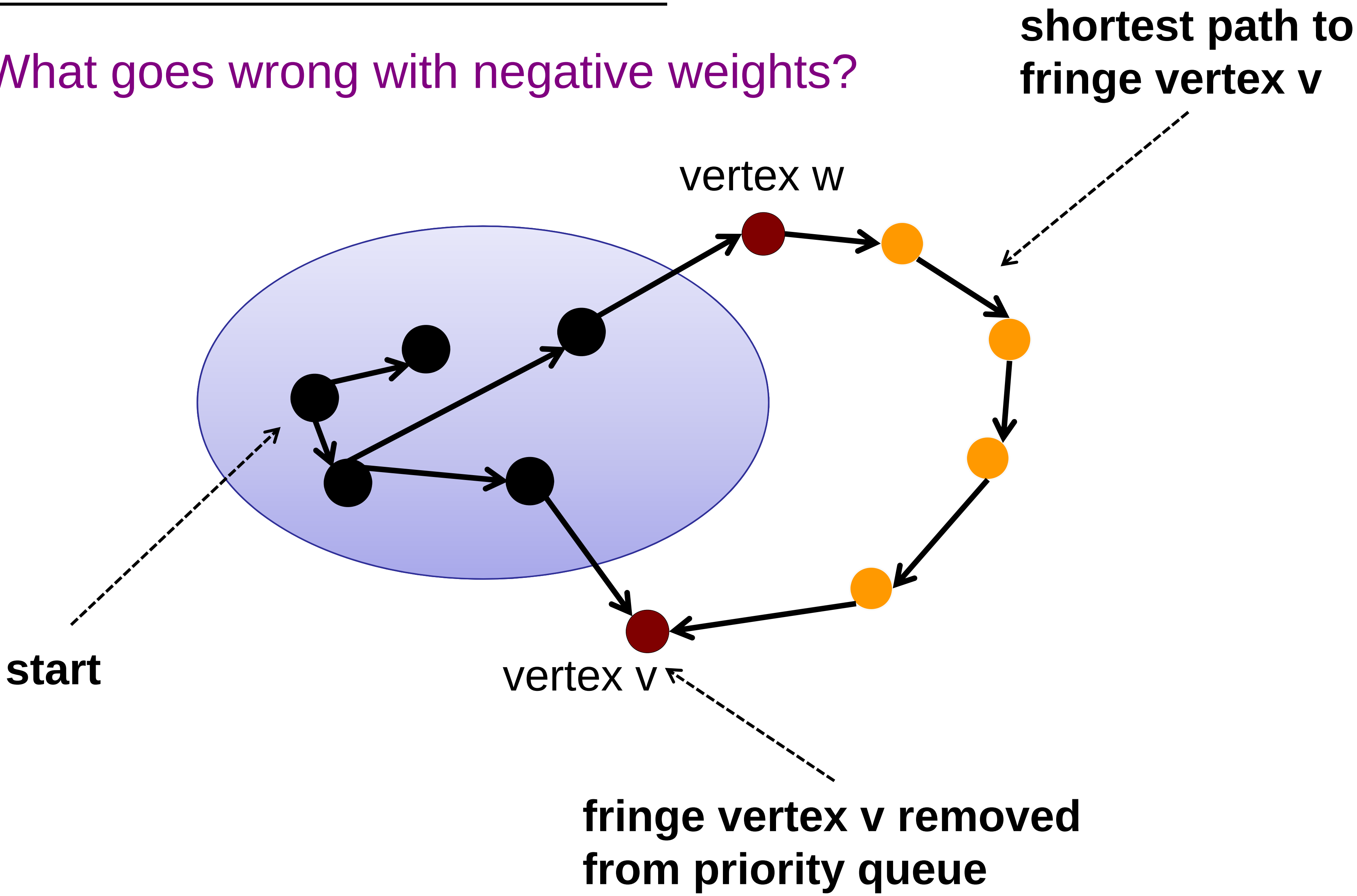
…

Step 4: Remove B.
Relax B.
Mark B done.

Oops: We need to update A.

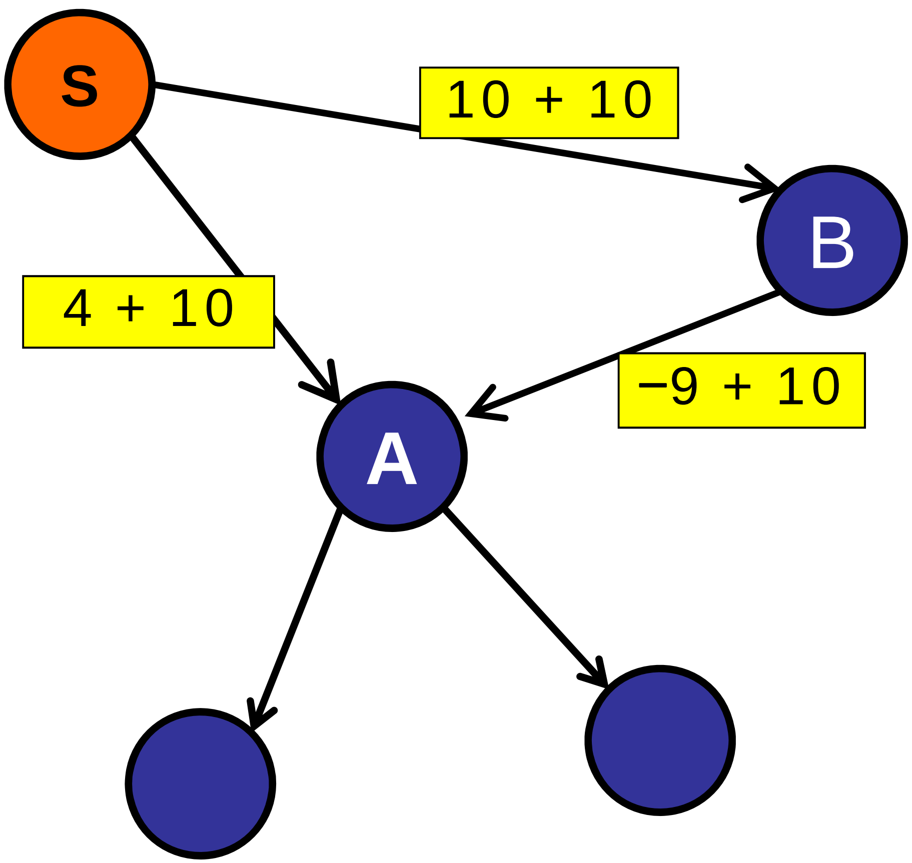
Dijkstra's Algorithm

What goes wrong with negative weights?



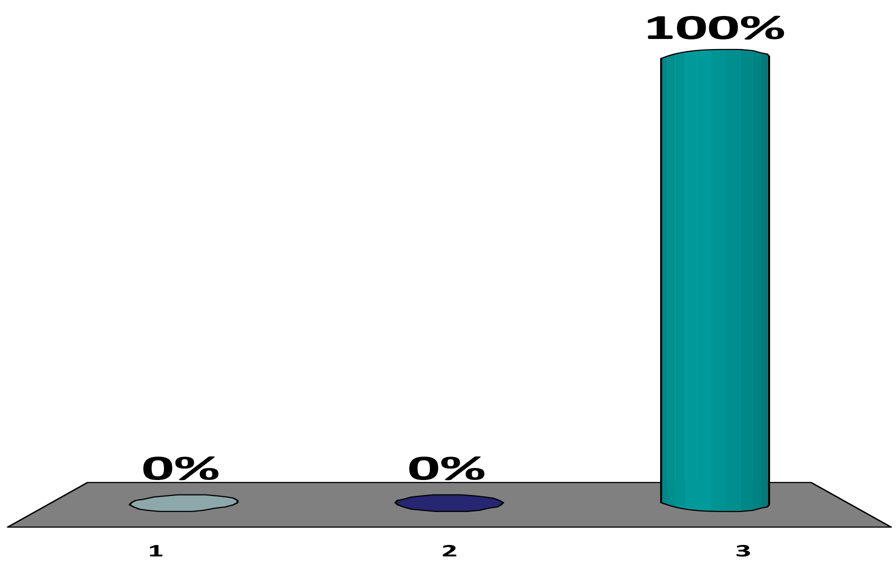
Dijkstra's Algorithm

Can we reweight? e.g.: weight += 10



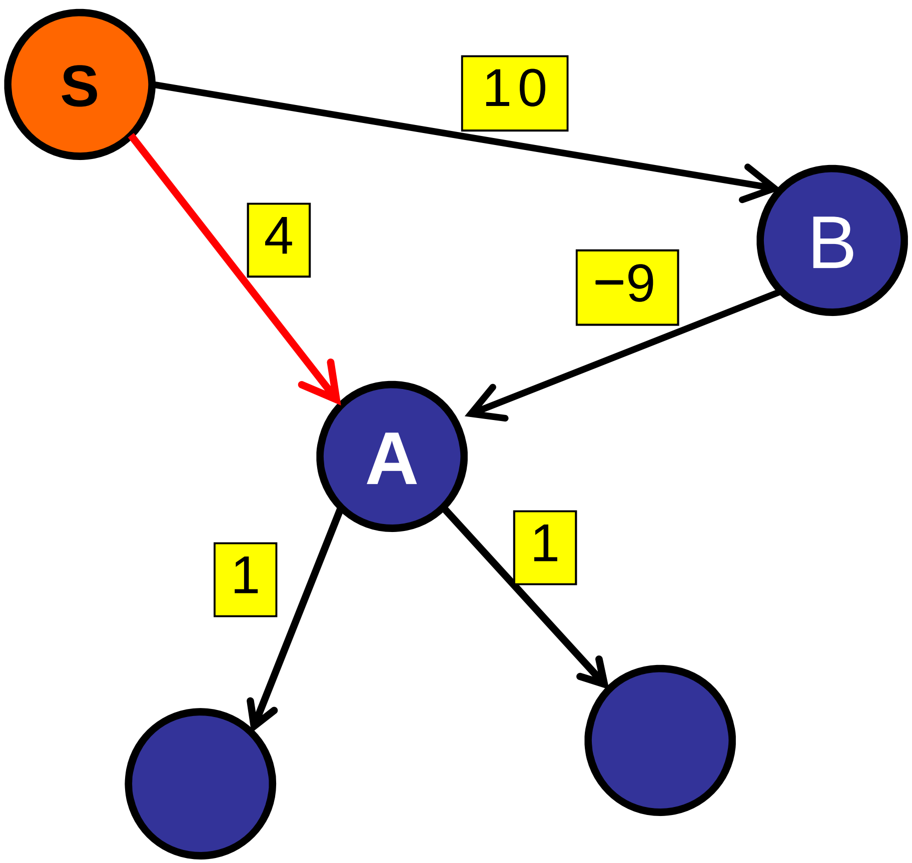
Can we reweight the graph?

1. Yes.
2. Only if there are no negative weight cycles.
3. No.



Dijkstra's Algorithm

Can we reweight?

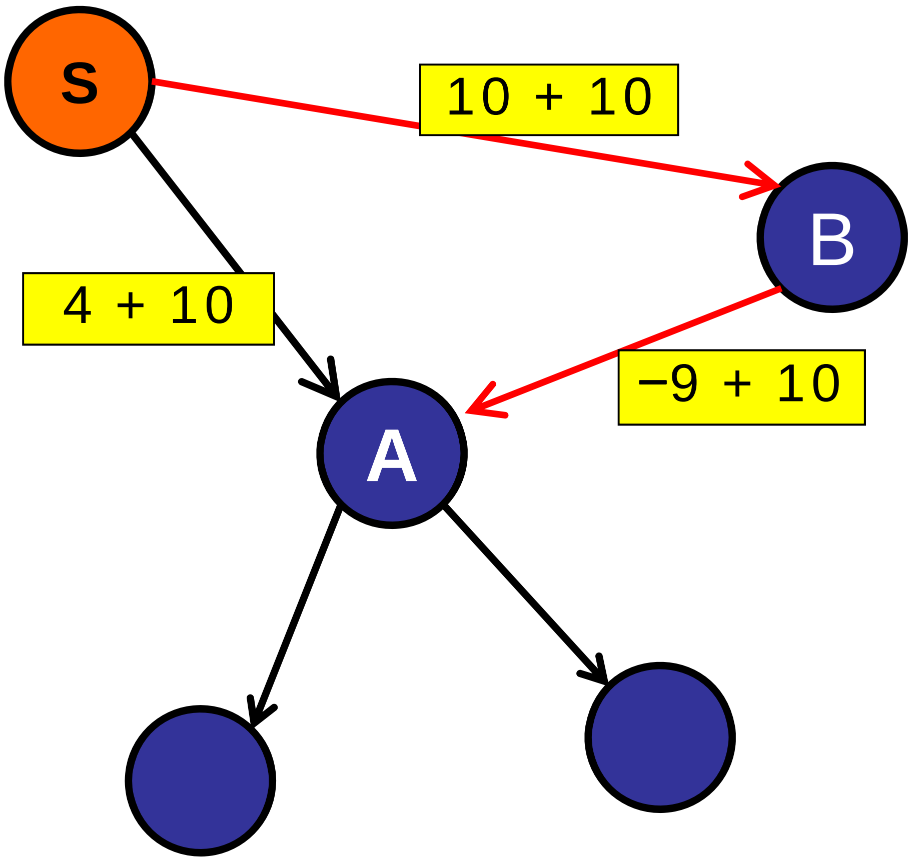


Path S-B-A: 1

Path S-A: 4

Dijkstra’s Algorithm

Can we reweight?



Path S-B-A: 21

Path S-A: 14

Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.

- O(E log V) time (with AVL tree Priority Queue).

- No negative weight edges!

Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Take edge from vertex that was discovered **least** recently.
  - DFS: Take edge from vertex that was discovered **most** recently.
  - Dijkstra's: Take edge from vertex that is **closest** to source.

Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Use queue.
  - DFS: Use stack.
  - Dijkstra's: Use priority queue.

Roadmap

Part I: Shortest Paths

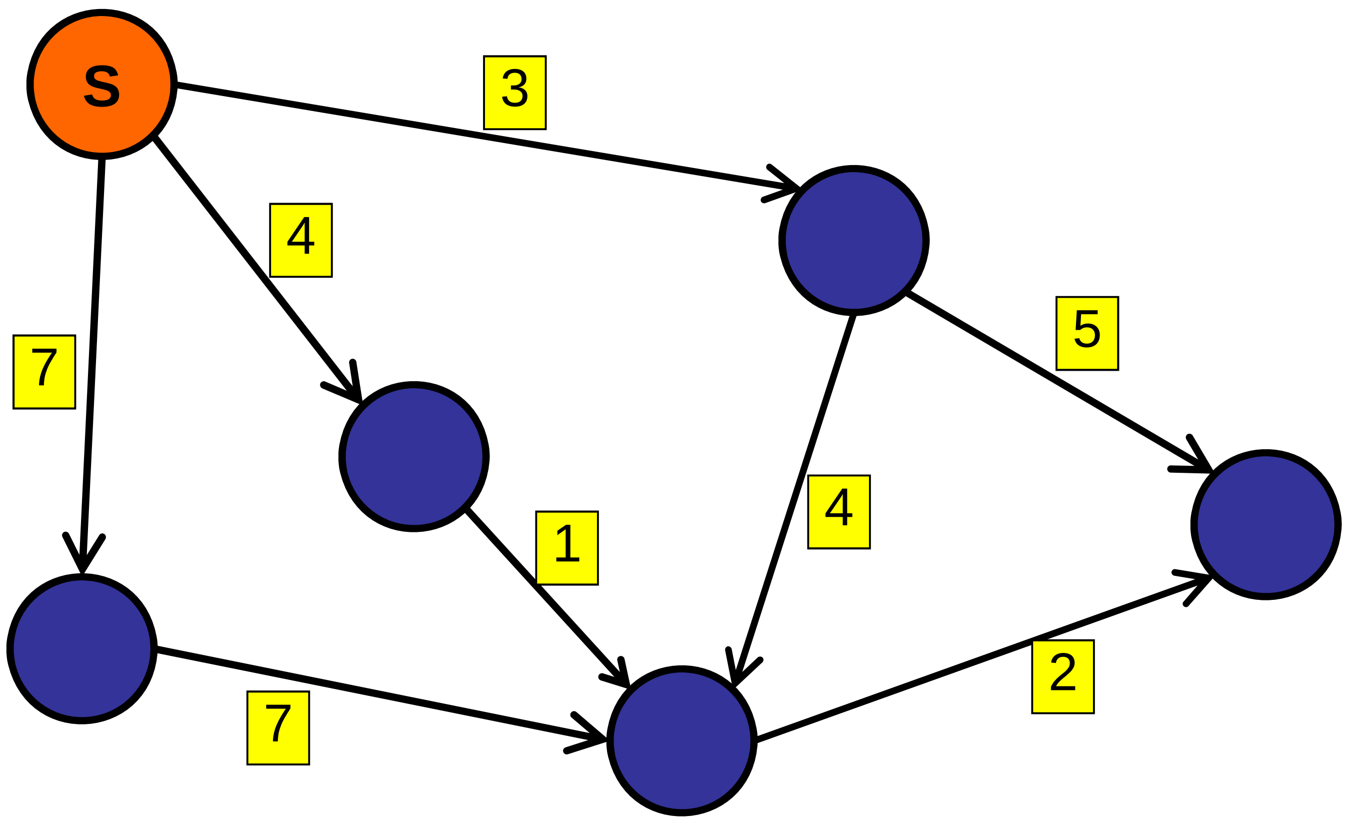
- Special Case: Tree
- Special Case: Non-negative weights (Dijkstra's)
- Special Case: Directed Acyclic Graphs

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

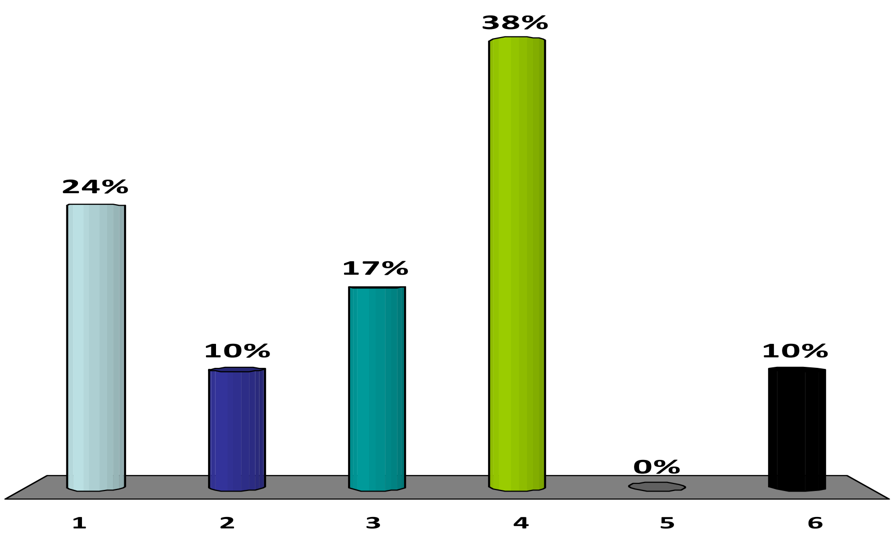
Shortest Paths

Acyclic Graph: Suppose the graph has no cycles.



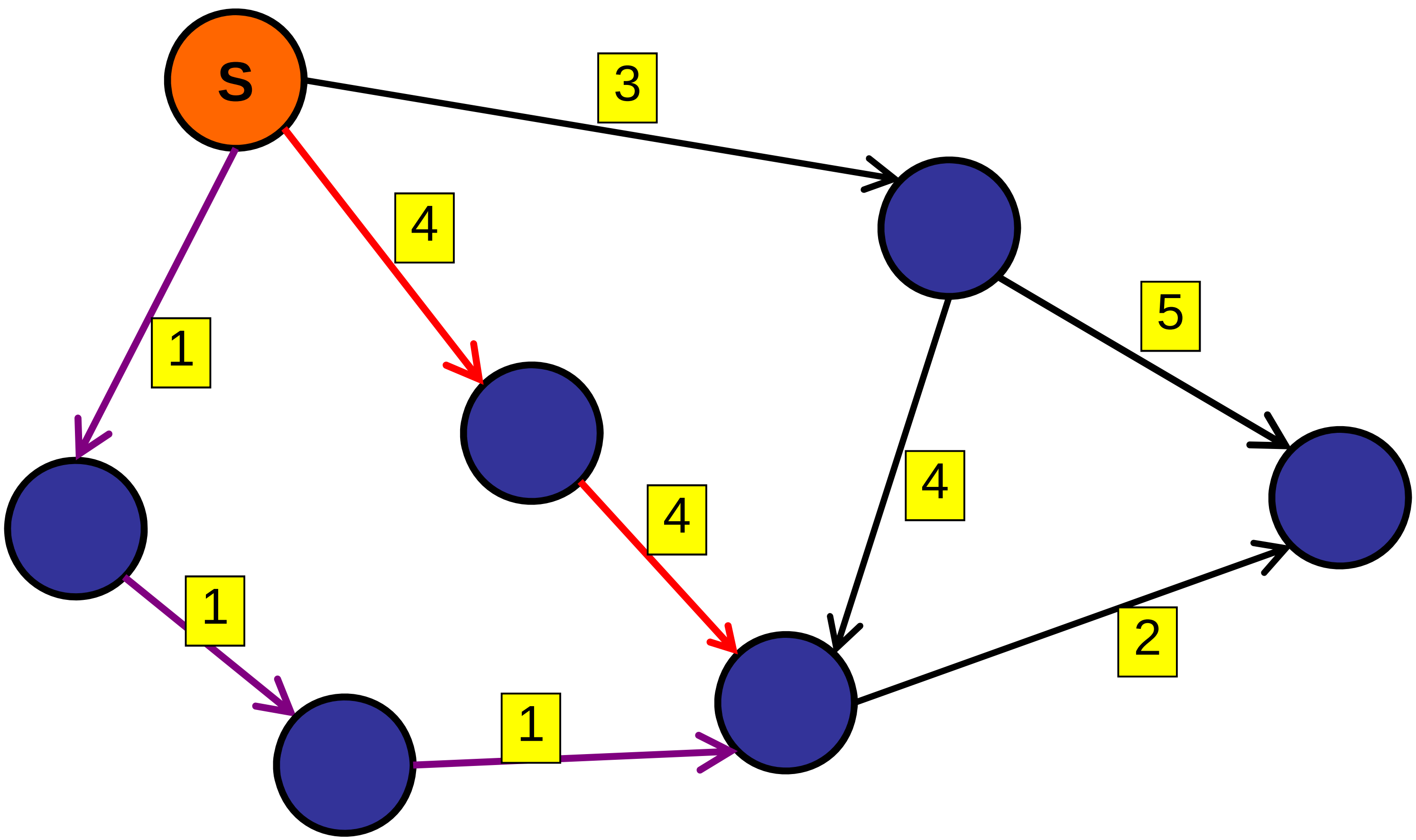
What order should we relax the nodes?

1. BFS
2. DFS pre-order
3. DFS post-order
4. Shortest edge
5. Longest edge
6. Other



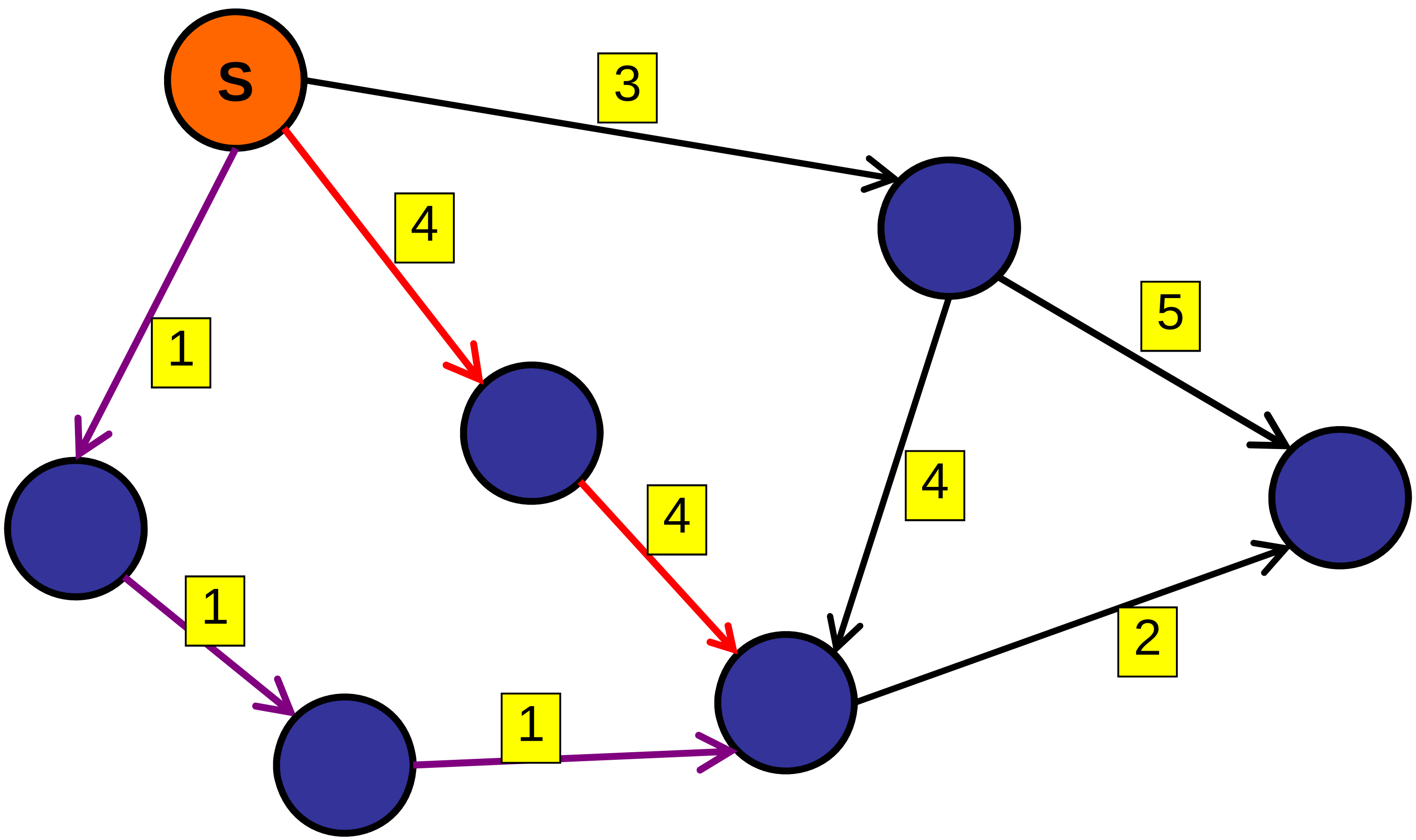
Shortest Paths

Acyclic Graph: Not BFS.



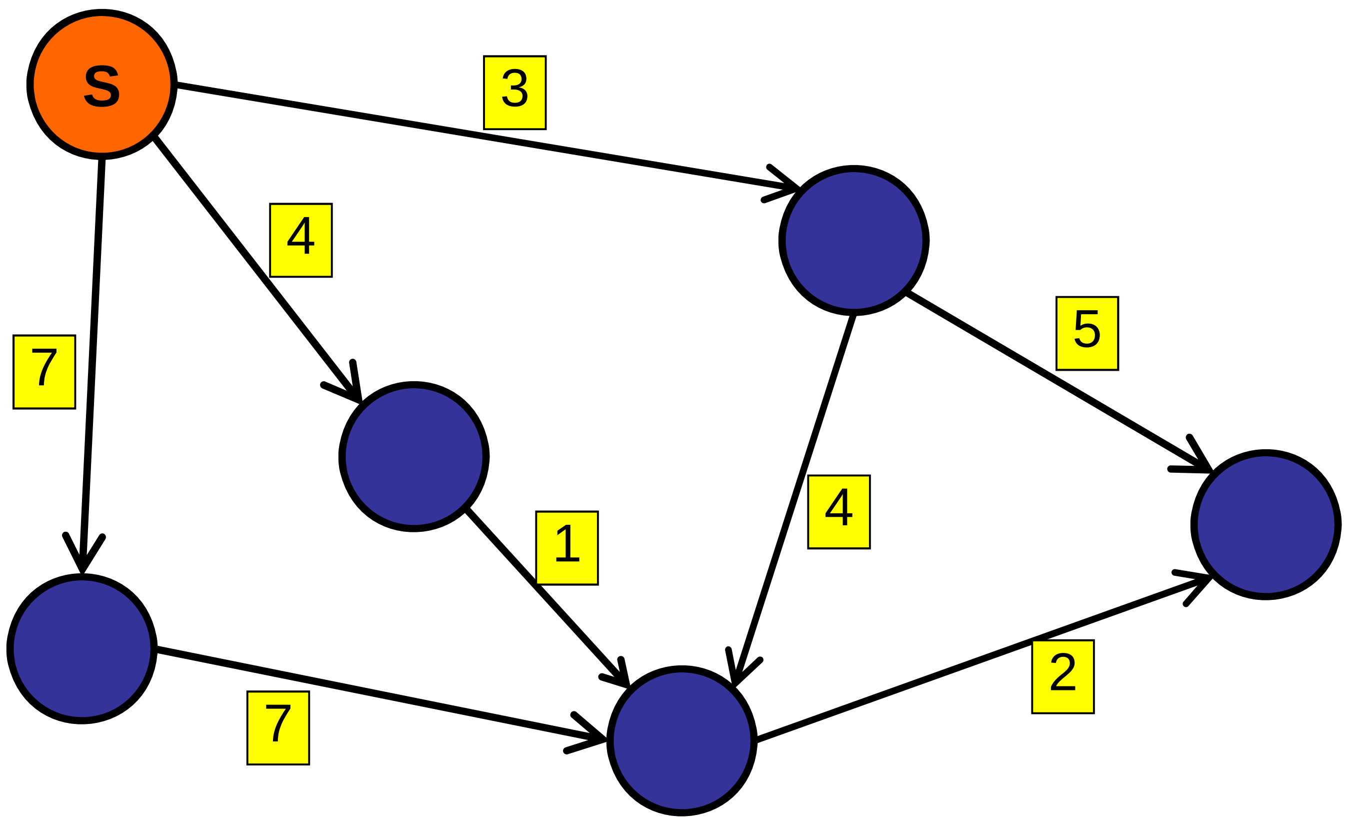
Shortest Paths

Acyclic Graph: Not DFS-preorder.



Shortest Paths

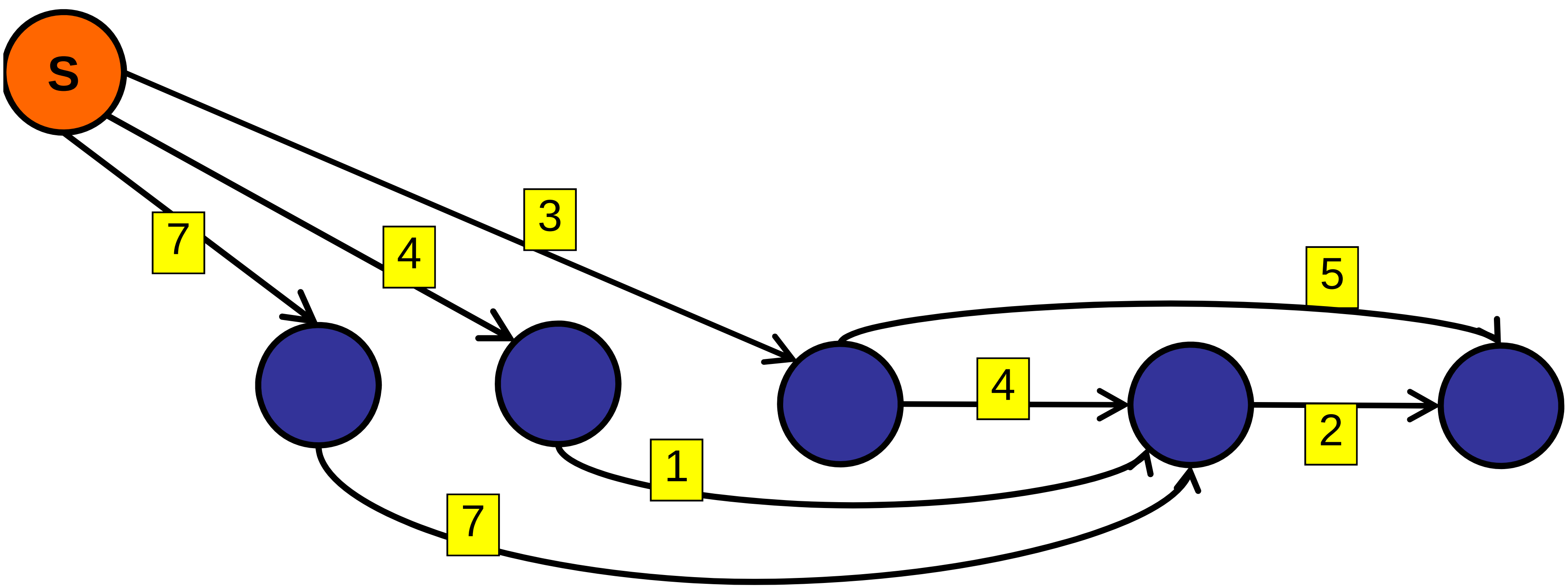
Acyclic Graph: has no cycles.



Shortest Paths

Acyclic Graph: has no cycles.

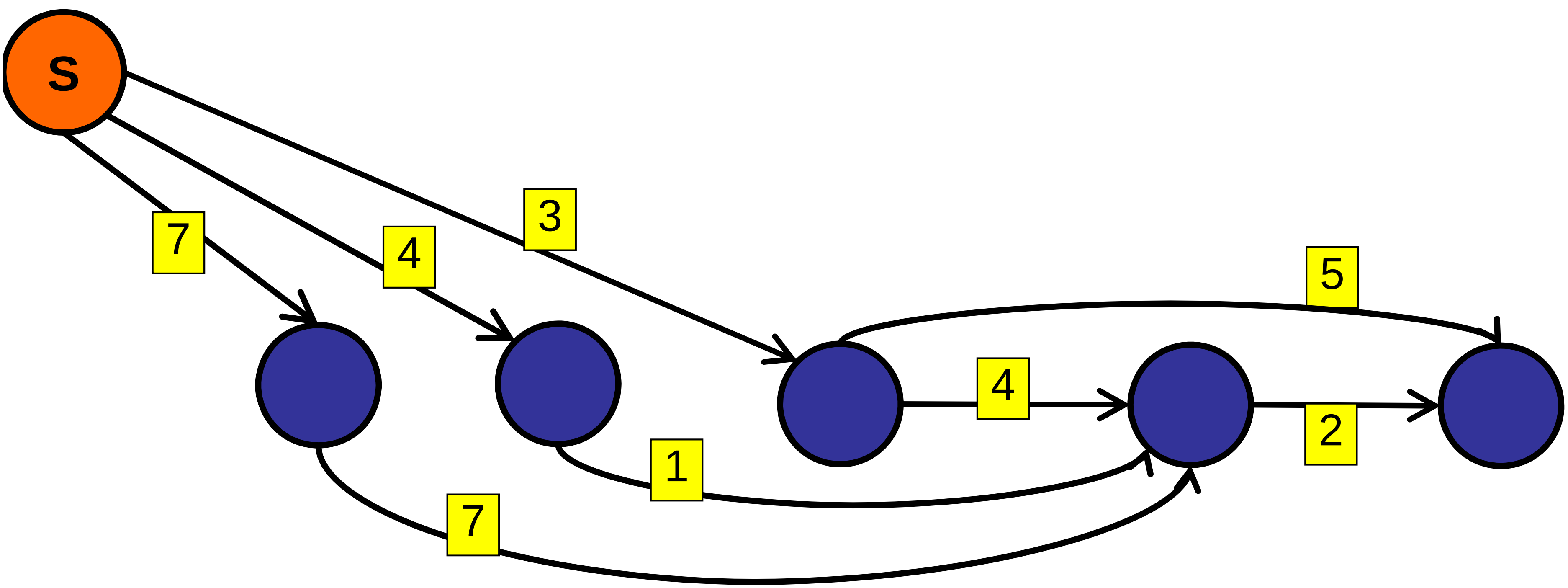
1. Topological sort



Shortest Paths

Acyclic Graph: has no cycles.

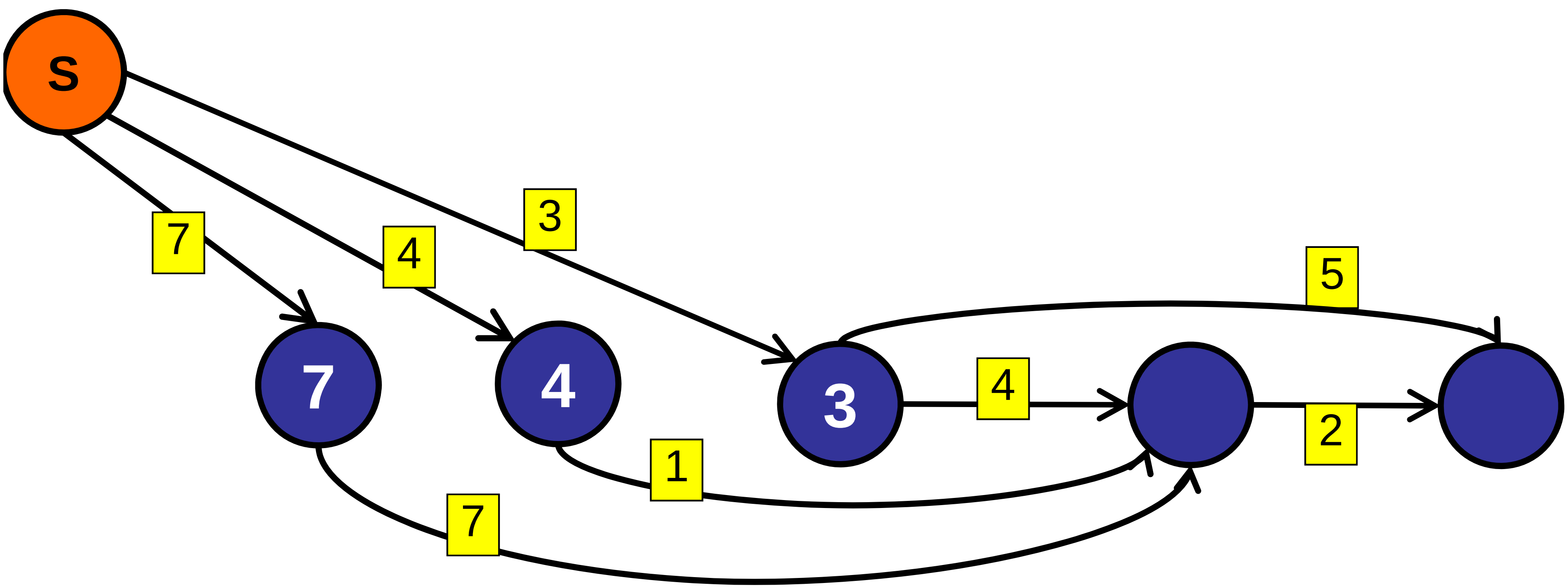
1. Topological sort
2. Relax in order.



Shortest Paths

Acyclic Graph: has no cycles.

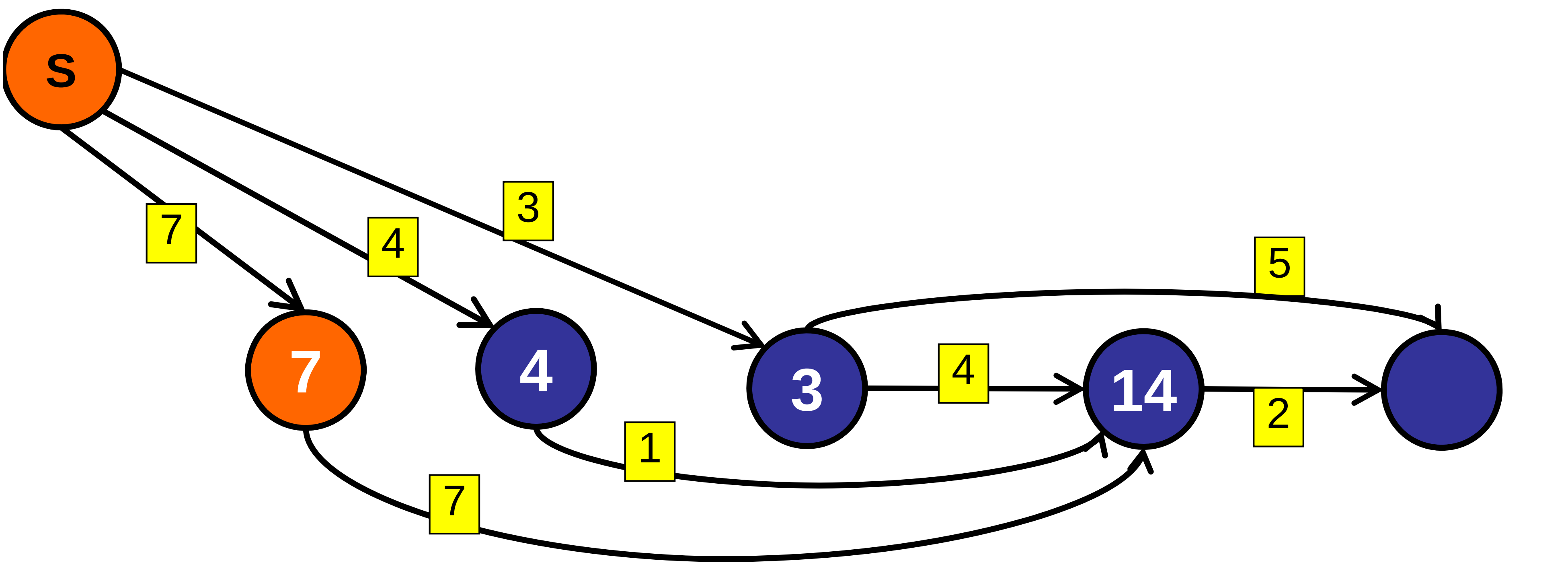
1. Topological sort
2. Relax in order.



Shortest Paths

Acyclic Graph: has no cycles.

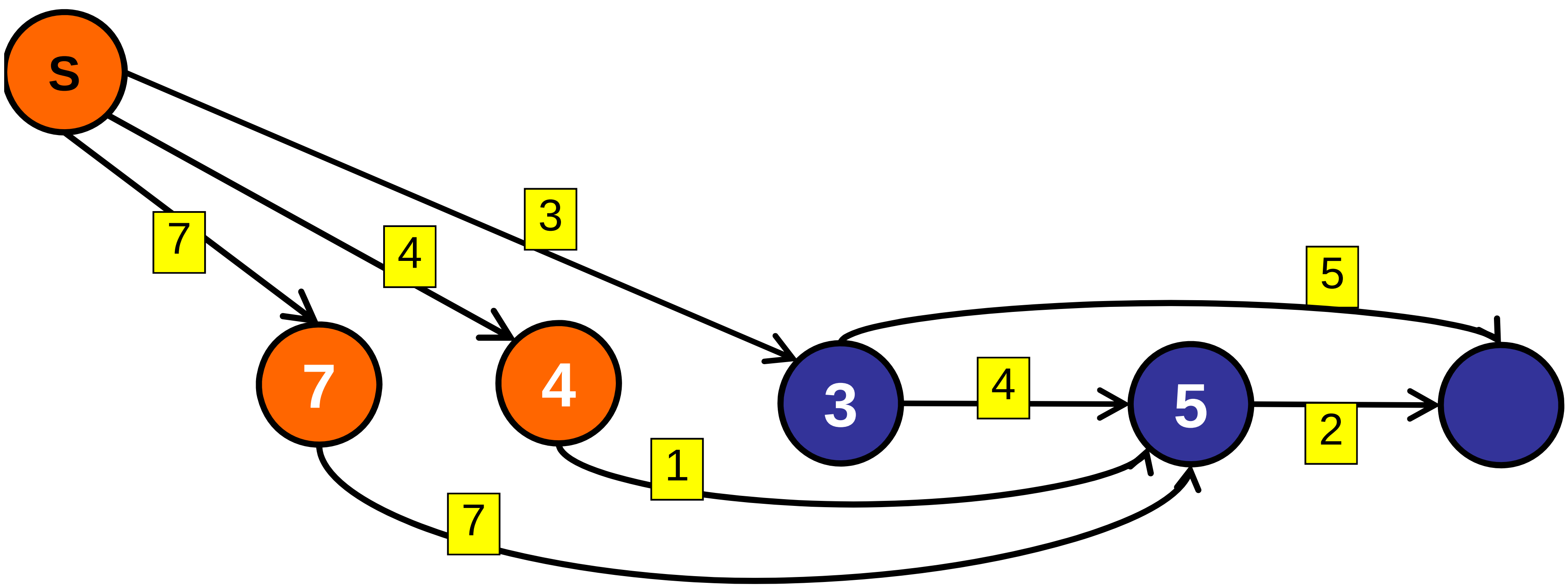
1. Topological sort
2. Relax in order.



Shortest Paths

Acyclic Graph: has no cycles.

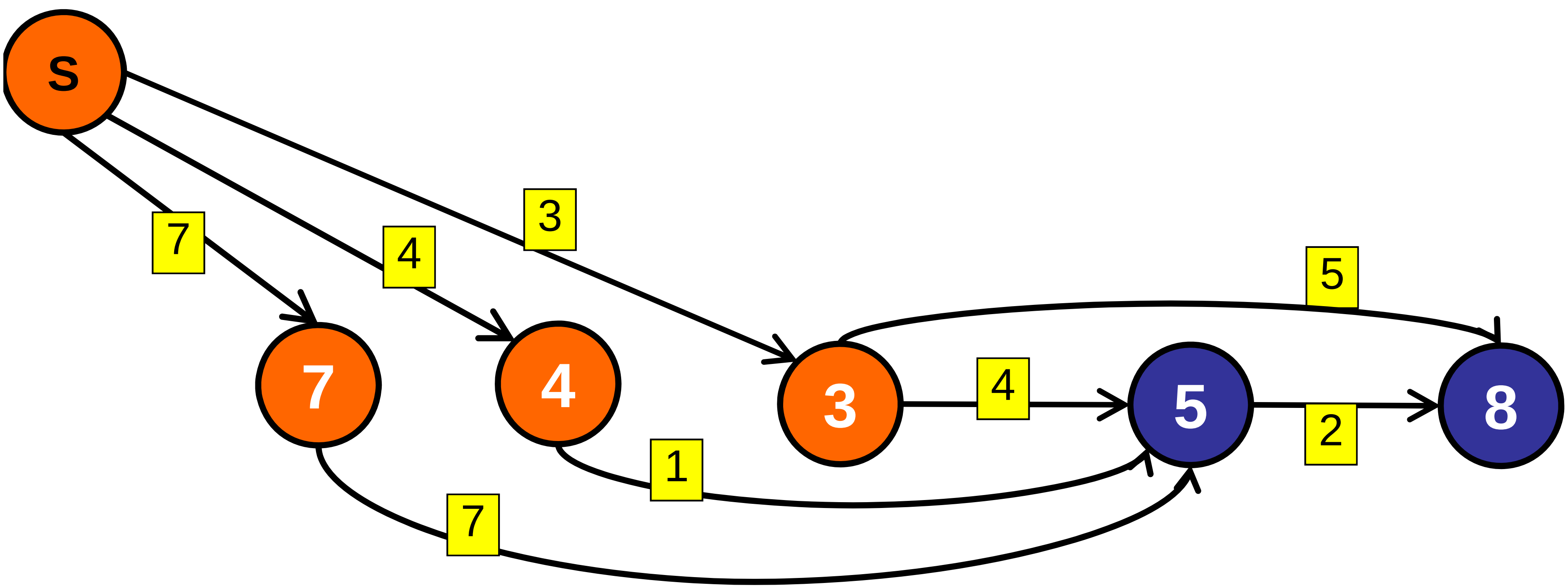
1. Topological sort
2. Relax in order.



Shortest Paths

Acyclic Graph: has no cycles.

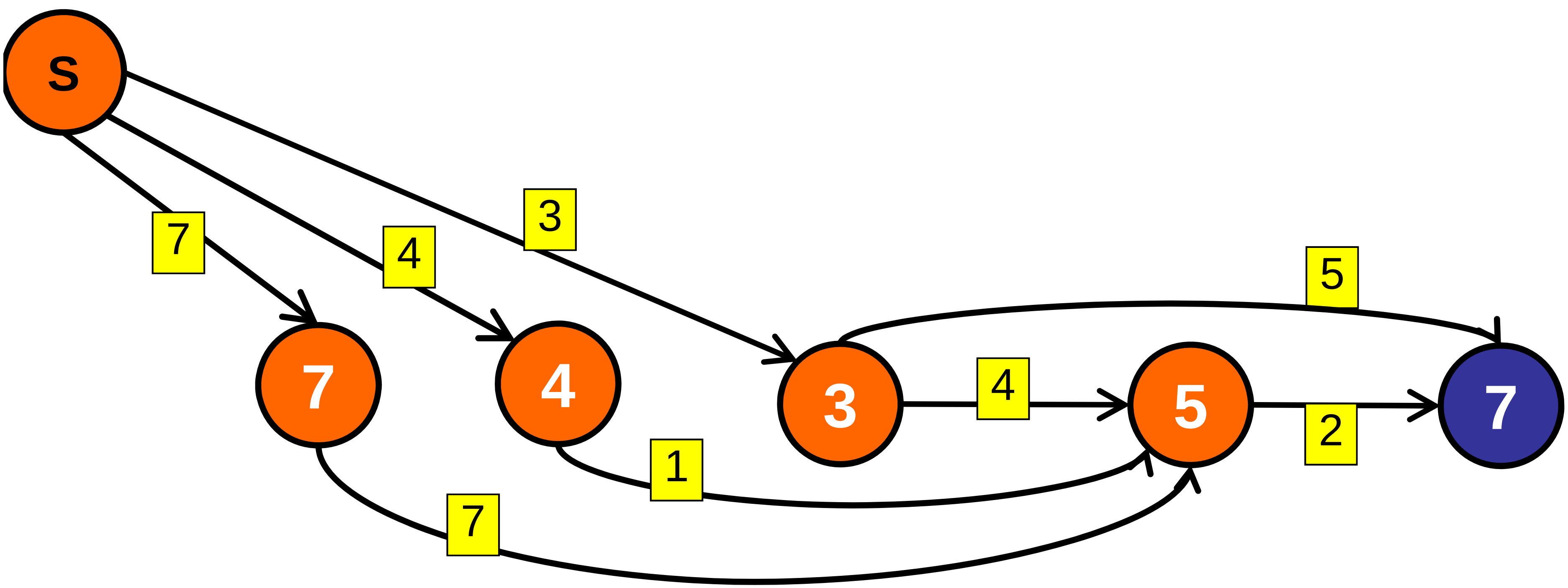
1. Topological sort
2. Relax in order.



Shortest Paths

Acyclic Graph: has no cycles.

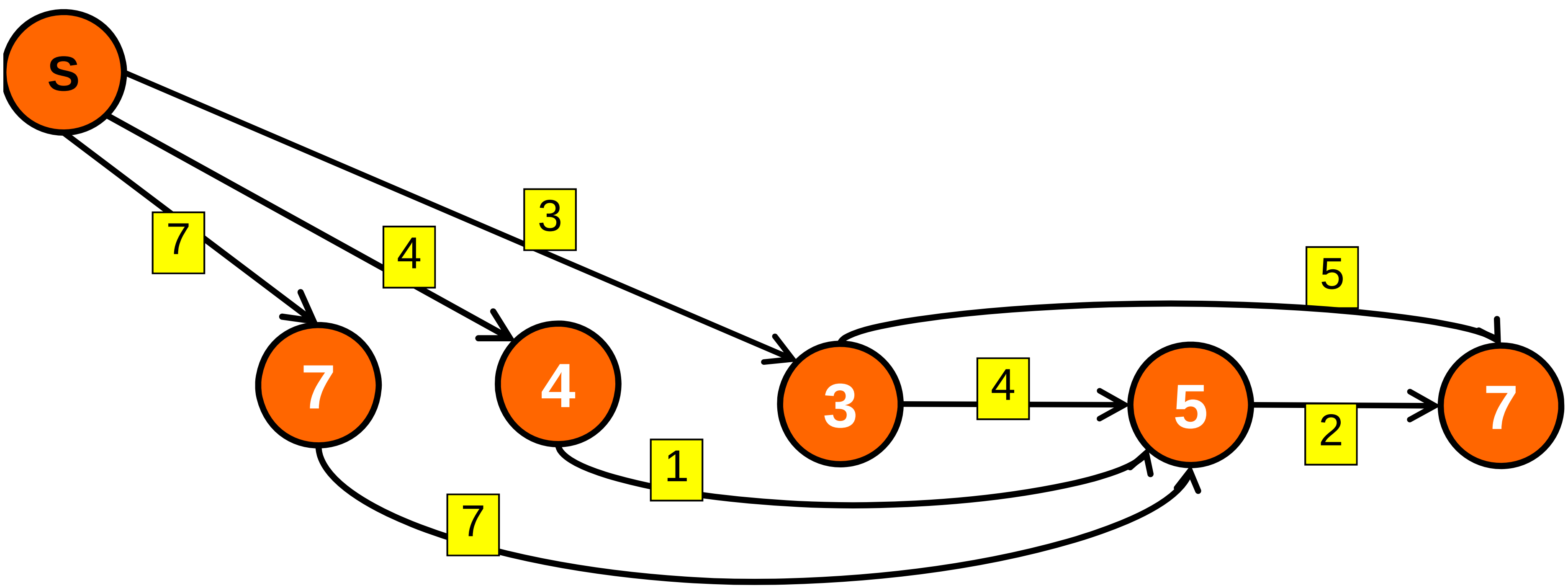
1. Topological sort
2. Relax in order.



Shortest Paths

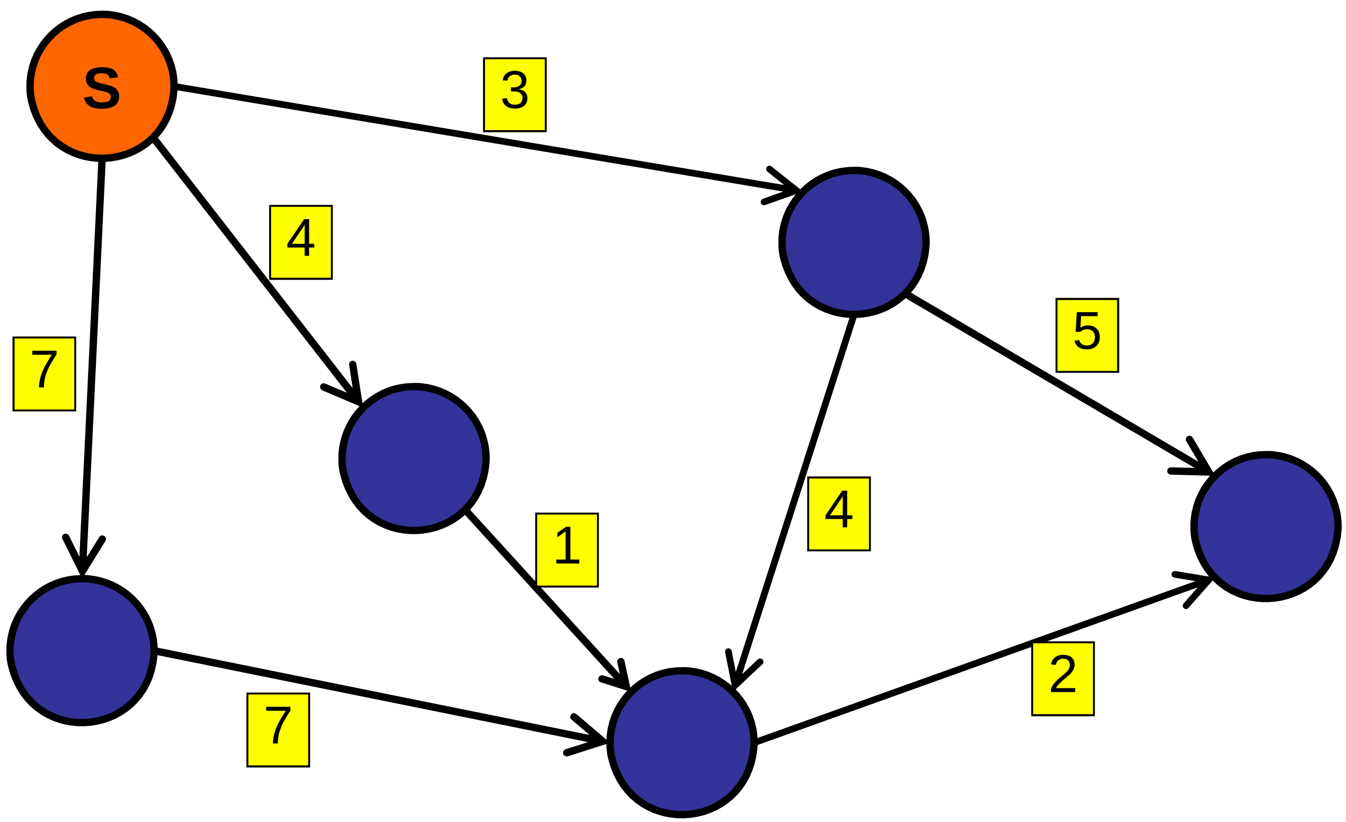
Acyclic Graph: has no cycles.

1. Topological sort
2. Relax in order.



Shortest Paths

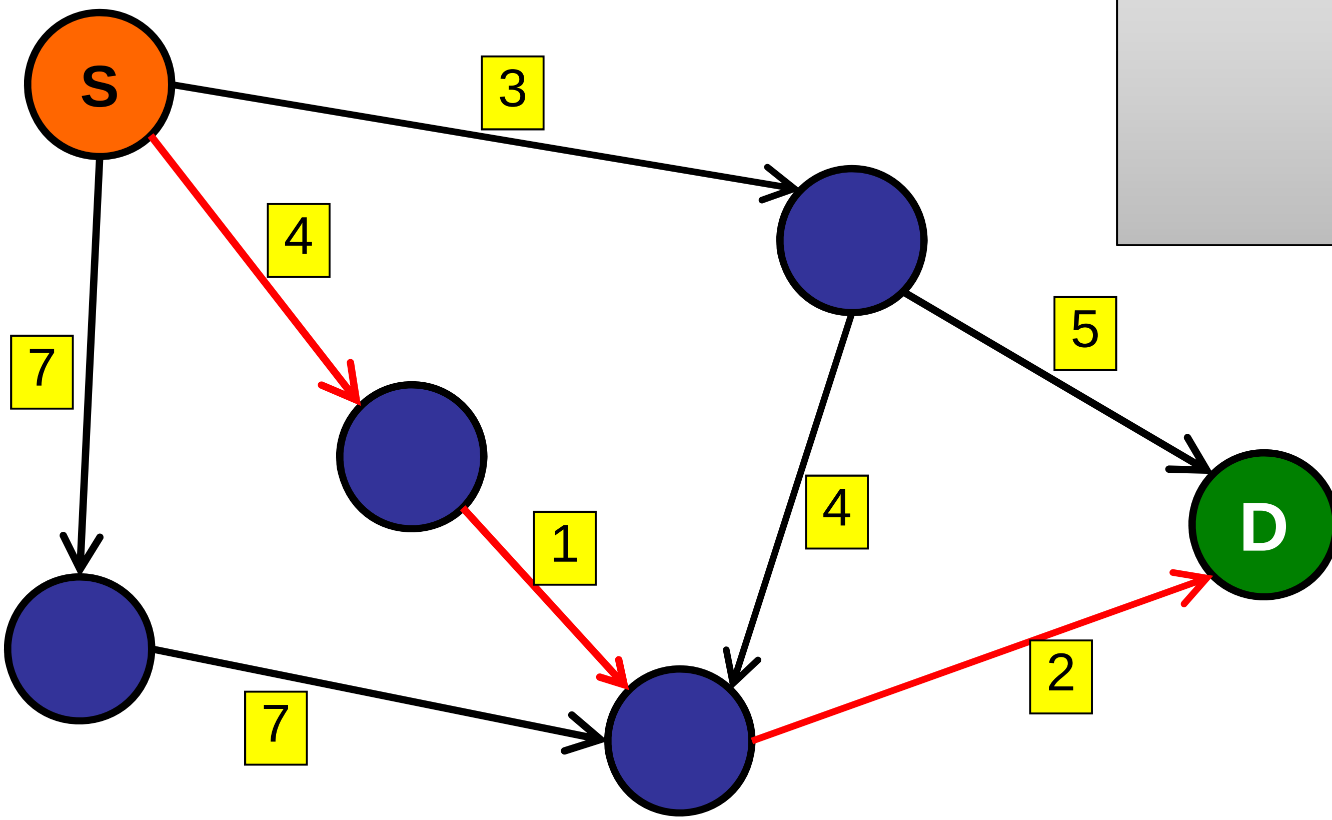
Acyclic Graph: Why topological order?



Shortest Paths

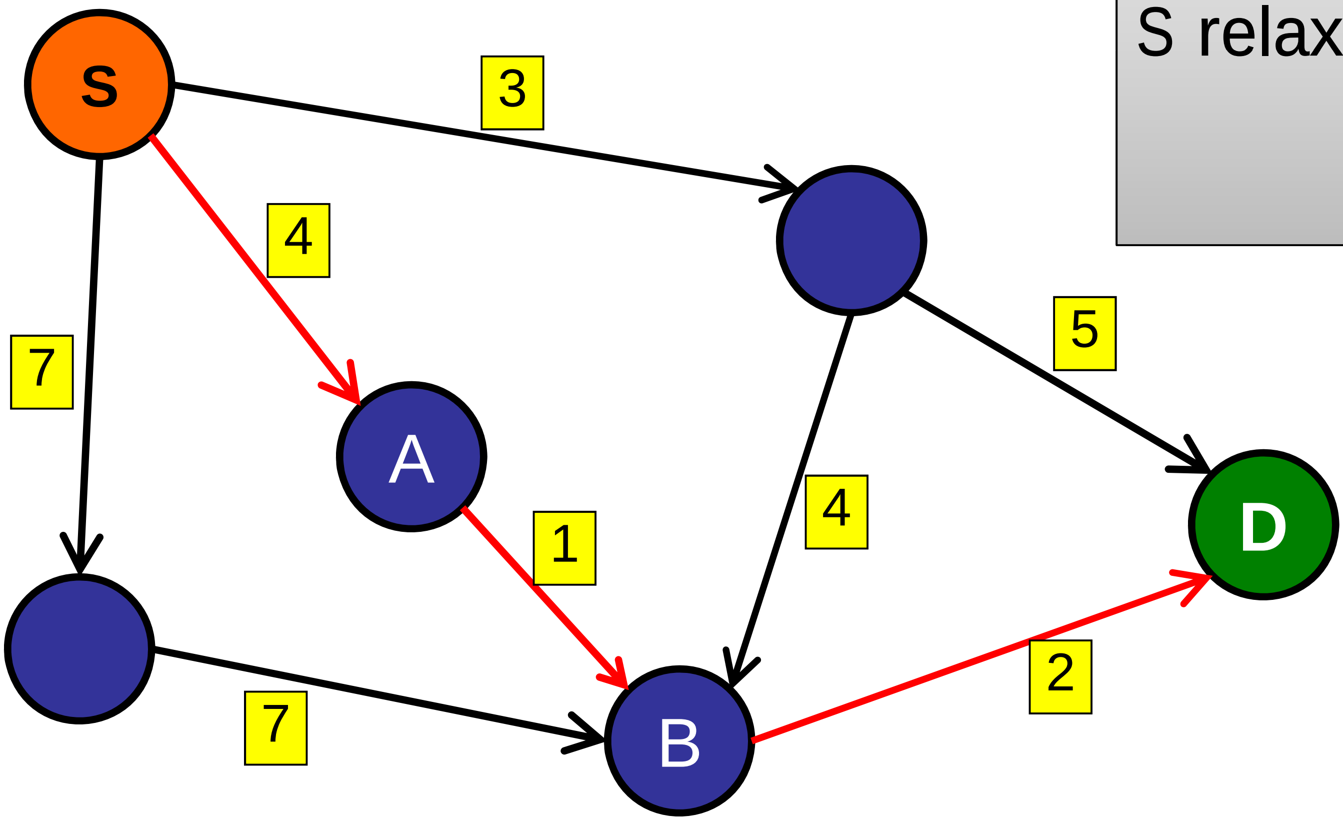
Acyclic Graph: Why topological order?

Fix S-D shortest path.



Shortest Paths

Acyclic Graph: Why topological order?



Fix S-D shortest path.

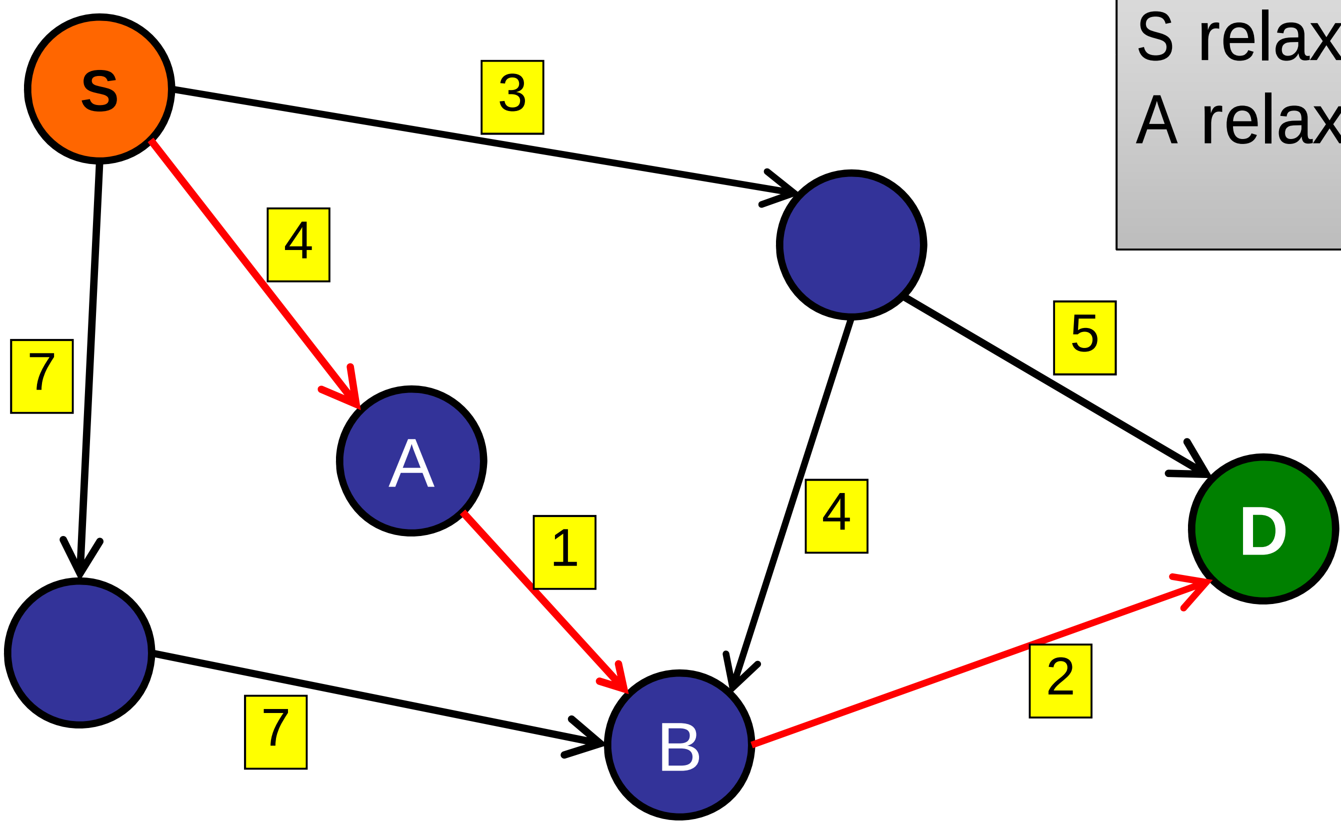
S relaxed before A.

Shortest Paths

Acyclic Graph: Why topological order?

Fix S-D shortest path.

S relaxed before A.
A relaxed before B.

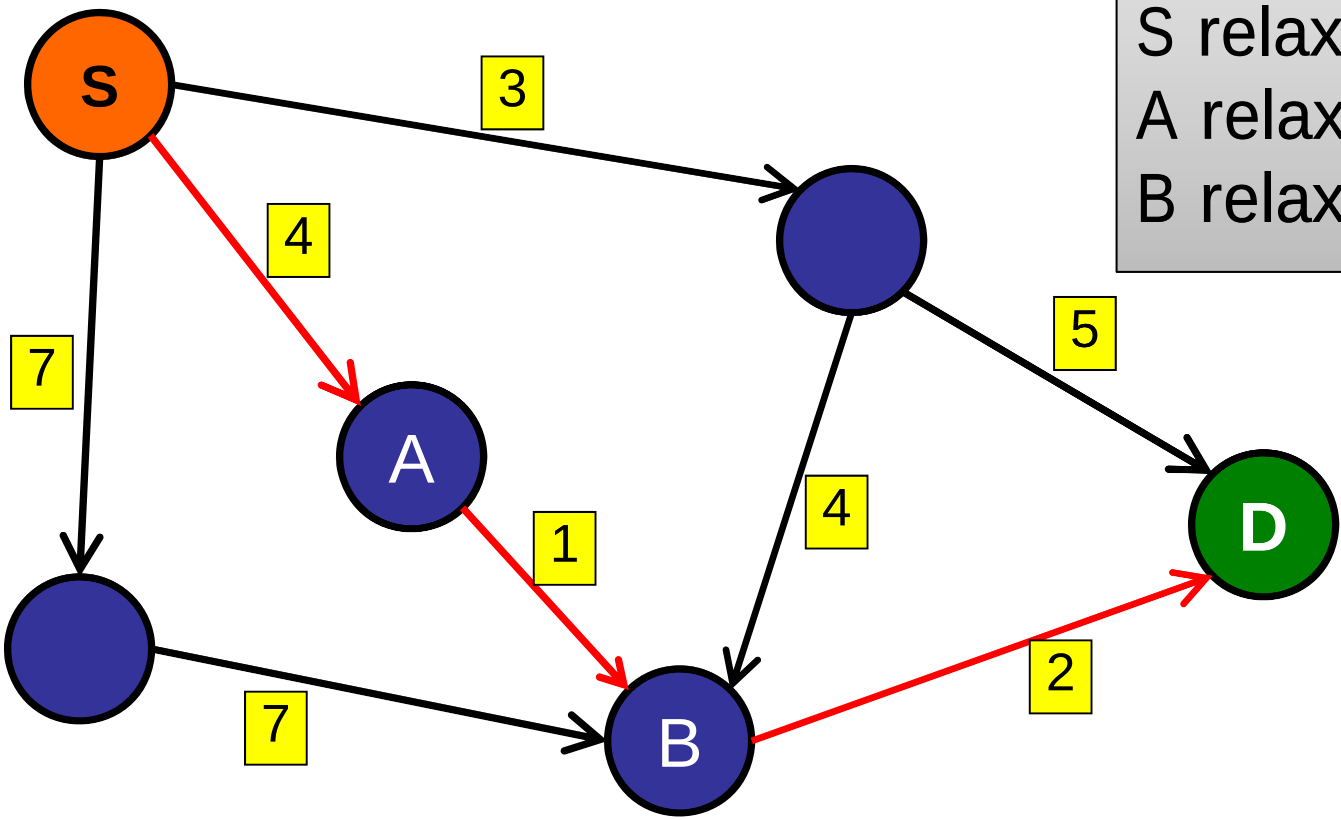


Shortest Paths

Acyclic Graph: Why topological order?

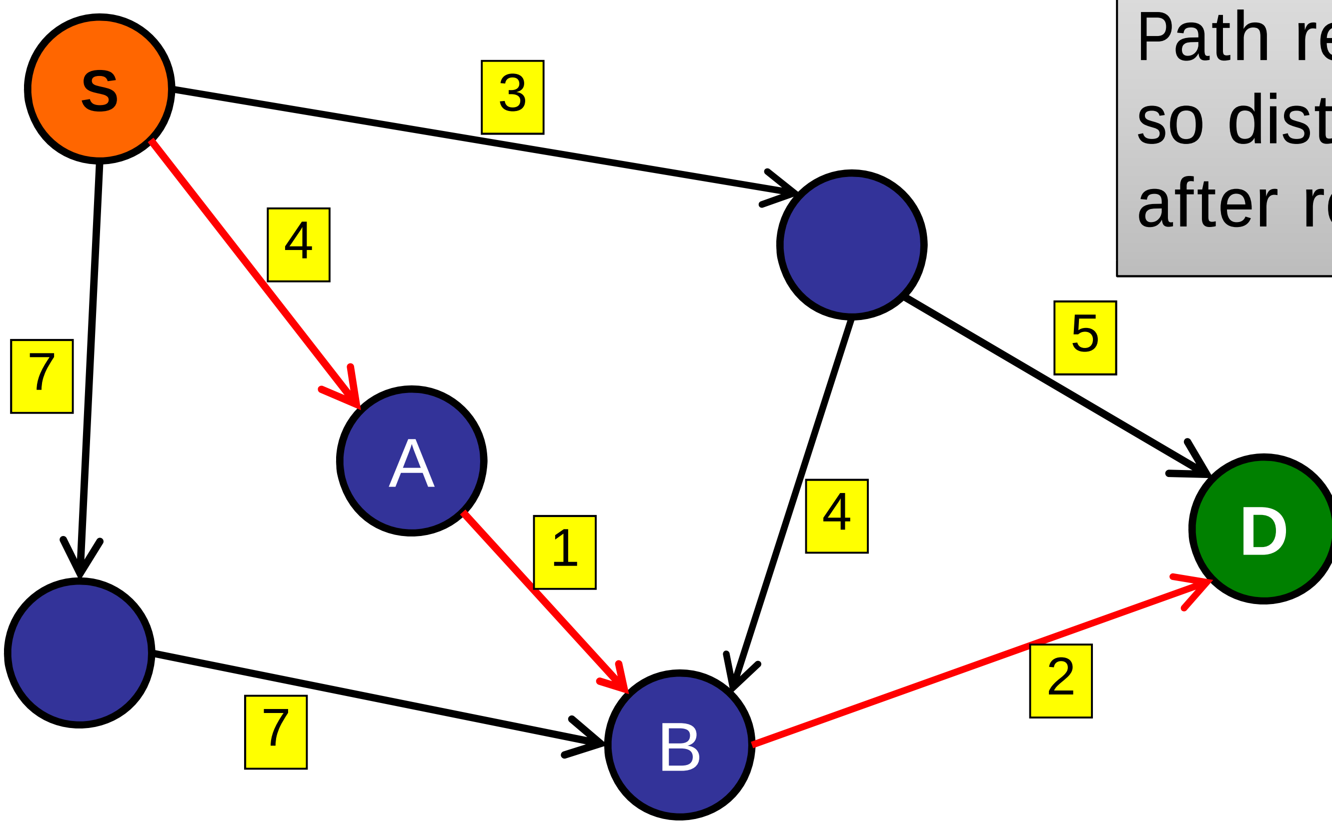
Fix S-D shortest path.

S relaxed before A.
A relaxed before B.
B relaxed before D.



Shortest Paths

Acyclic Graph: Why topological order?

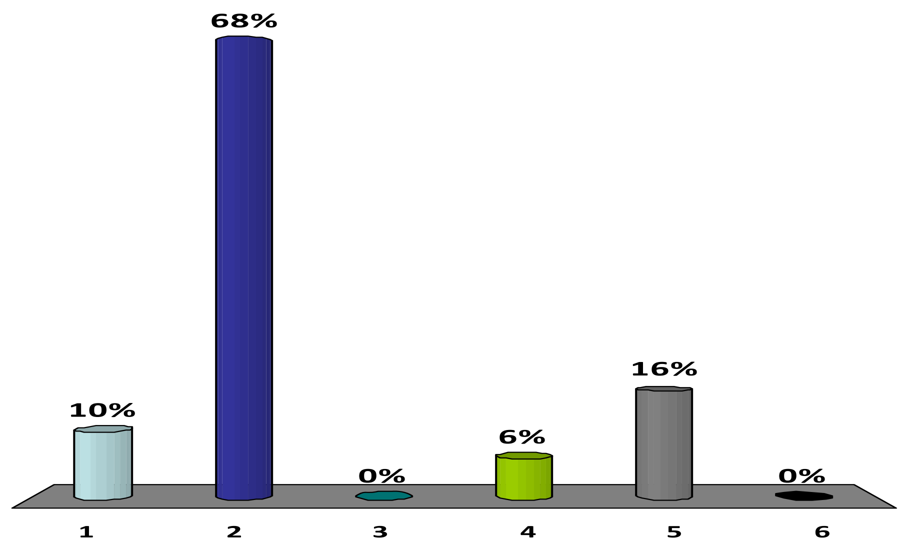


Fix S-D shortest path.

Path relaxed in-order, so distance is correct after relaxation.

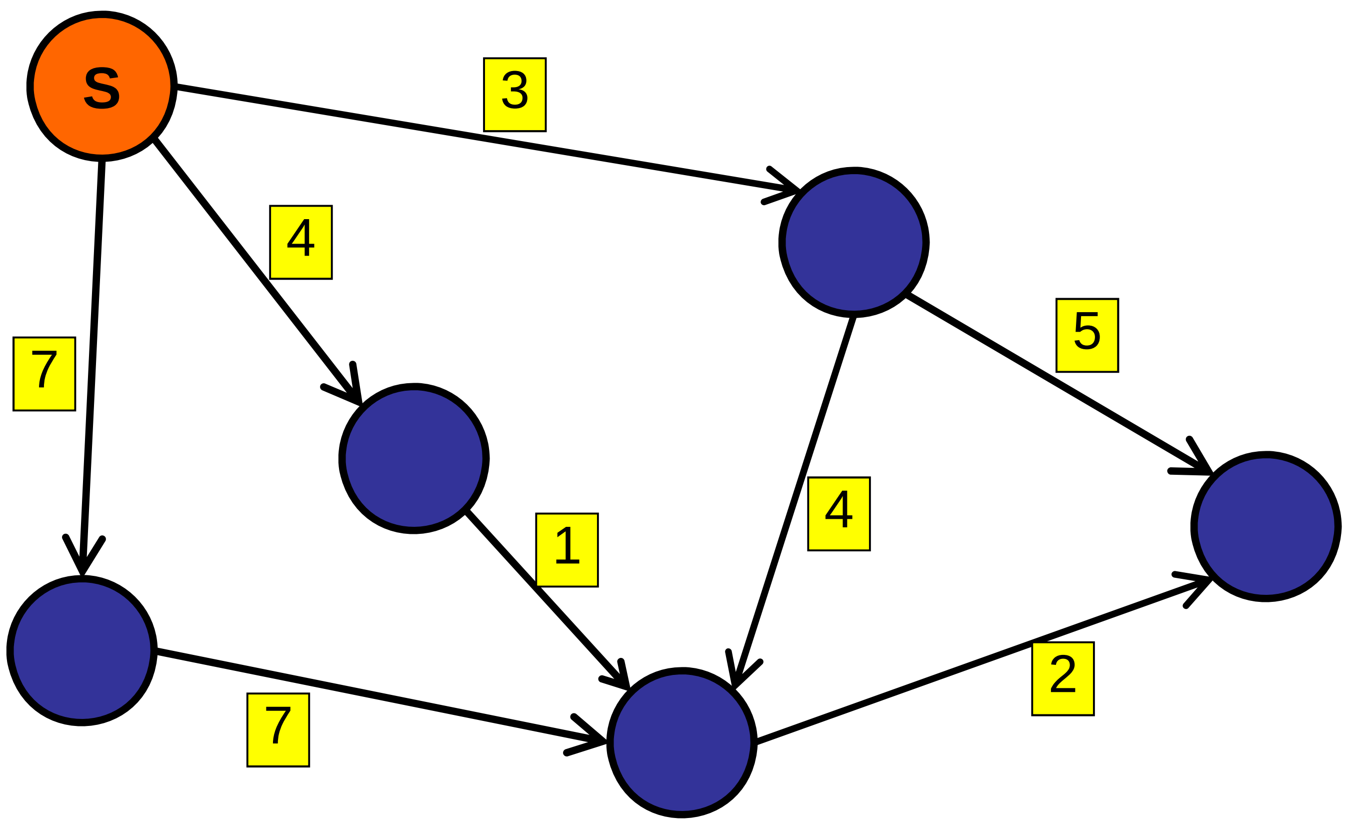
What is the running time of shortest paths on a DAG?

1. O(V)
2. O(E)
3. O(V^2)
4. O(E log V)
5. O(V log E)
6. O(VE)



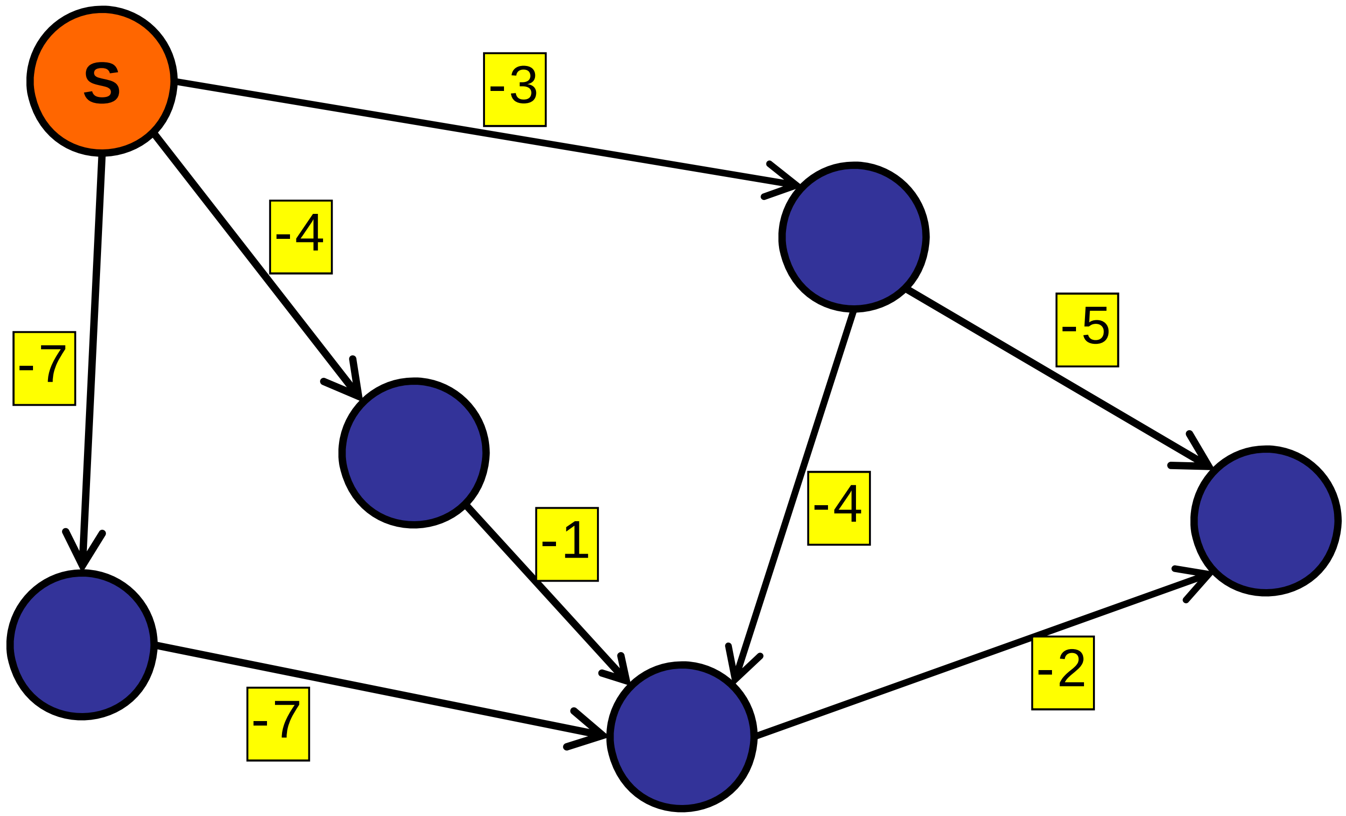
Longest Paths

Acyclic Graph: Any ideas?



Longest Paths

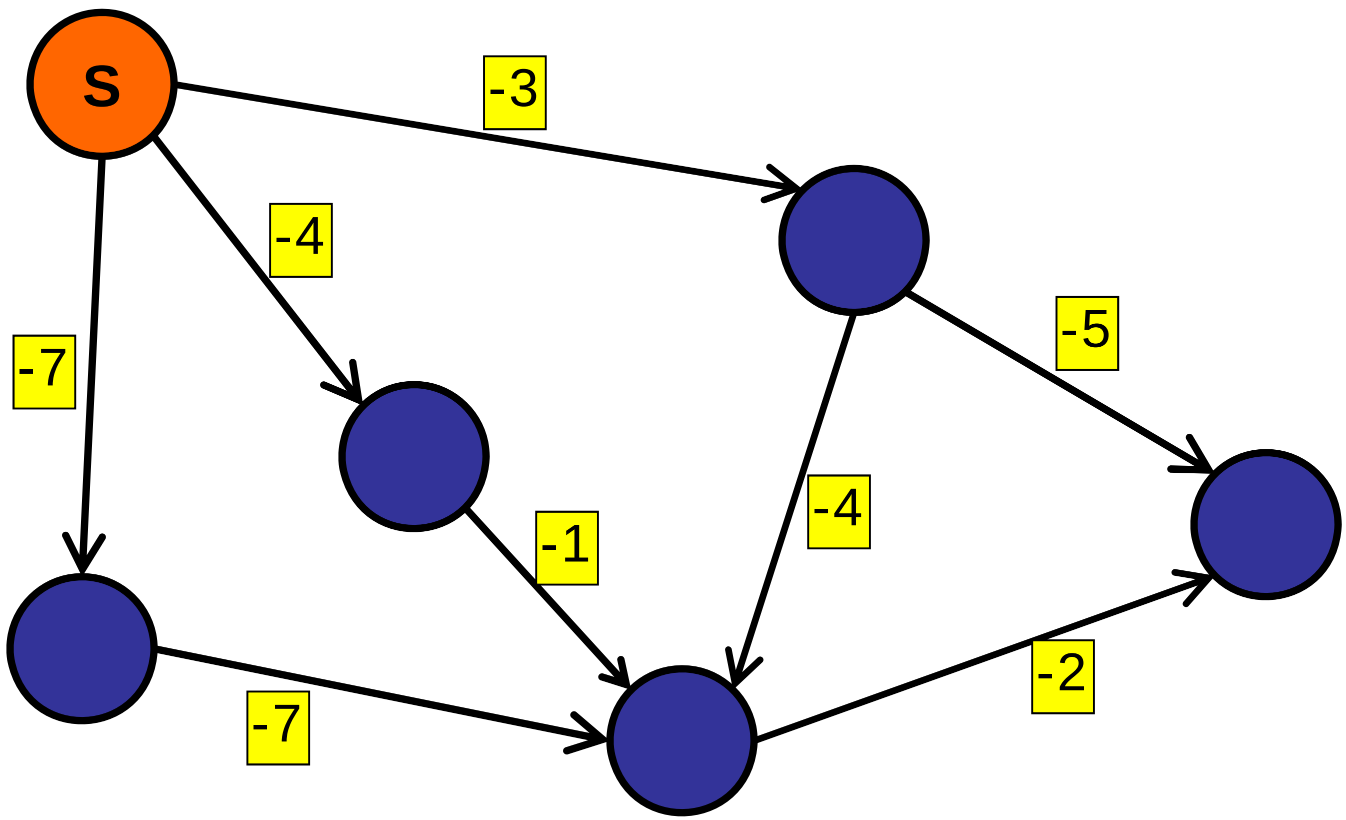
Acyclic Graph: Negate the edges!



Longest Paths

Acyclic Graph:

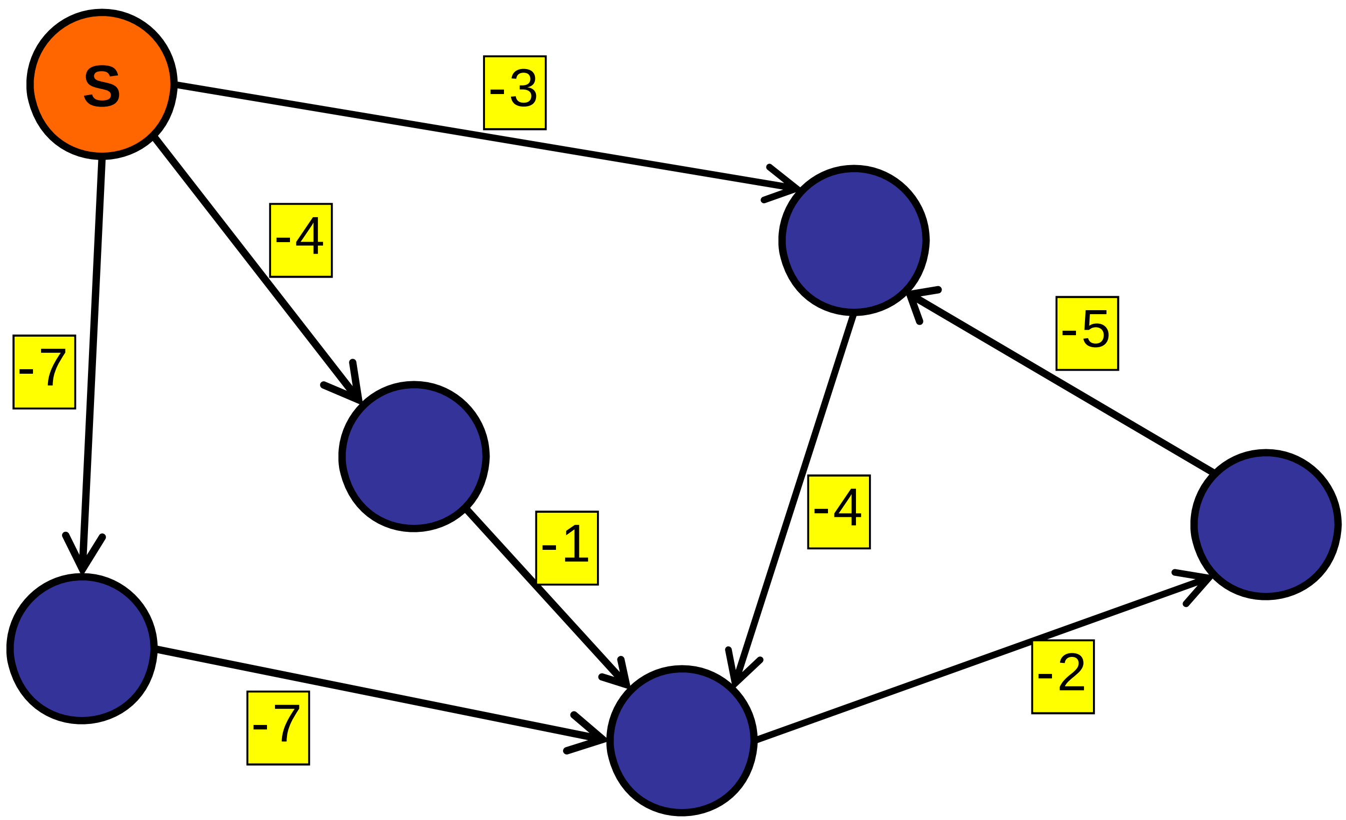
shortest path in negated=longest path in regular



Longest Paths

General (cyclic) Graph: (positive weights)

Can we use the same trick?

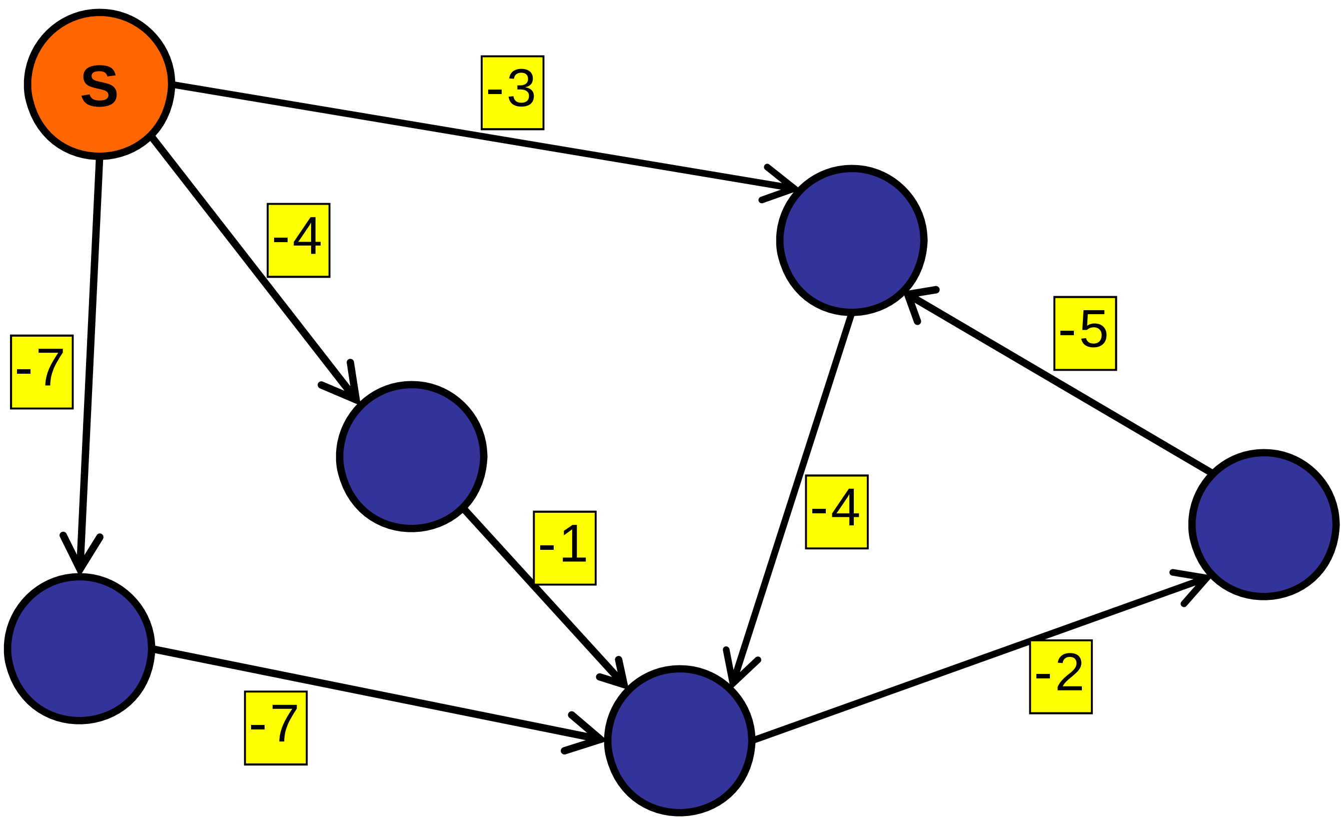


Longest Paths

General (cyclic) Graph: (positive weights)

Can we use the same trick? NO

Negative weight cycles!



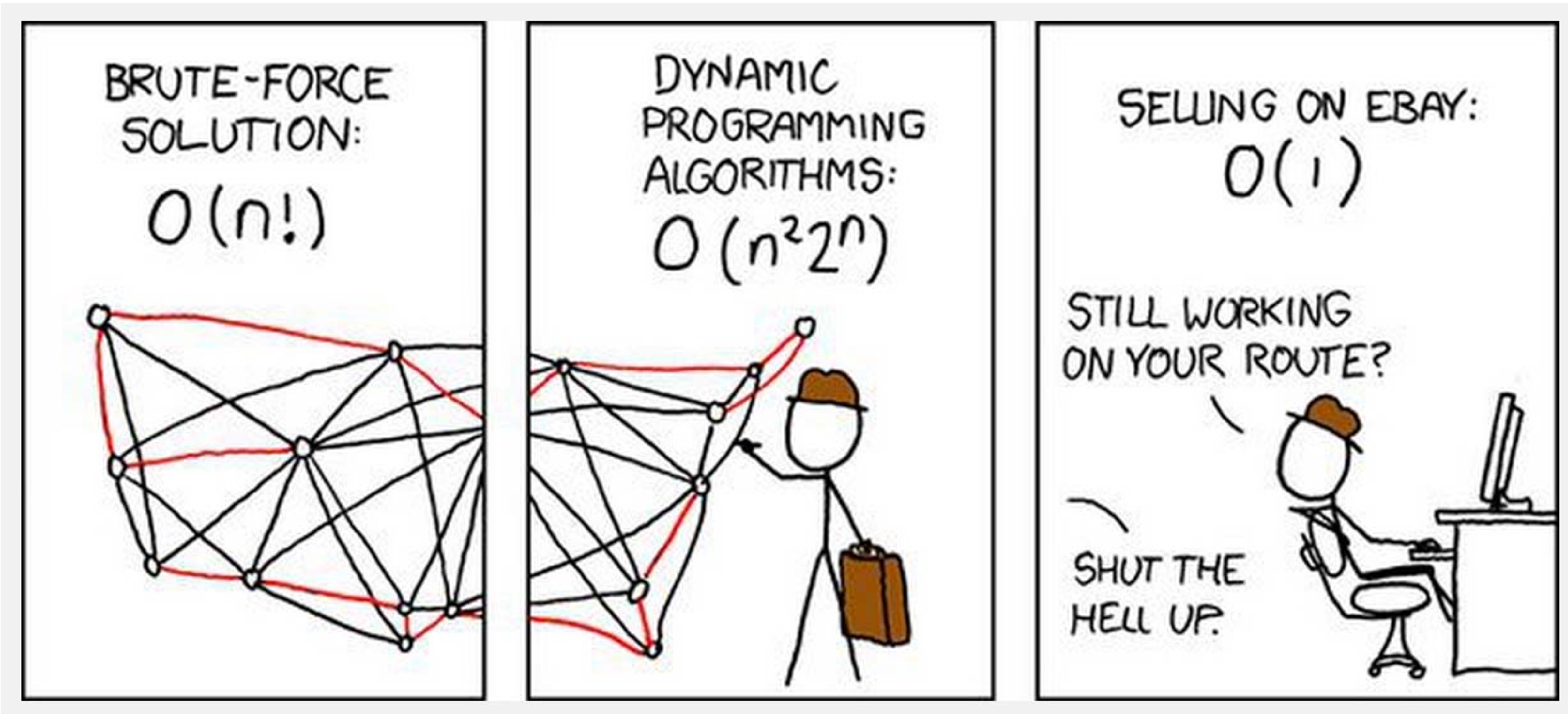
Longest Path

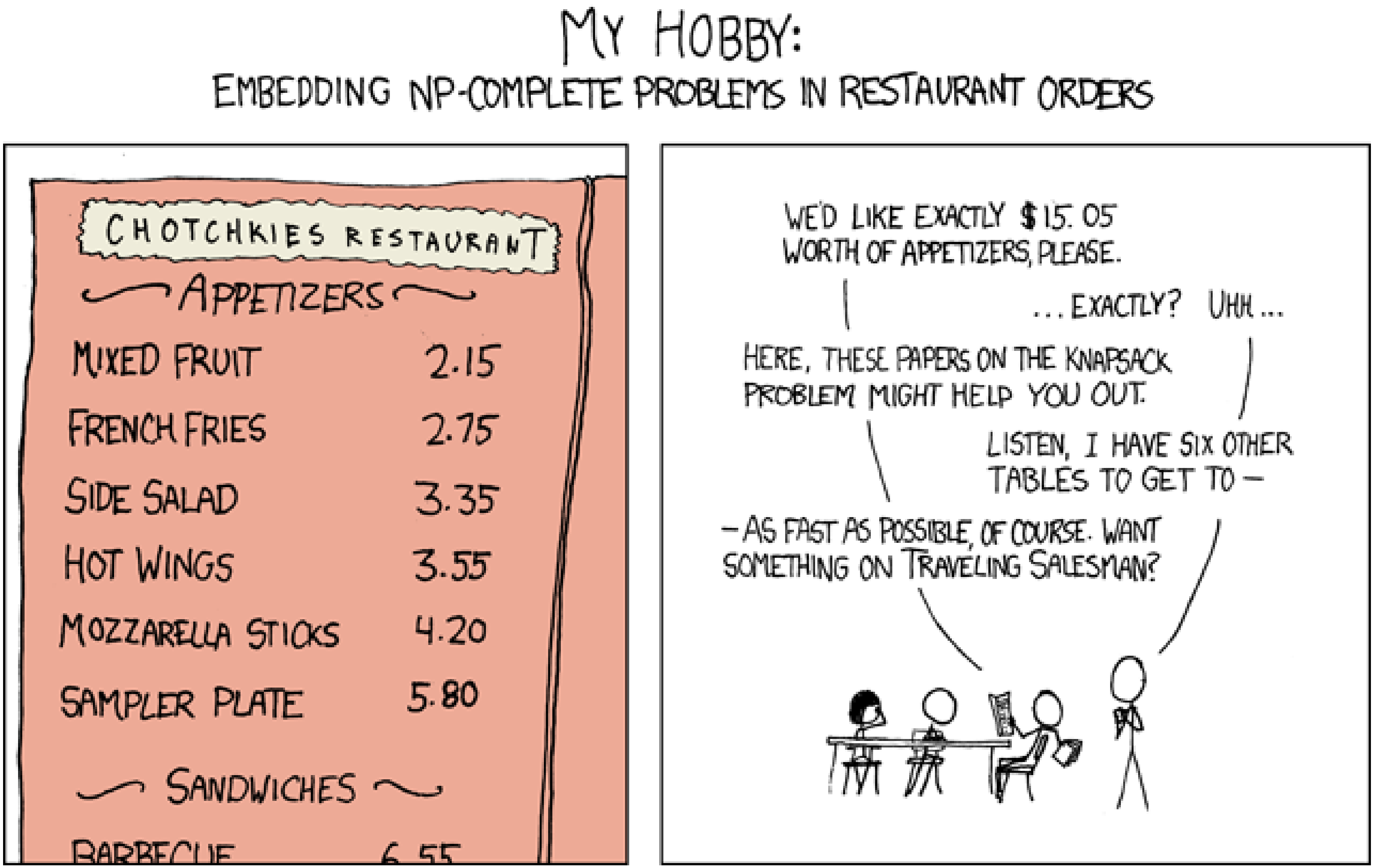
Directed Acyclic Graph:

- Solvable efficiently using topological sort

General (cyclic) Graphs:

- NP-Hard
- Reduction from Hamiltonian Path:
  - If you could find the longest simple path, then you could decide if there is a path that visits every vertex.
  - Any polynomial time algorithm for longest path thus implies a polynomial time algorithm for HAMPATH.


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Roadmap

Part I: Shortest Paths

- Special Case: Tree
- Special Case: Non-negative weights (Dijkstra's)
- Special Case: Directed Acyclic Graphs

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA
- AGAATCCGAA

How similar are they?

- Metric: edit distance

  How many operations to transform one DNA string into another?

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA ← delete G, delete T
- AGAATCCGA ← add T

Three operations:

- Delete a character
- Add a character
- Transform a character

Example: DNA Alignment

Input: two DNA strings:

- AGGAACCGTA ← delete G, delete T
- AGAATCCGA ← add T

Three operations:

- Delete a character cost = d
- Add a character cost = a
- Transform a character cost = t

OR: minimum *cost* to transform A to B?

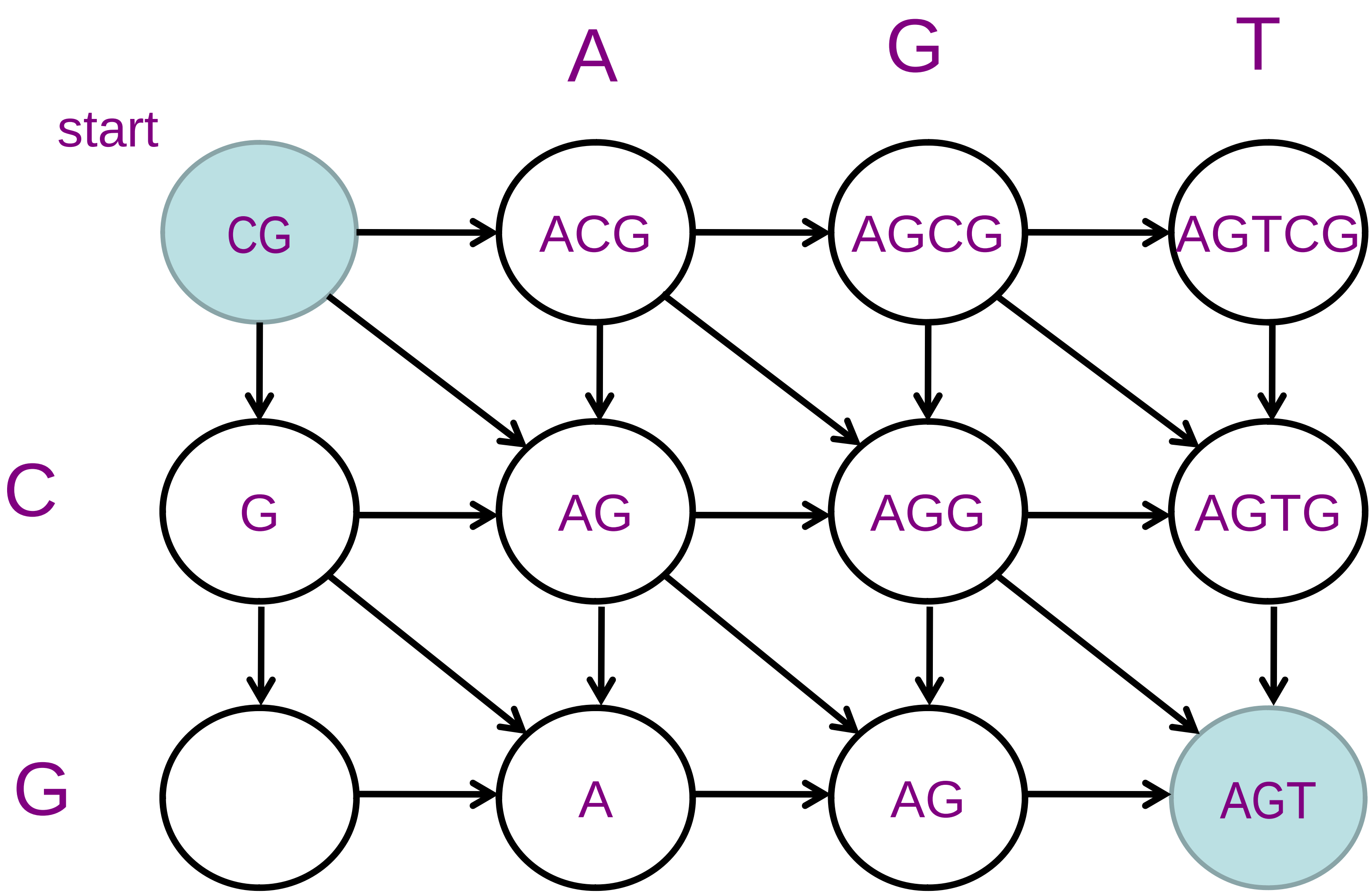
Example: DNA Alignment

Model question as a directed graph:

- For each character i, character j:
  - Create a node in the graph N(i,j)
  - N(i,j) represents adapting position i of the old string to match position j of the new string.
- For node N(i,j), three outgoing edges:
  - insert character j+1 from new string after position i
  - delete character i+1 from old string
  - transform character i+1 to character j+1

Example: DNA Alignment

Transform: CG to AGT



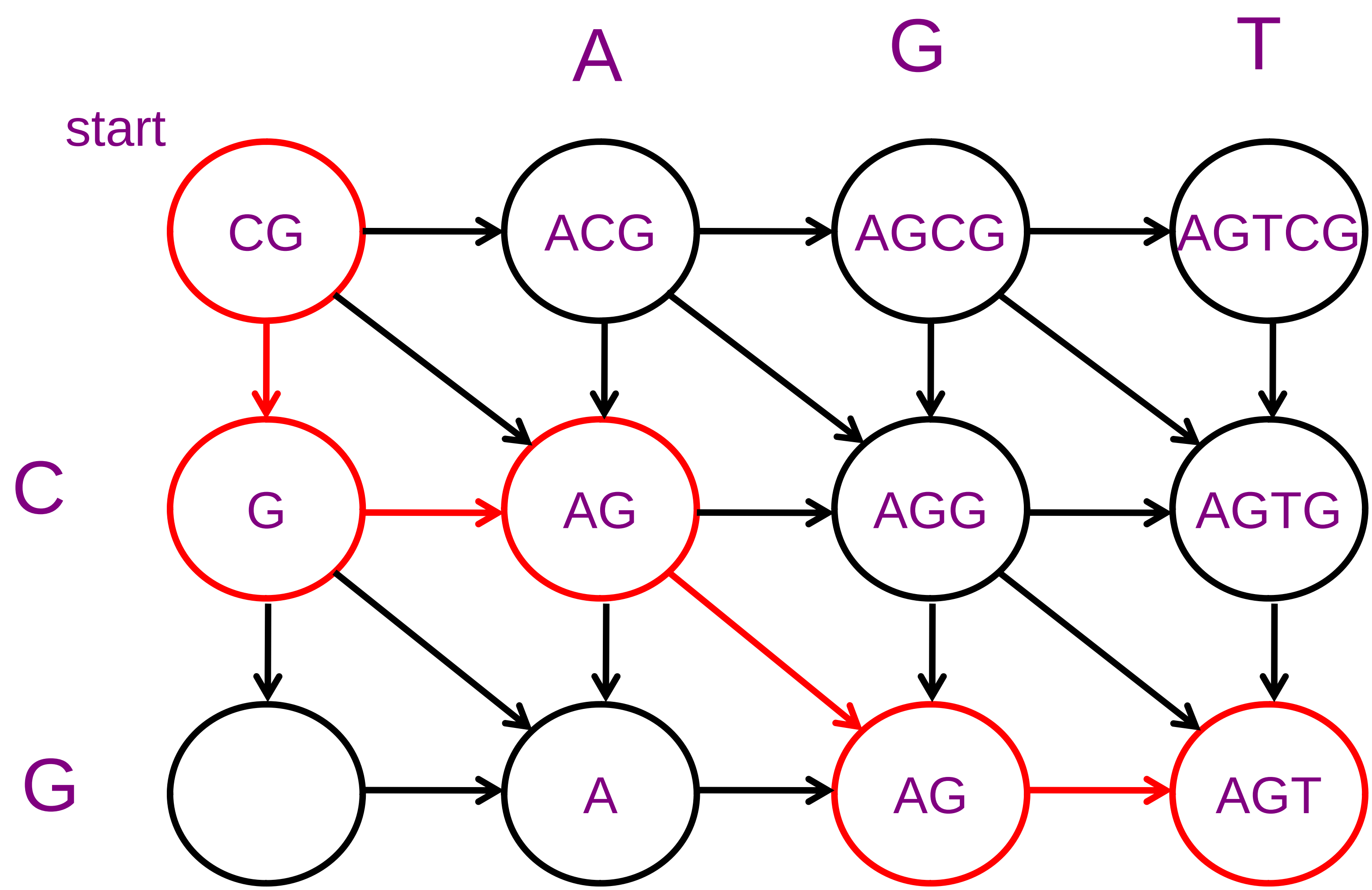
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Delete C, Add A, Leave G, Add T:



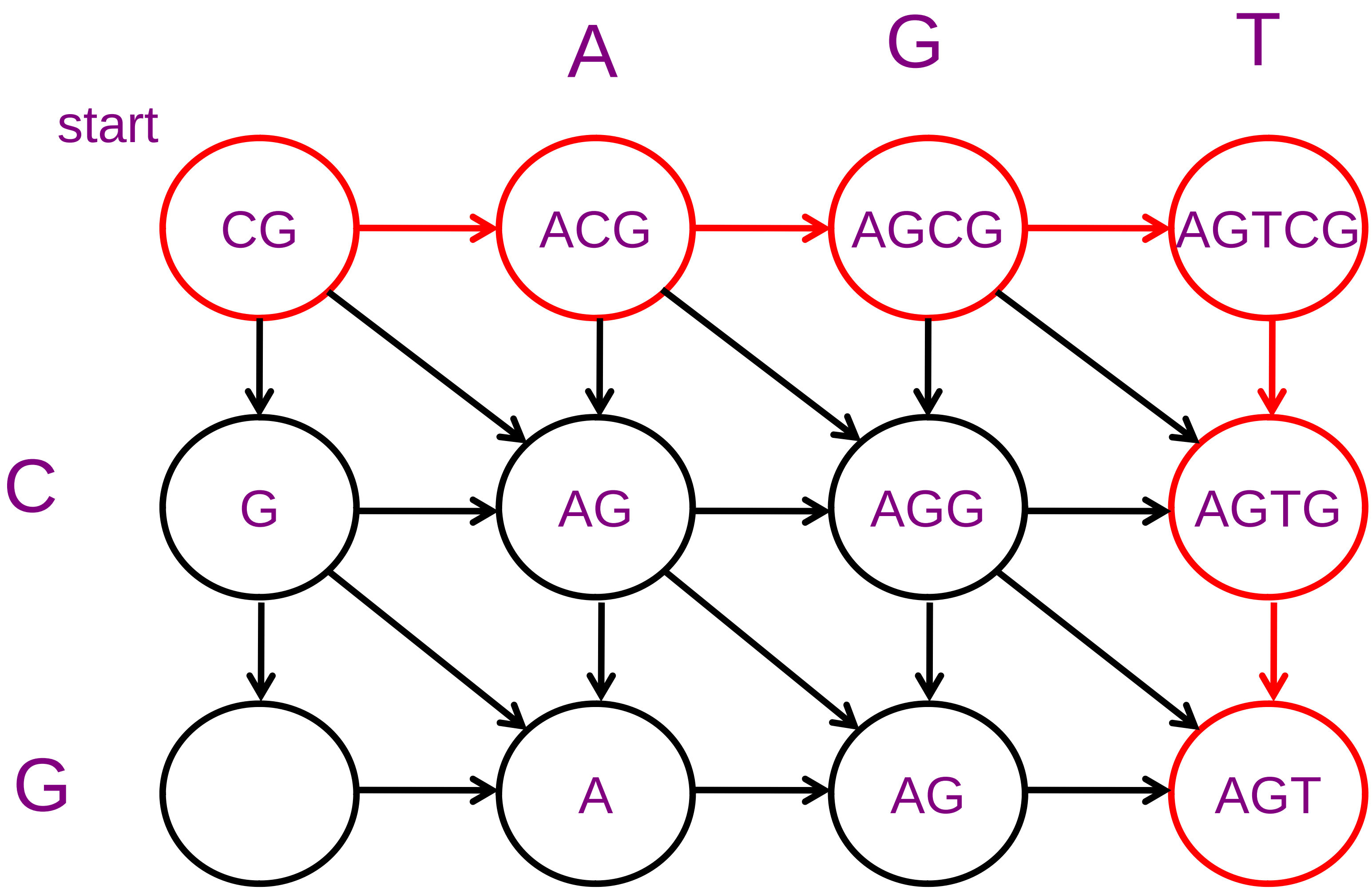
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Add A, Add G, Add T, Delete C, Delete G:



Vertical: delete character

Horizontal: add character

Diagonal: transform character

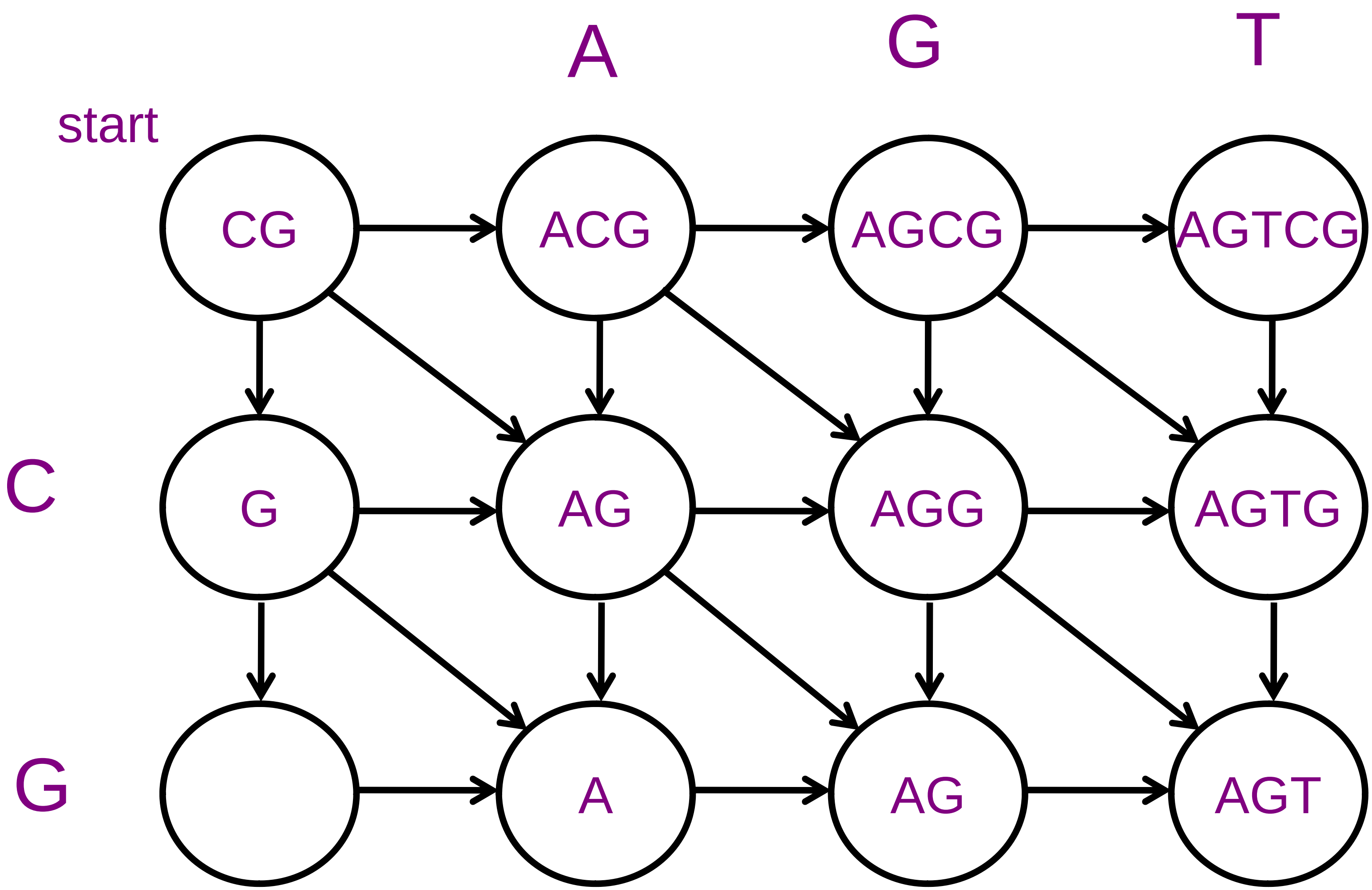
Example: DNA Alignment

Model question as a directed graph:

- For node N(i,j):
  - The first i letters of the old string have been replaced with the first j letters of the new string.
  - The shortest path to N(i,j) is the shortest set of changes to change the first i letters of the old string to the first j letters of the new string.

Example: DNA Alignment

Transform: CG to AGT



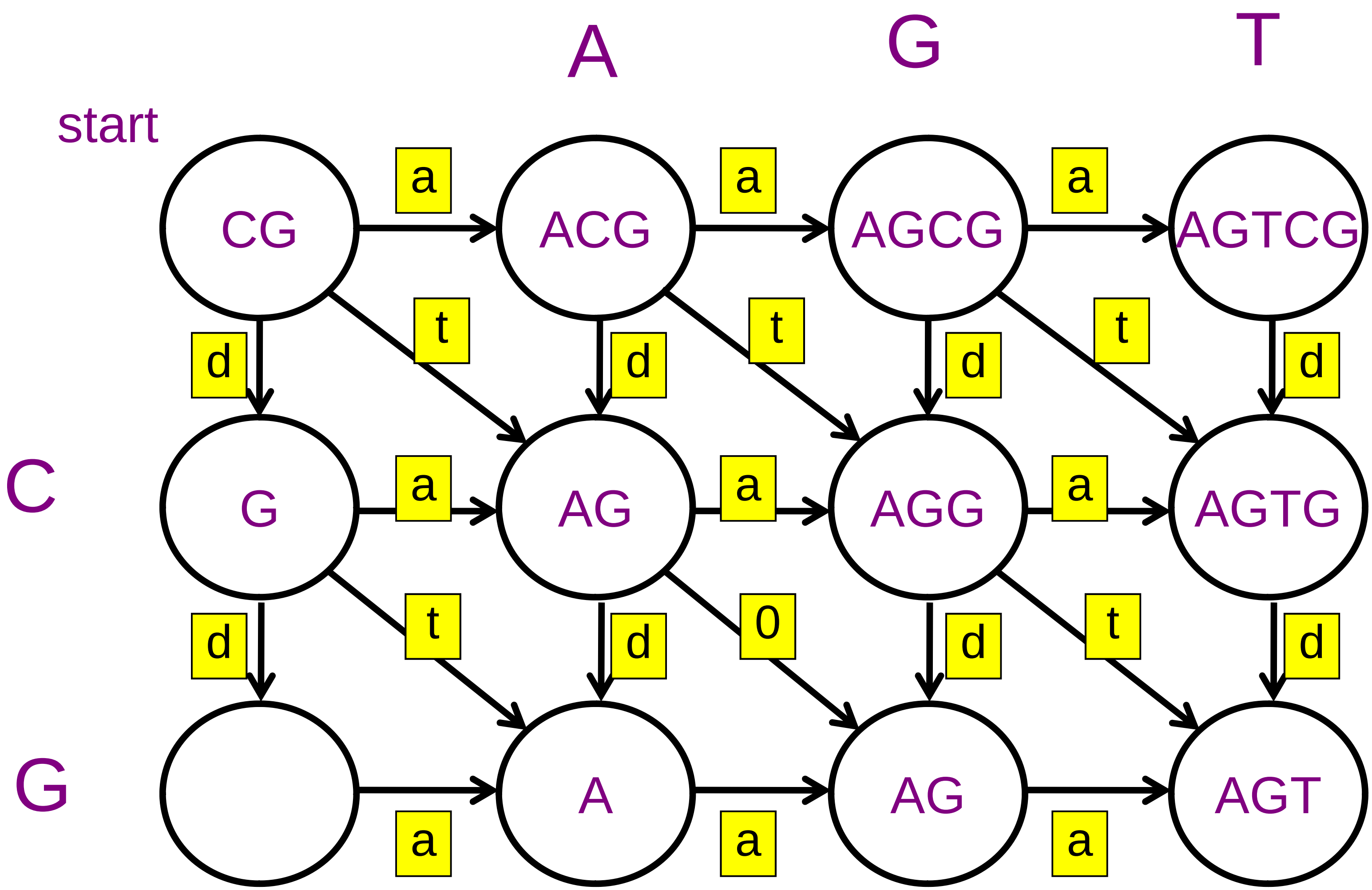
Vertical: delete character

Horizontal: add character

Diagonal: transform character

CG to AGT

Edge costs:



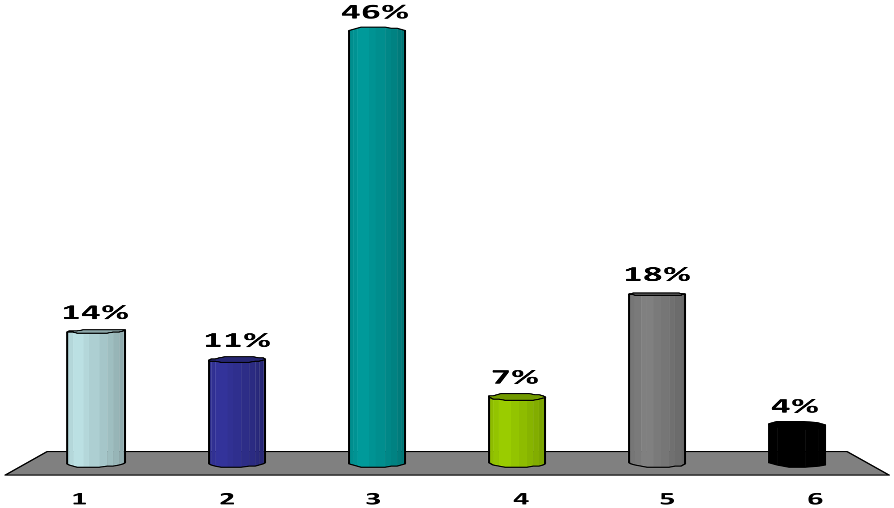
Vertical: delete character

Horizontal: add character

Diagonal: transform character

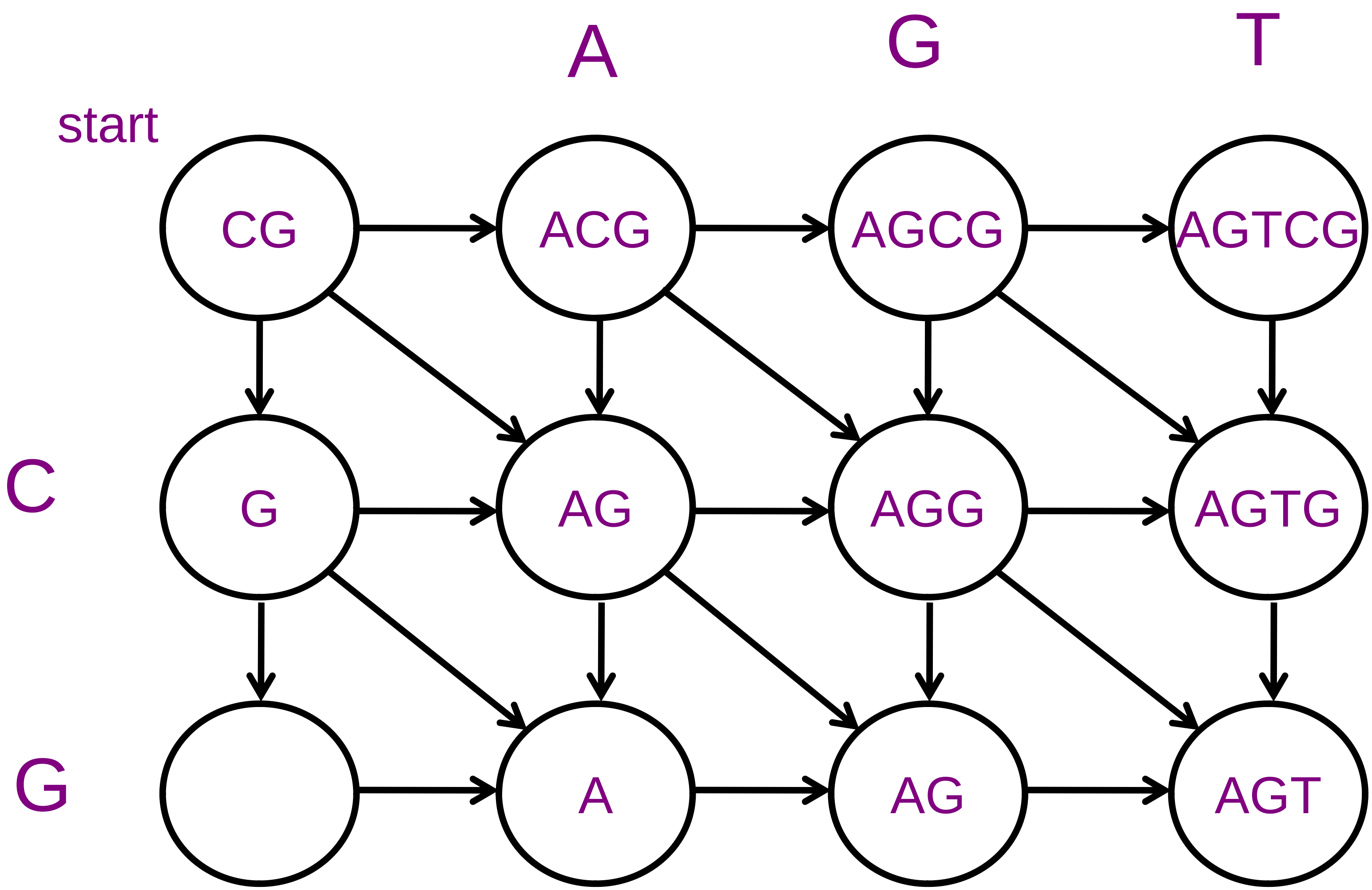
What is the running time to find the minimum edit distance from a string of length n to a string of length n?

1. O(n)
2. O(n log n)
3. $O(n^2)$
4. $O(n^2 \log n)$
5. $O(n^3)$
6. $O(n^4)$



Example: DNA Alignment

Transform: CG to AGT



Vertical: delete character

Horizontal: add character

Diagonal: transform character

Roadmap

Part I: Shortest Paths

- Special Case: Tree
- Special Case: Non-negative weights (Dijkstra’s)
- Special Case: Directed Acyclic Graphs

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems

Example: Scheduling

Input:

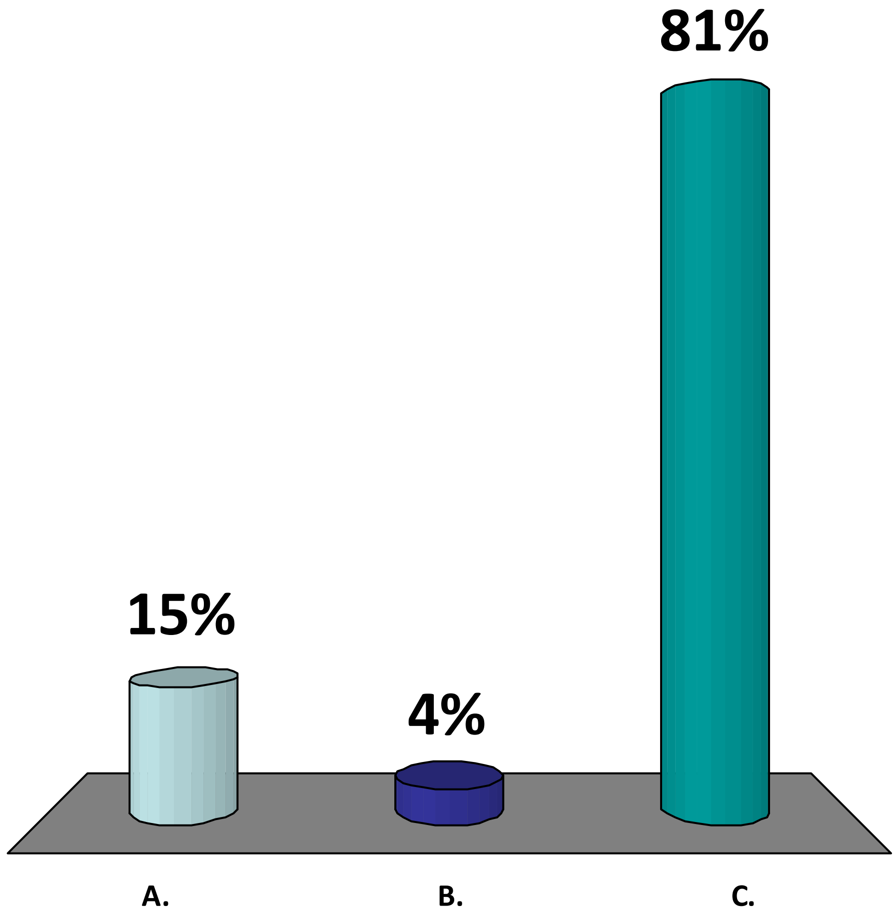
- Set of tasks: A, B, C, D, E, F
- Constraints:
  - A must be done at least 10 minutes before C
  - D must be done at most 20 minutes after E
  - B must be done after F

Output:

- Feasible?
- Schedule?

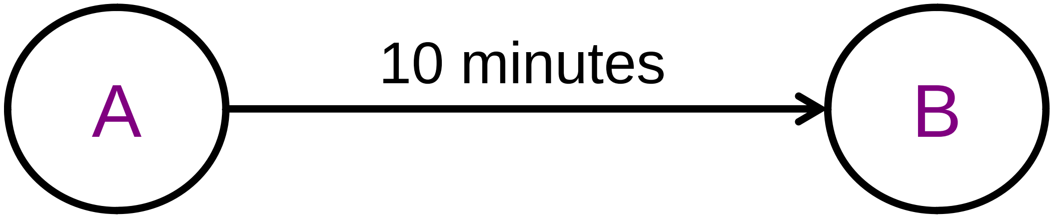
We can assume there is no negative cycle because...

A. we said so
B. our algorithms cannot solve it
C. Negative cycles make the scheduling problem meaningless



Example: Scheduling

B must be executed **at most** 10 minutes **after** A

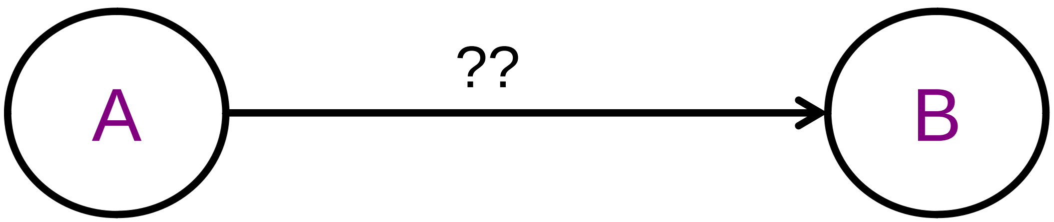


Shortest path = schedule time

triangle inequality: shortest path to B is at most 10 longer than shortest path to A

Example: Scheduling

B must be executed **at least** 10 minutes **after** A

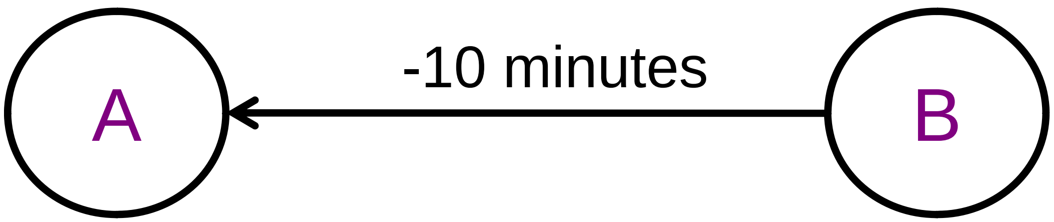


Shortest path = schedule time

triangle inequality: shortest path to B is at most 10 longer than shortest path to A

Example: Scheduling

B must be executed **at least** 10 minutes **after** A

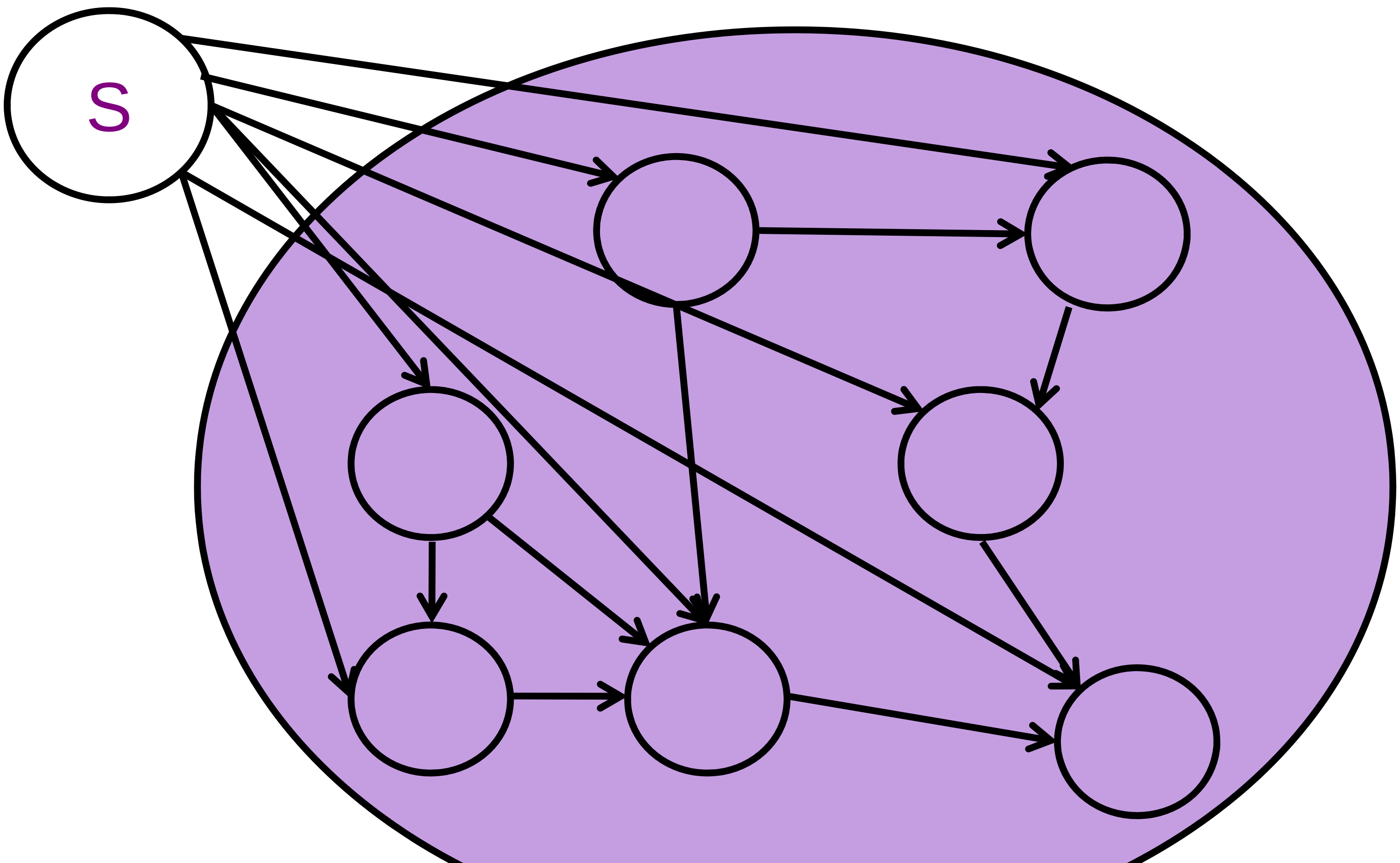


Shortest path = schedule time

triangle inequality: shortest path to B is at least 10 longer than shortest path to A

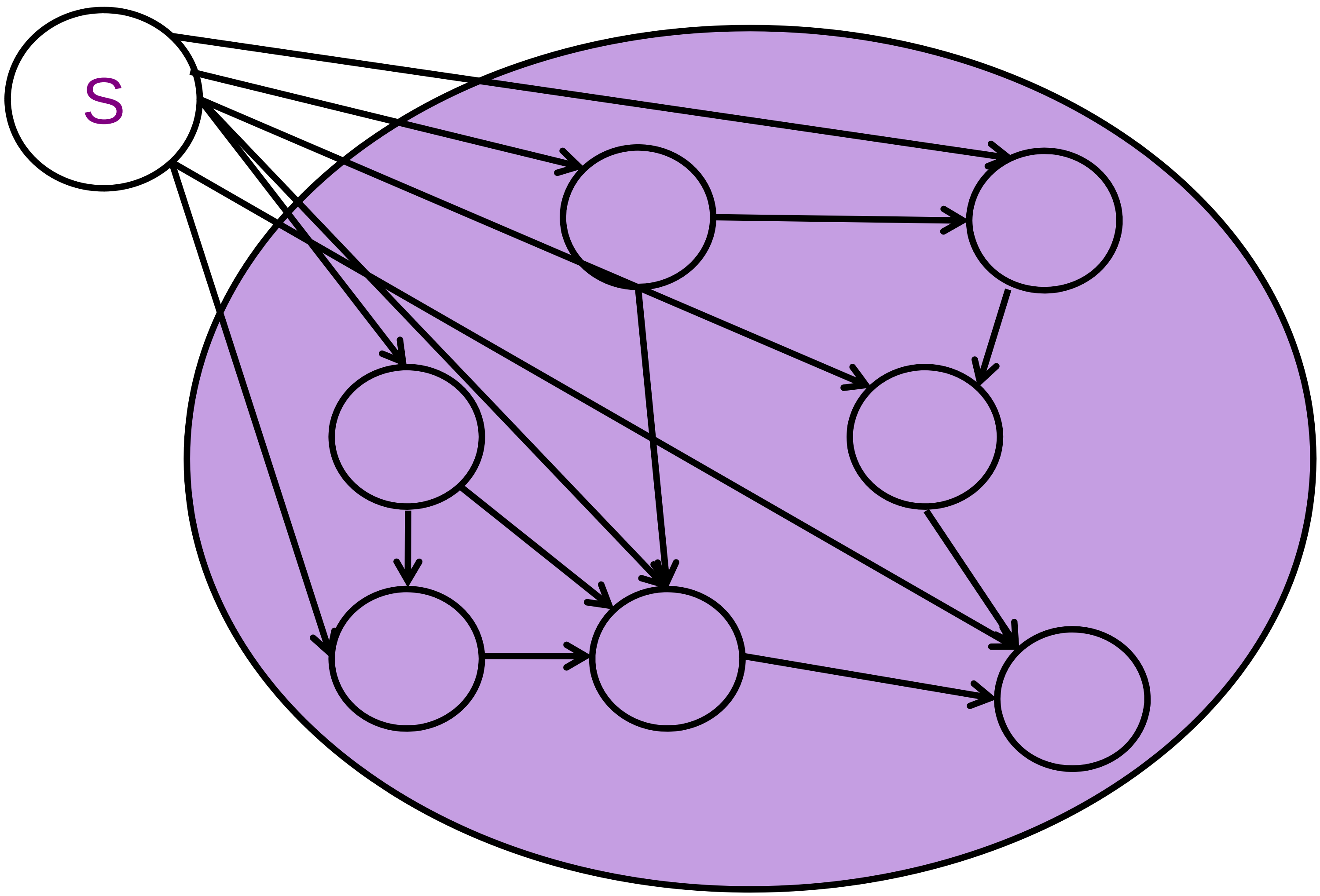
Example: Scheduling

Add source S connected by 0 weight edges to all.



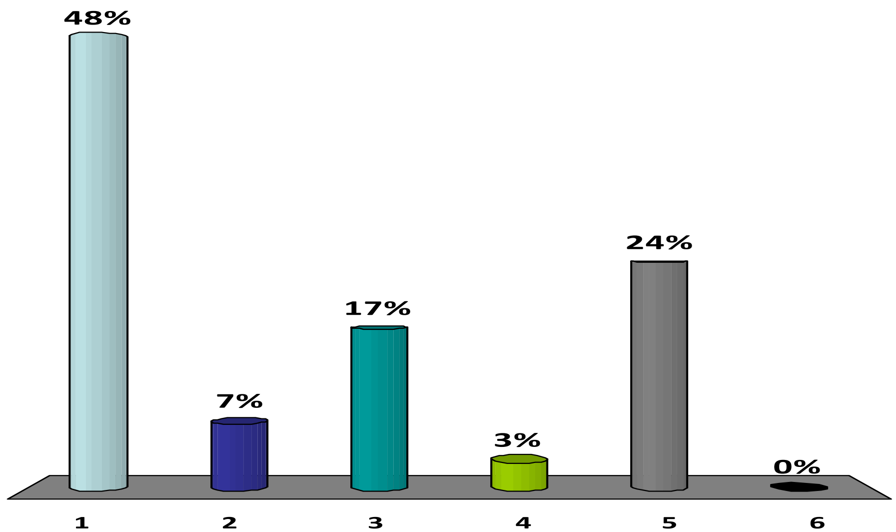
Example: Scheduling

Solve shortest paths.



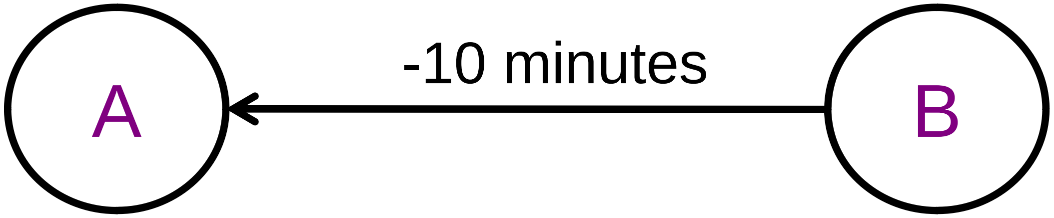
What is the running time to find the schedule for n jobs and m constraints?

1. $O(n + m)$
2. $O(n \log m)$
3. $O(m \log n)$
4. $O(n^2)$
5. $O(nm)$
6. $O(n^m)$



Example: Scheduling

B must be executed **at least** 10 minutes **after** A



Negative edges: use Bellman-Ford!

Running time: O(nm)

Example: Scheduling

Input:

- Set of tasks: A, B, C, D, E, F
- Constraints:
  - A must be done at least 10 minutes before C
  - D must be done at most 20 minutes after E
  - B must be done after F

Output:

- Shortest path guarantees constraints are met.
- Shortest path finishes all tasks in minimum time.

Roadmap

Part I: Shortest Paths

- Special Case: Tree
- Special Case: Non-negative weights (Dijkstra's)
- Special Case: Directed Acyclic Graphs

Part II: Applications of Shortest Paths

- DNA Alignment
- Constraint Systems