CS2040S
Data Structures and Algorithms

Shortest Paths & DAGs!

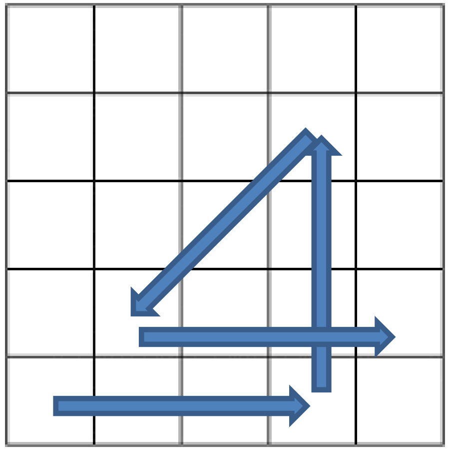
(Try writing a program to solve this!)

Puzzle of the week:

- 5 x 5 grid
- Choose a starting square
- Move: 3 cells vertically or horizontally OR
- Move: 2 cells diagonally.
- Cannot visit same cell twice.
- Cannot exit grid

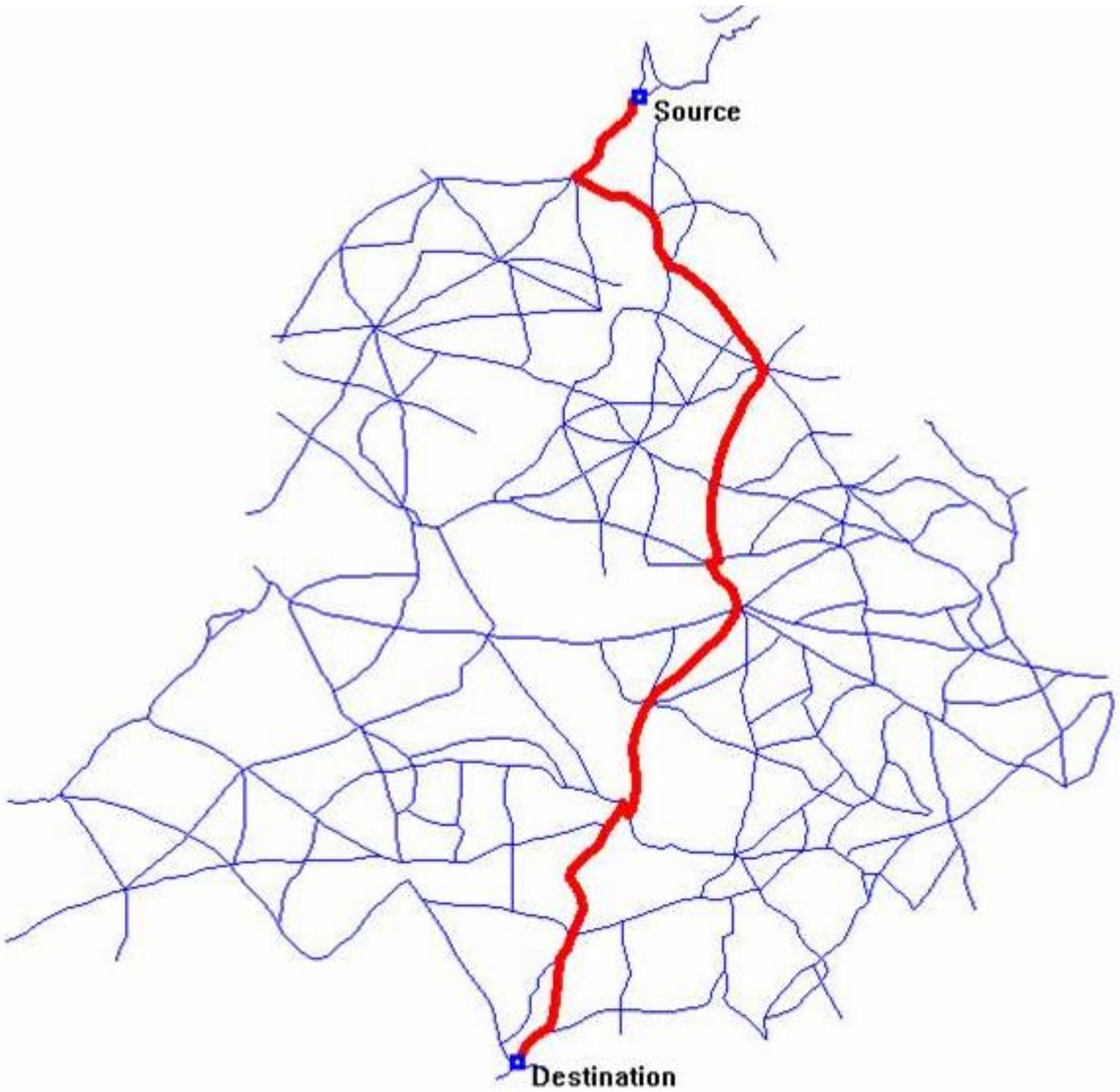
To win: visit all cells.

Example:



** What's the *worst* you can do?

SHORTEST PATHS



Shortest Path Problem

Basic question: find the shortest path!

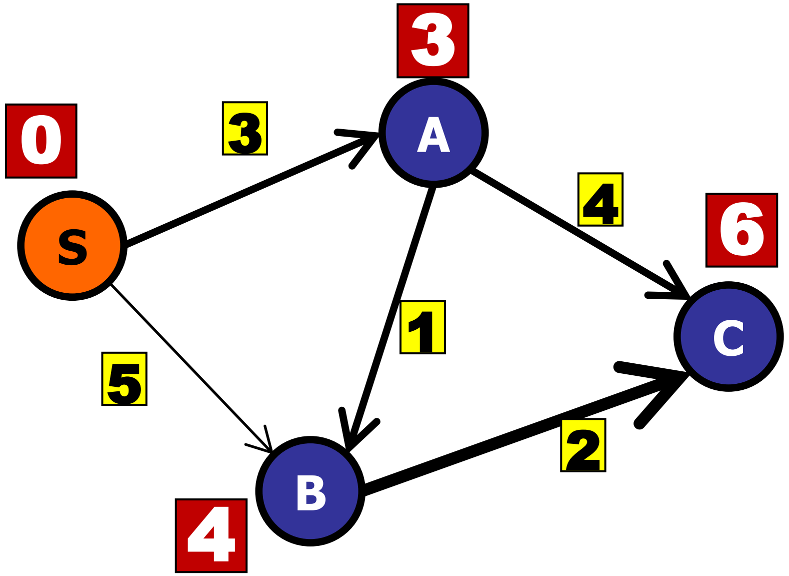
- Source-to-destination: one vertex to another
- Single source: one vertex to every other
- All pairs: between all pairs of vertices

Variants:

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

Bellman-Ford

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



Does Bellman-Ford always work in graphs with negative weights?

1. Yes
2. No
3. I forget

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.

Path relaxation property

- **CLAIM.** If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order
    - $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- Then $d[v_k] = \delta[v_k]$.
- This property holds ***regardless of any other relaxation steps that occur*** (even **intermixed**)
  - E.g., $(v_0, v_1), (v_i, v_j), (v_1, v_2), \dots, (v_{k-1}, v_k)$ will *still* result in $d[v_k] = \delta[v_k]$.

Special Cases

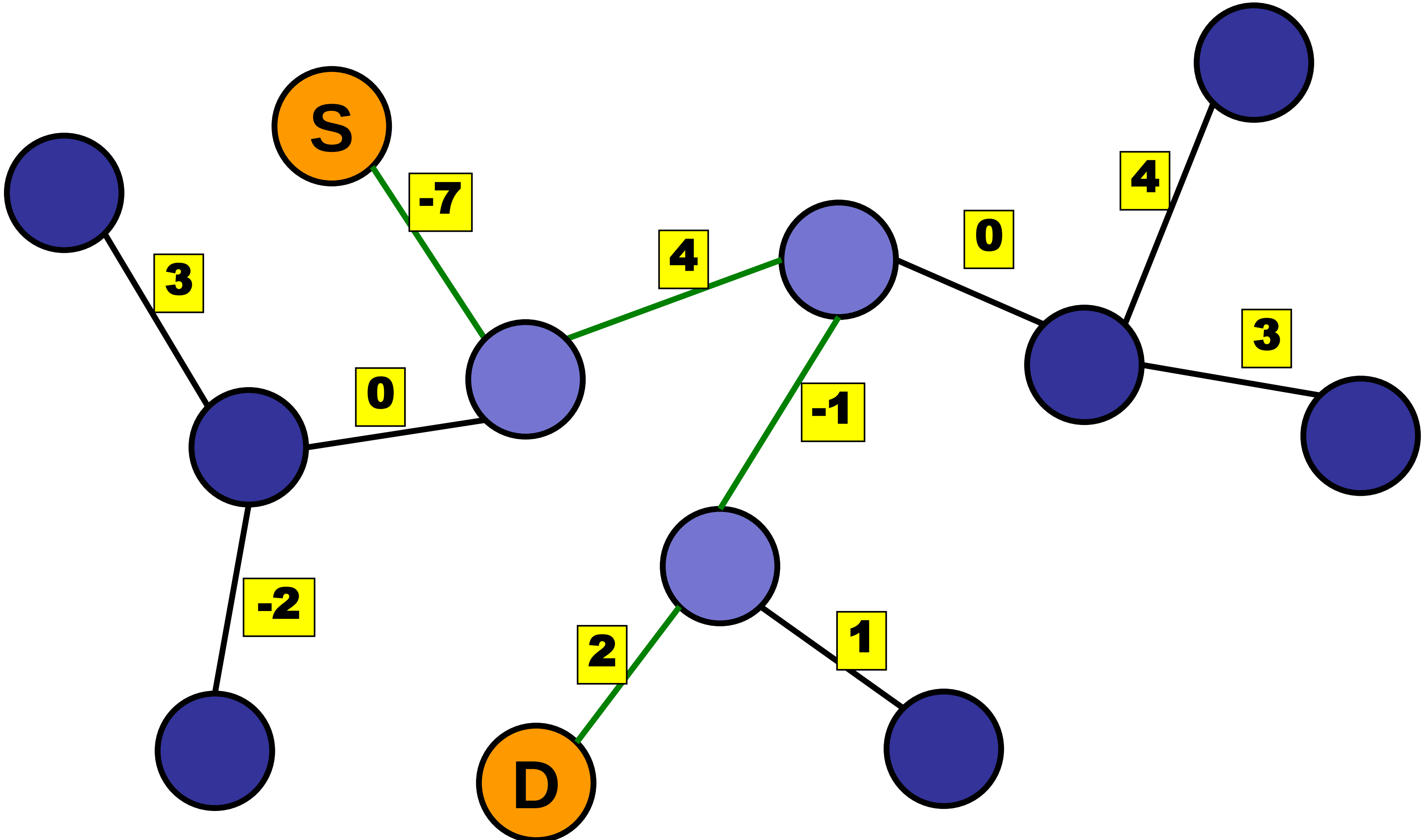
Condition	**Algorithm**	**Time Complexity**
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V+E)$
No Negative Weights	Dijkstra’s Algorithm	
On Tree		
On DAG		

Trees

For weighted trees (possibly with negative weights), design an $O(V)$ time SSSP algorithm.

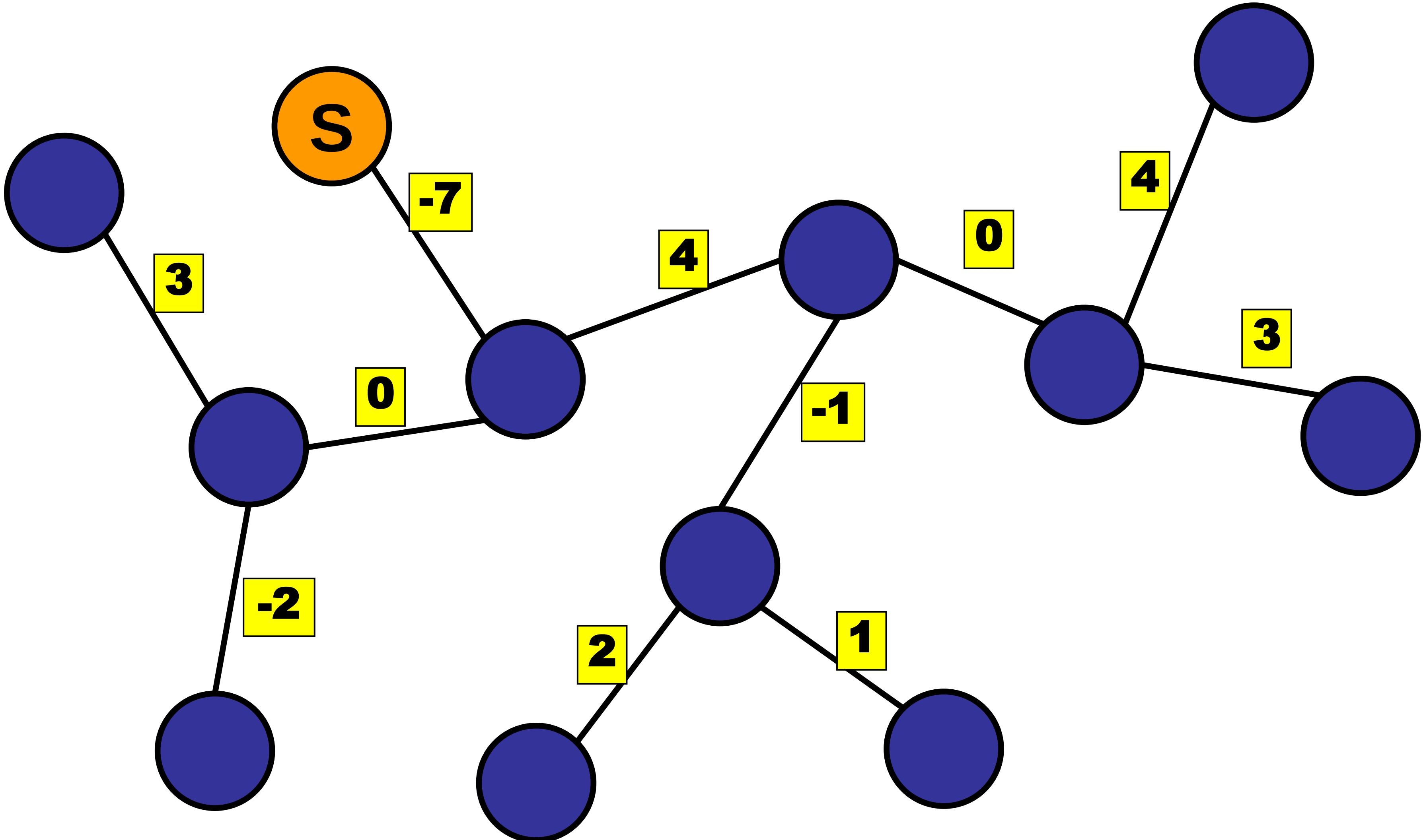
Special Case: Tree

source-to-destination: only one possible path!



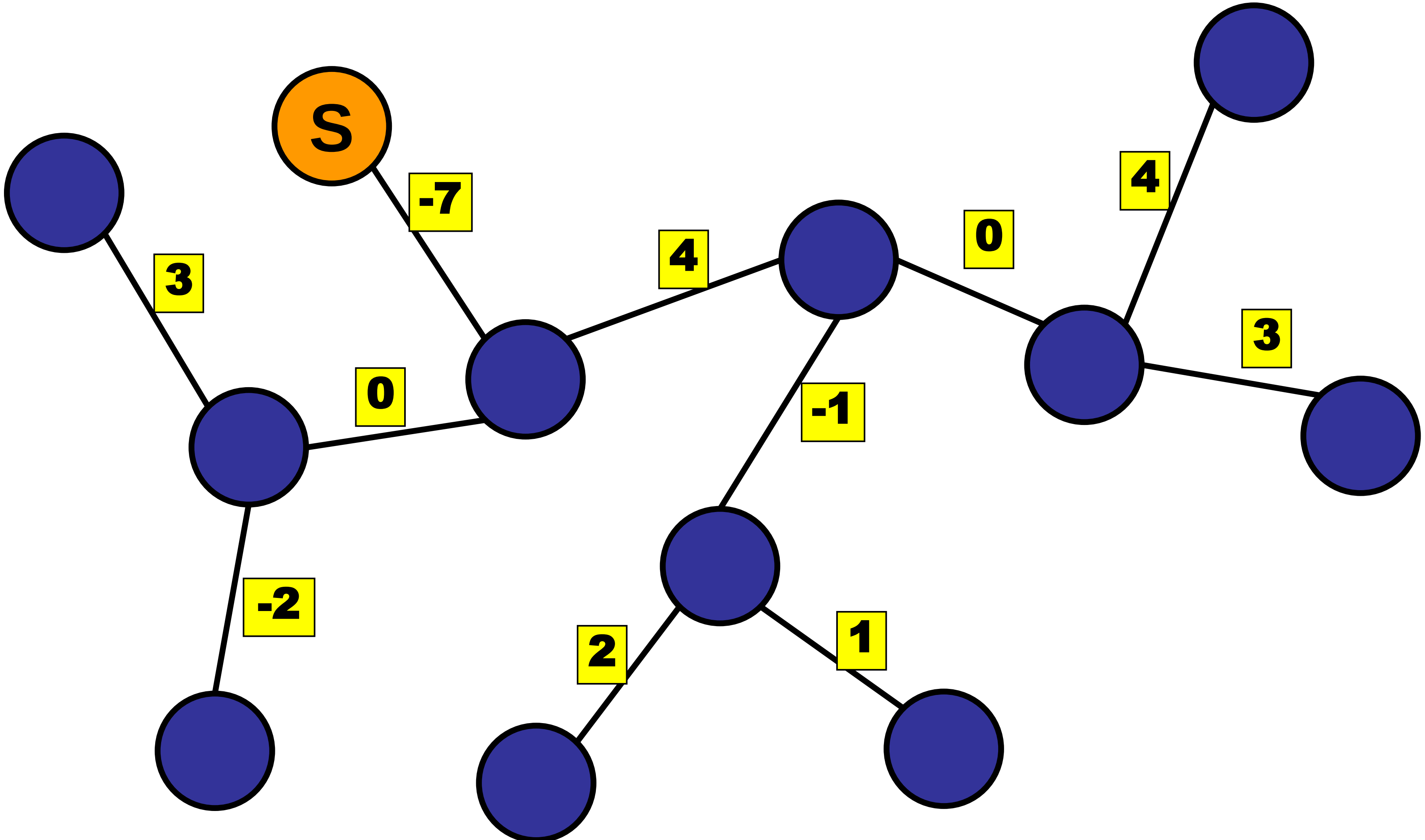
Special Case: Tree

source-to-all: what order to relax?



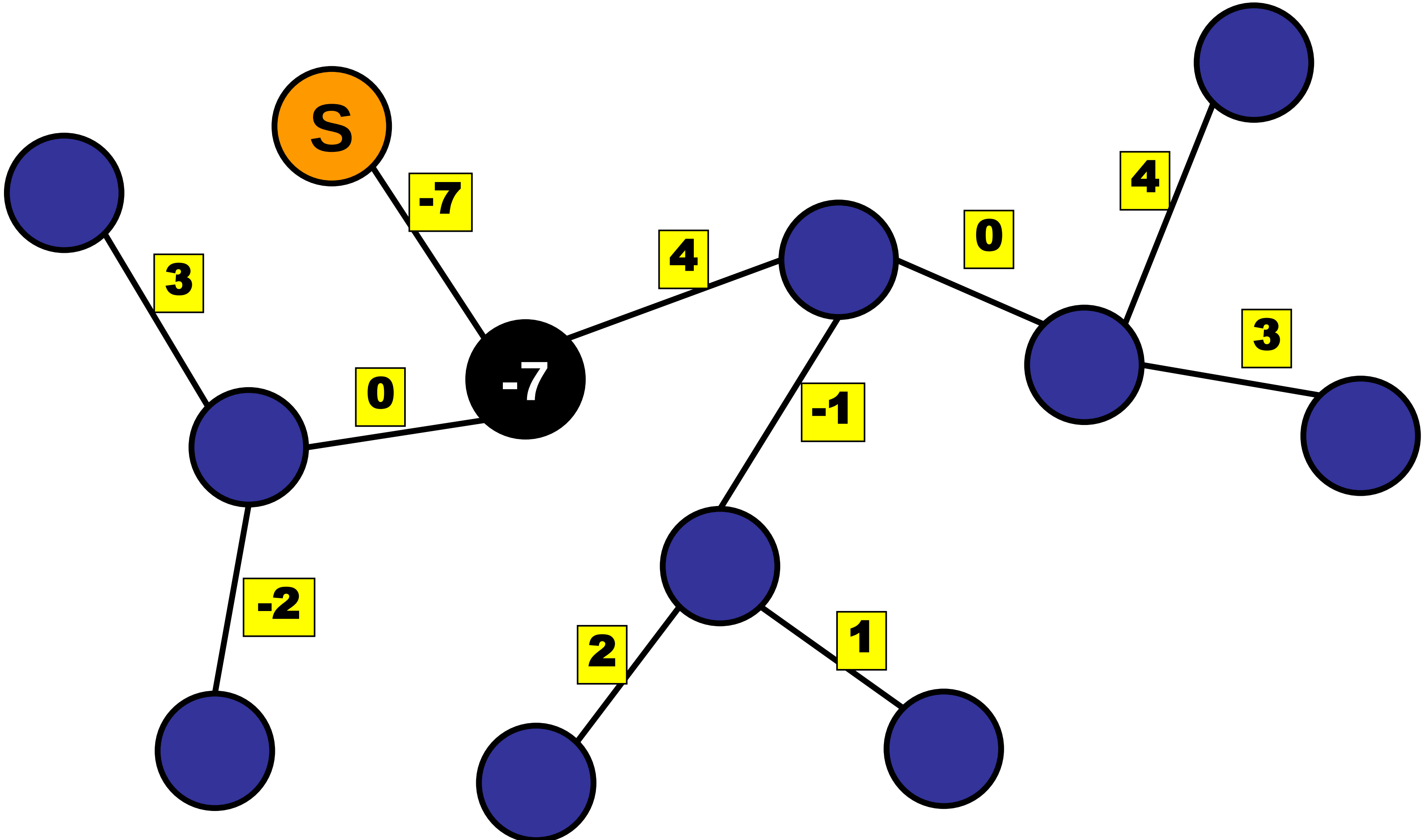
Special Case: Tree

Relax edges in (BFS or DFS order).



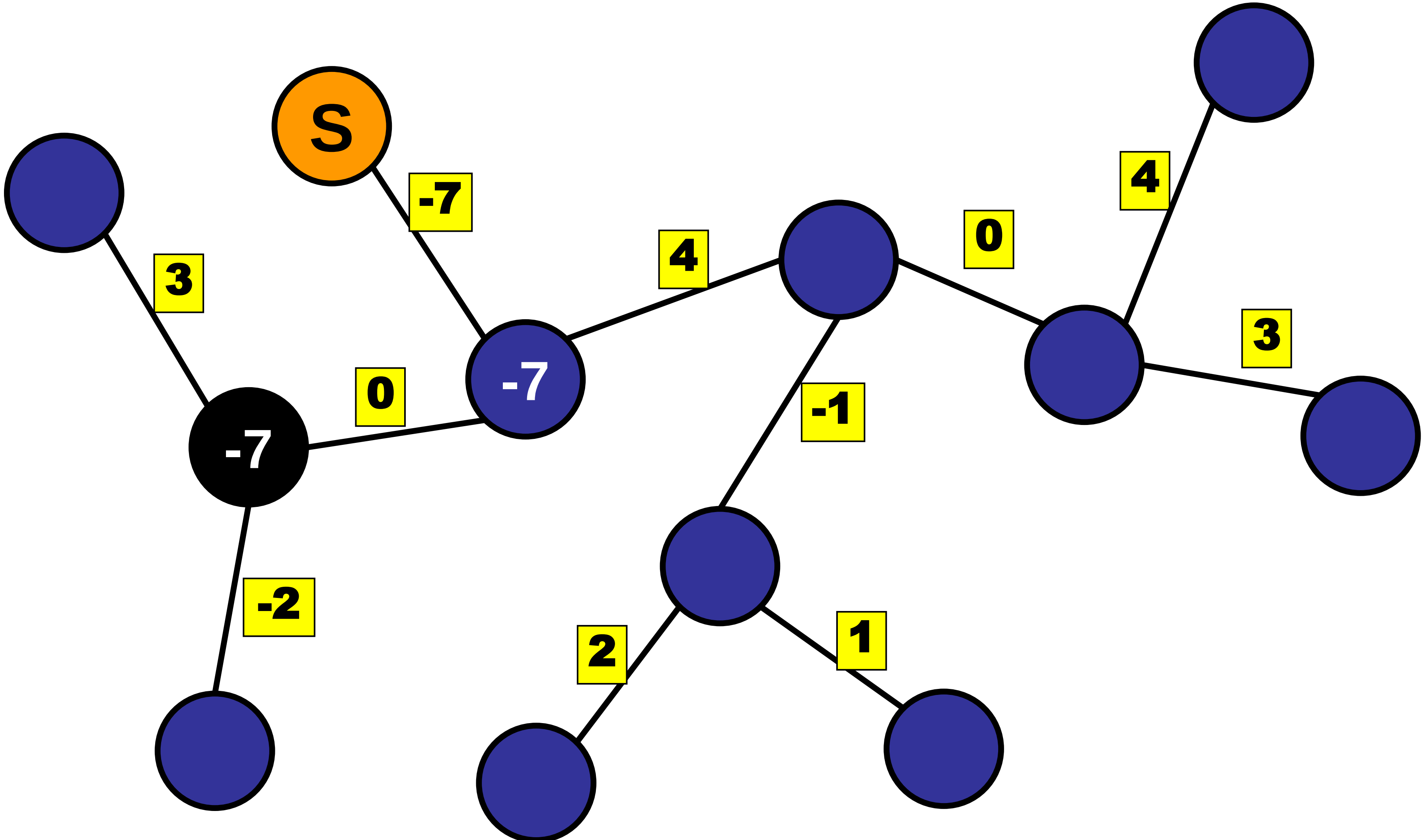
Special Case: Tree

Relax edges in DFS order.



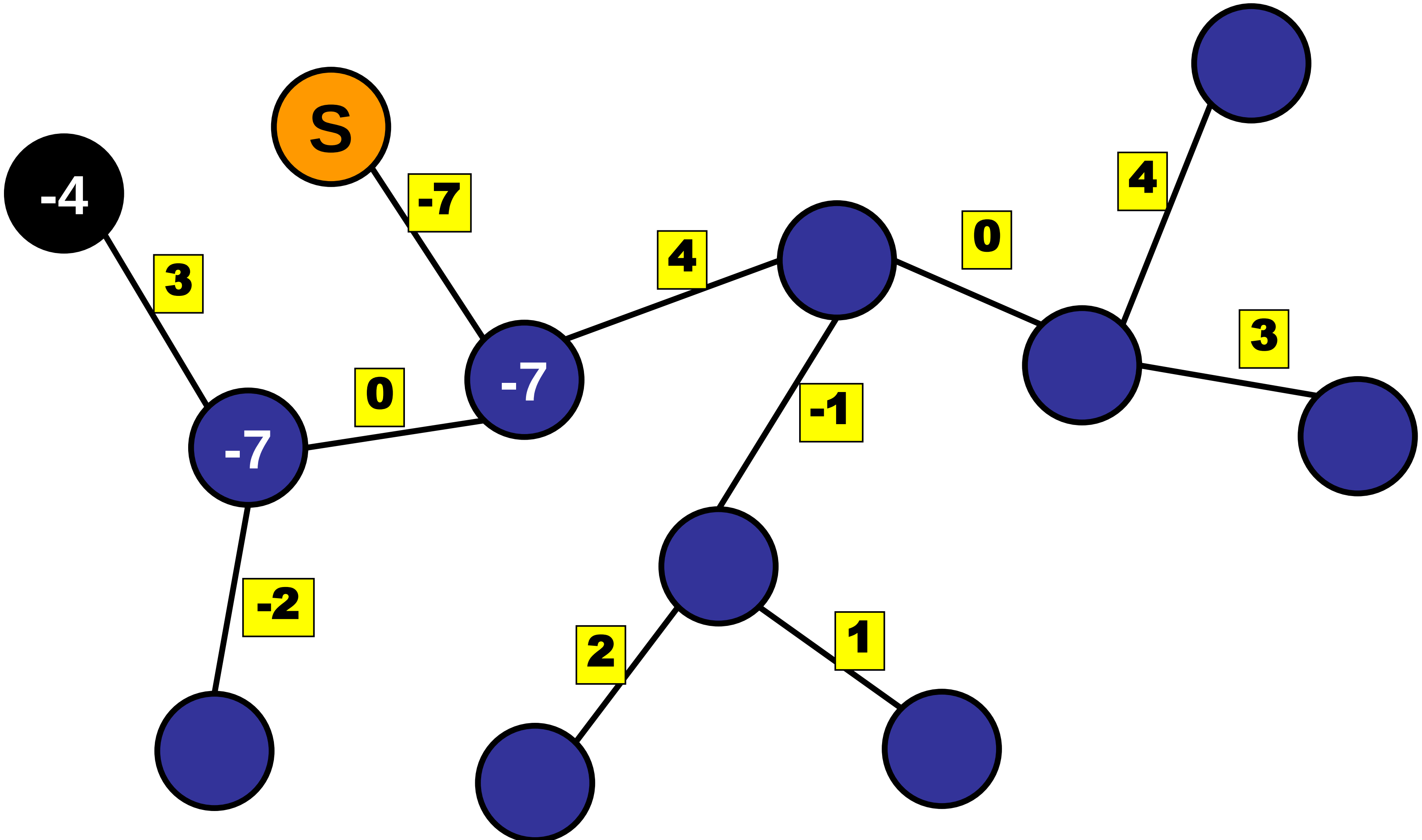
Special Case: Tree

Relax edges in DFS order.



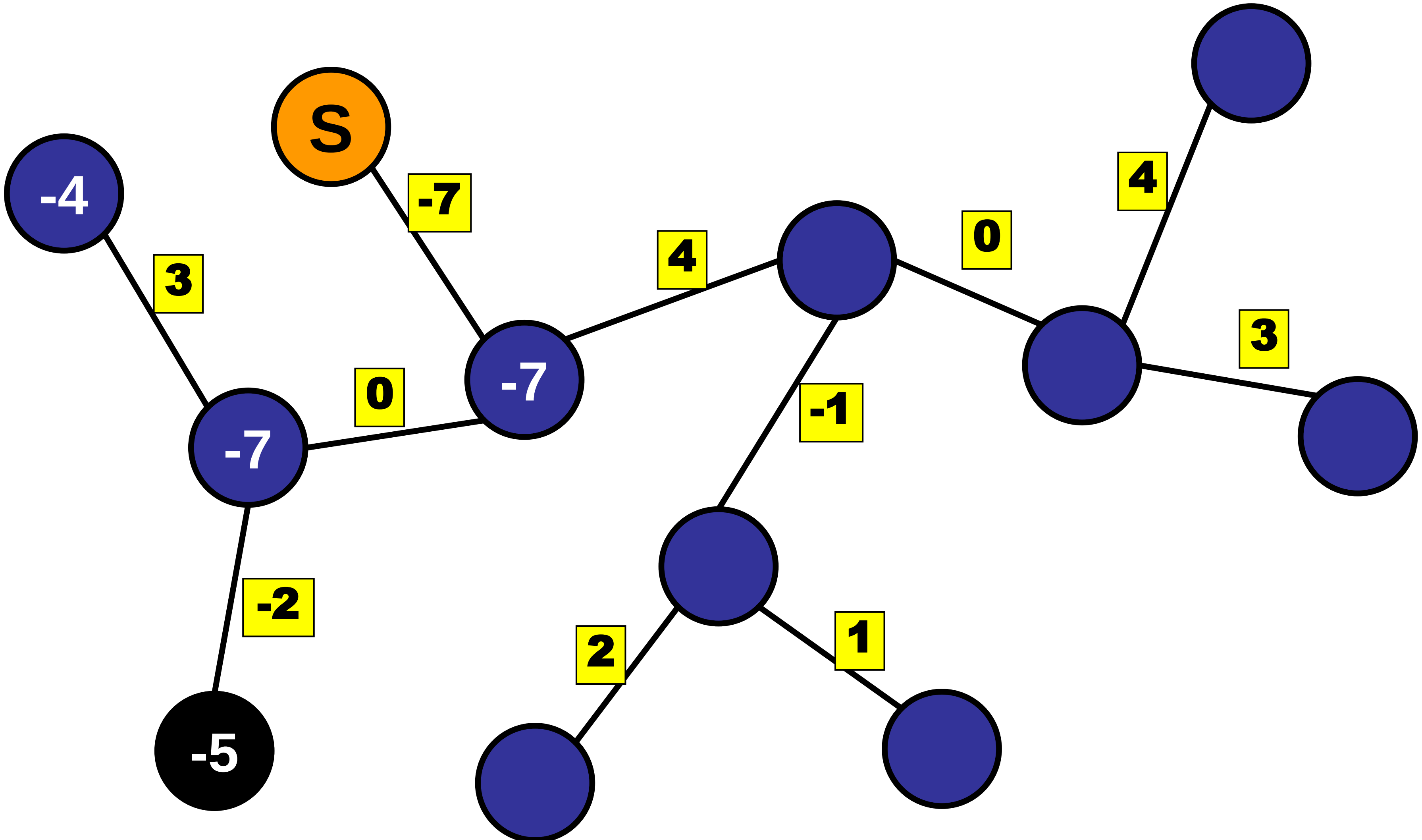
Special Case: Tree

Relax edges in DFS order.



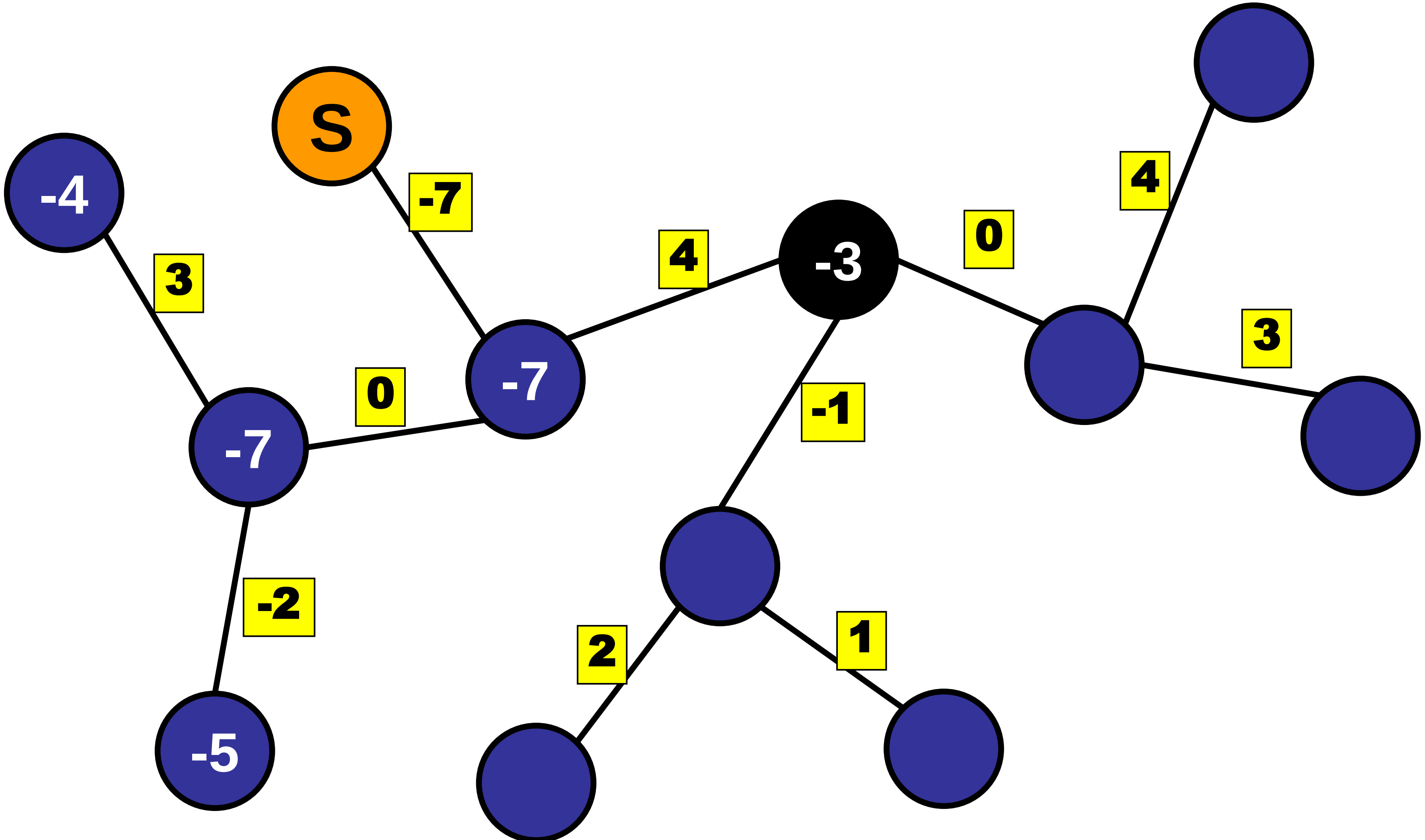
Special Case: Tree

Relax edges in DFS order.



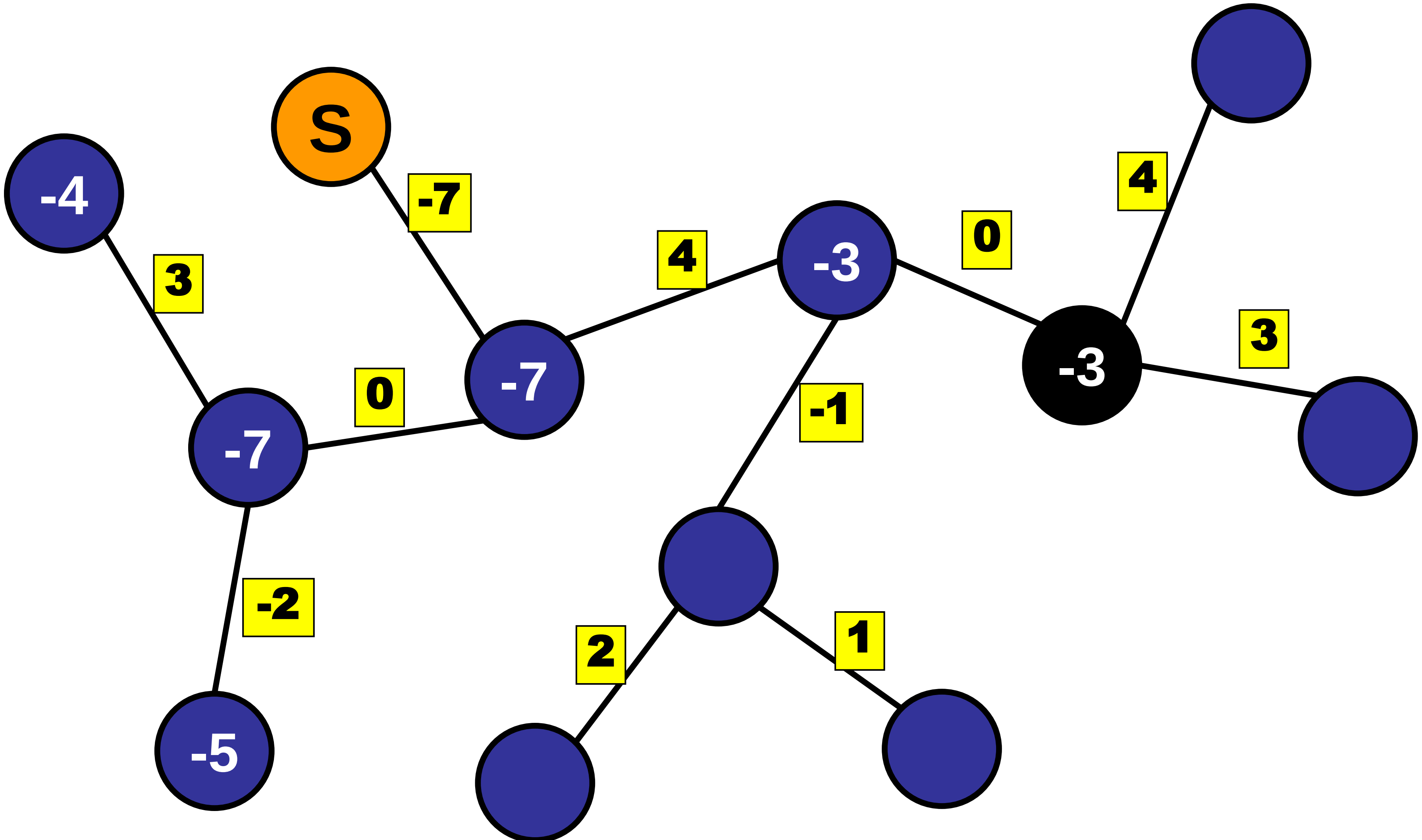
Special Case: Tree

Relax edges in DFS order.



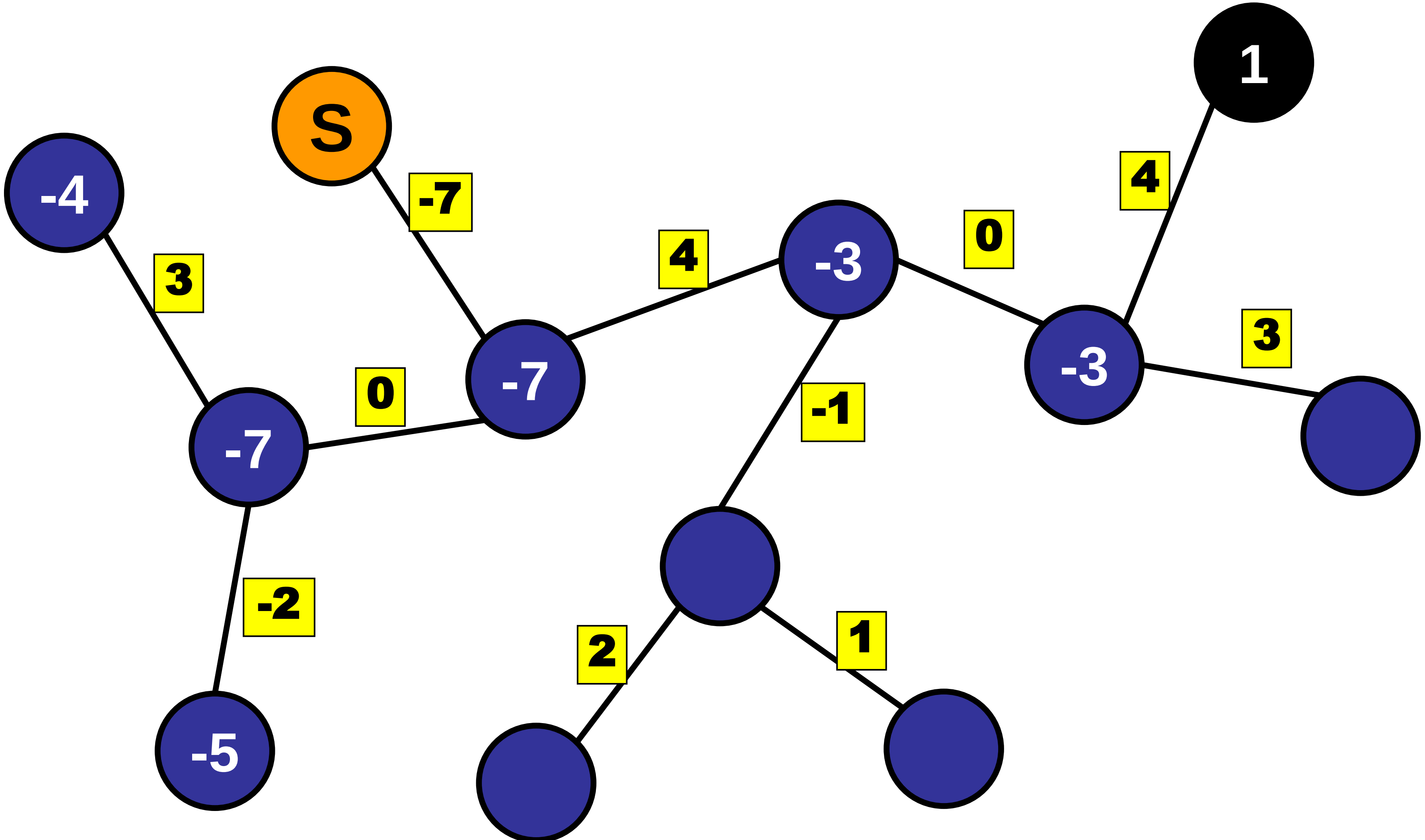
Special Case: Tree

Relax edges in DFS order.



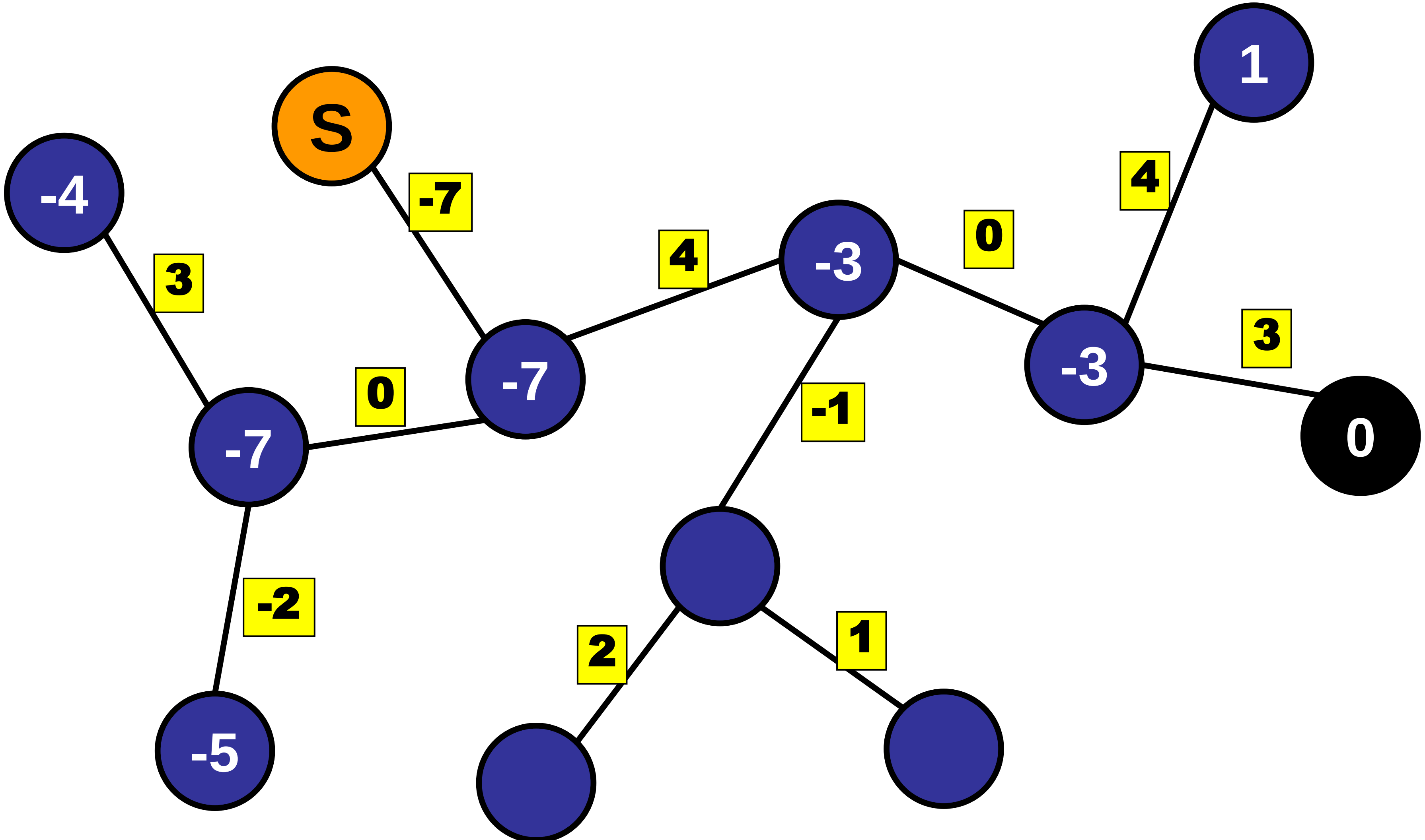
Special Case: Tree

Relax edges in DFS order.



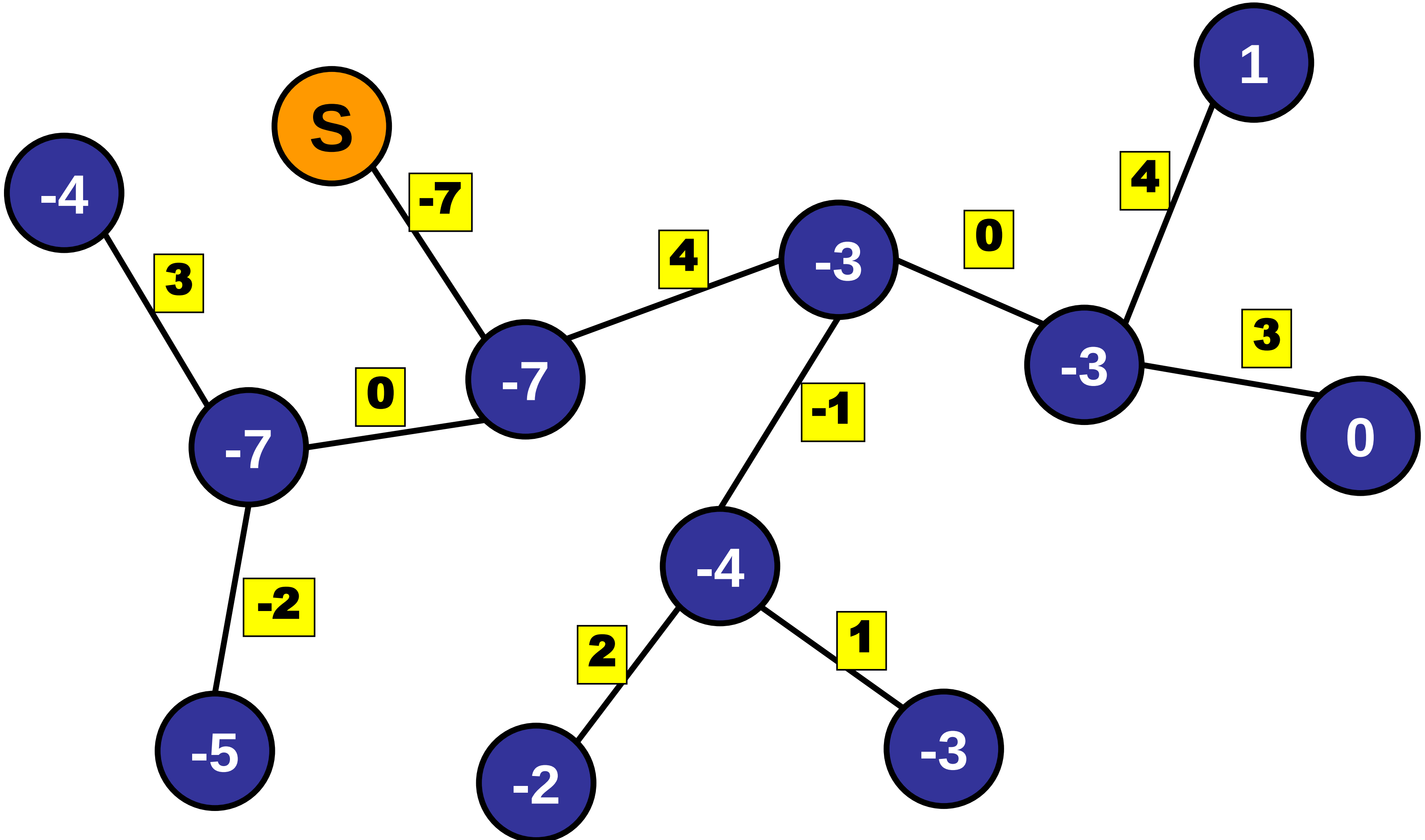
Special Case: Tree

Relax edges in DFS order.



Special Case: Tree

Relax edges in DFS order.



Special Cases

Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V+E)$
No Negative Weights	Dijkstra's Algorithm	
On Tree	BFS / DFS order	$O(V)$
On DAG		

```
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

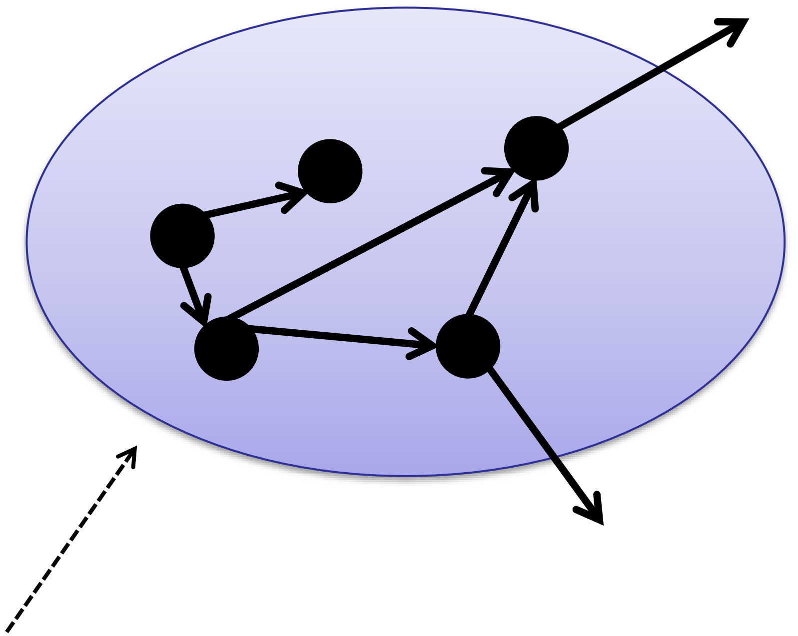
```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w,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

Dijkstra’s Algorithm

Why does it work?

Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.

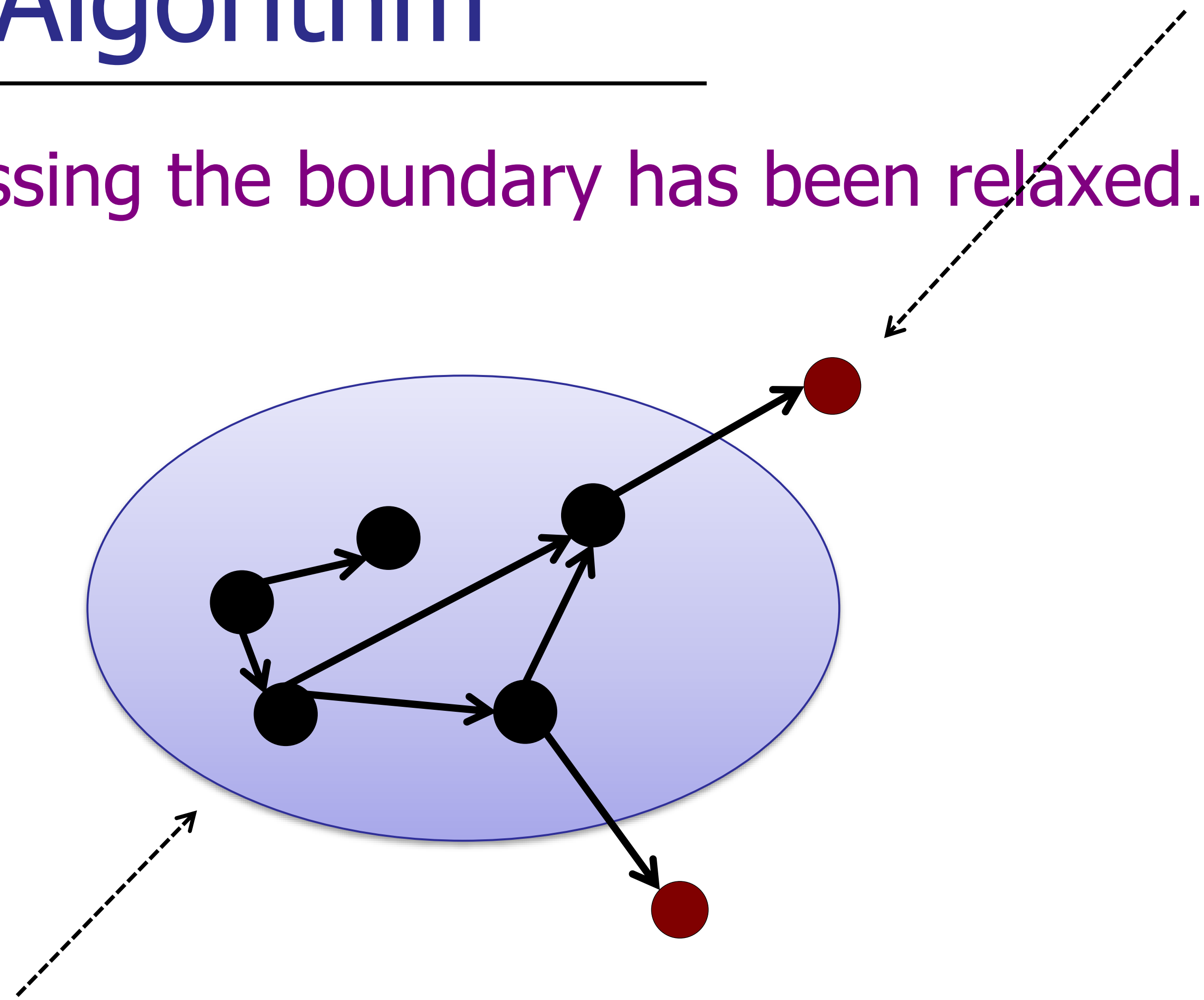


finished vertices:
distance is accurate.
Initially: just the source.

Dijkstra's Algorithm

Every edge crossing the boundary has been relaxed.

fringe vertices: neighbor of a finished vertex.

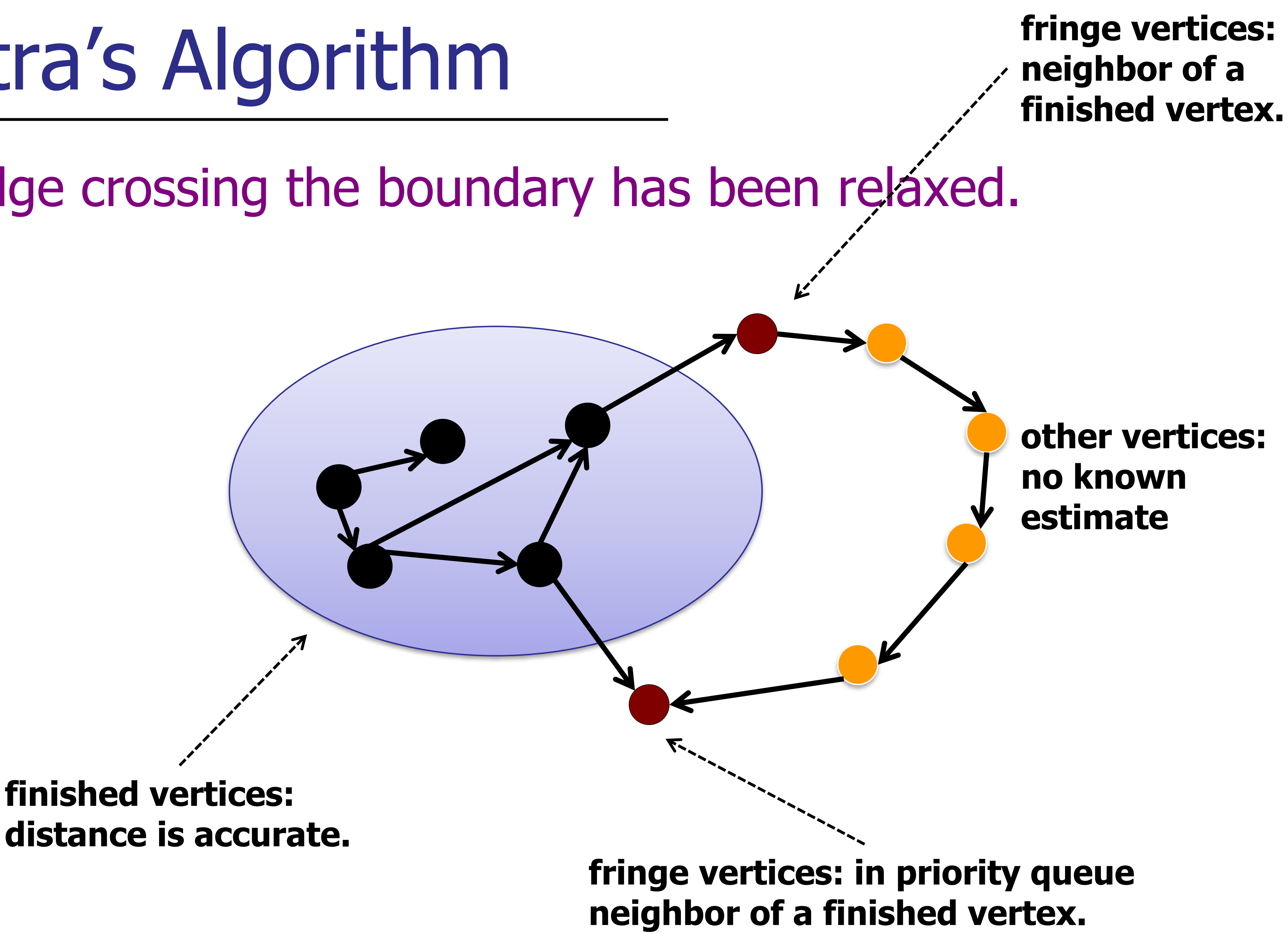


finished vertices: distance is accurate.

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

Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.



Dijkstra’s Algorithm

Proof by induction:

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

Dijkstra’s Algorithm

Proof by induction:

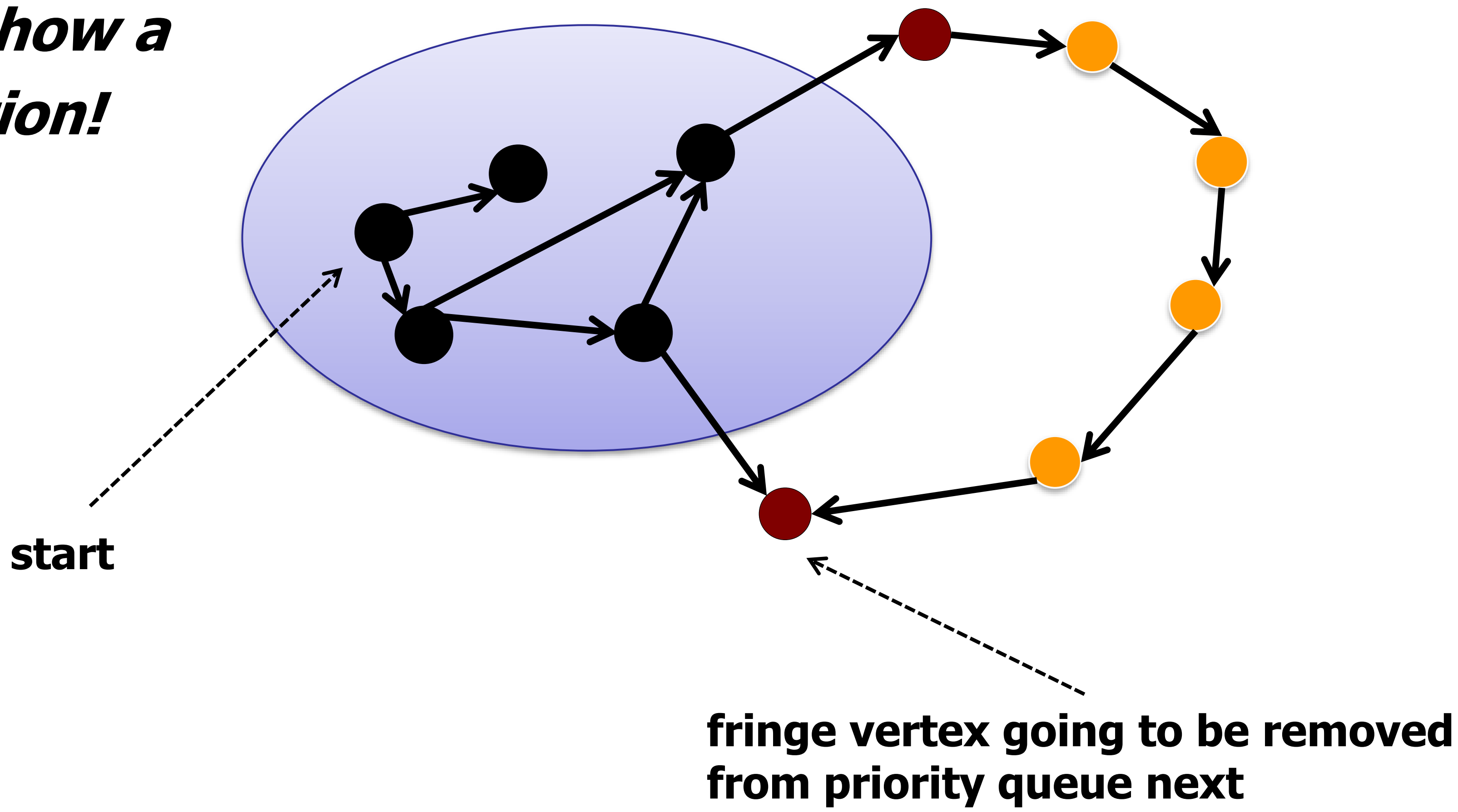
- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

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

Dijkstra's Algorithm

Assume not: *fringe vertex is removed but not finished*

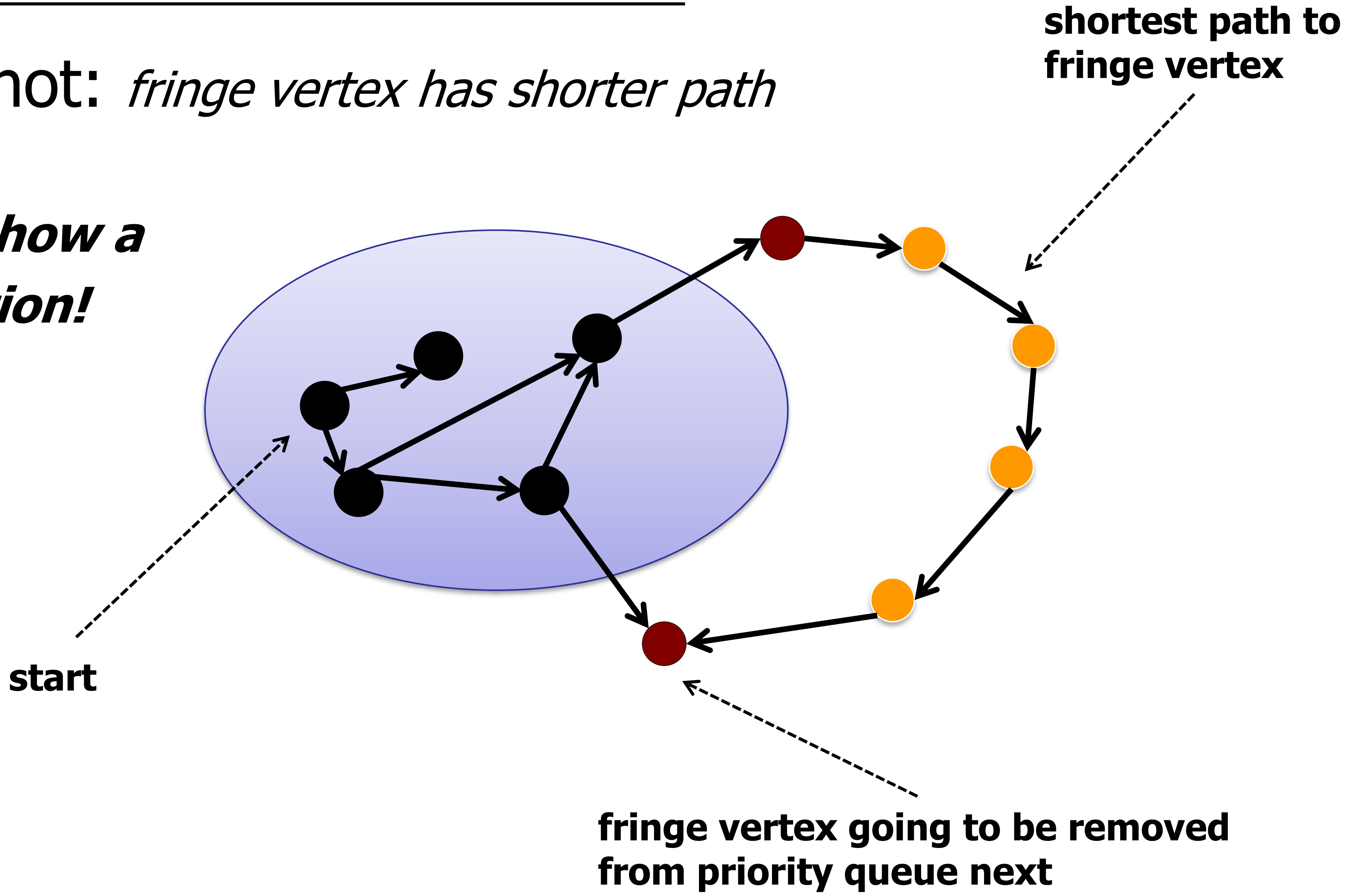
Going to show a contradiction!



Dijkstra's Algorithm

Assume not: *fringe vertex has shorter path*

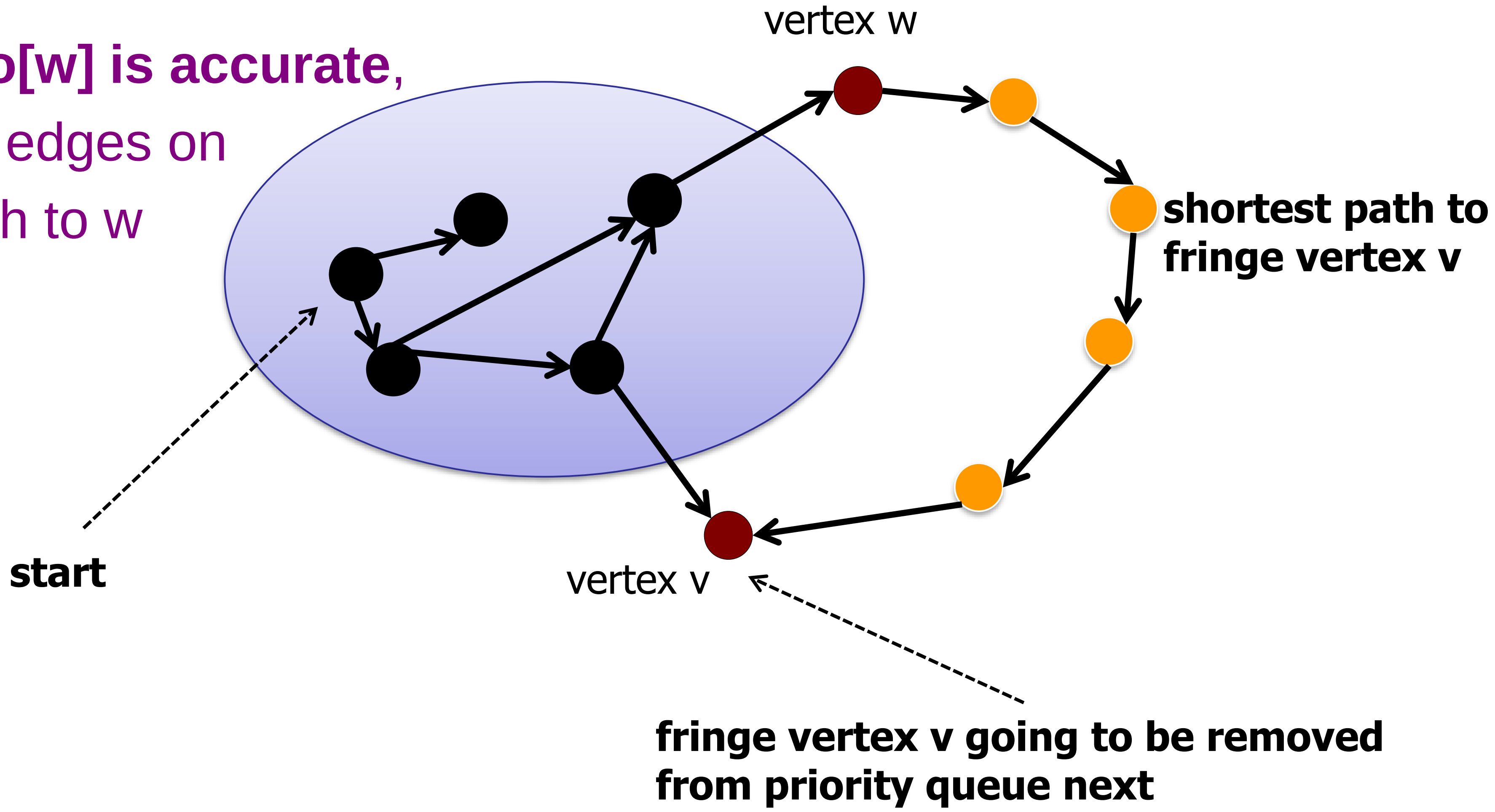
Going to show a contradiction!



Dijkstra's Algorithm

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

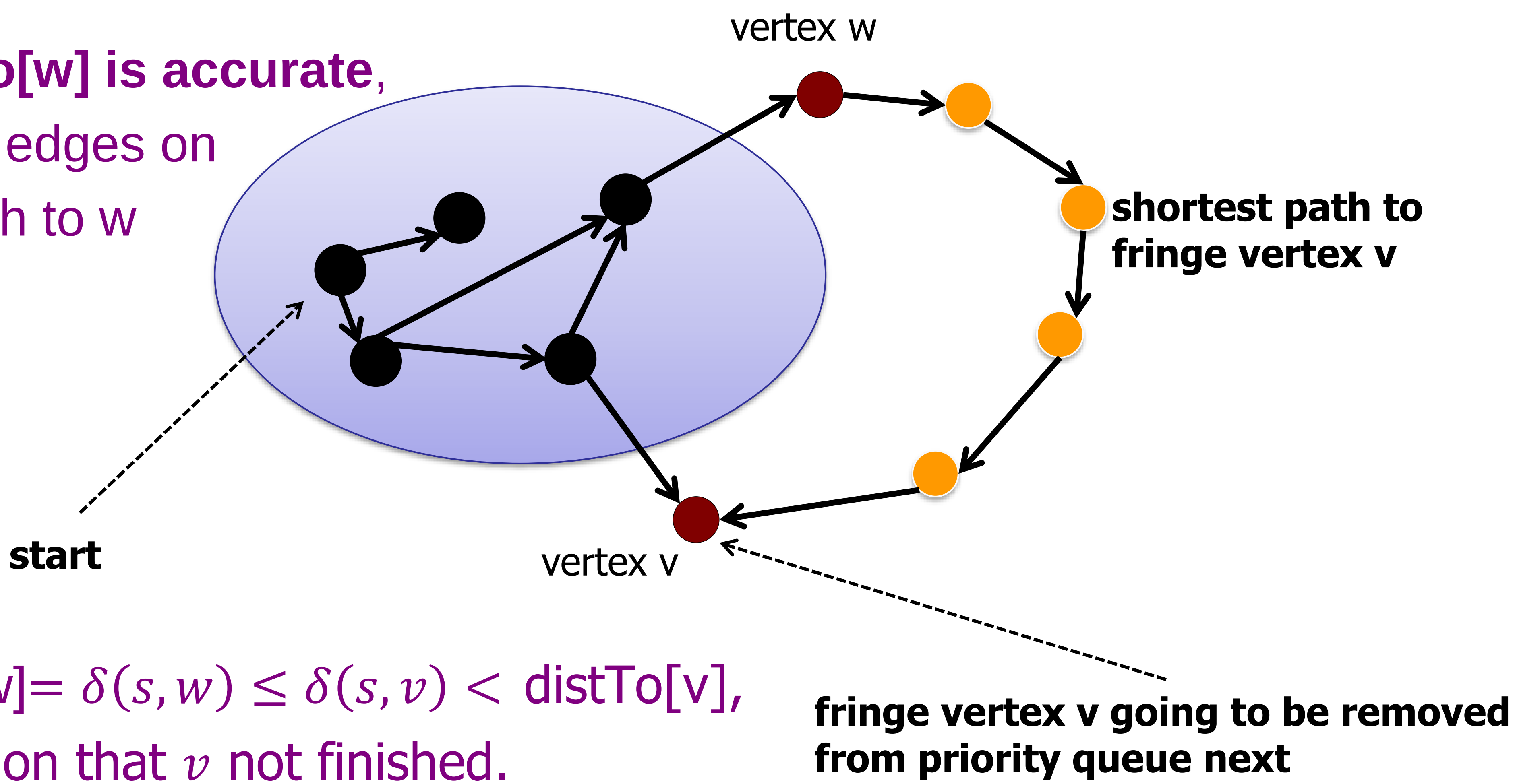
Then **distTo[w] is accurate**, because all edges on shortest path to w relaxed.



Dijkstra's Algorithm

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

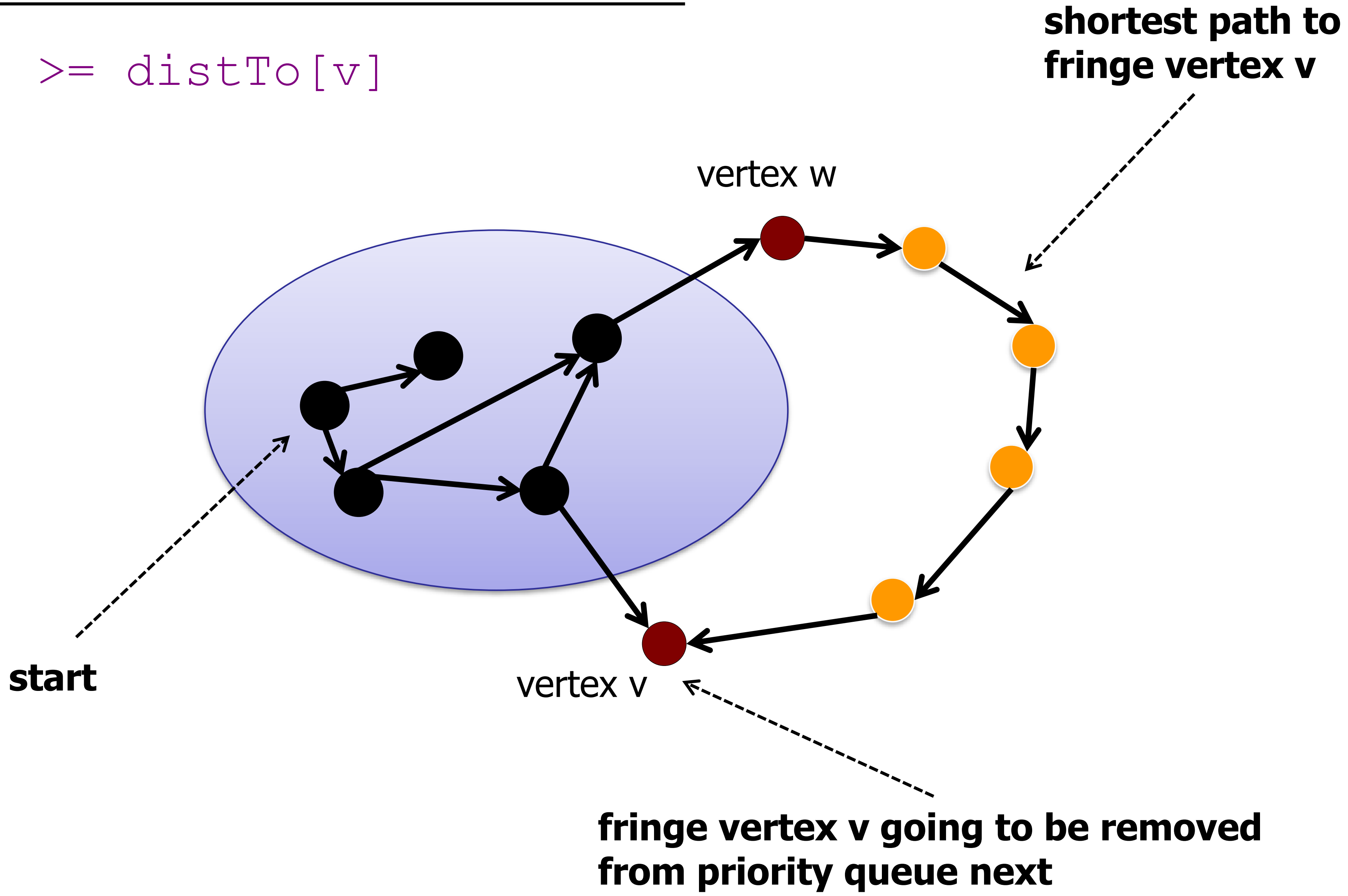
Then **distTo[w] is accurate**, because all edges on shortest path to w relaxed.



So, distTo[w]= $\delta(s, w) \leq \delta(s, v) <$ distTo[v], by assumption that v not finished.

Dijkstra’s Algorithm

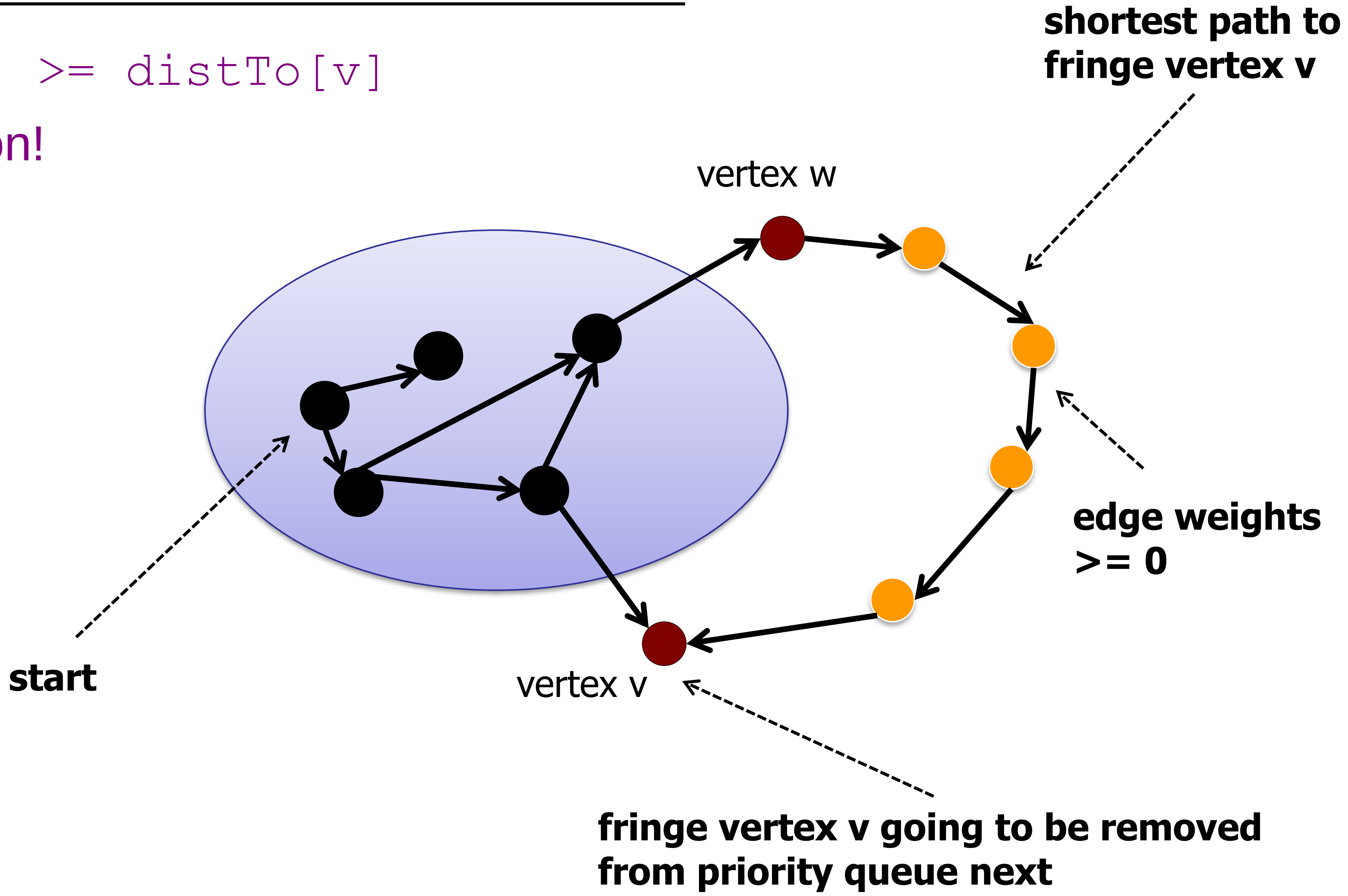
`distTo[w] >= distTo[v]`



Dijkstra's Algorithm

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

Contradiction!



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]);
    }
}
```

Extending a path does not make it shorter!

Roadmap

Directed Graphs

- Directed acyclic graphs
- Topological Sort
- Connected Components

What is a directed graph?

Graph consists of two types of elements:

Nodes (or vertices)

- At least one.

Edges (or arcs)

- Each edge connects two nodes in the graph
- Each edge is unique.
- Each edge is **directed**.

What is a directed graph?

Graph $G = \langle V, E \rangle$

- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

Order matters!

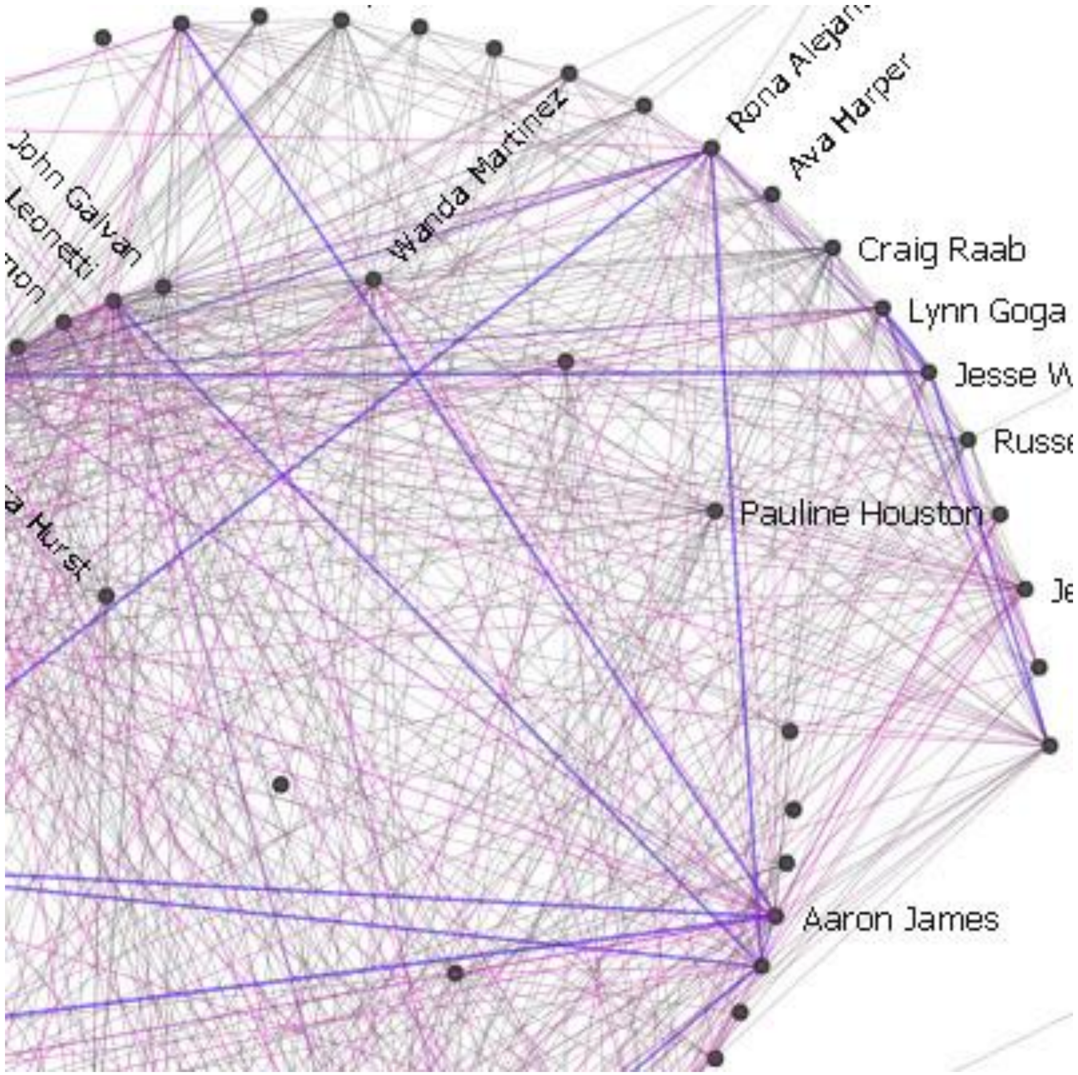
Directed Graphs

Is friendship always bidirectional?:

- Nodes are people
- Edge = friendship

Facebook: yes

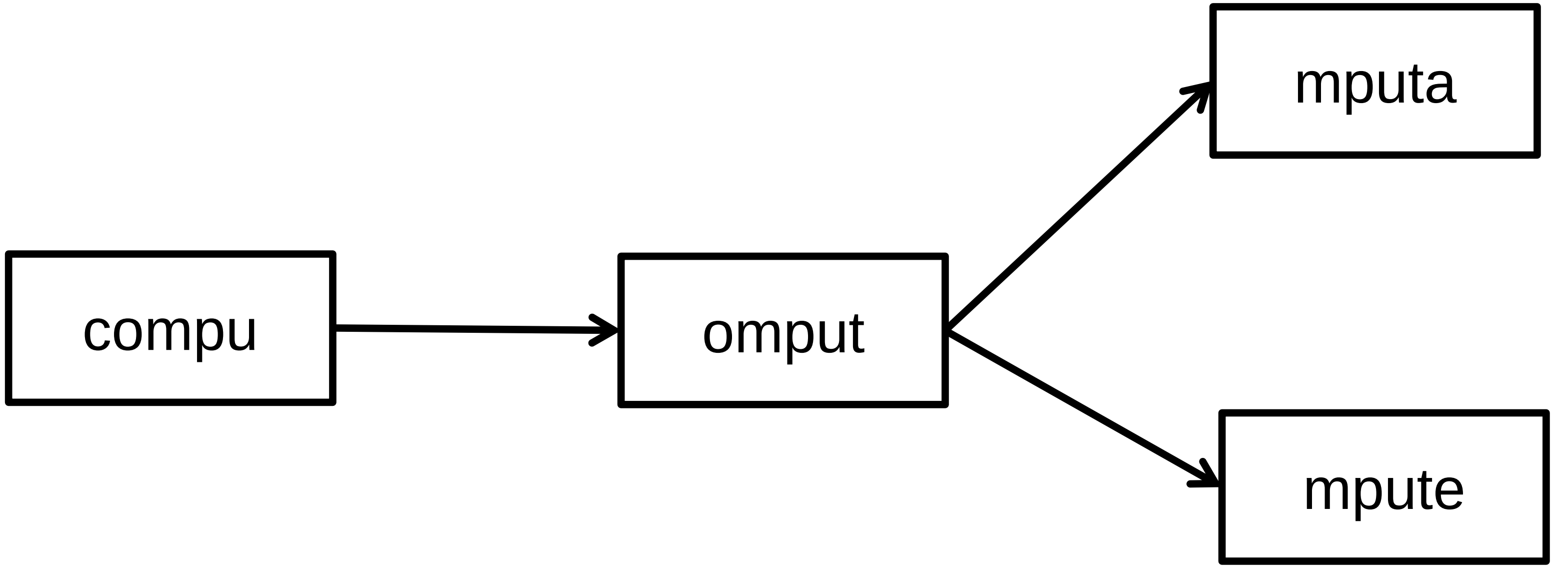
Twitter: no



Directed Graphs

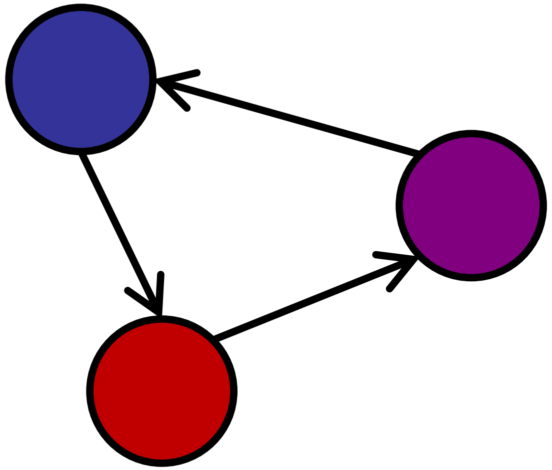
Markov text generation:

- Nodes are kgrams
- Edge = one kgram follows another

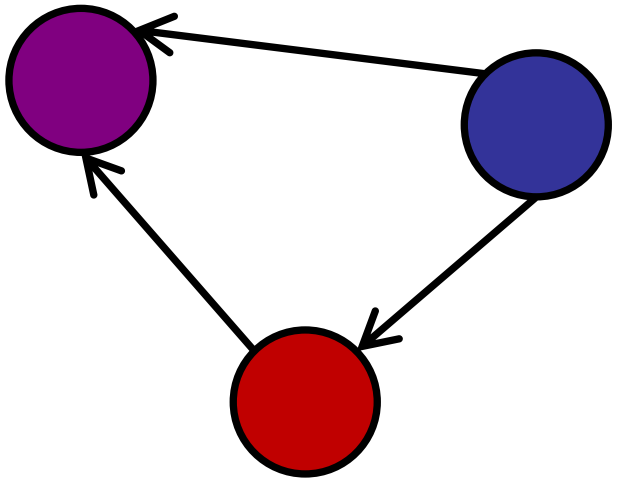


Directed Acyclic Graphs

Cyclic

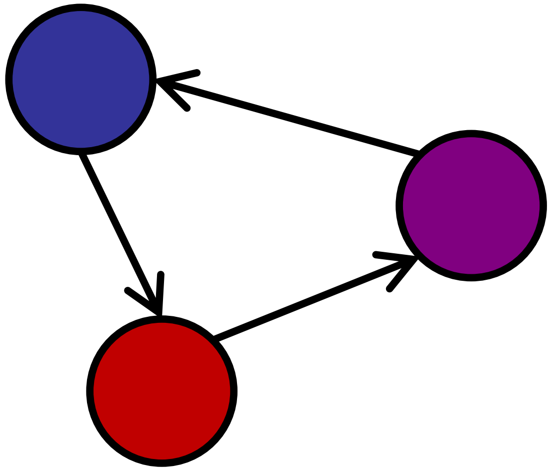


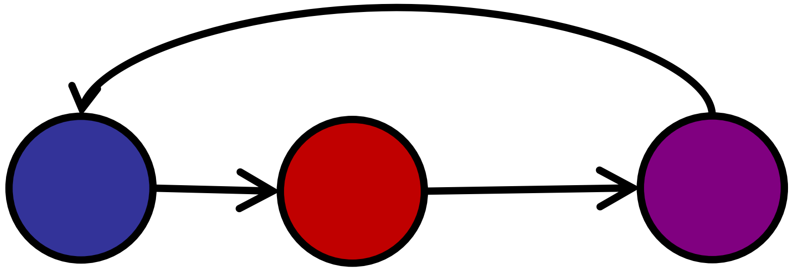
Acyclic



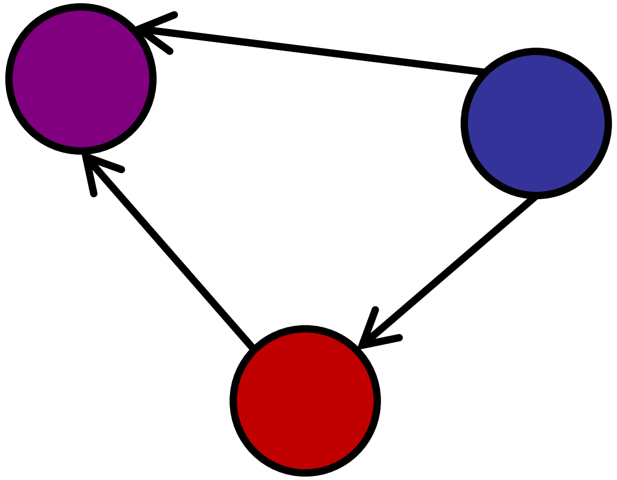
Directed Acyclic Graphs

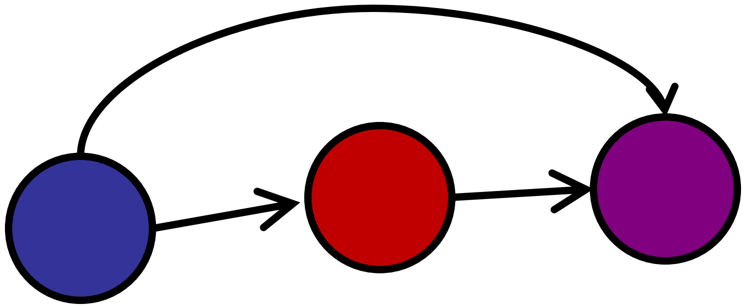
Cyclic





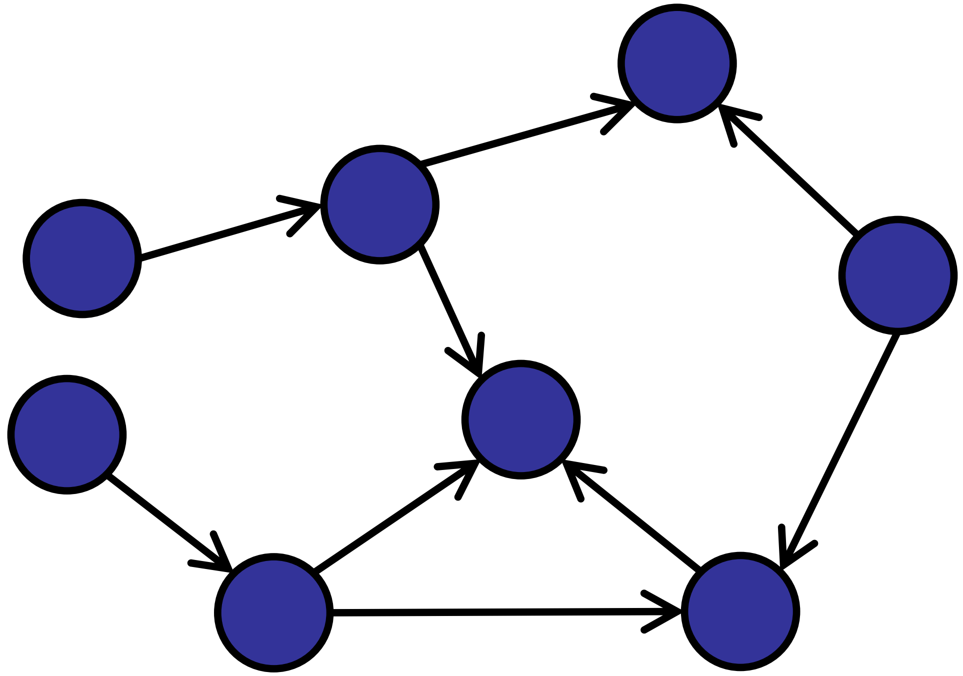
Acyclic





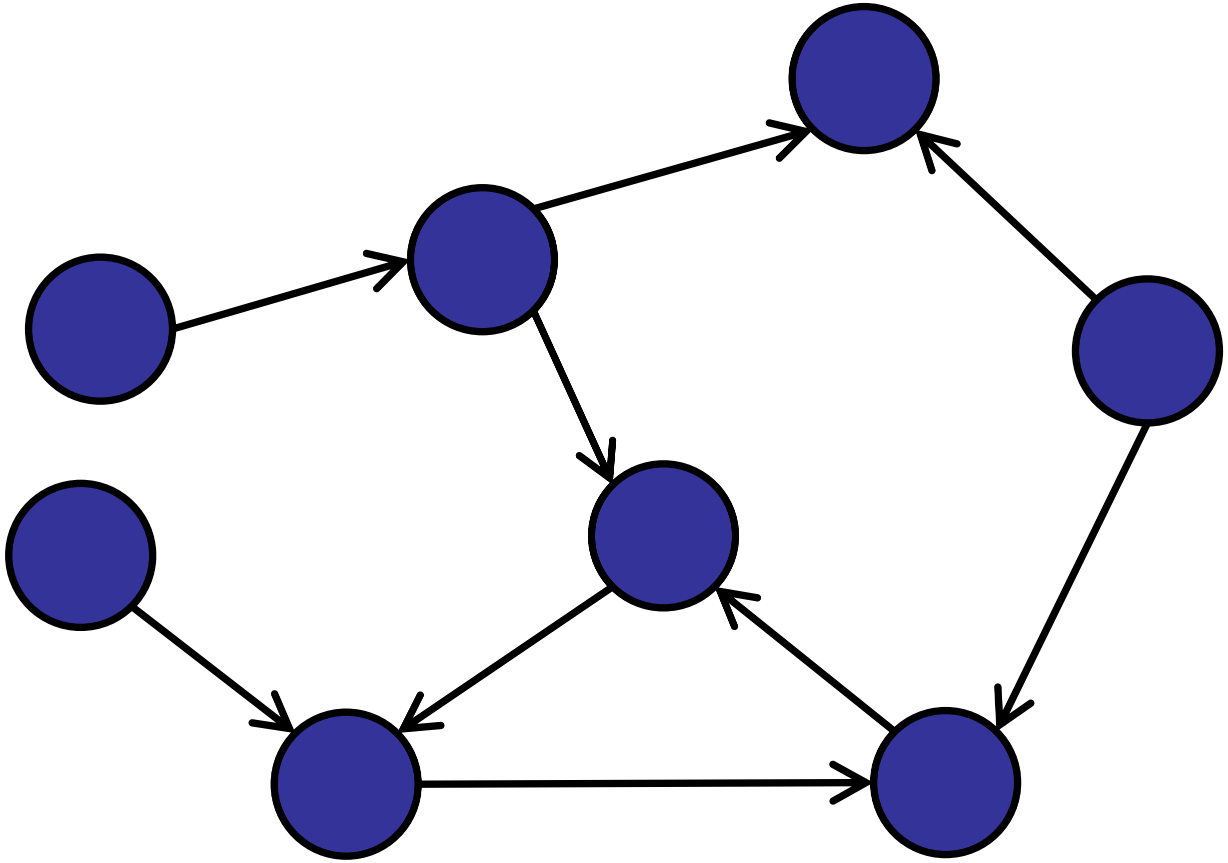
Is this graph:

1. Cyclic
2. Acyclic
3. Transcendental



Directed Acyclic Graphs

Cyclic or Acyclic?



Scheduling

Set of tasks for baking cookies:

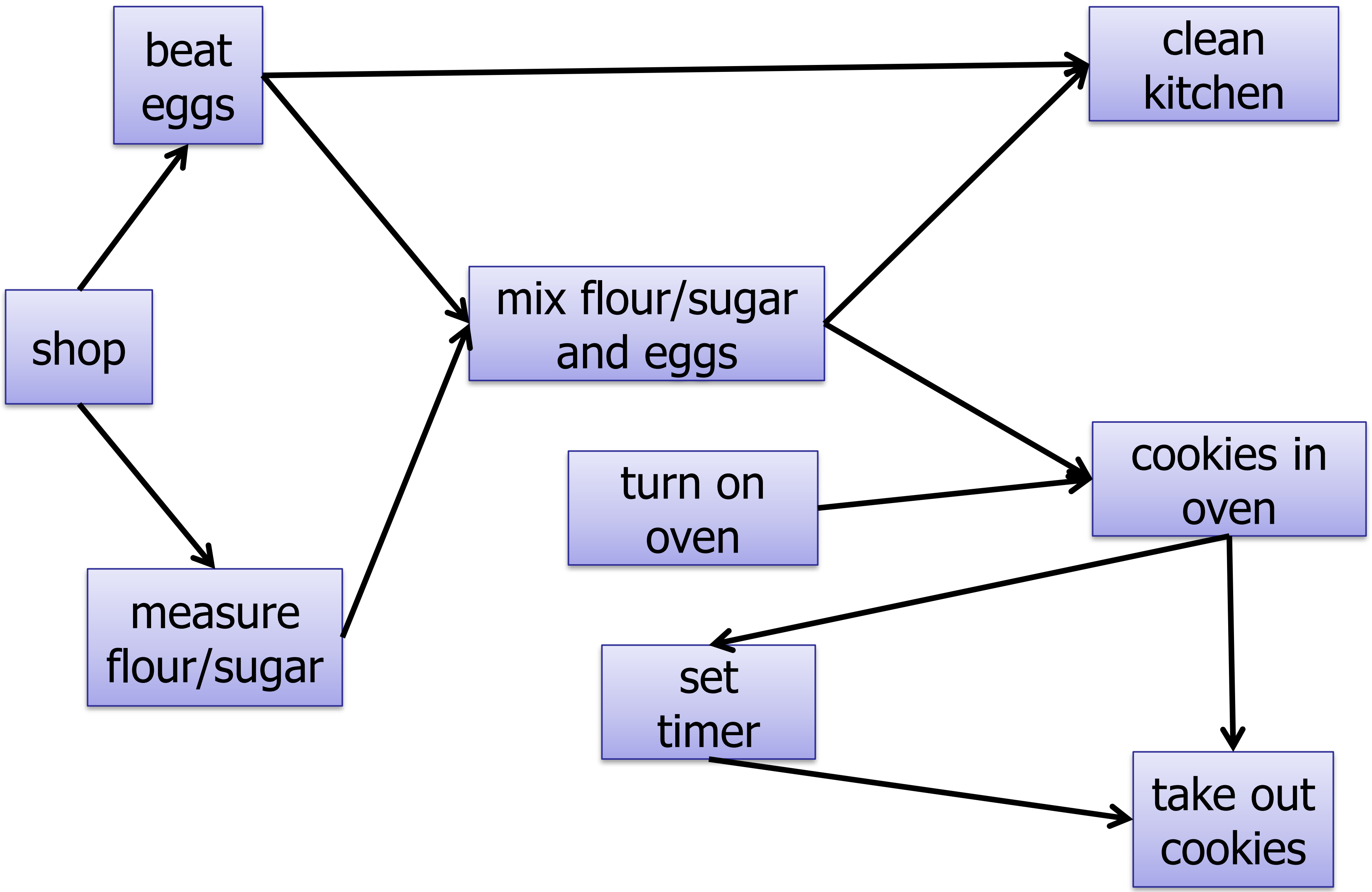
- Shop for groceries
- Put the cookies in the oven
- Clean the kitchen
- Beat the eggs in a bowl
- Measure the flour and sugar in a bowl
- Mix the eggs with the flour and sugar
- Turn on the oven
- Set the timer
- Take out the cookies

Scheduling

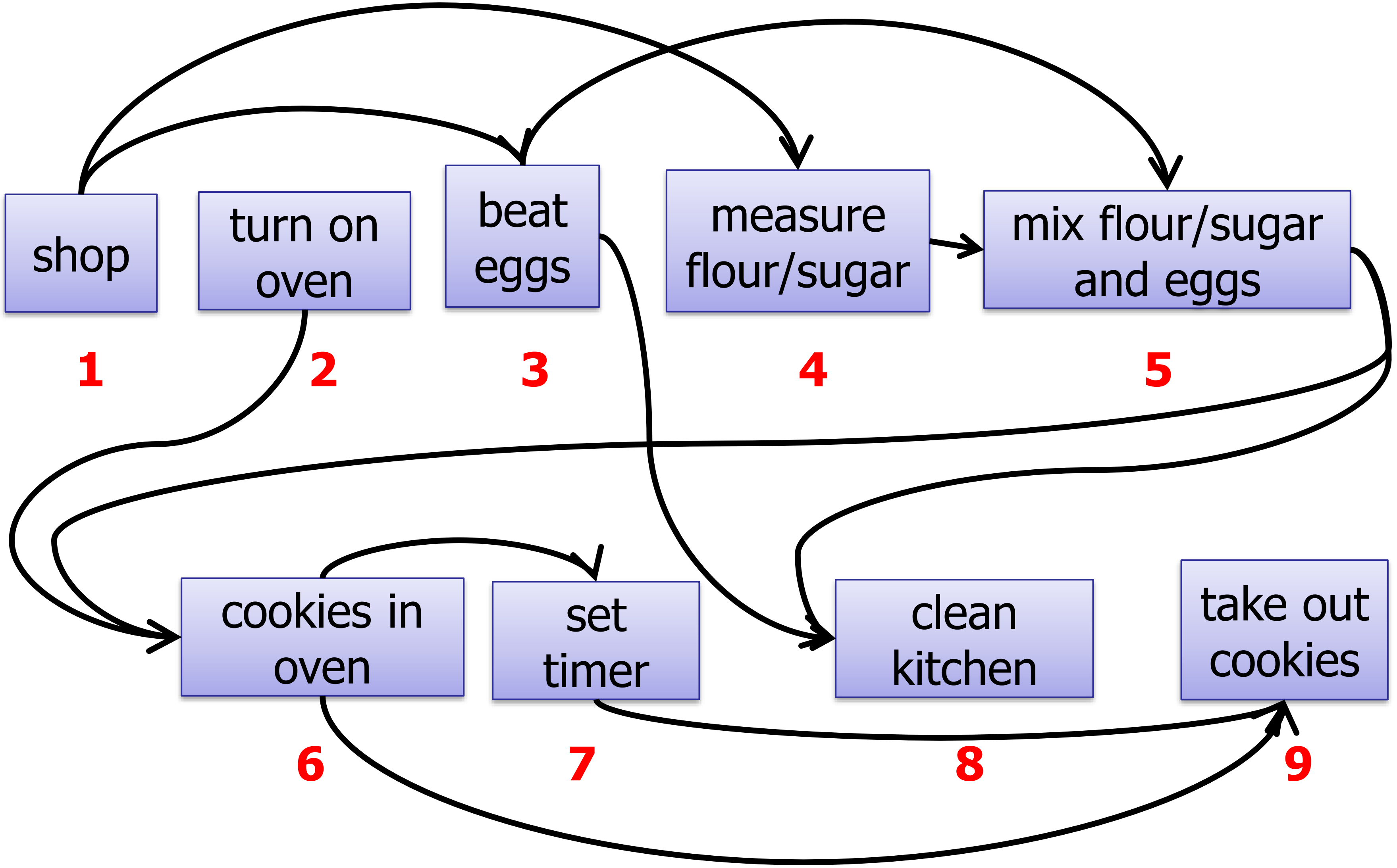
Ordering:

- Shop for groceries before beat the eggs
- Shop for groceries before measure the flour
- Turn on the oven before put the cookies in the oven
- Beat the eggs before mix the eggs with the flour
- Measure the flour before mix the eggs with the flour
- Put the cookies in the oven before set the timer
- Measure the flour before clean the kitchen
- Beat the eggs before clean the kitchen
- Mix the flour and the eggs before clean the kitchen

Scheduling


beat eggs
clean kitchen
shop
mix flour/sugar and eggs
turn on oven
cookies in oven
measure flour/sugar
set timer
take out cookies

Topological Ordering



Topological Order

Properties:

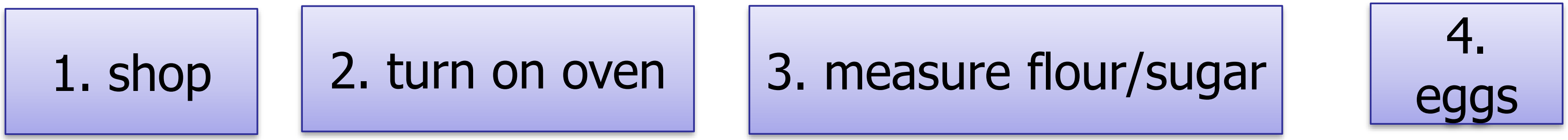
1. Sequential total ordering of all nodes

1. shop | 2. turn on oven | 3. measure flour/sugar | 4. eggs

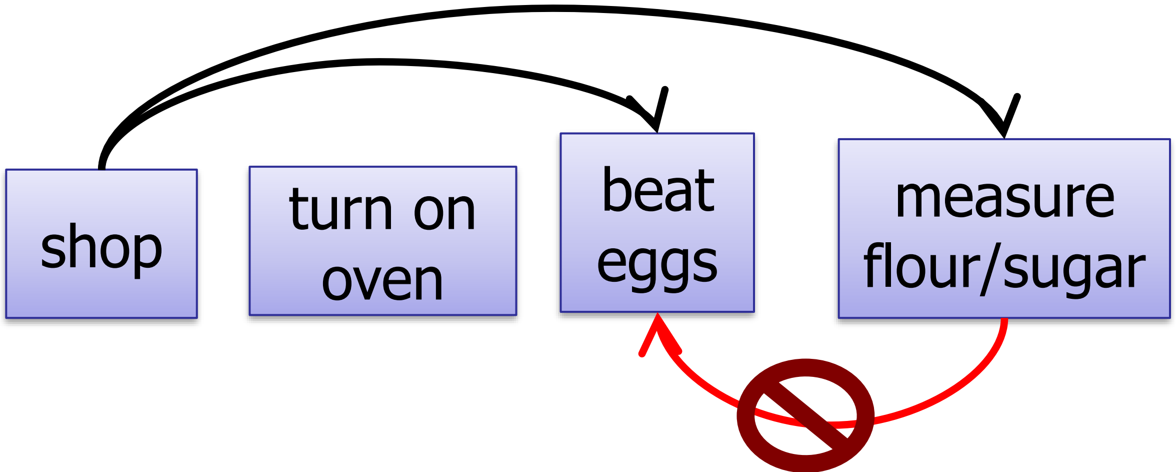
Topological Order

Properties:

1. Sequential total ordering of all nodes



2. Edges only point forward

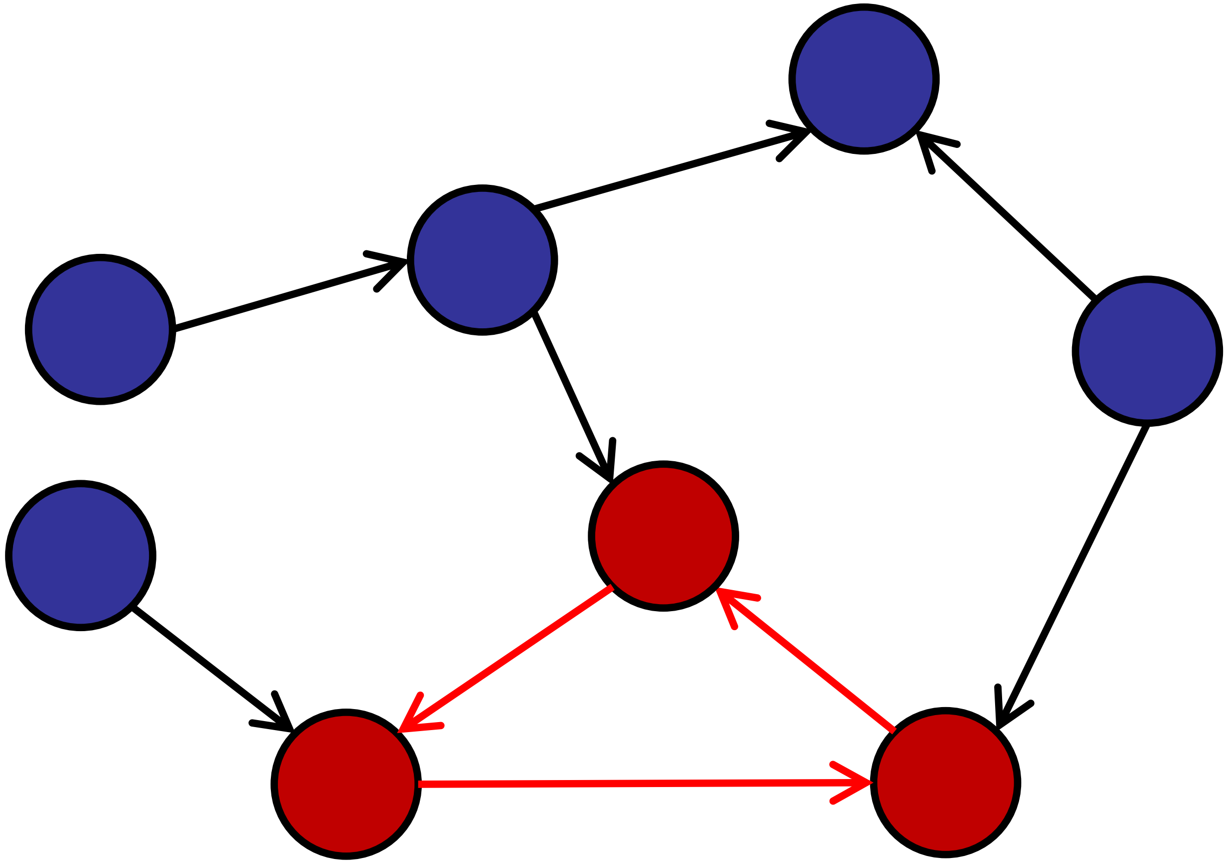


Does every directed graph have a topological ordering?

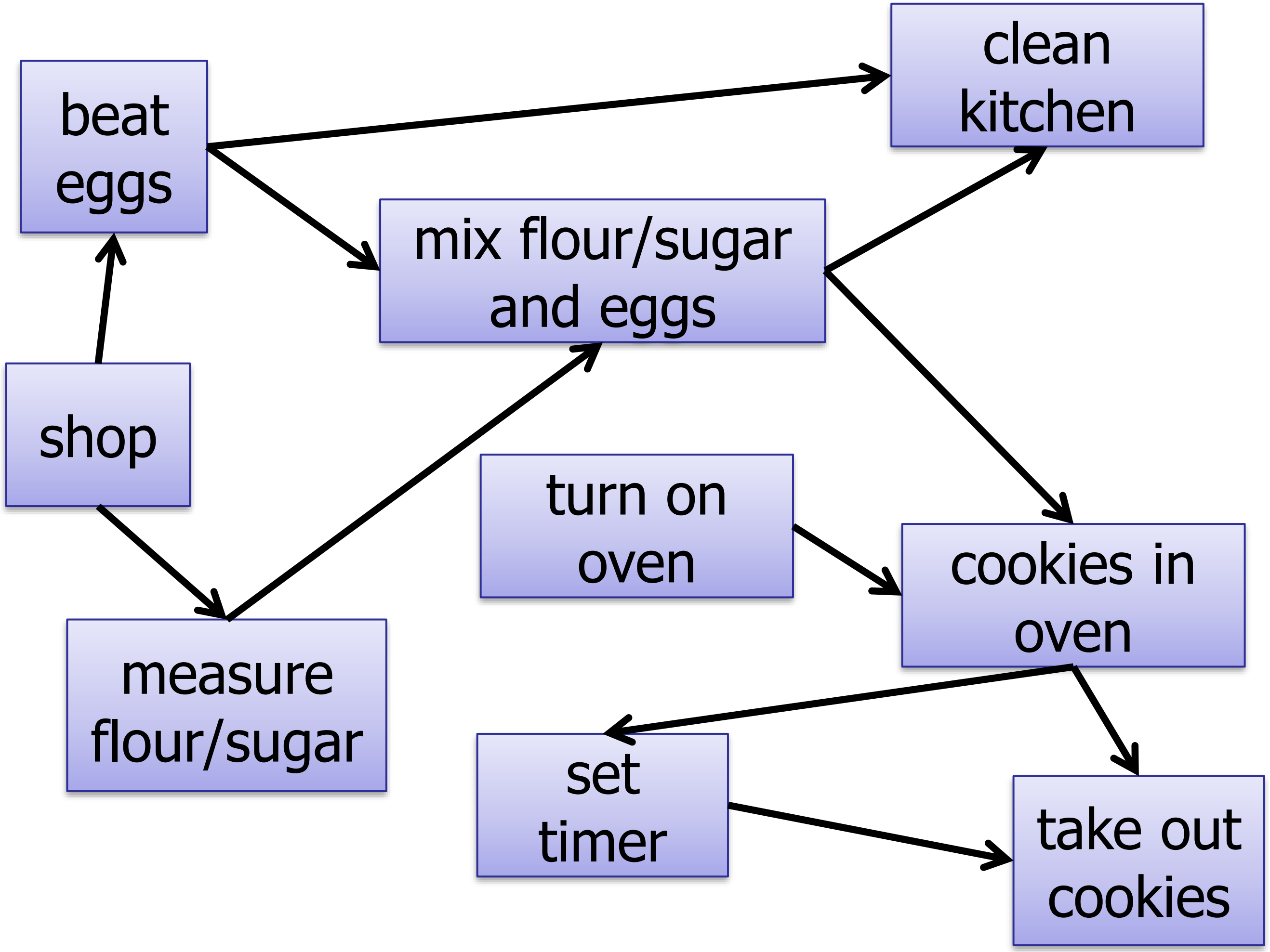
1. Yes
2. ✓ No
3. Only if the adjacency matrix has small second eigenvalue.

Directed Acyclic Graphs

Does it have a topological ordering?



Directed Acyclic Graph



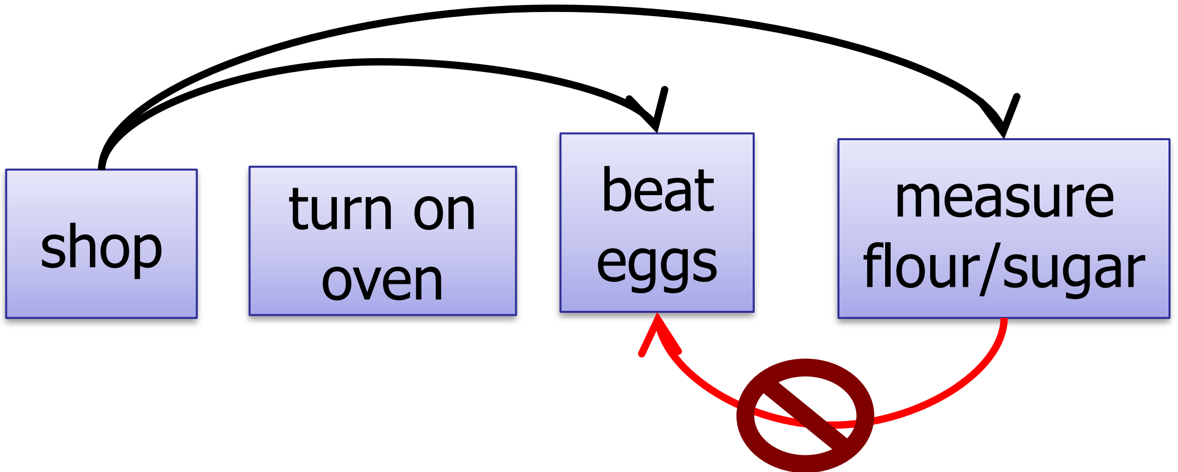
Topological Order

Properties:

1. Sequential total ordering of all nodes

1. shop | 2. turn on oven | 3. measure flour/sugar | 4. eggs

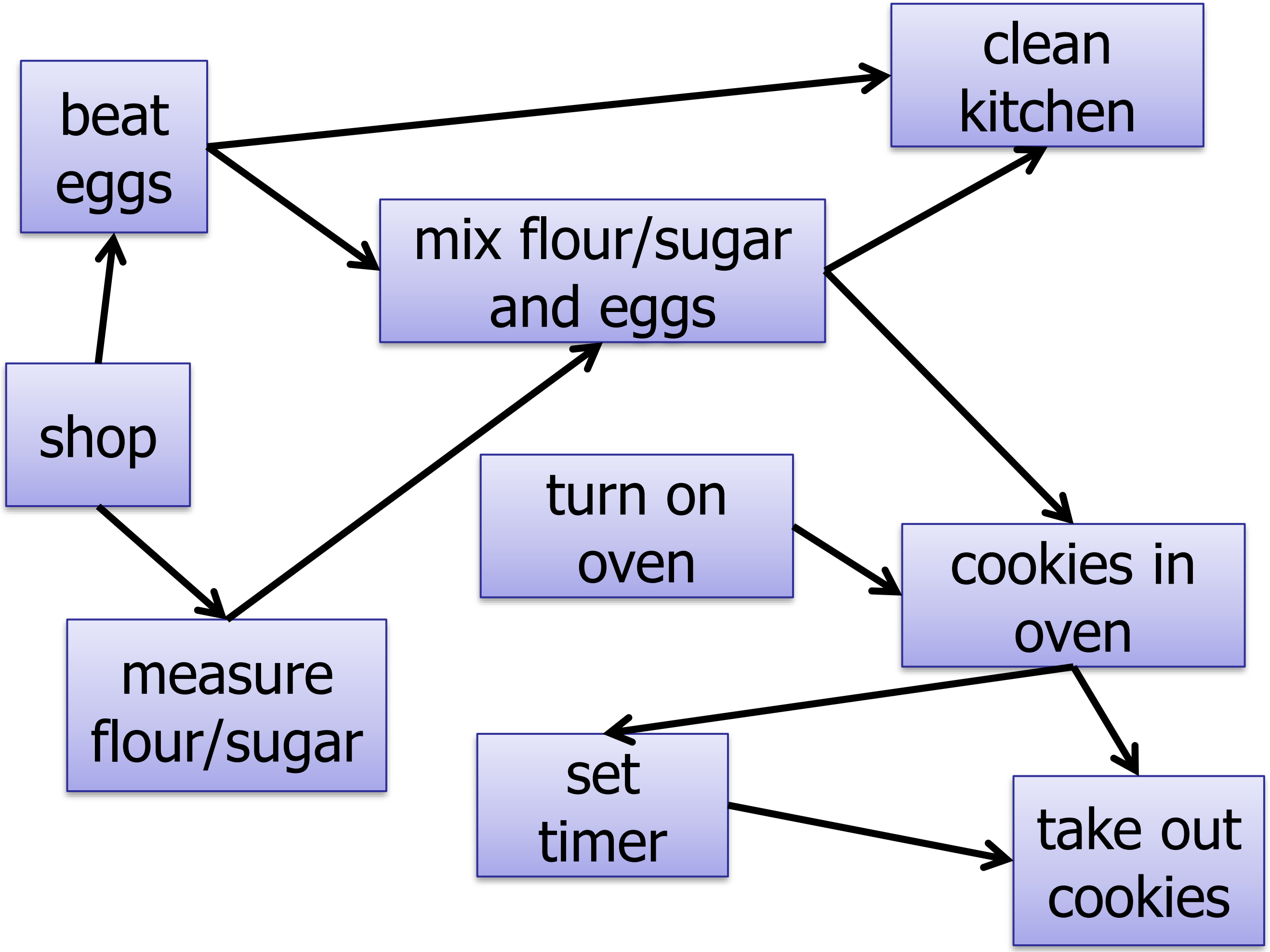
2. Edges only point forward



Which algorithm is best for finding a Topological Ordering in a DAG?

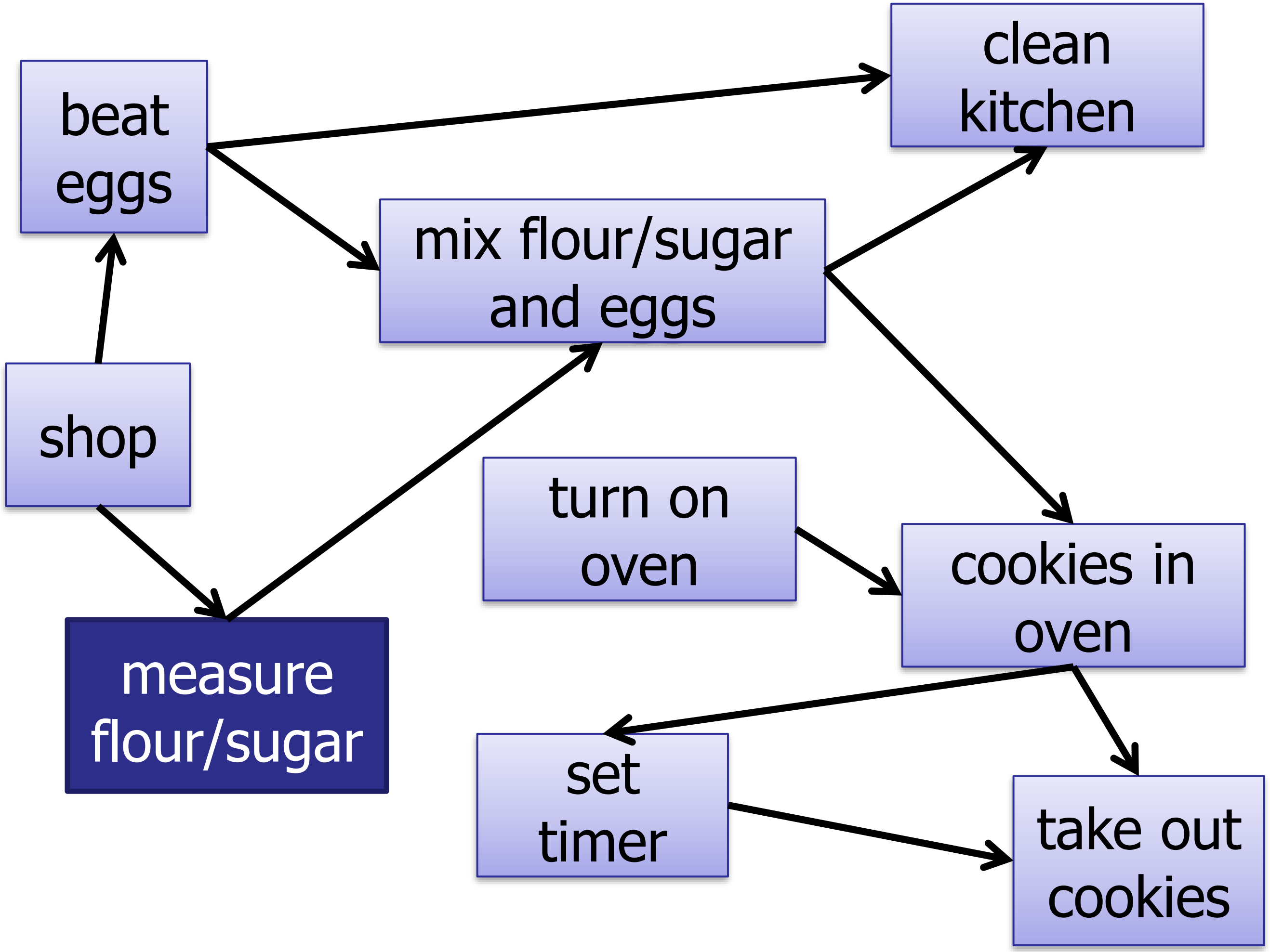
1. Breadth-first search
2. ✓ Depth-first search
3. Bloom Filter
4. Karatsuba algorithm
5. Something else

Depth-First Search



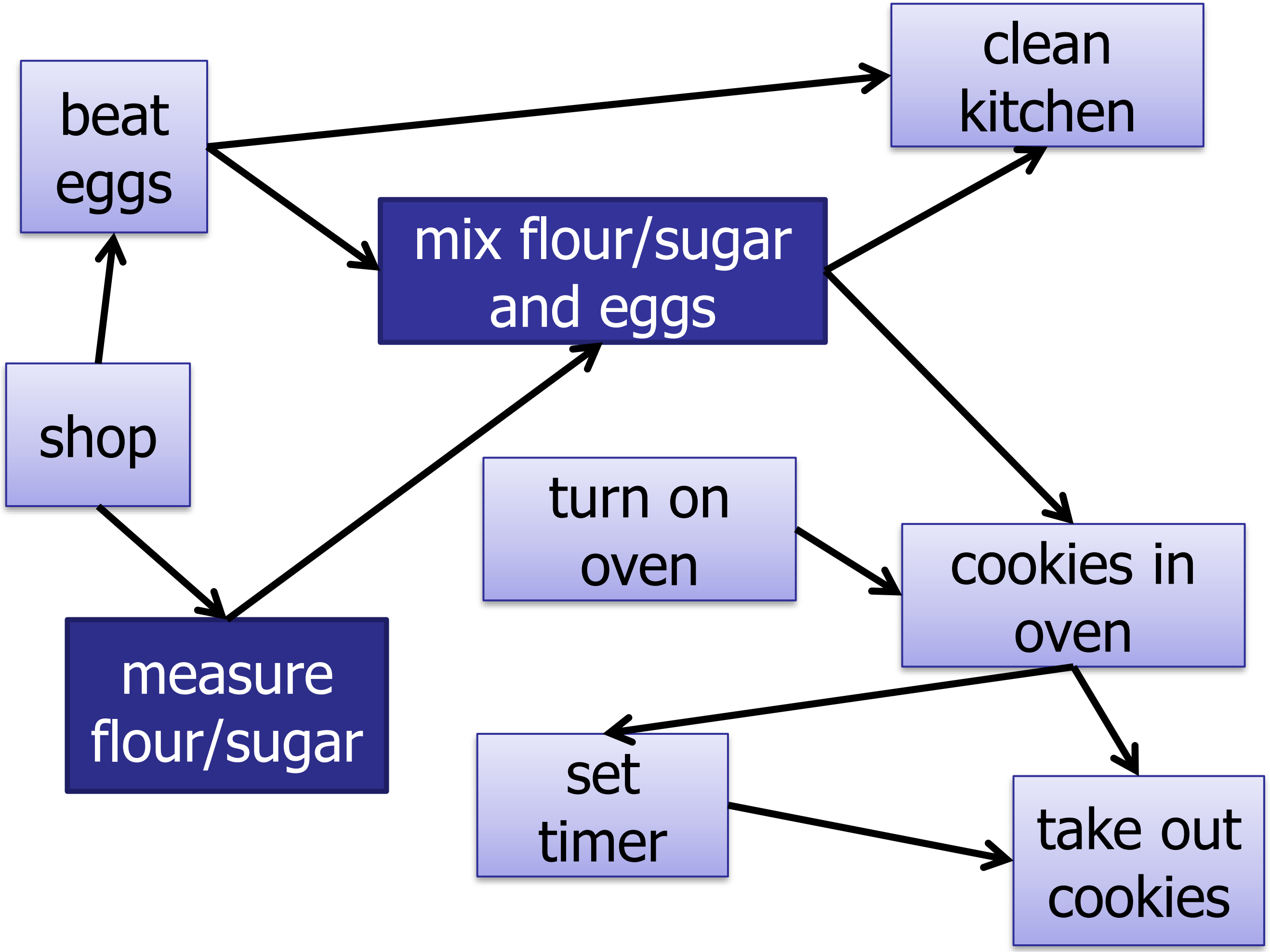
Depth-First Search

1. measure



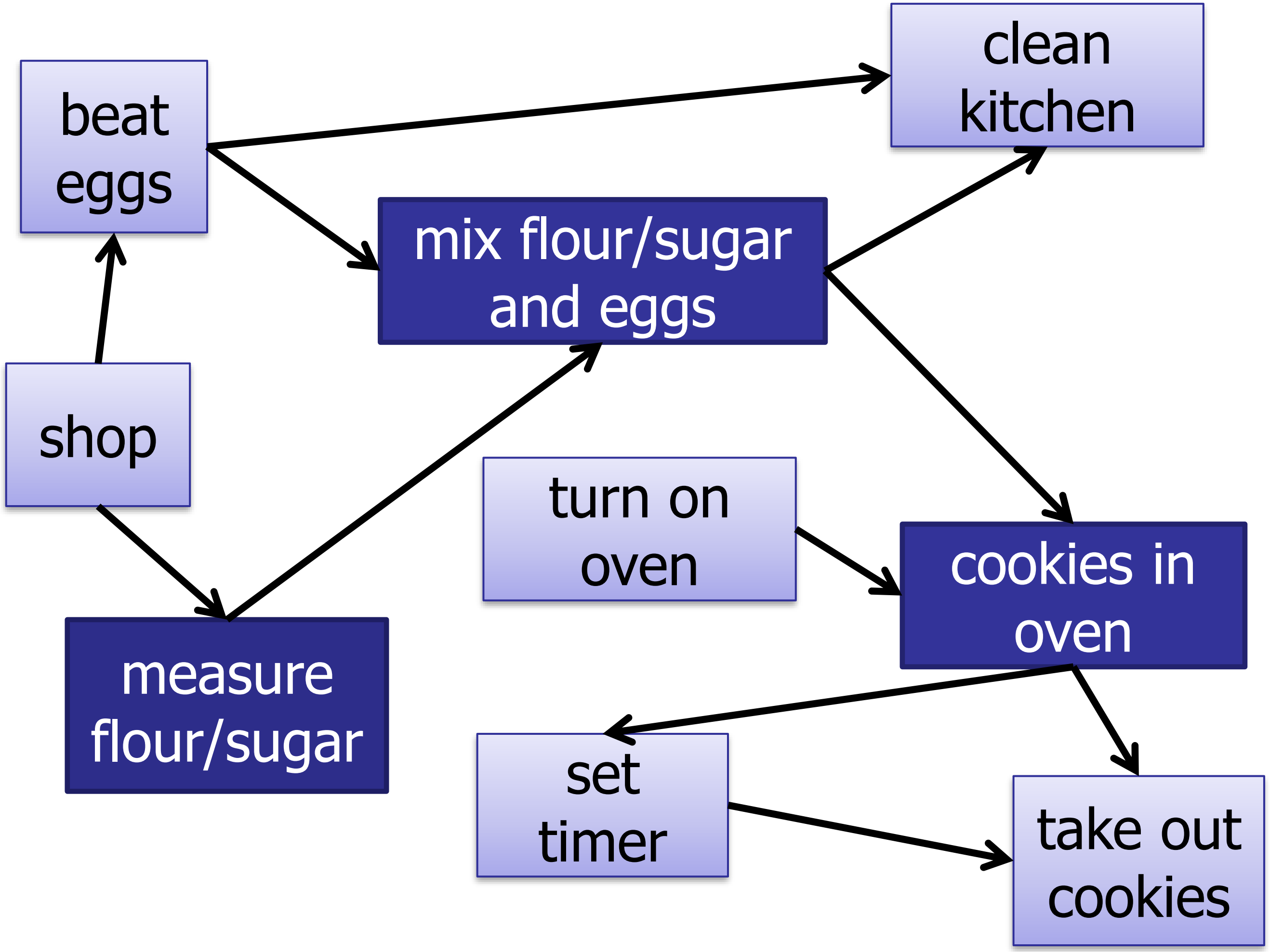
Depth-First Search

1. measure
2. mix



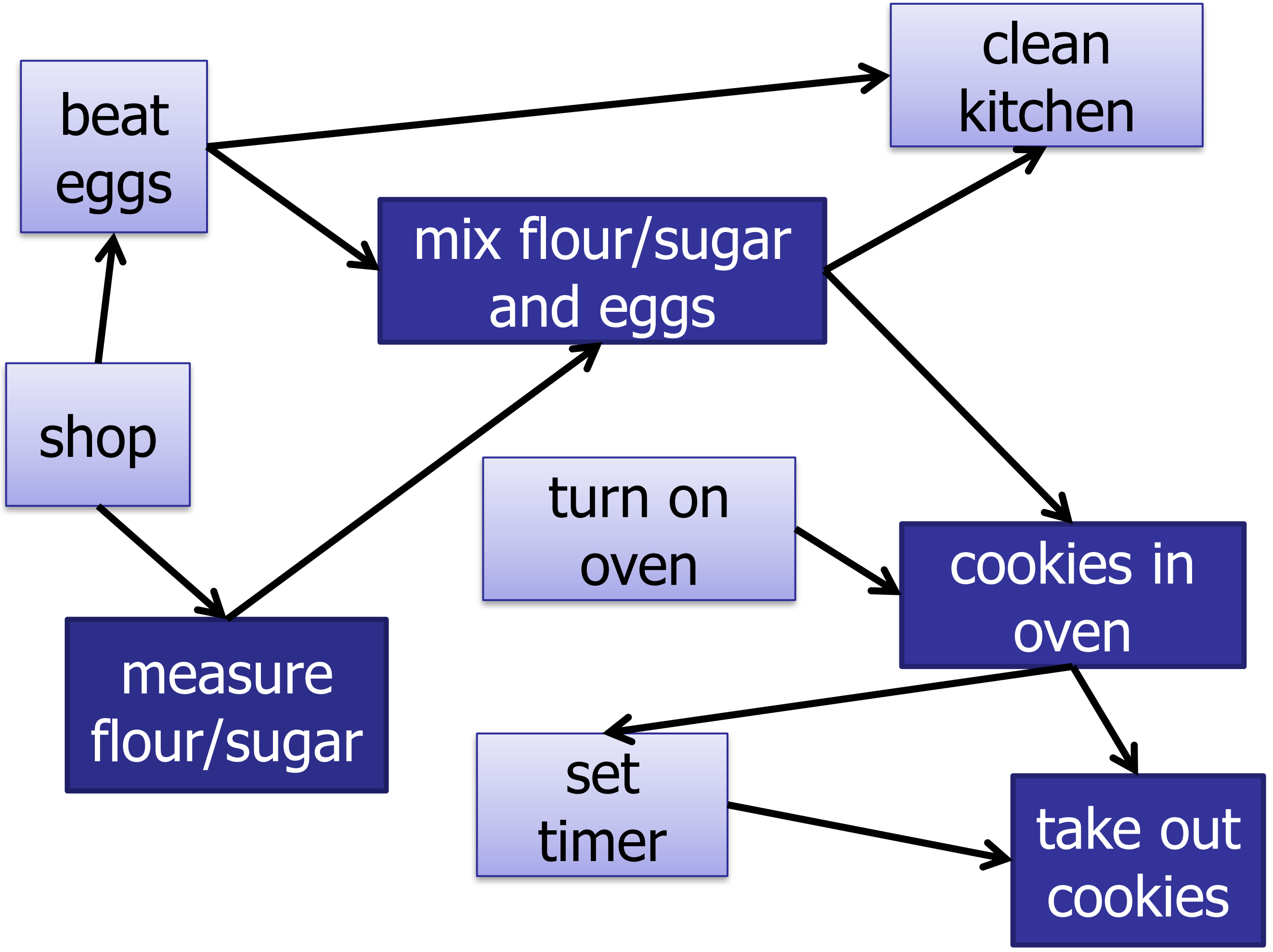
Depth-First Search

1. measure
2. mix
3. in oven



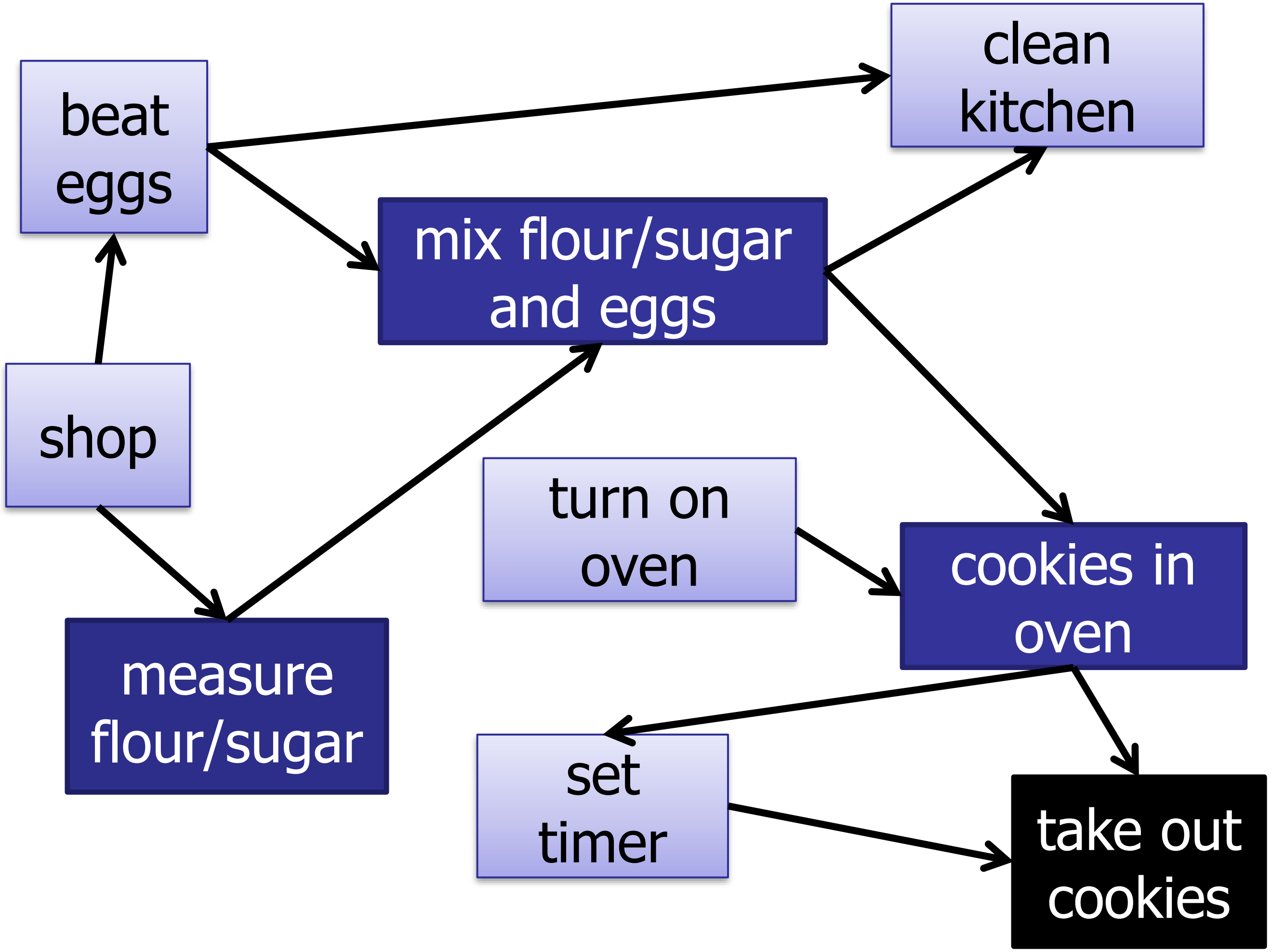
Depth-First Search

1. measure
2. mix
3. in oven
4. take out



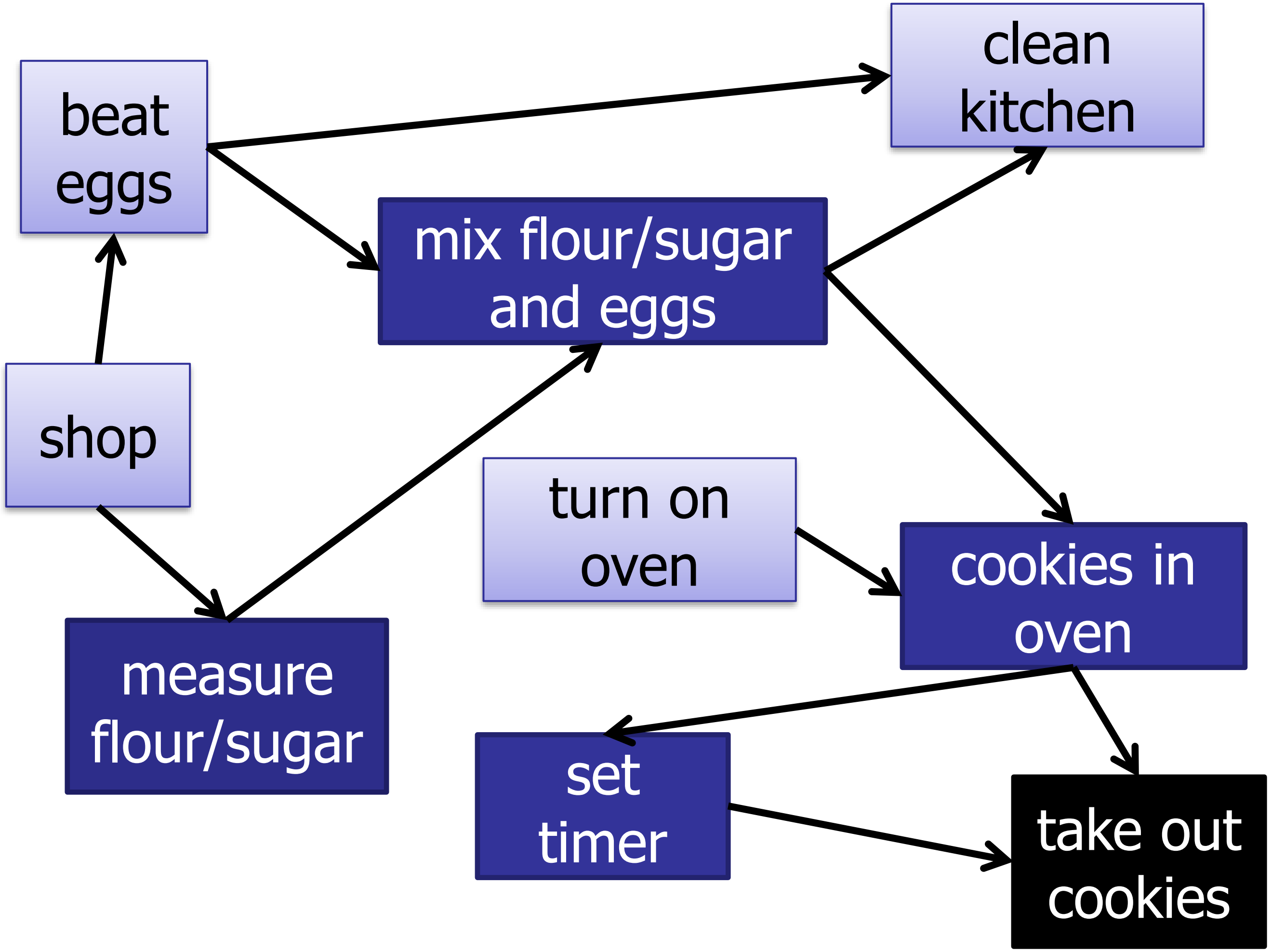
Depth-First Search

1. measure
2. mix
3. in oven
4. take out



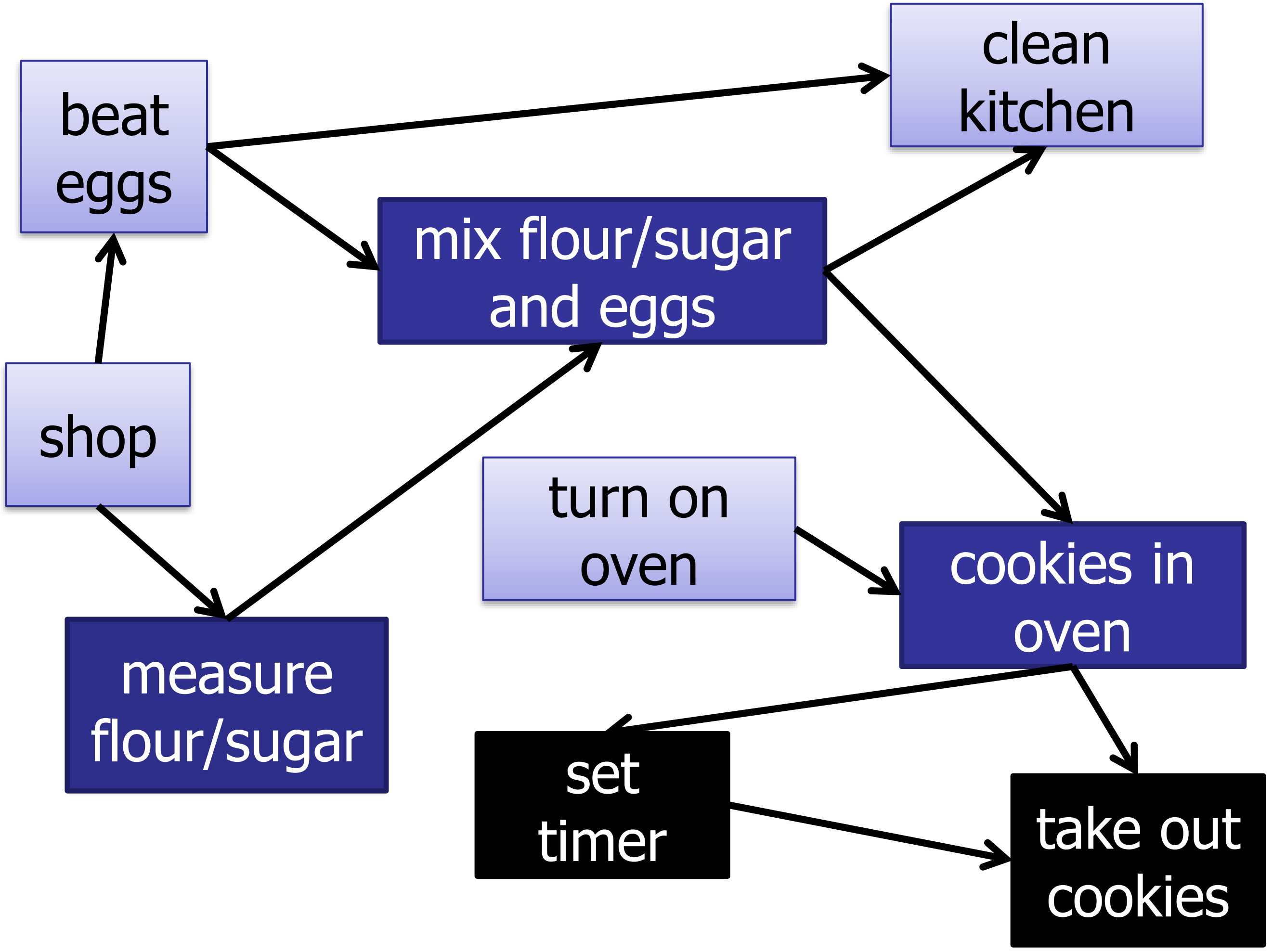
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



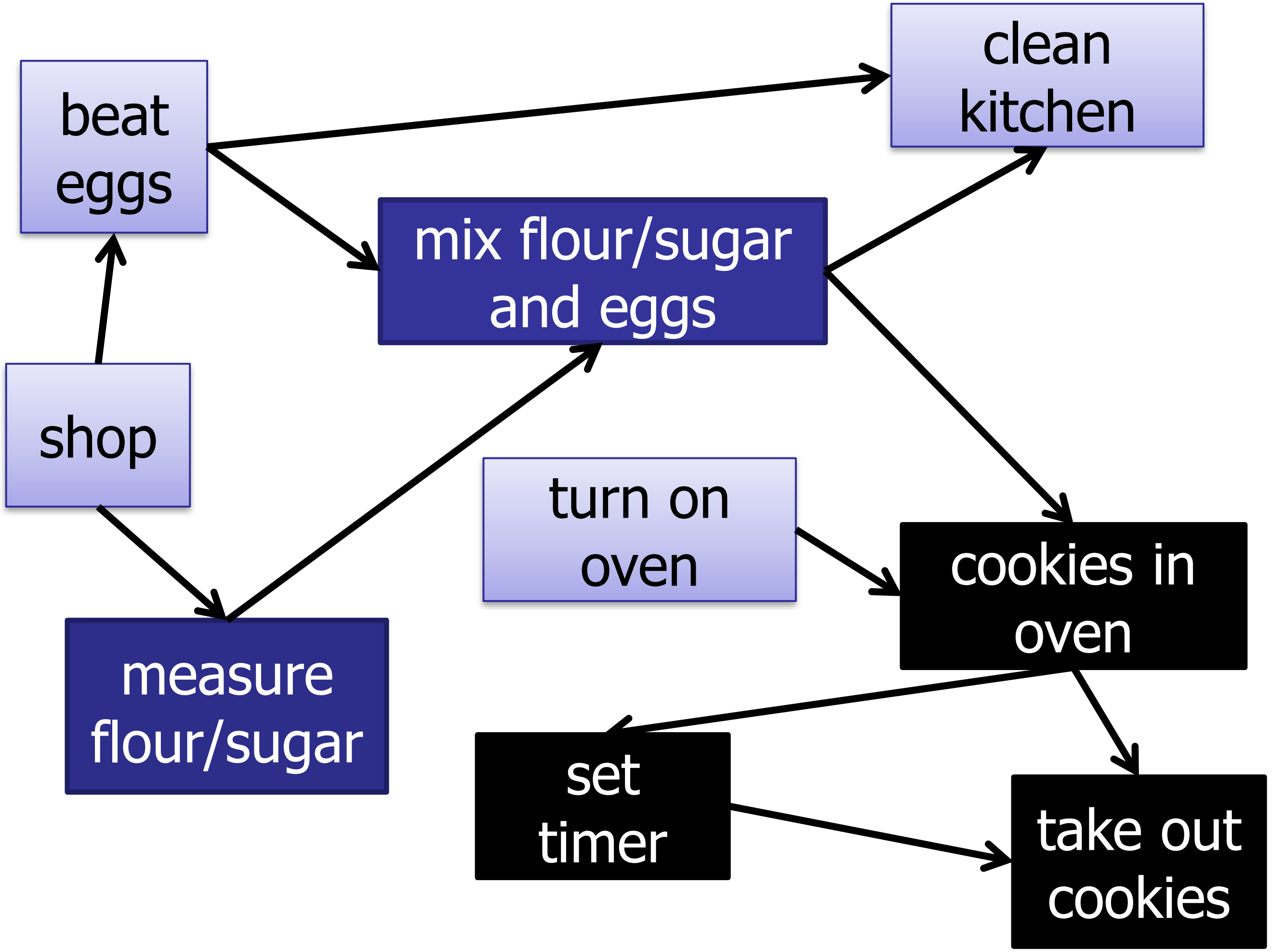
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



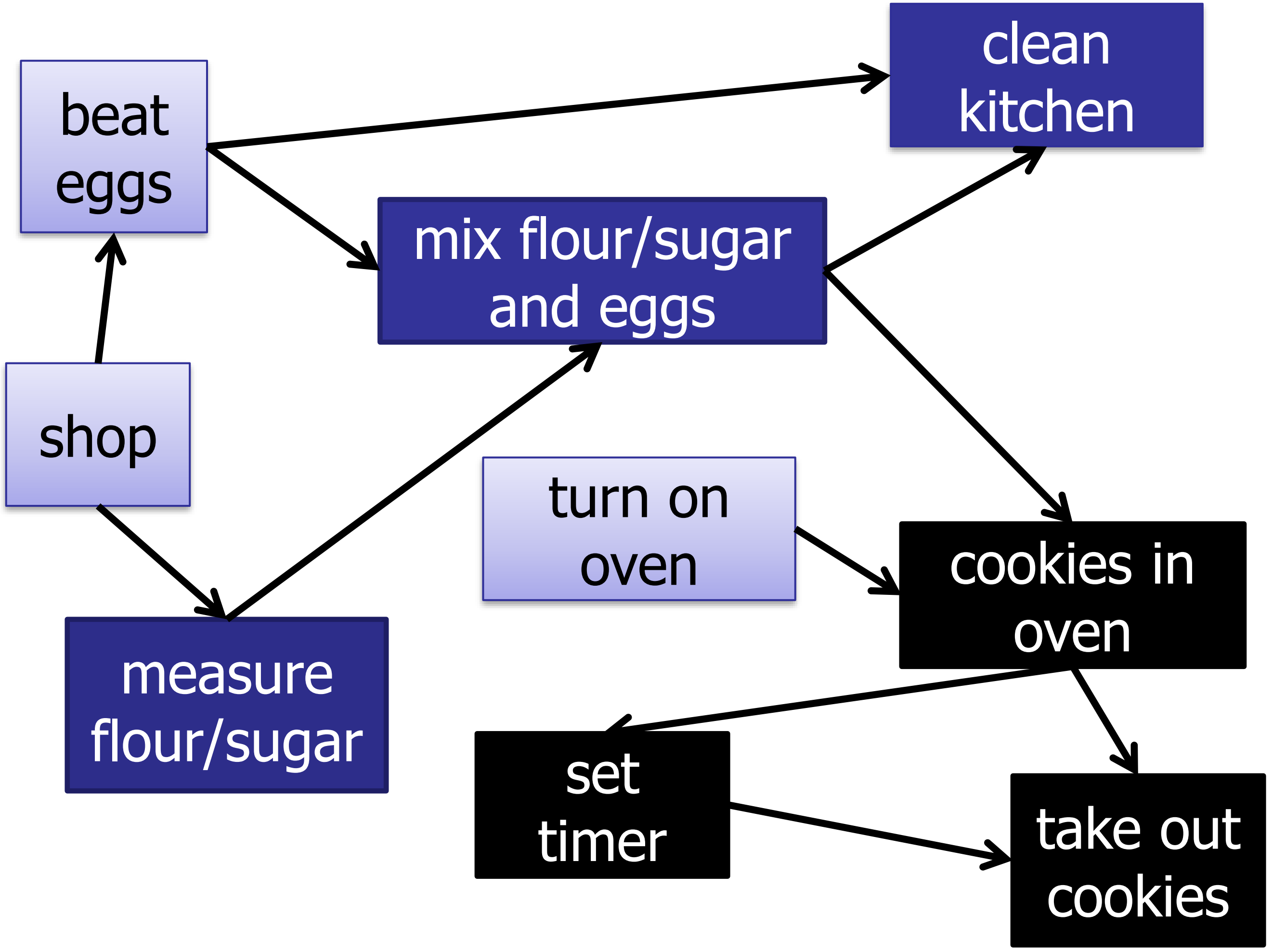
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



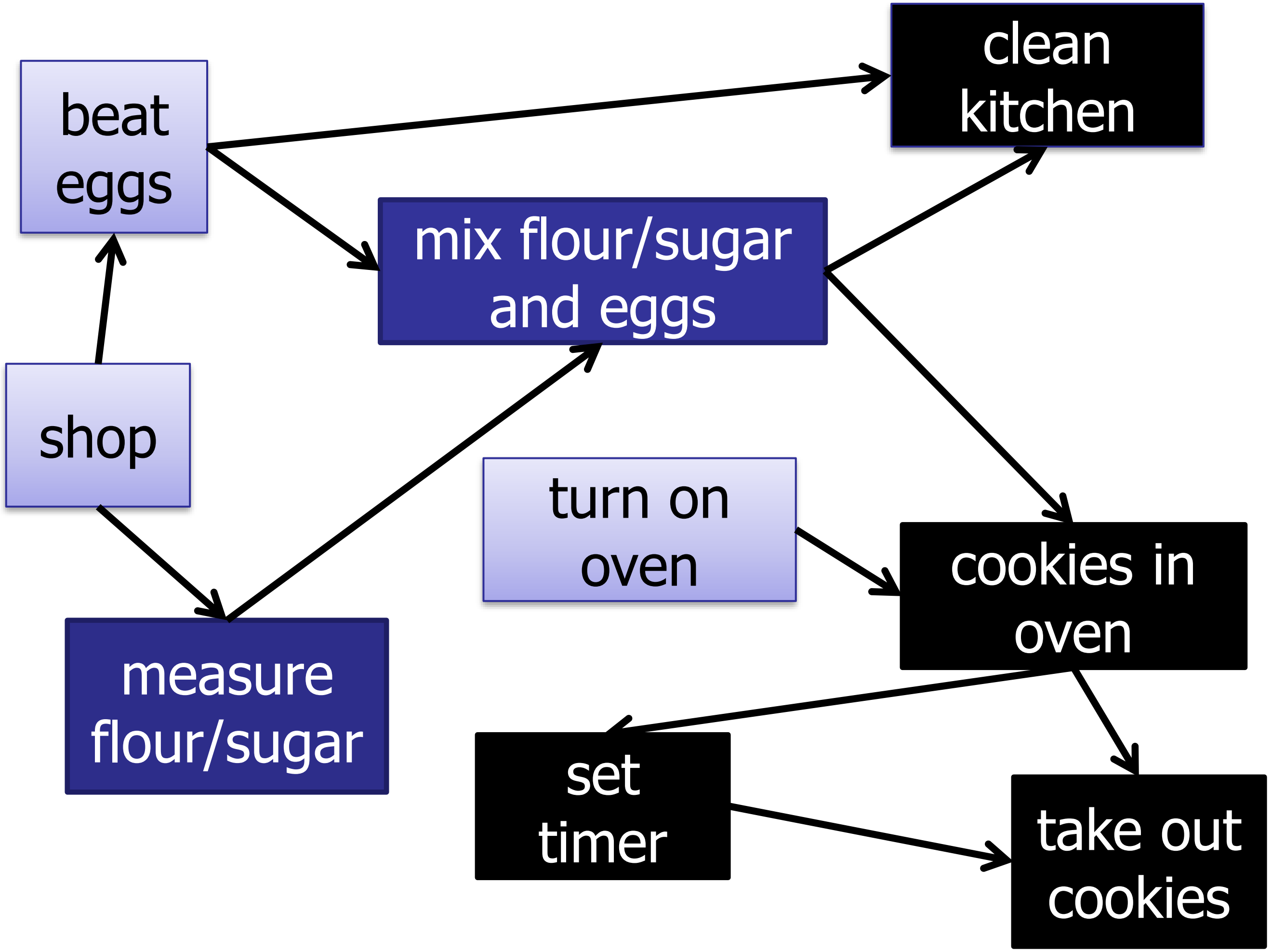
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



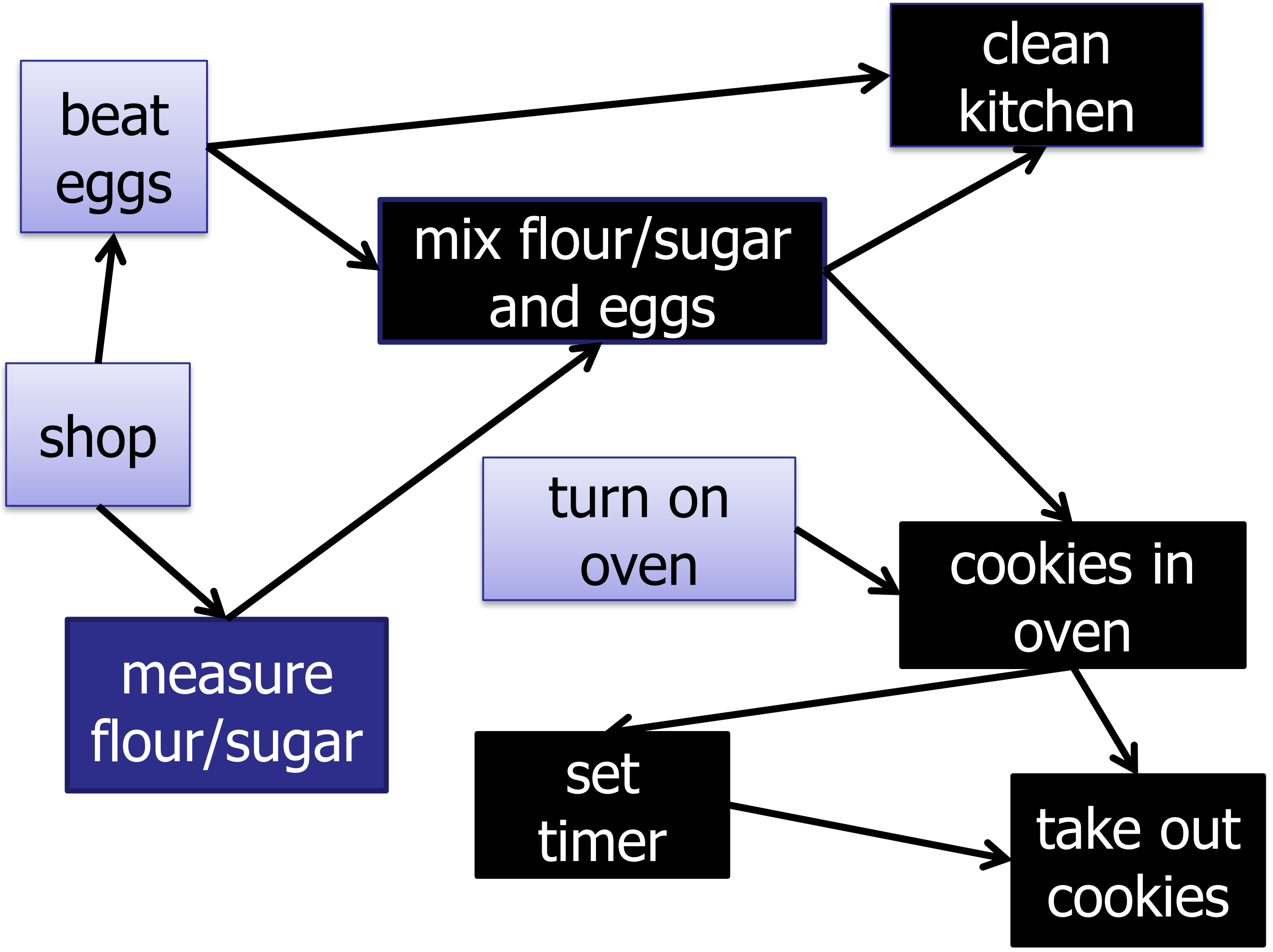
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



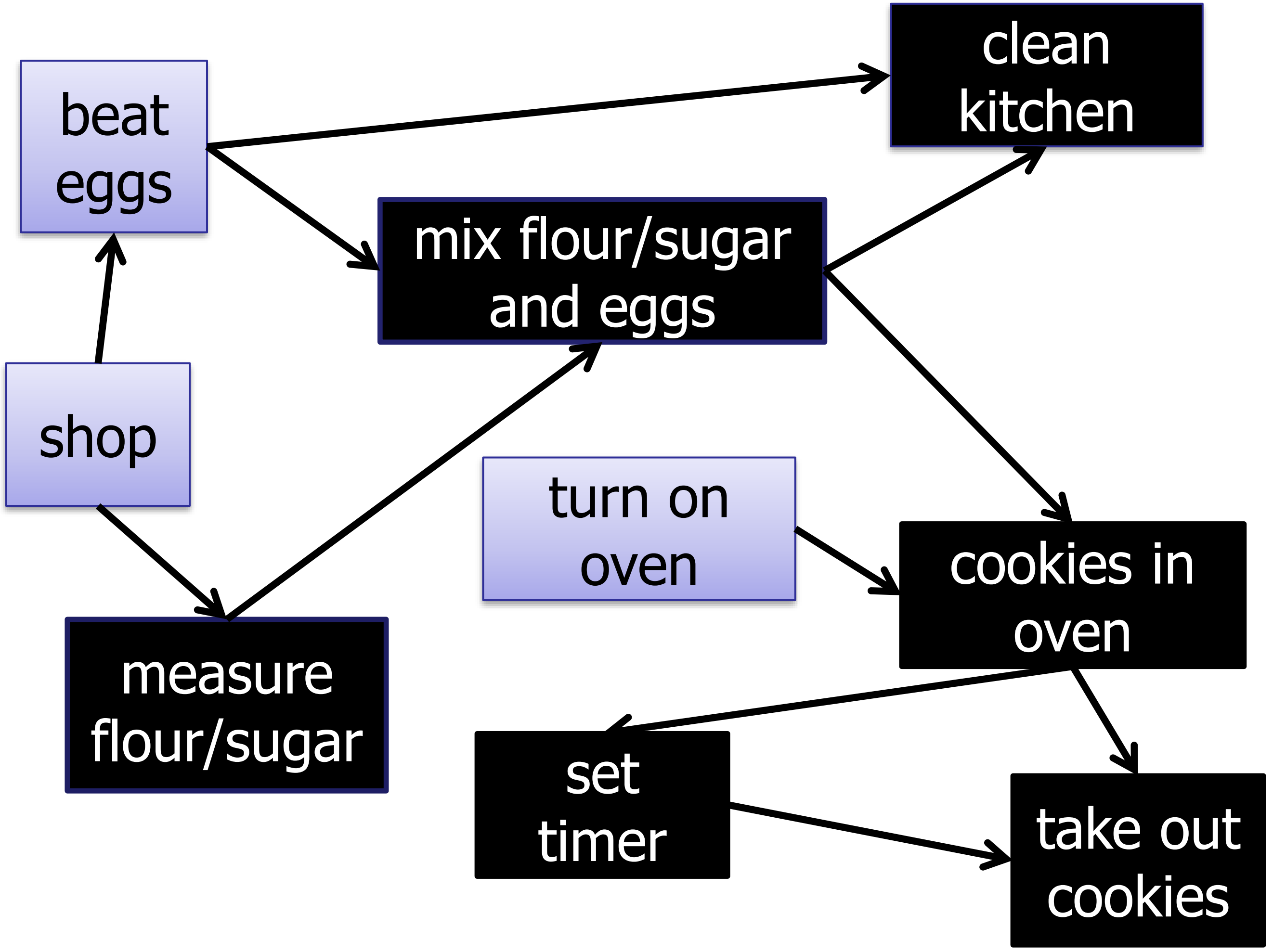
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



Searching a (Directed) Graph

Pre-Order Depth-First Search:

– Process each node when it is *first* visited.

Searching a (Directed) Graph

Pre-Order Depth-First Search:

- Process each node when it is *first* visited.

Post-Order Depth-First Search:

- Process each node when it is *last* visited.

DFS: Pre-Order

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
     if (!visited[v]){
           visited[v] = true;
           ProcessNode(v);
           DFS-visit(nodeList, visited, v);
     }
  }
}
```

DFS Post-Order

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
     if (!visited[v]){
           visited[v] = true;
           DFS-visit(nodeList, visited, v);
           ProcessNode(v);
     }
  }
}
```

Searching a (Directed) Graph

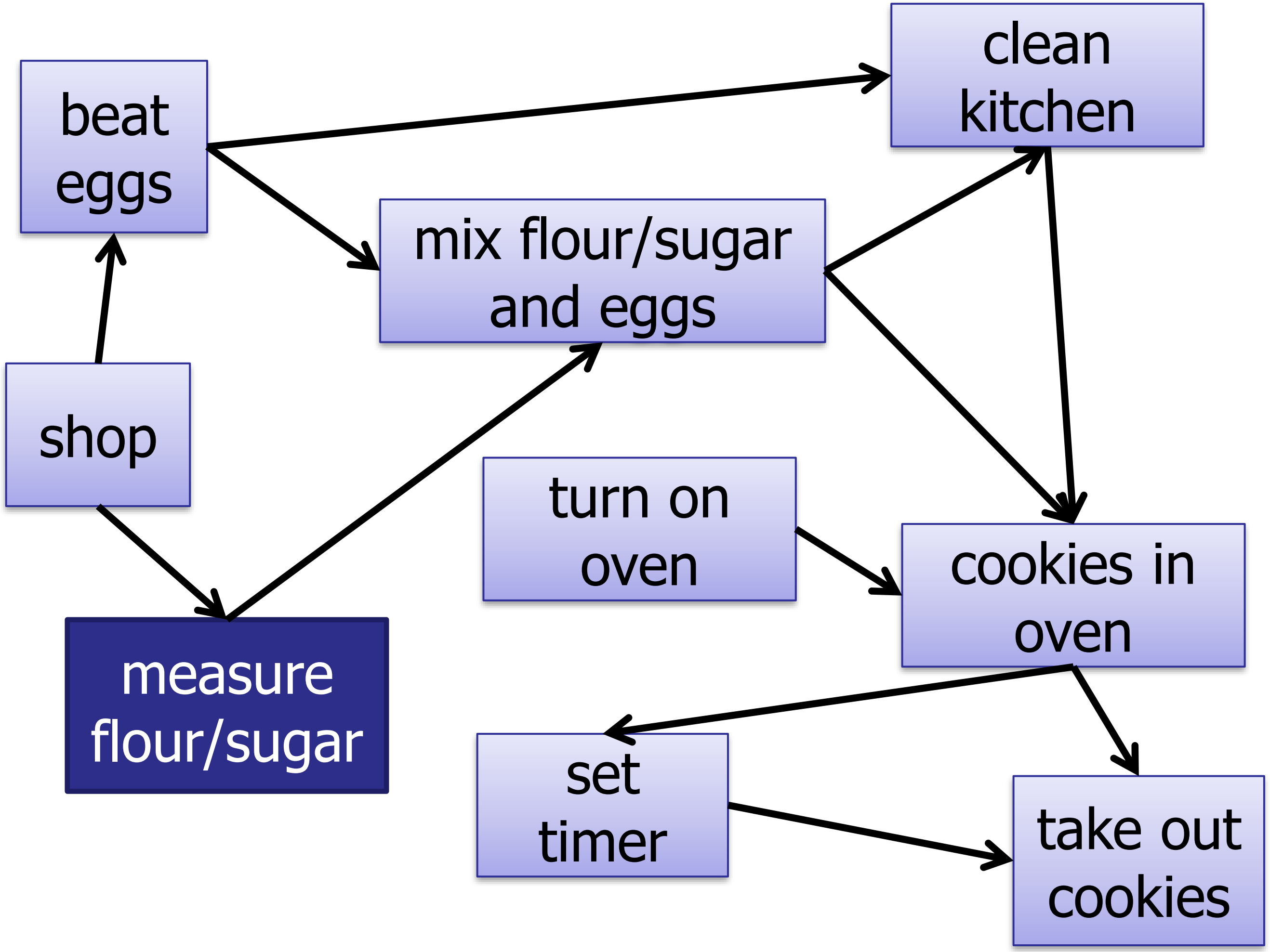
Pre-Order Depth-First Search:

– Process each node when it is *first* visited.

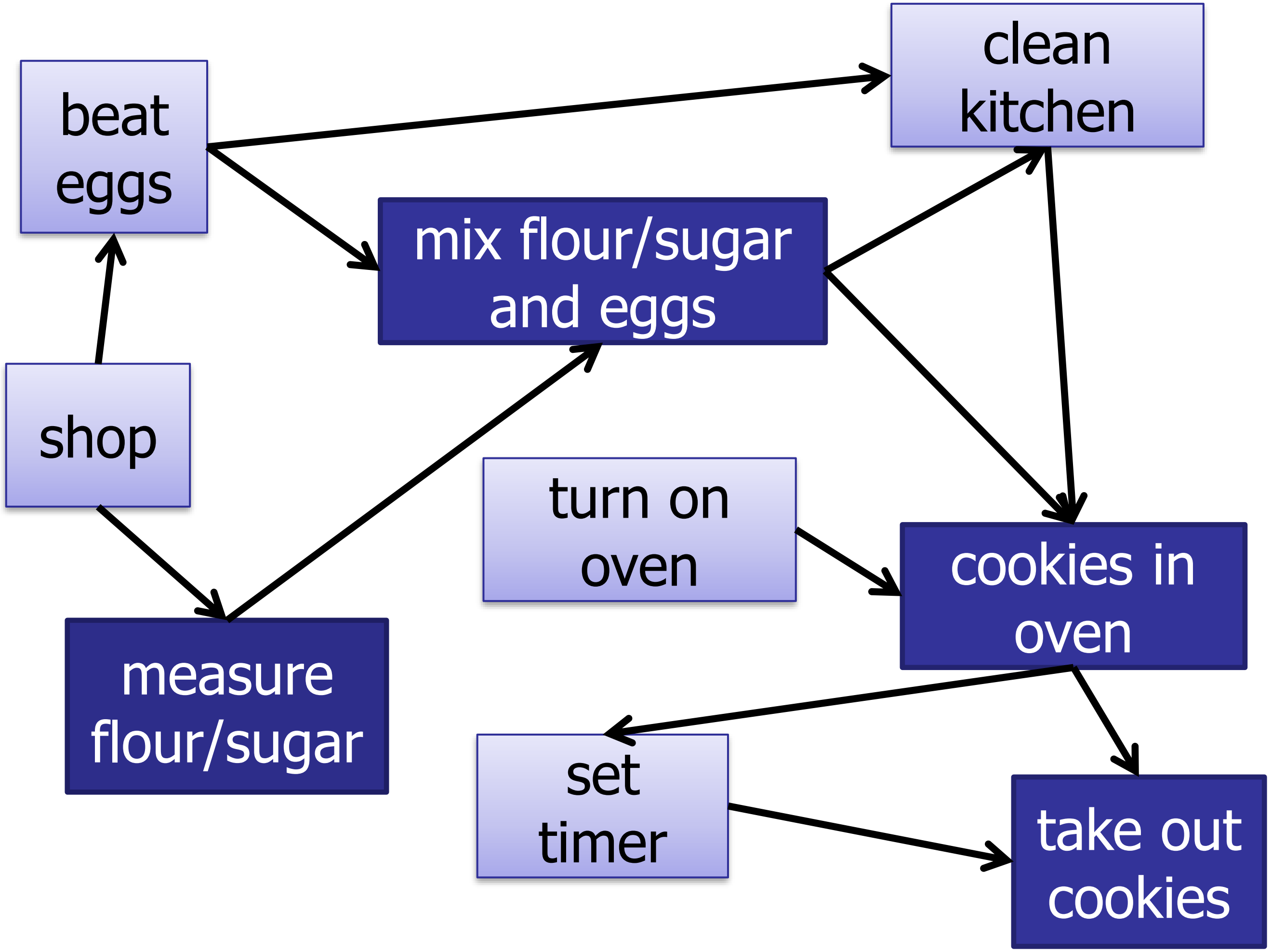
Post-Order Depth-First Search:

– Process each node when it is *last* visited.

Post-Order Depth-First Search

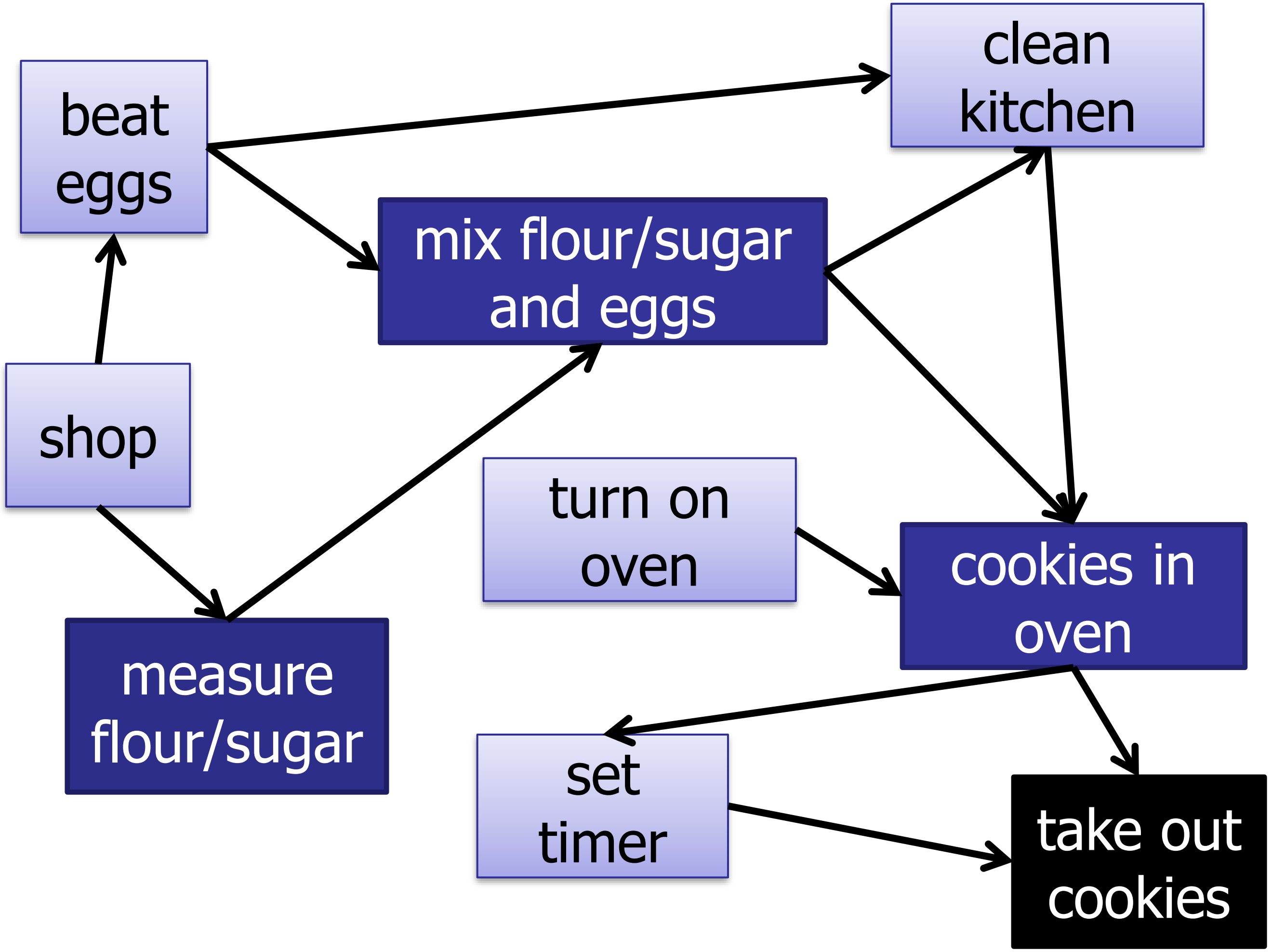


Post-Order Depth-First Search



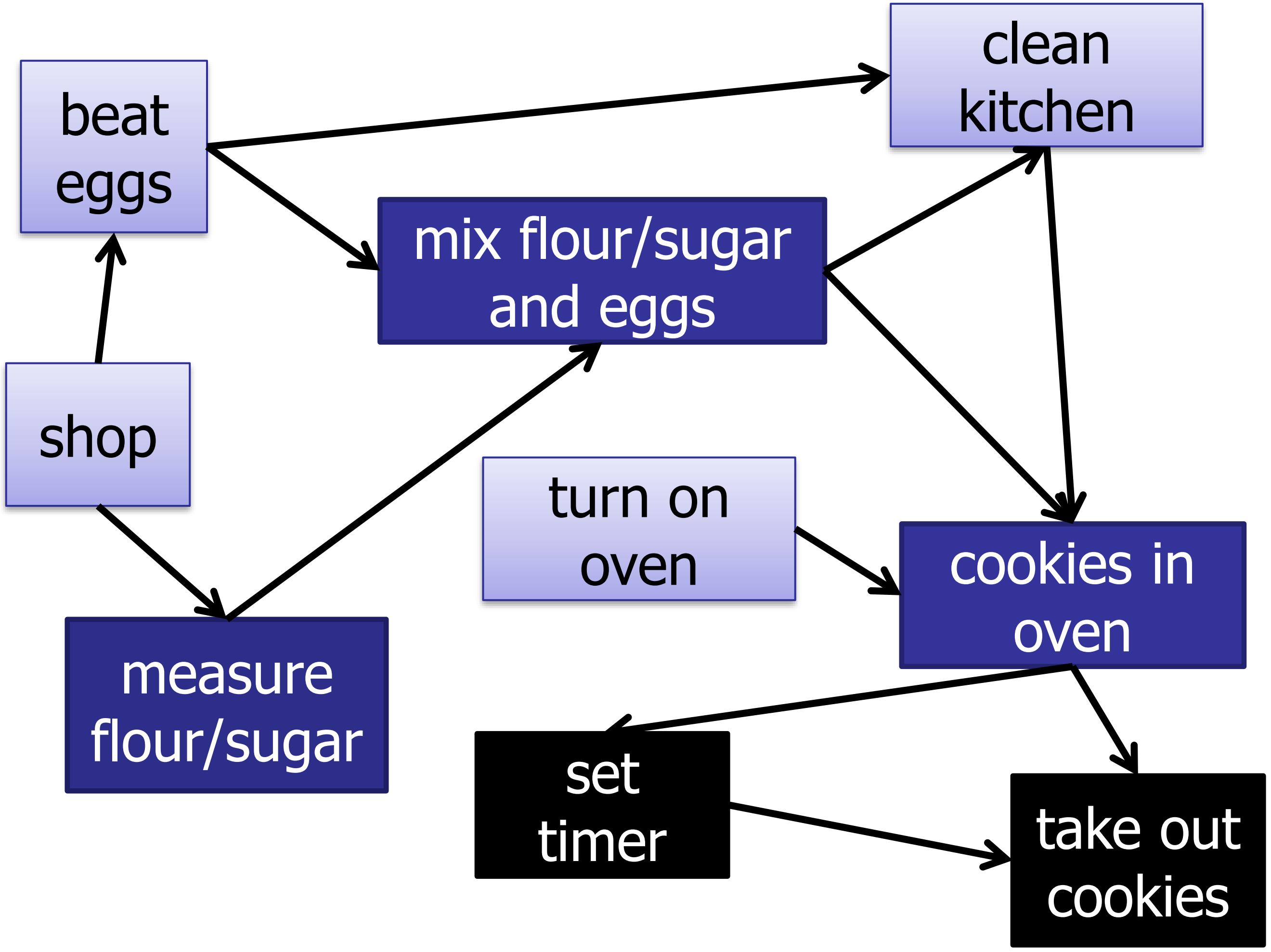
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7.
8.
9. take out



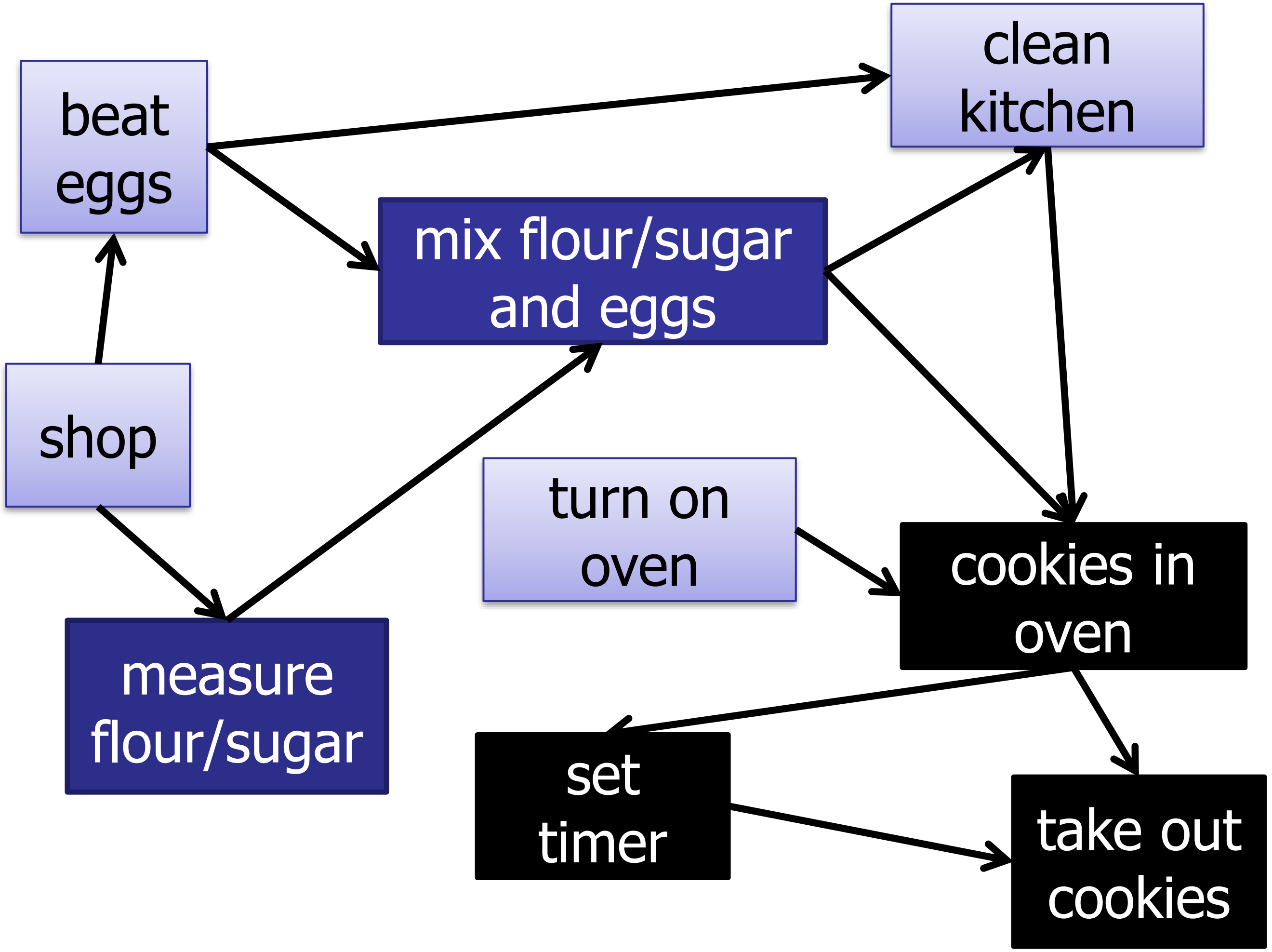
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7.
8. set timer
9. take out



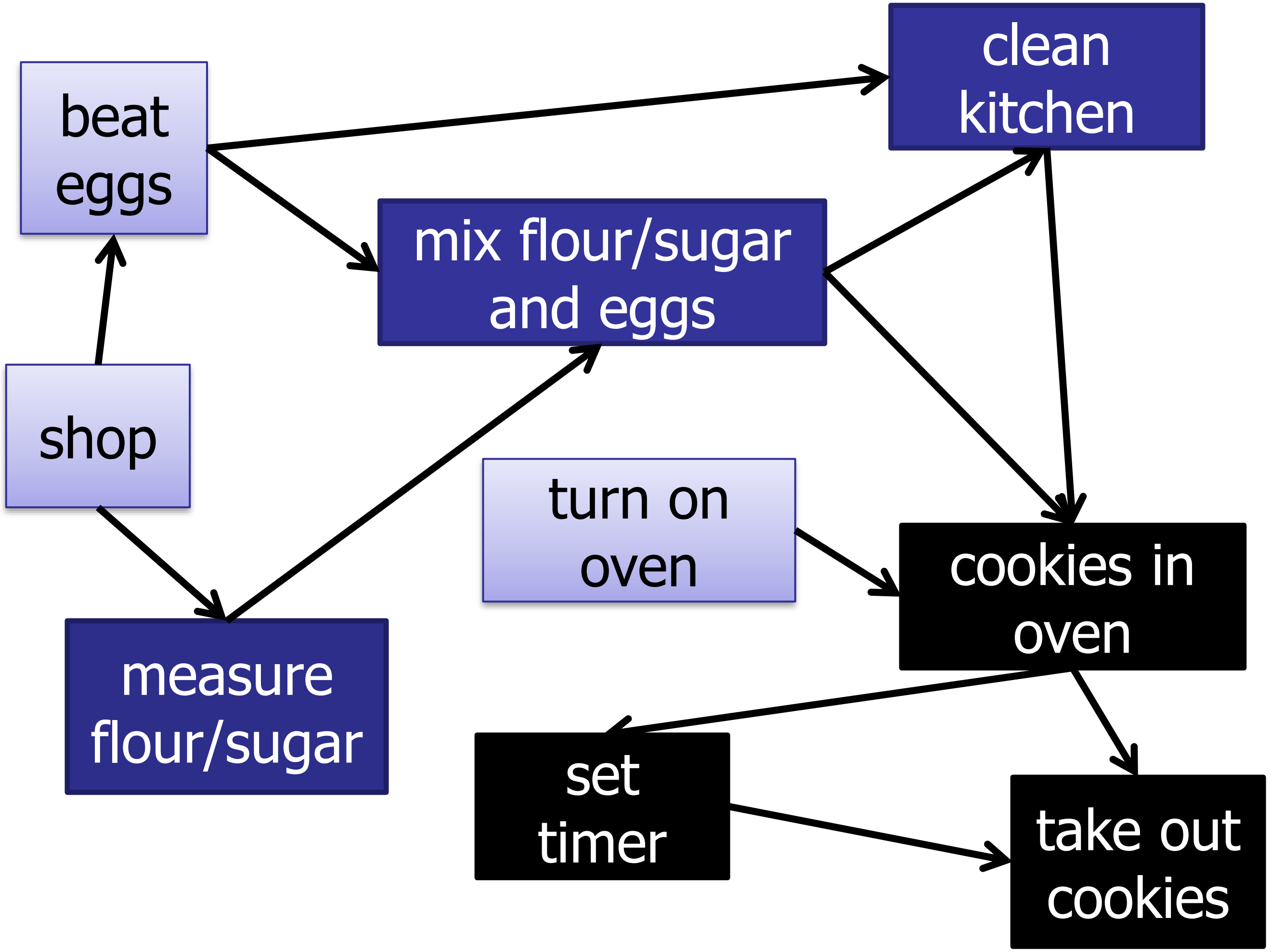
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



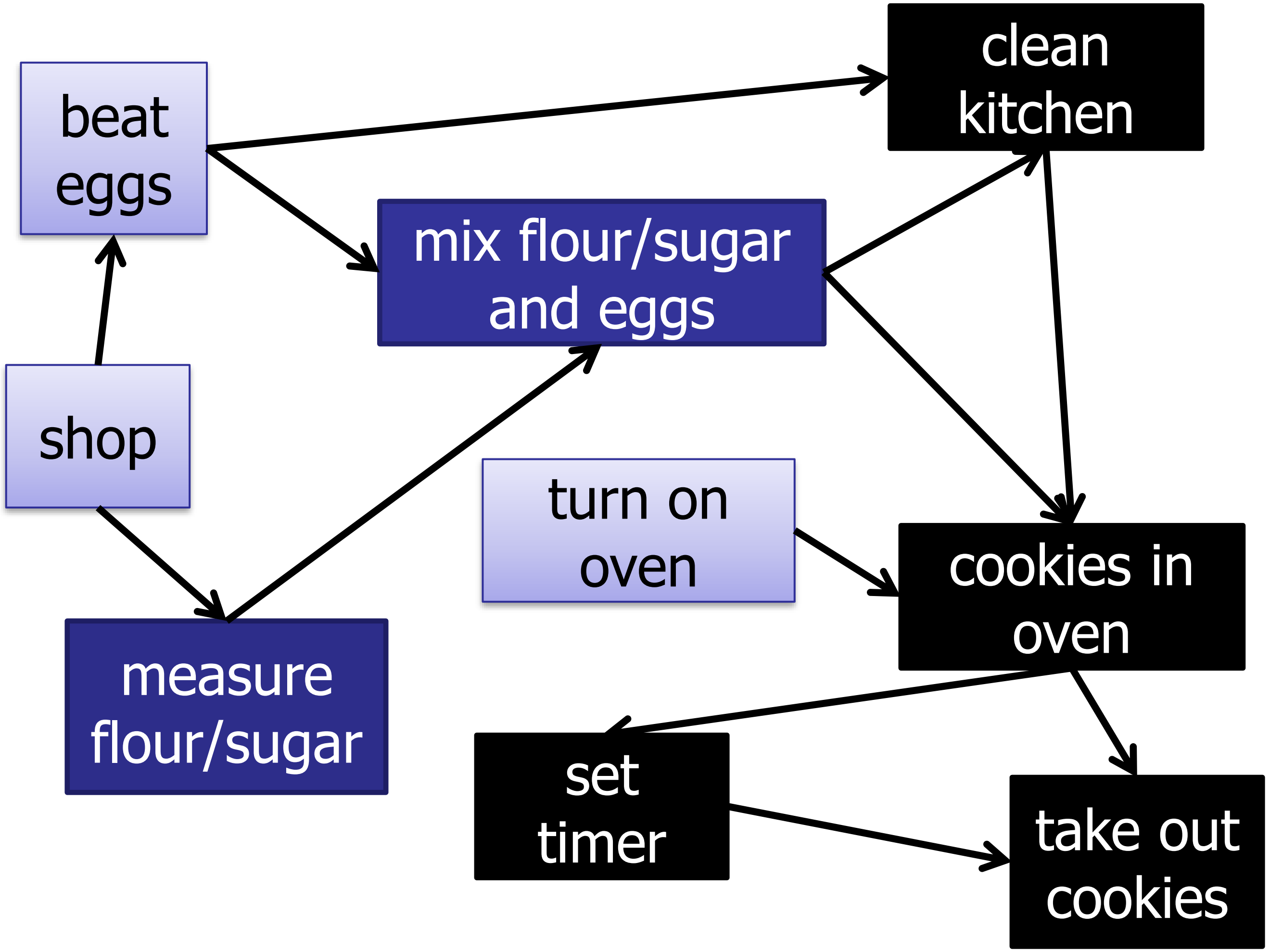
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



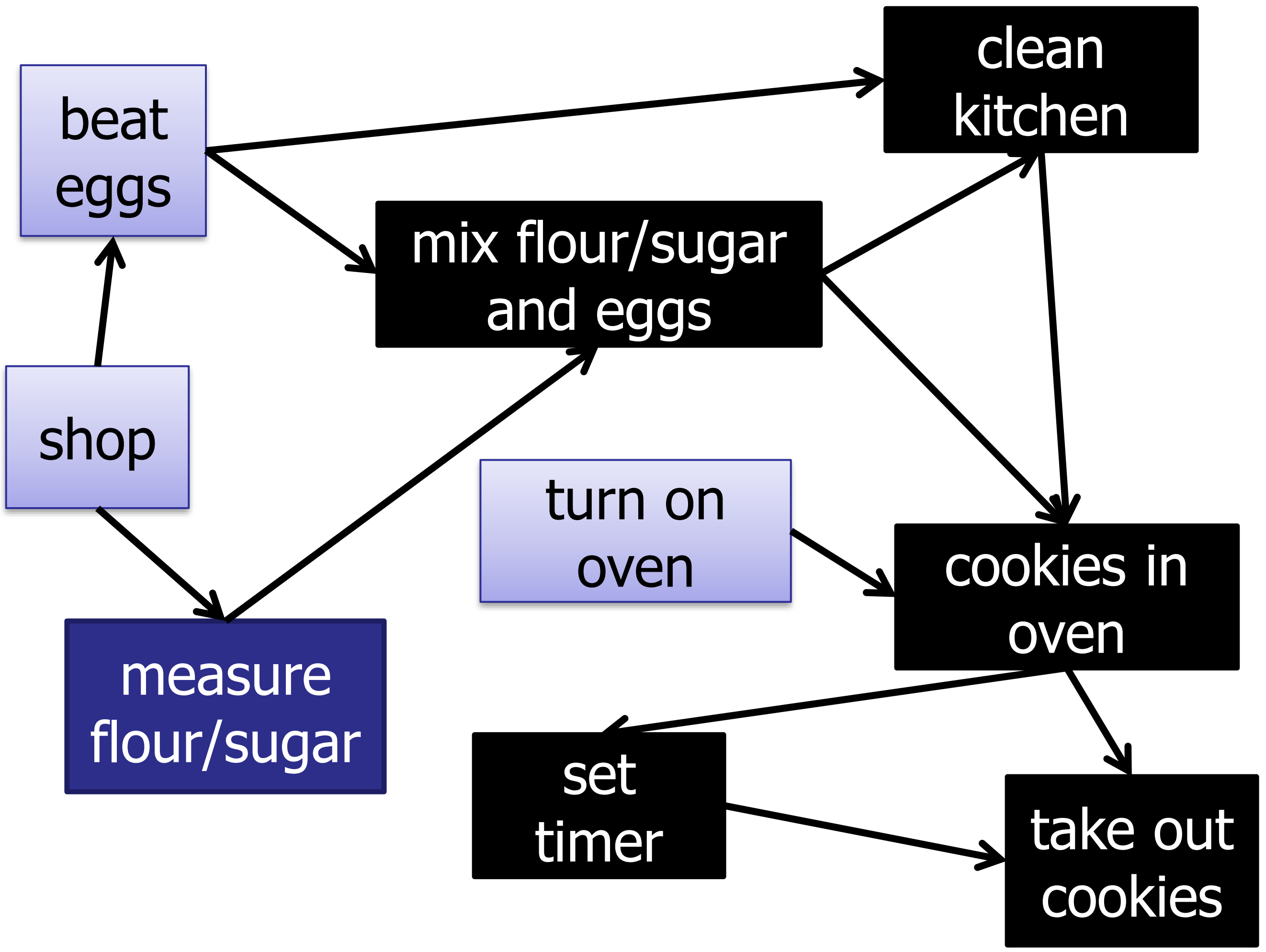
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6. clean
7. in oven
8. set timer
9. take out



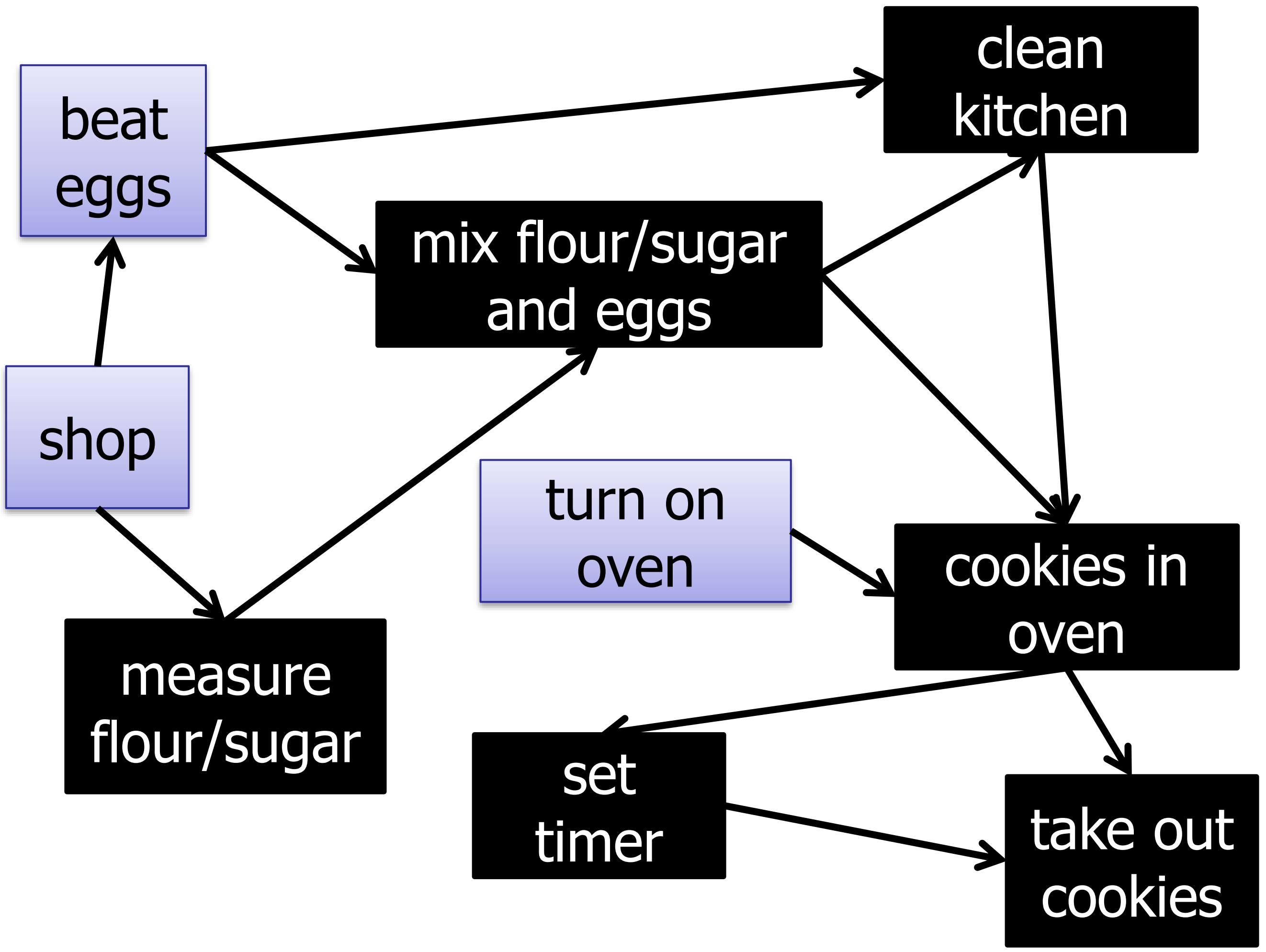
Post-Order Depth-First Search

1.
2.
3.
4.
5. mix
6. clean
7. in oven
8. set timer
9. take out



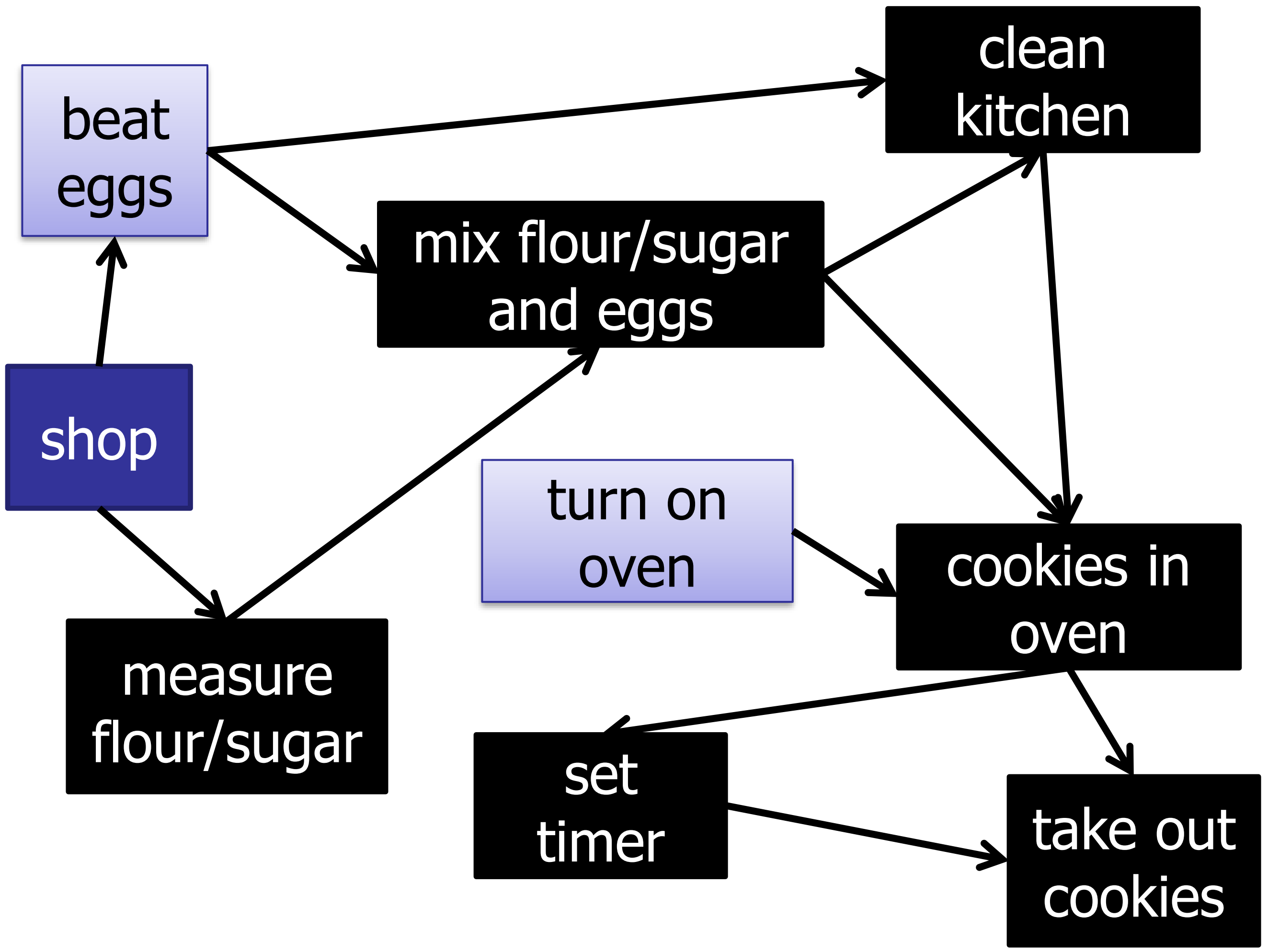
Post-Order Depth-First Search

1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



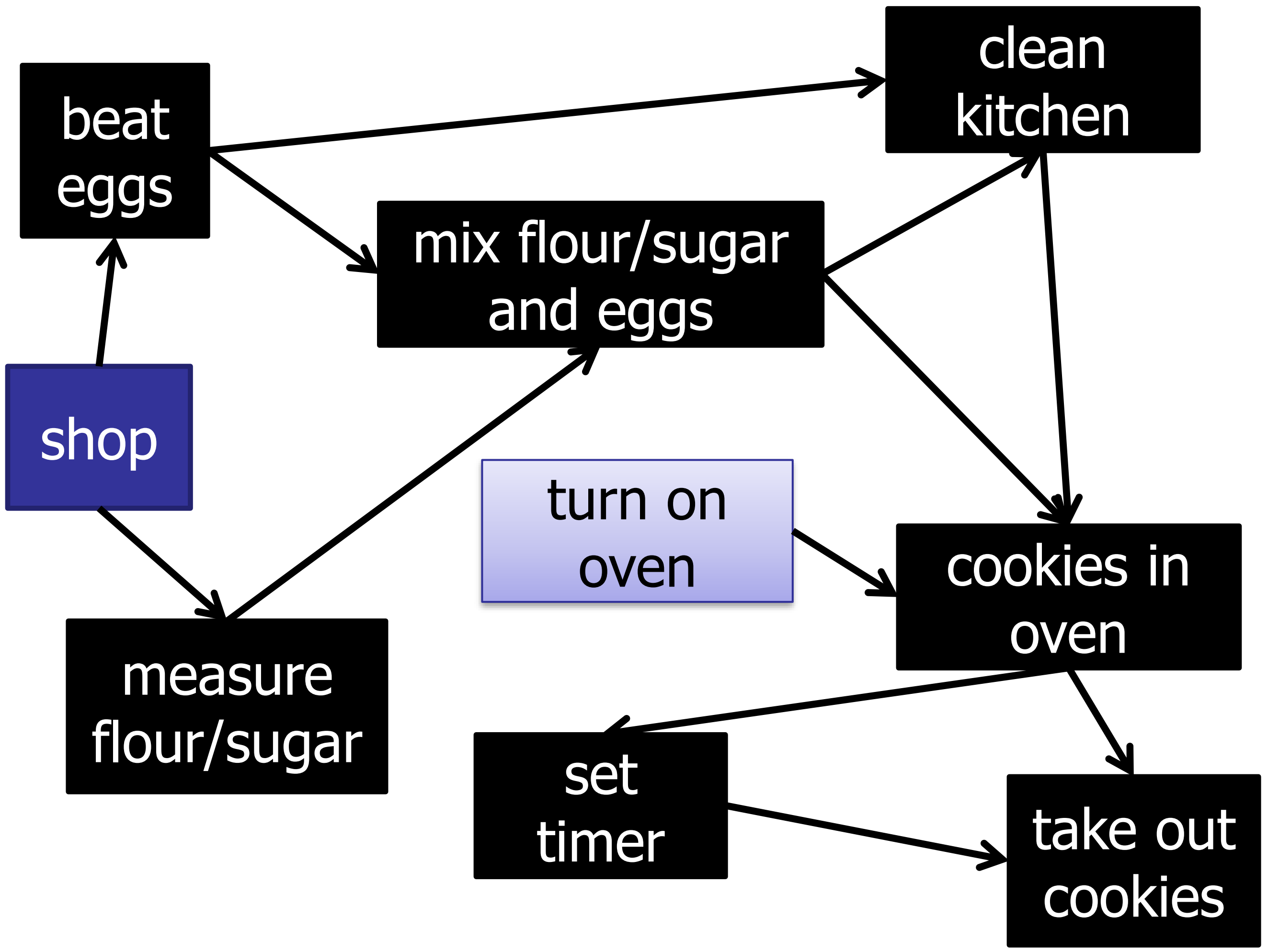
Post-Order Depth-First Search

1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



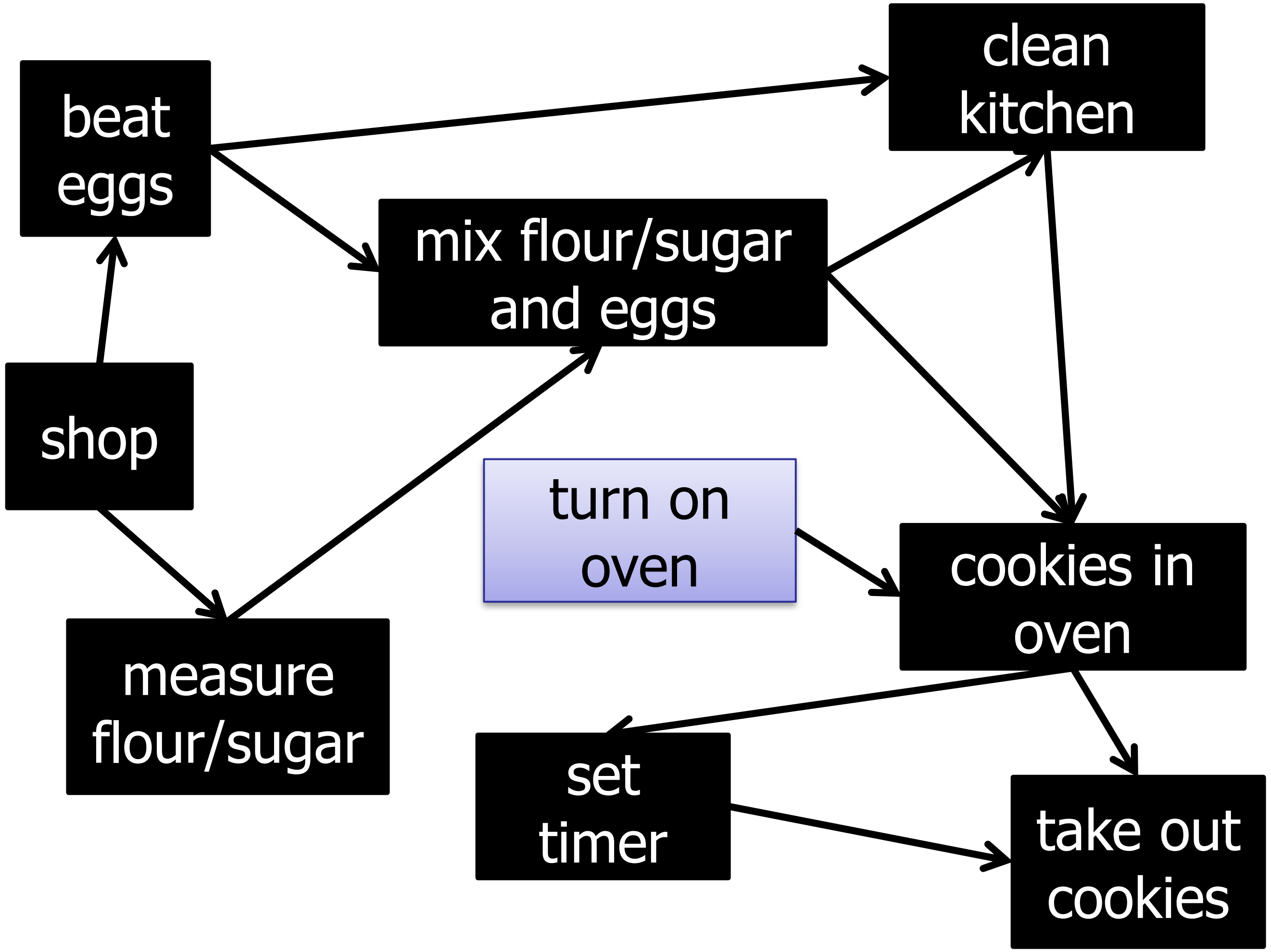
Post-Order Depth-First Search

1.
2.
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



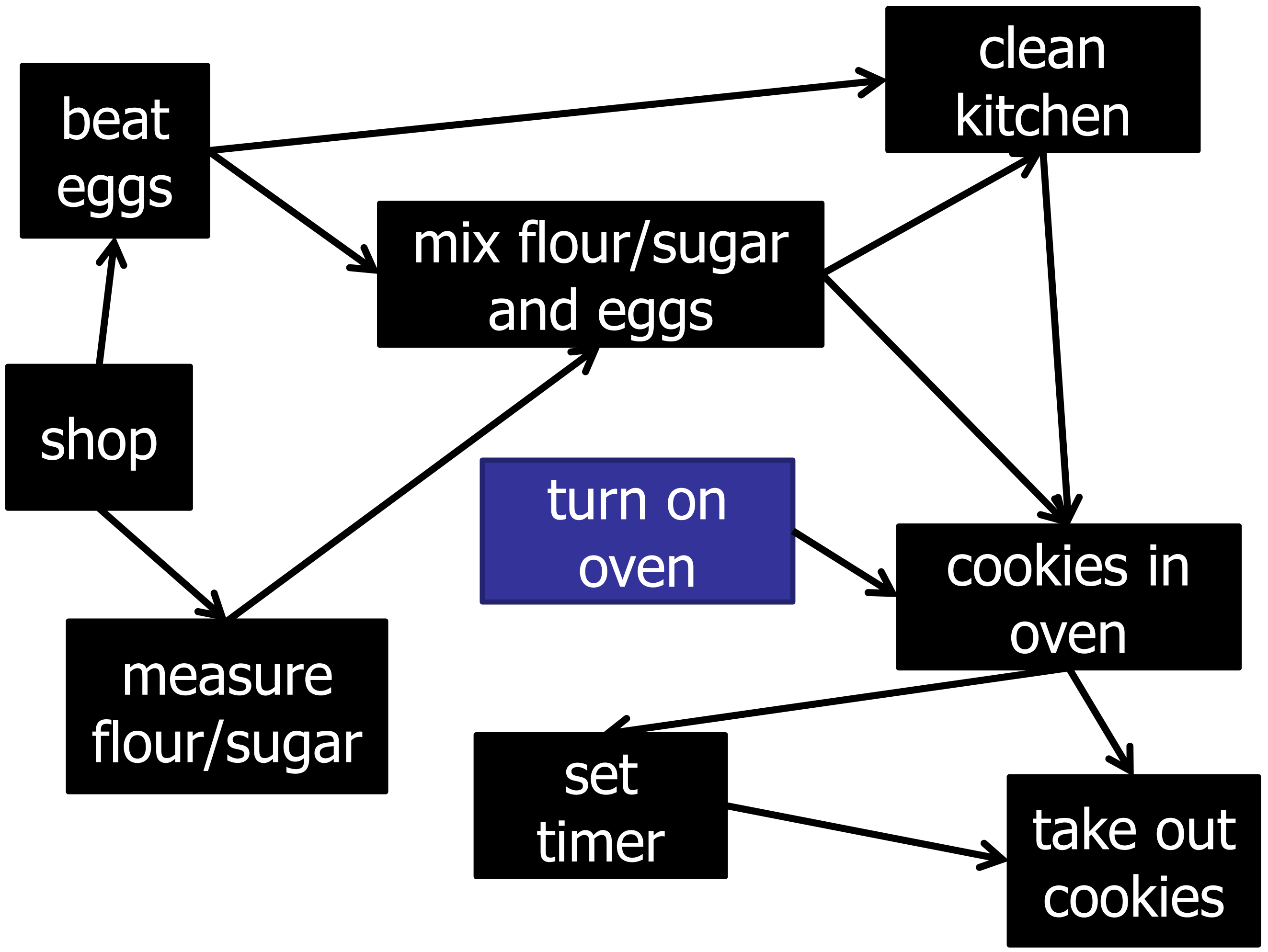
Post-Order Depth-First Search

1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



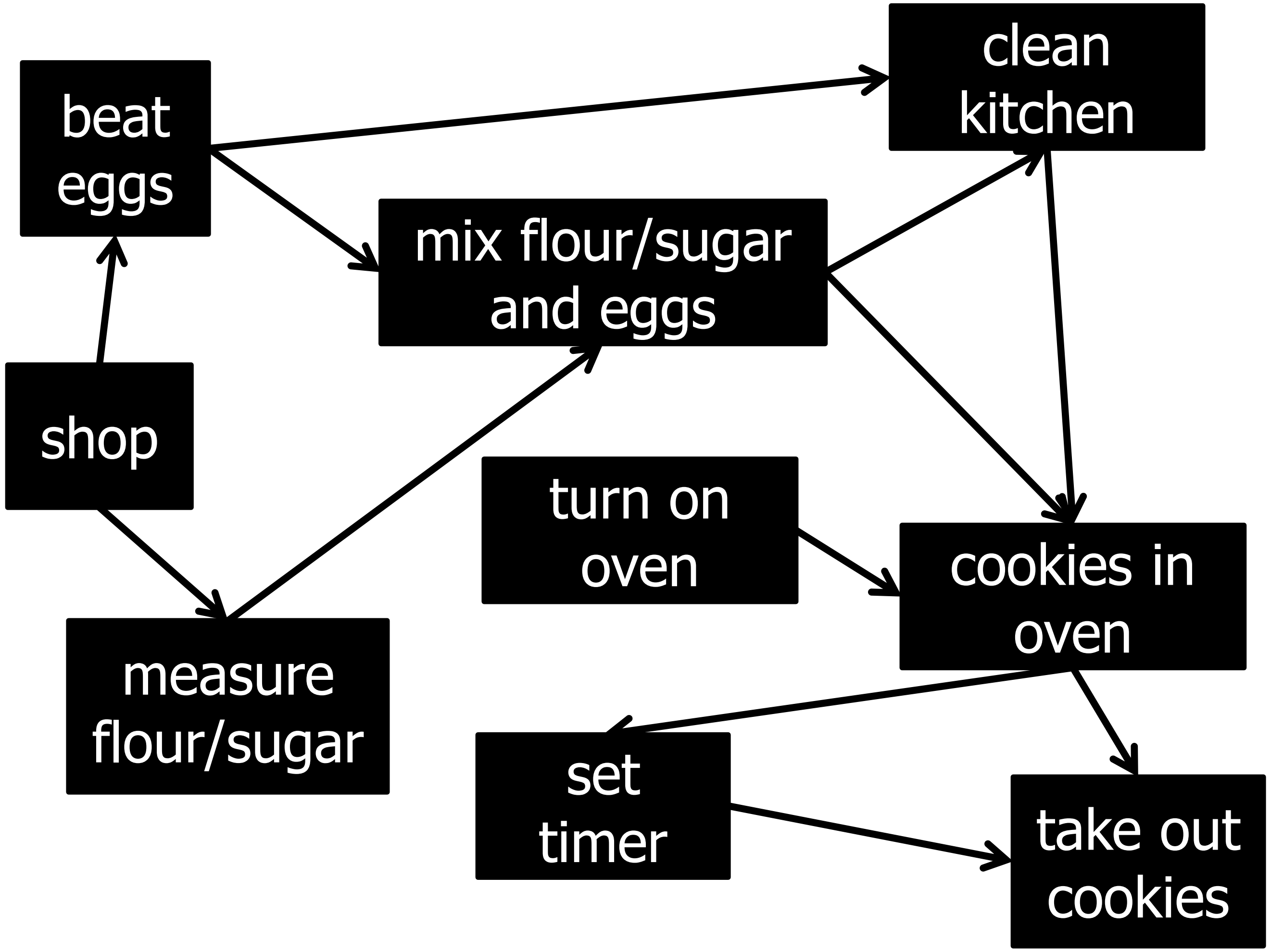
Post-Order Depth-First Search

1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



Post-Order Depth-First Search

1. on oven
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



Topological Sort

What is the time complexity of topological sort?

DFS: $O(V+E)$

Depth-First Search

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
     if (!visited[v]){
          visited[v] = true;
          DFS-visit(nodeList, visited, v);
          schedule.prepend(v);
     }
  }
}
```

Depth-First Search

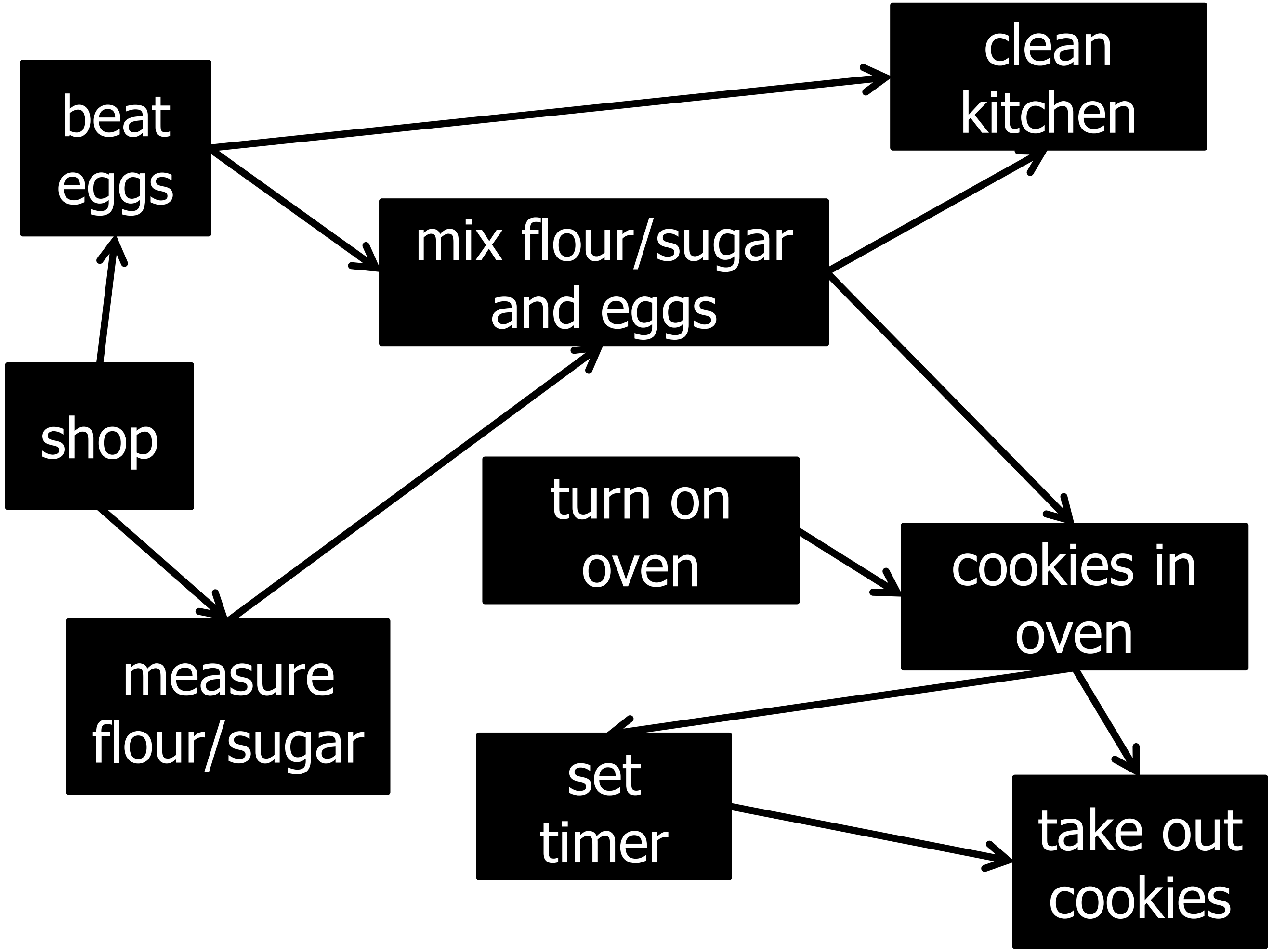
```
DFS(Node[] nodeList){
boolean[] visited = new boolean[nodeList.length];
Arrays.fill(visited, false);

  for (start = i; start<nodeList.length; start++) {
     if (!visited[start]){
           visited[start] = true;
           DFS-visit(nodeList, visited, start);
           schedule.prepend(v);
     }
  }
```

Is a topological ordering unique?

1. Yes
2. No
3. Only on Wednesdays.

Post-Order Depth-First Search

1. **on oven**
2. **shop**
3. beat
4. measure
5. mix
6. **clean**
7. in oven
8. **set timer**
9. take out



Topological Sort

Input:

- Directed Acyclic Graph (DAG)

Output:

- Total ordering of nodes, where all edges point forwards.

Algorithm:

- Post-order Depth-First Search
- $O(V + E)$ time complexity

Topological Sort

Alternative algorithm:

Input: directed graph G

Repeat:

- S = all nodes in G that have *no* incoming edges.
- Add nodes in S to the topo-order
- Remove all edges adjacent to nodes in S
- Remove nodes in S from the graph

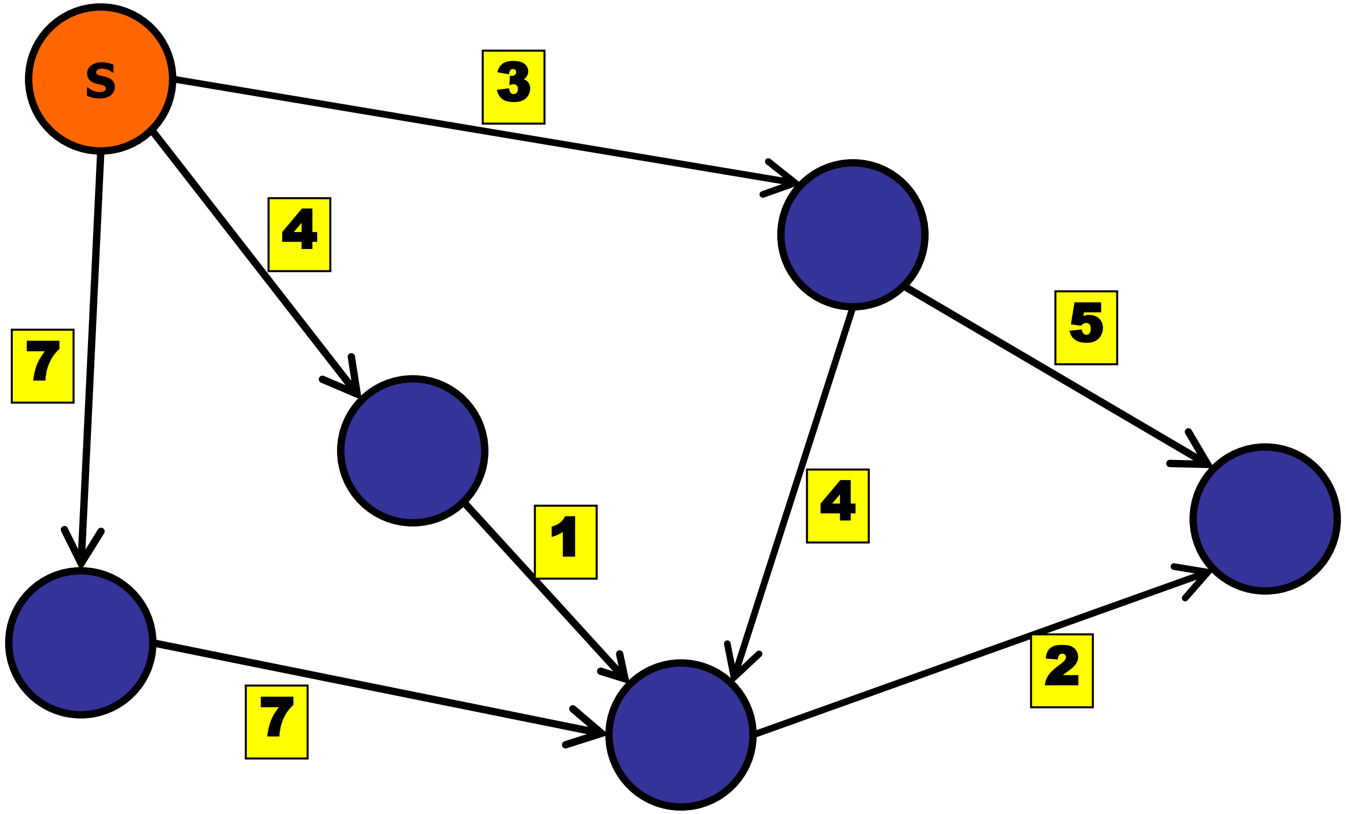
Time:

- O(V + E) time complexity

Special Cases

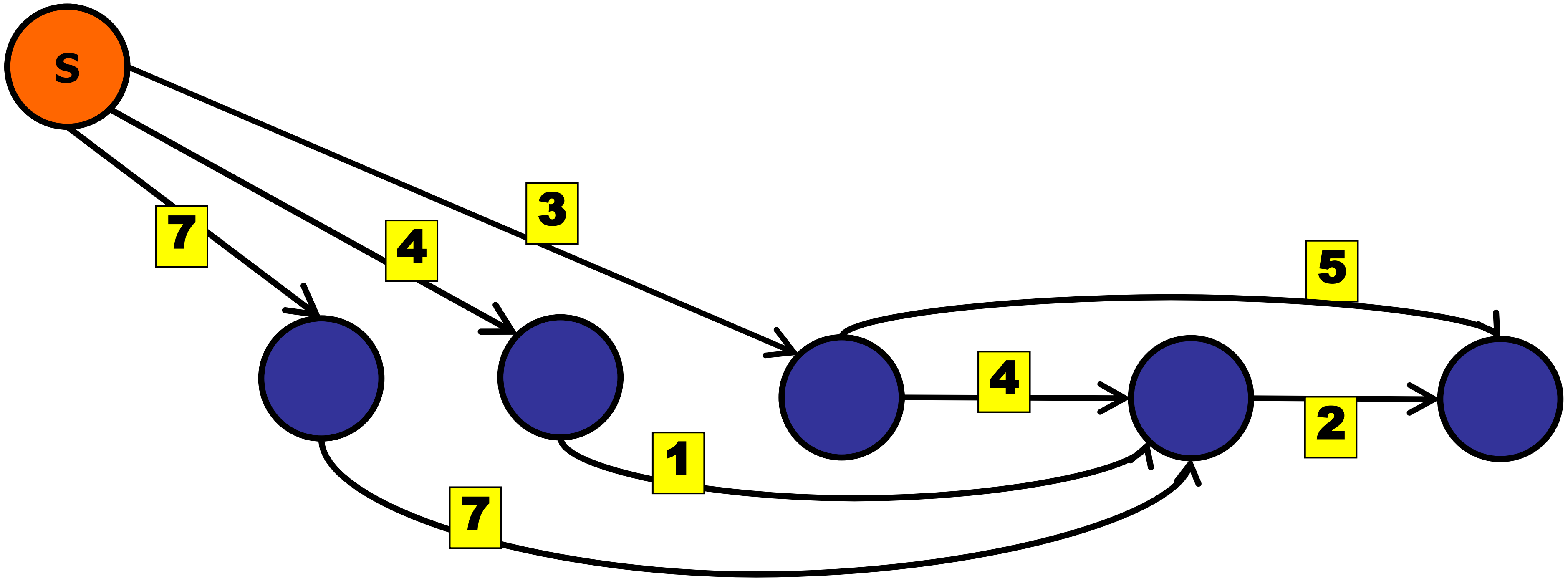
Condition	Algorithm	Time Complexity
No Negative Weight Cycles	Bellman-Ford Algorithm	$O(VE)$
On Unweighted Graph (or equal weights)	BFS	$O(V+E)$
No Negative Weights	Dijkstra's Algorithm	
On Tree	BFS / DFS order	$O(V)$
On DAG	Topological sort order	$O(V+E)$

Directed Acyclic Graph (DAG)



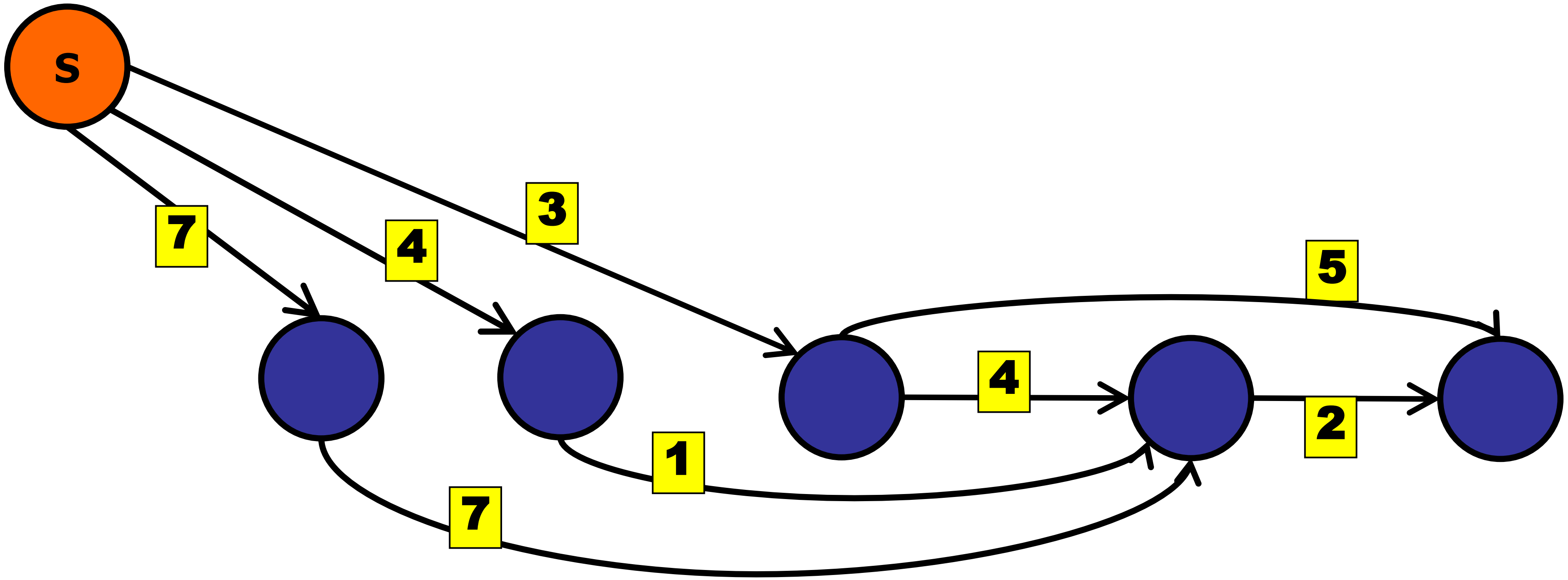
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



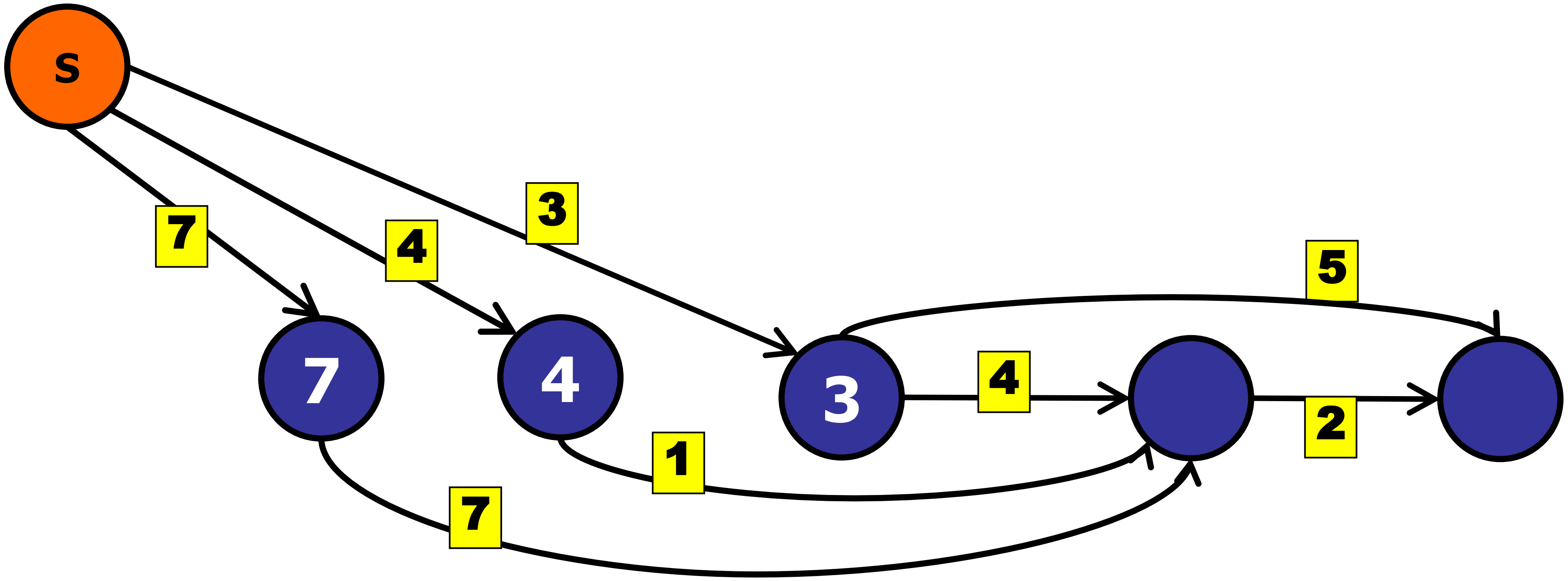
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



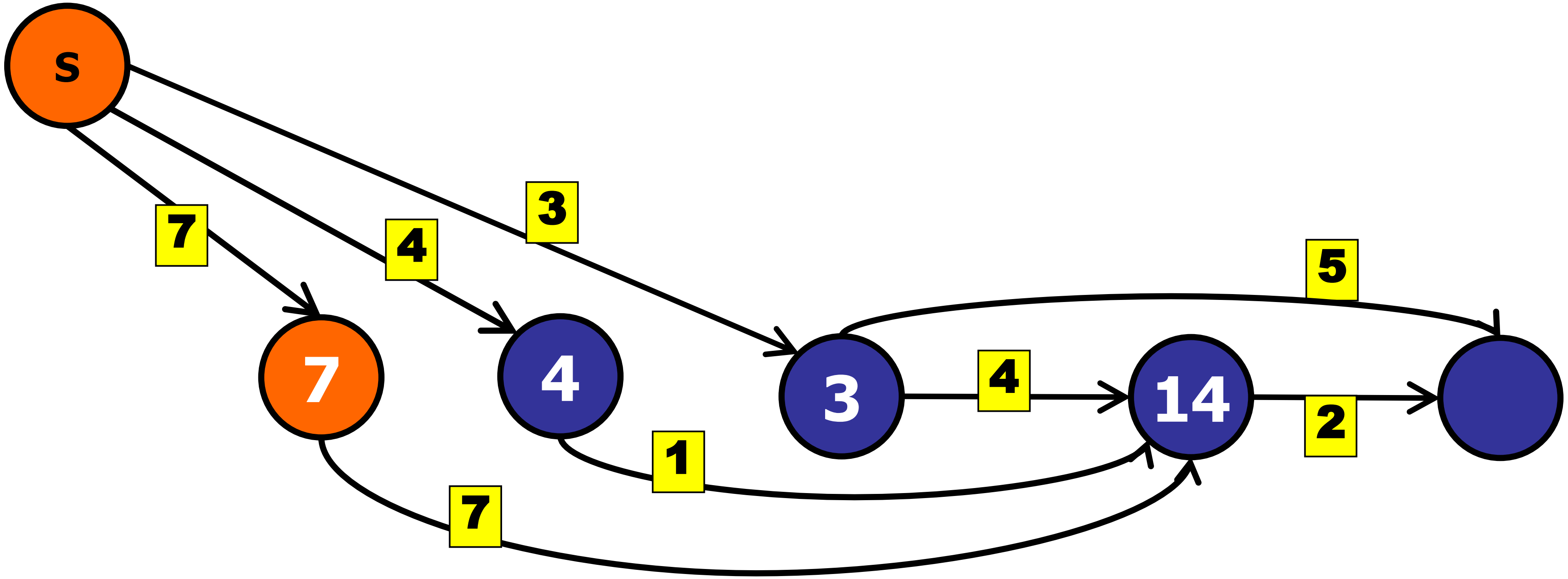
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



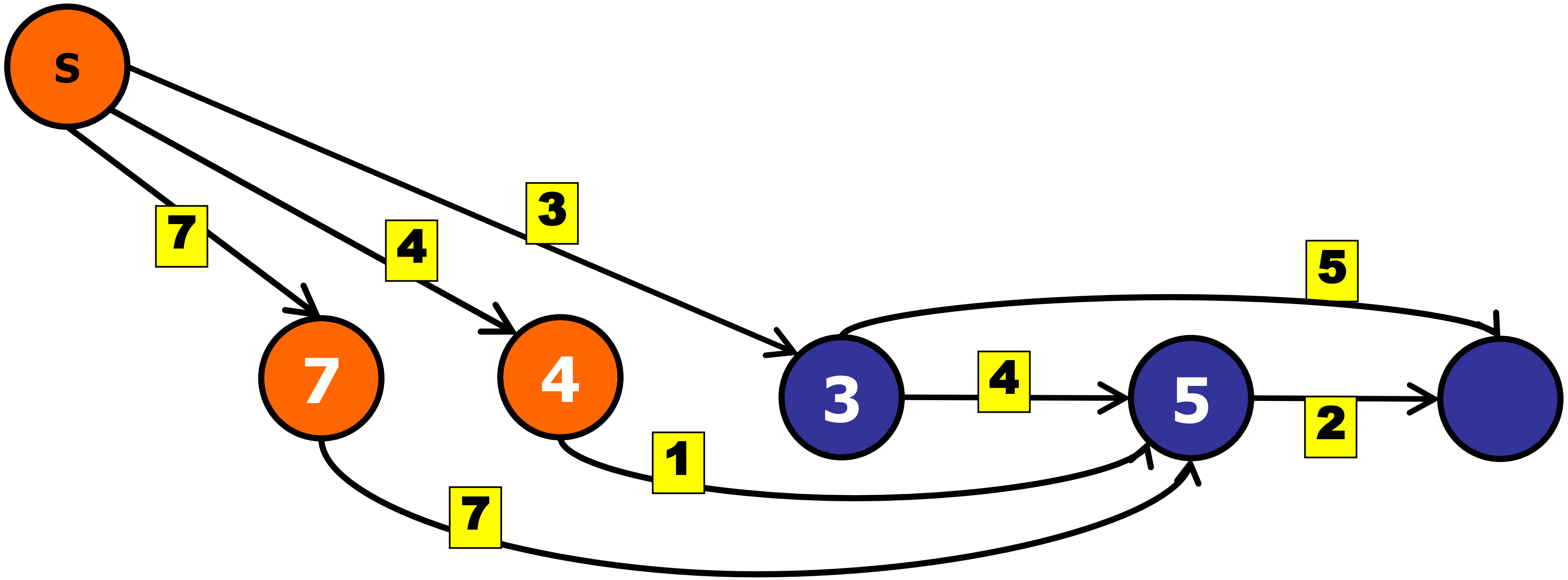
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



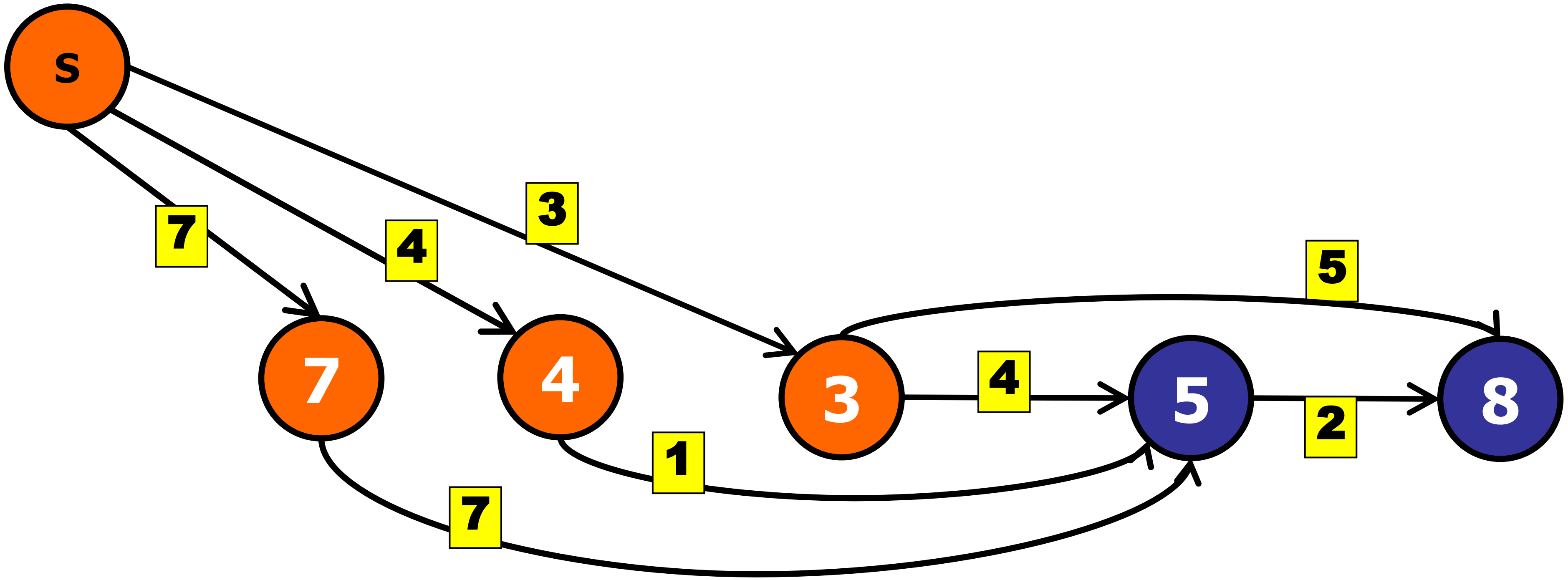
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



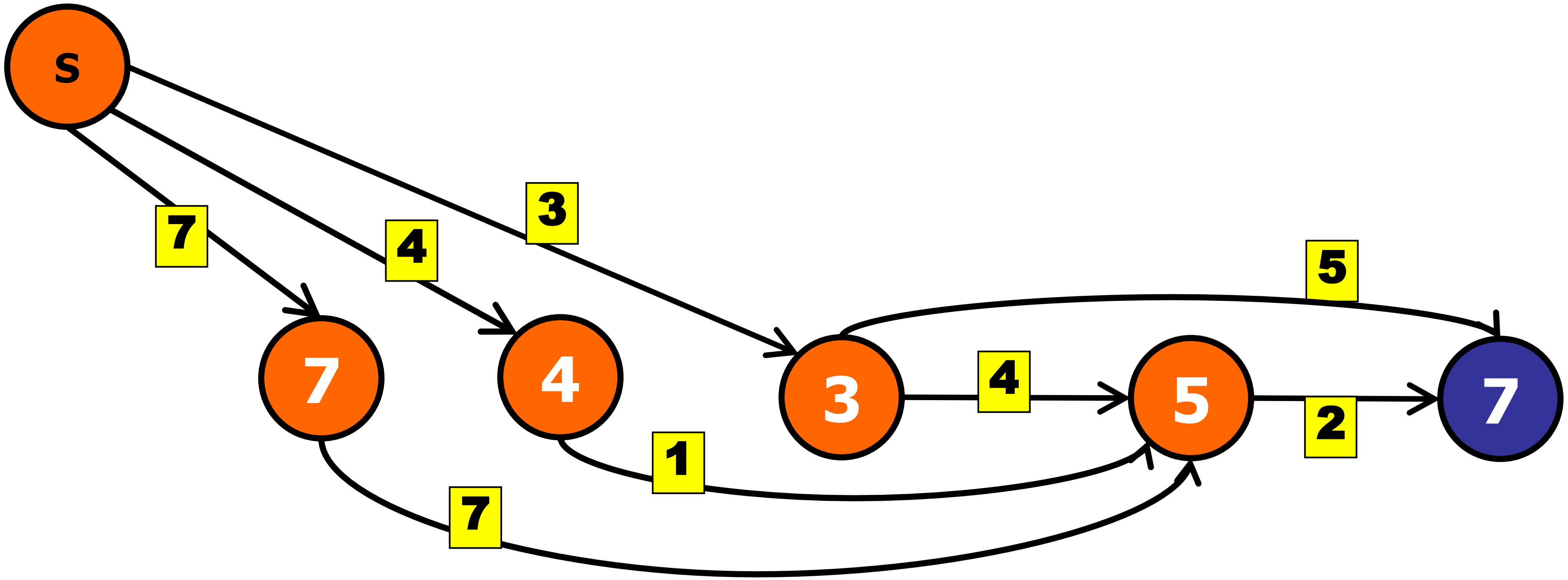
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



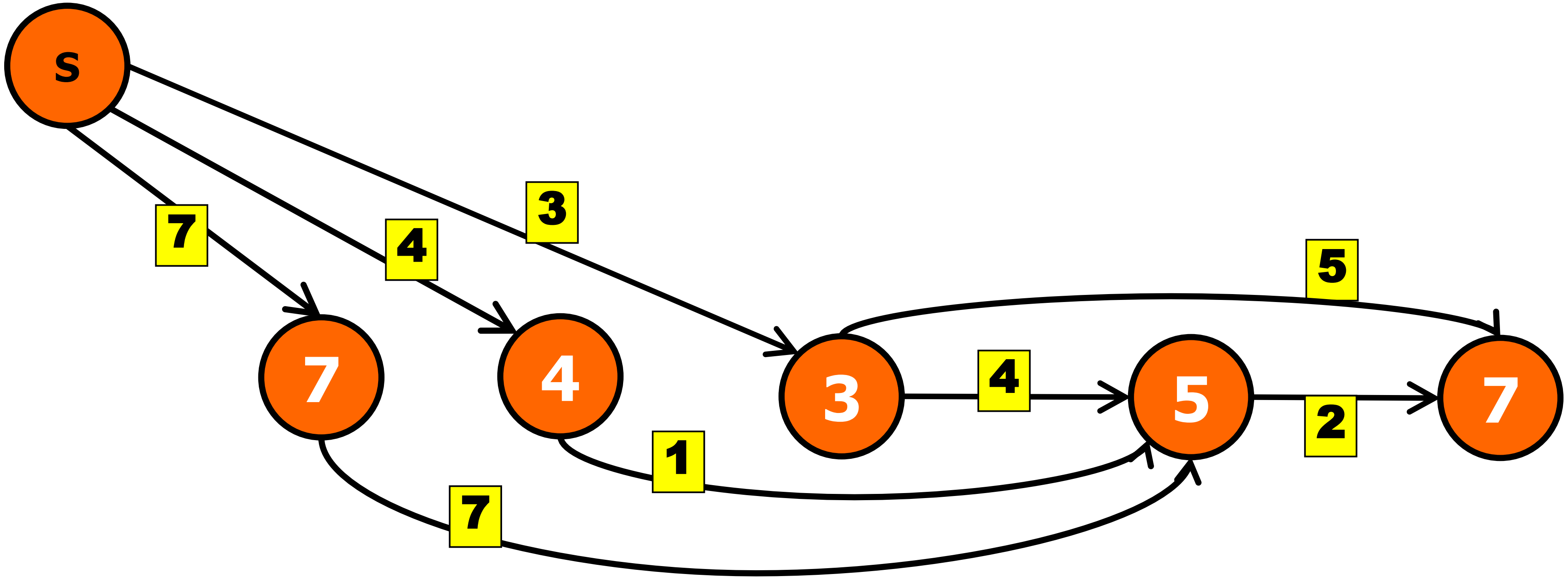
Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.

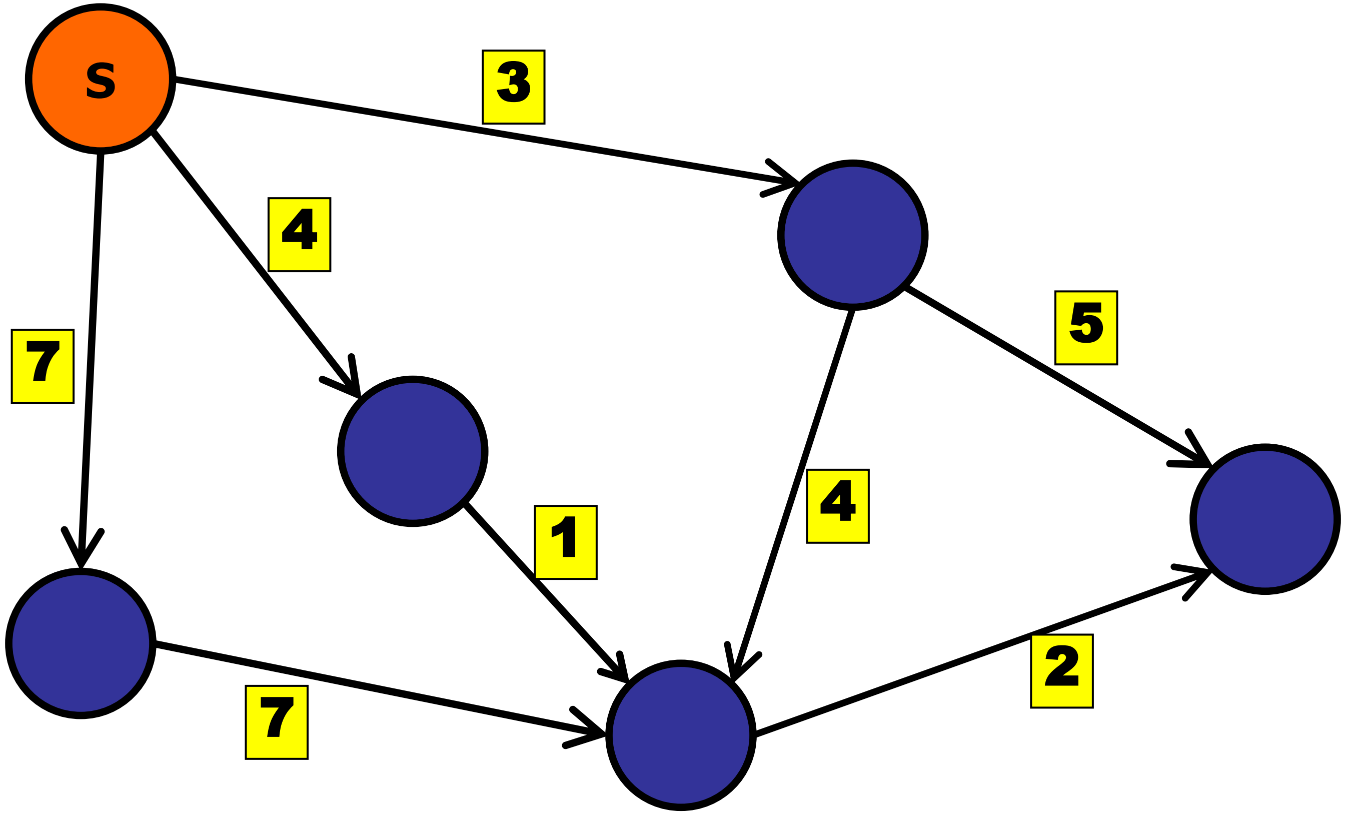


Directed Acyclic Graph (DAG)

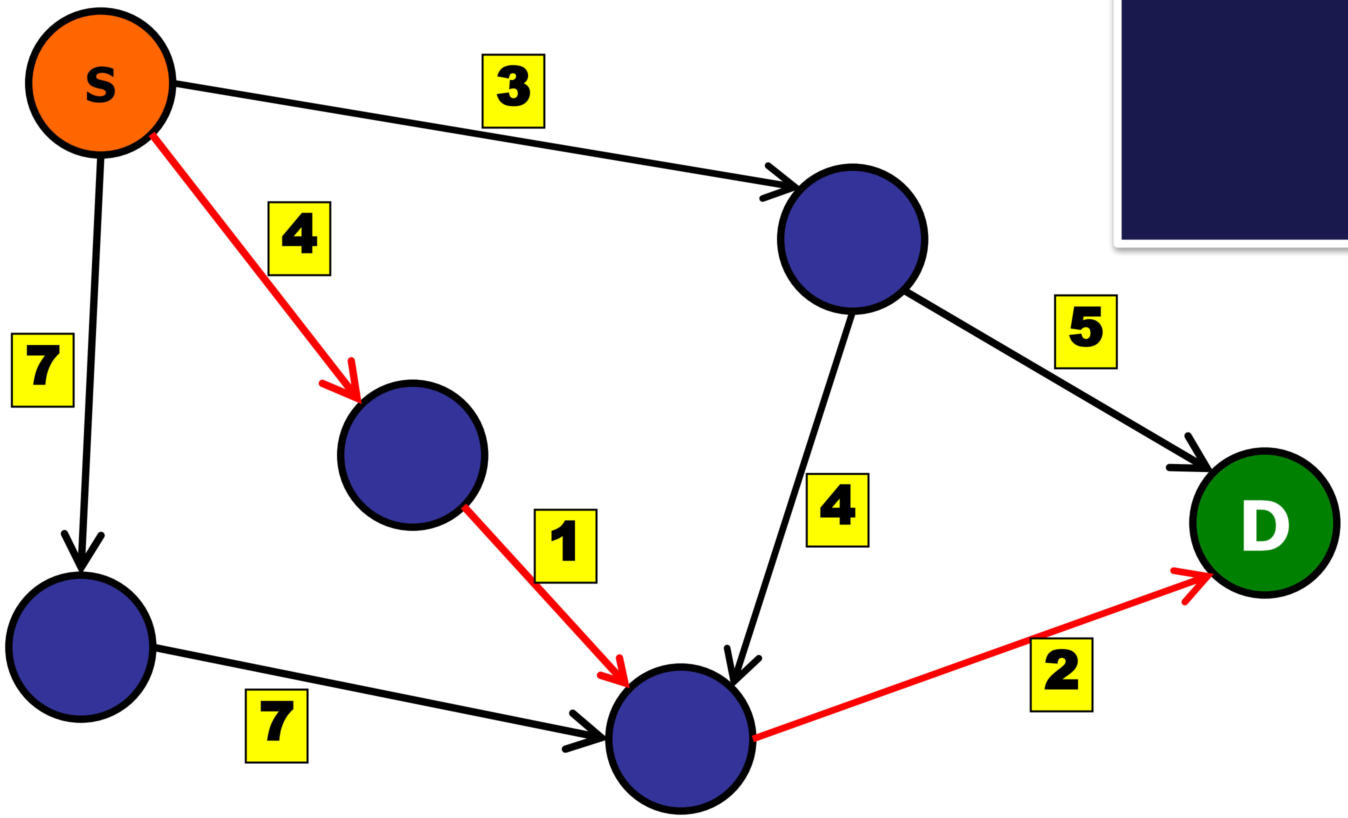
1. Topological sort
2. Relax in order.



Why topological order?

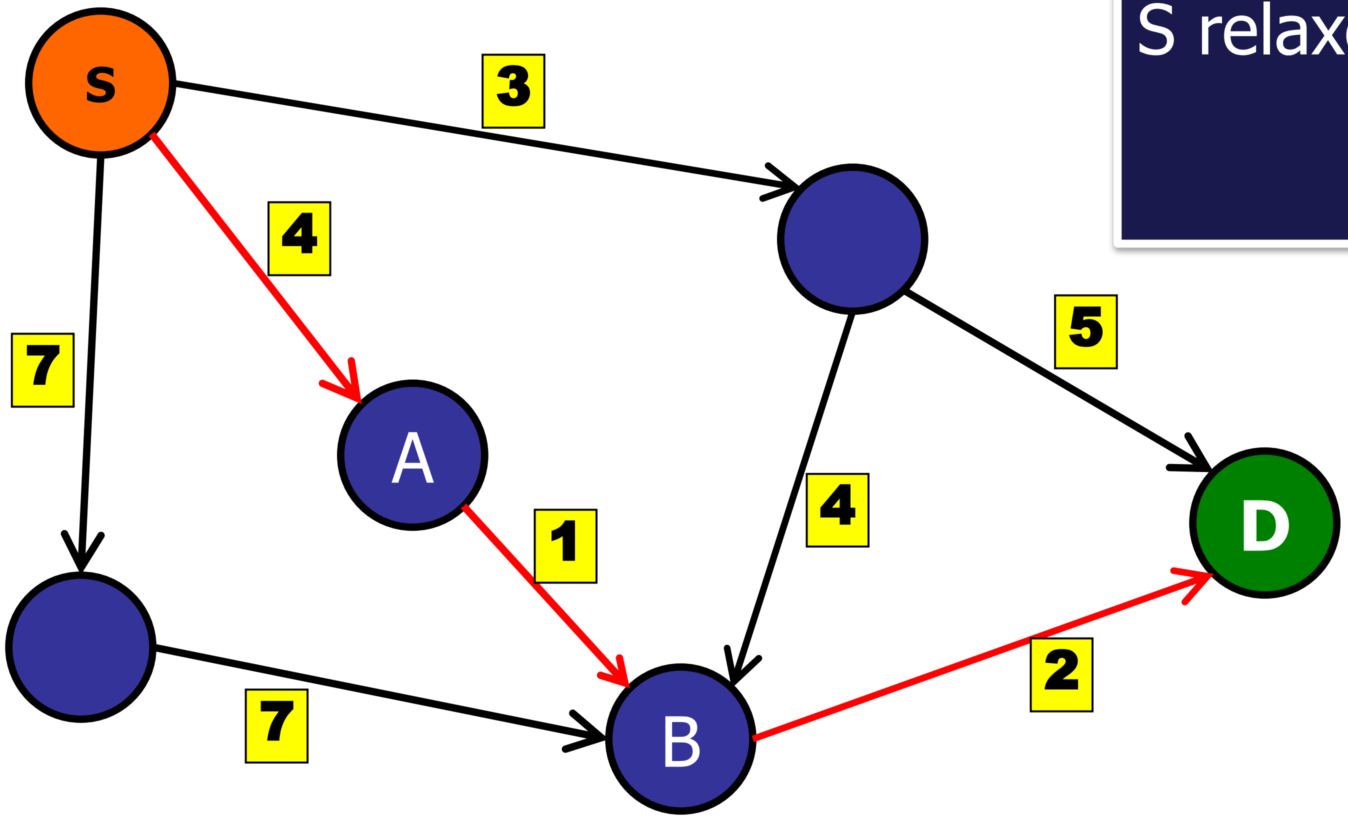


why topological order?



Fix S-D shortest path.

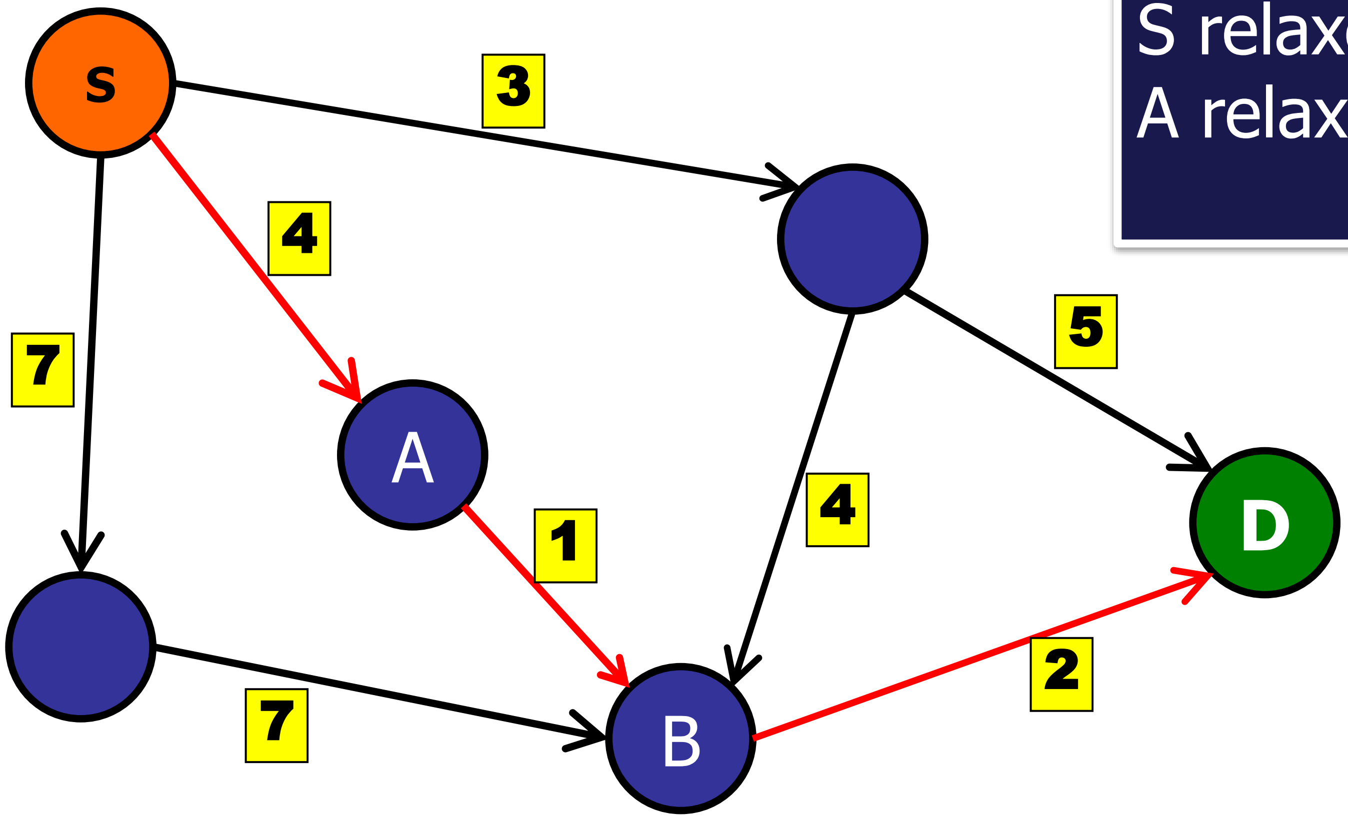
Why topological order?



Fix S-D shortest path.

S relaxed before A.

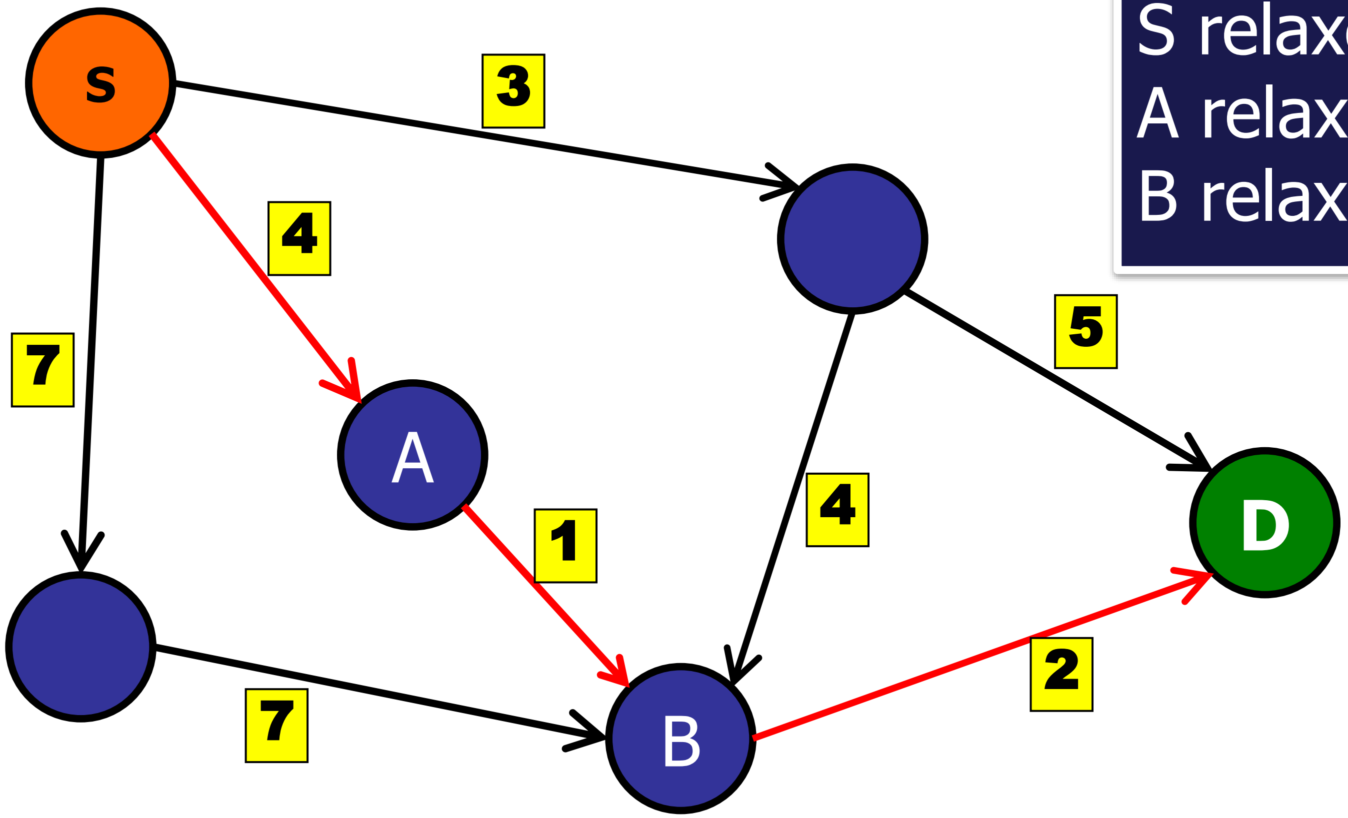
Why topological order?



Fix S-D shortest path.

S relaxed before A.
A relaxed before B.

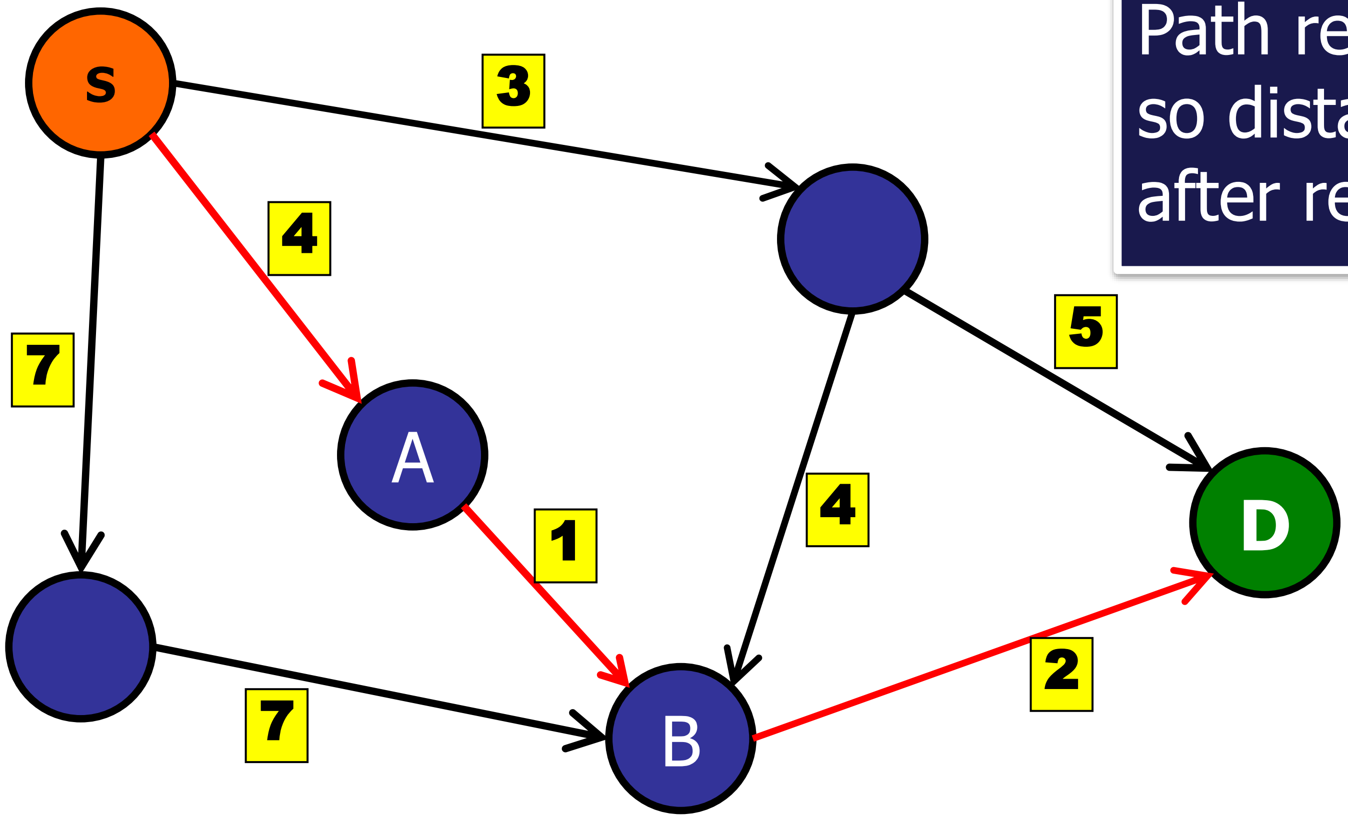
Why topological order?



Fix S-D shortest path.

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

Why topological order?



Fix S-D shortest path.

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