COP 3530: Computer Science III
Summer 2005

Graphs and Graph Algorithms – Part 4

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790

http://www.cs.ucf.edu/courses/cop3530/summer05

School of Computer Science
University of Central Florida

All-Pairs Shortest Paths

- Dijkstra's algorithm was a single source shortest path algorithm.
- Rather than defining one vertex as a starting vertex, suppose that we would like to find the distance between every pair of vertices in a weighted graph G. In other words, the shortest path from every vertex to every other vertex in the graph.
- One option would be to run Dijkstra's algorithm in a loop considering each vertex once as the starting point. A better option would be to use the Floyd-Warshall dynamic programming algorithm (typically referred to as Floyd's algorithm).

Floyd's Shortest Path Algorithm

- Idea #1: Number the vertices 1, 2, …, n.
- Idea #2: Shortest path between vertex *i* and vertex *j* without passing through any other vertex is **weight(edge(i,j))**.
  - Let it be $\mathbf{D^0(i,j)}$. (note: textbook uses A^k, I use D^k)
- Idea #3: If $\mathbf{D^k(i,j)}$ is the shortest path between vertex *i* and vertex *j* using vertices numbered 1, 2, …, k, as intermediate vertices, then $\mathbf{D^n(i,j)}$ is the solution to the shortest path problem between vertex *i* and vertex *j*.

Floyd's Shortest Path Algorithm (cont.)

Assume that we have $\mathbf{D^{k-1}(i,j)}$ for all i, and j.

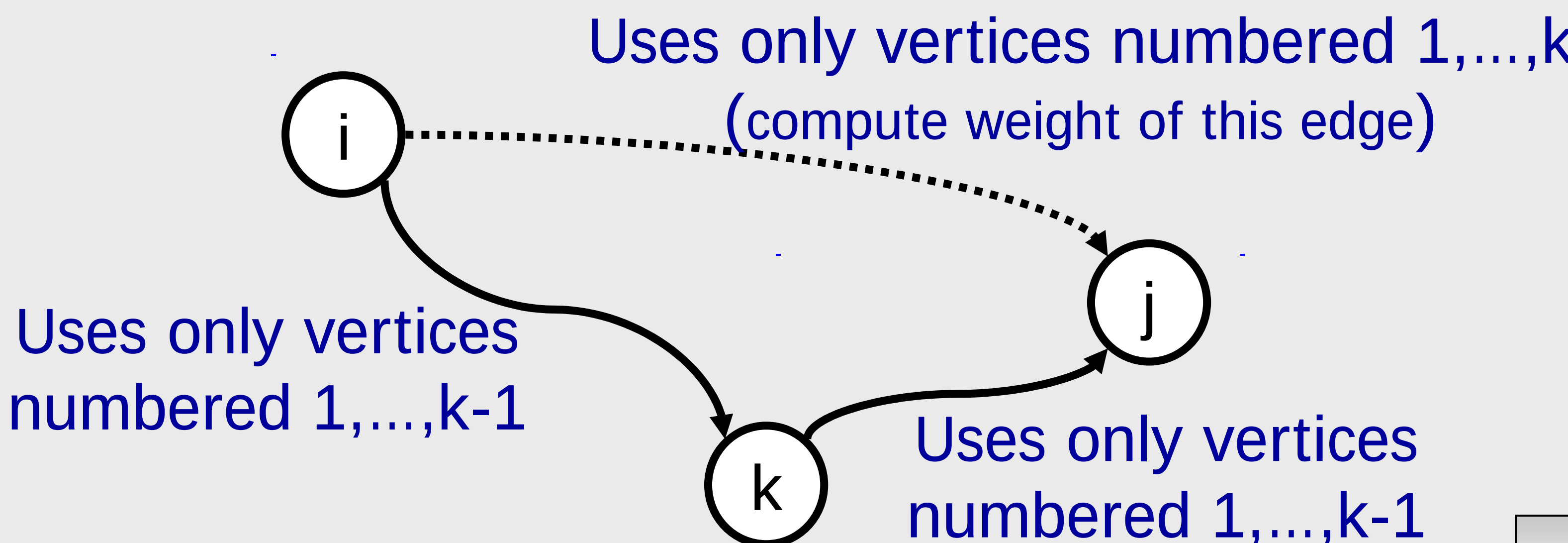
We wish to compute $\mathbf{D^{k}(i,j)}$. What are the possibilities? Choose between:

(1) a path through vertex k.

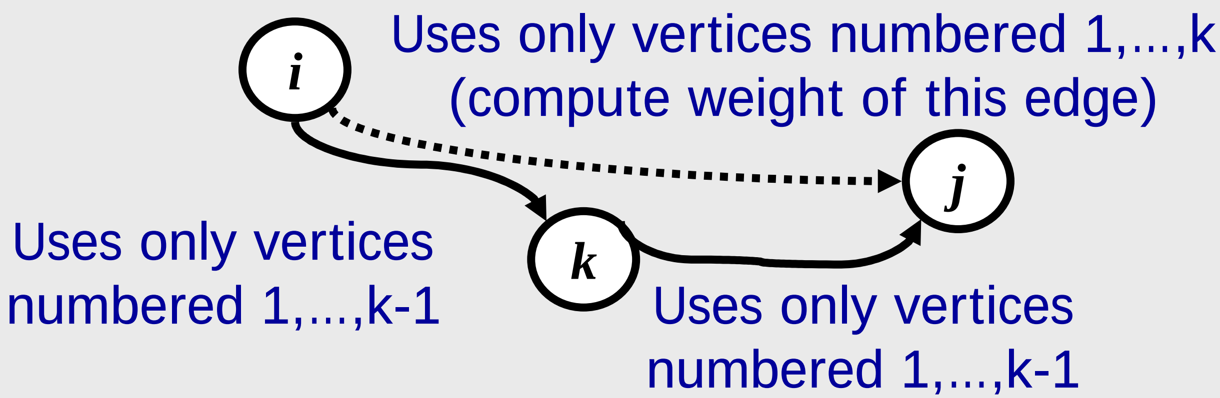
Path Length: $\mathbf{D^{k}(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)}$.

(2) Skip vertex k altogether.

Path Length: $\mathbf{D^{k}(i,j) = D^{k-1}(i,j)}$.



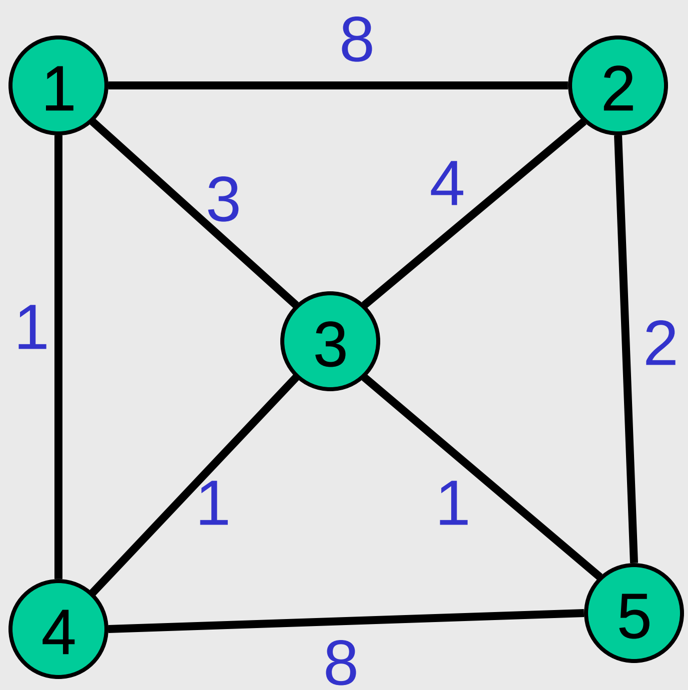
Floyd's Shortest Path Algorithm (cont.)



$$D^{0}_{i,j} = \begin{cases} 0 & \text{if } i = j \\ weight\big(edge(v_i, v_j)\big) & \text{if } (v_i, v_j) \text{ is an edge} \\ \infty & \text{otherwise} \end{cases}$$

$$D^{k}_{i,j} = \min\left\{ D^{k-1}_{i,j}, \quad D^{k-1}_{i,k} + D^{k-1}_{k,j} \right\}$$

Floyd's Shortest Path Algorithm EXAMPLE

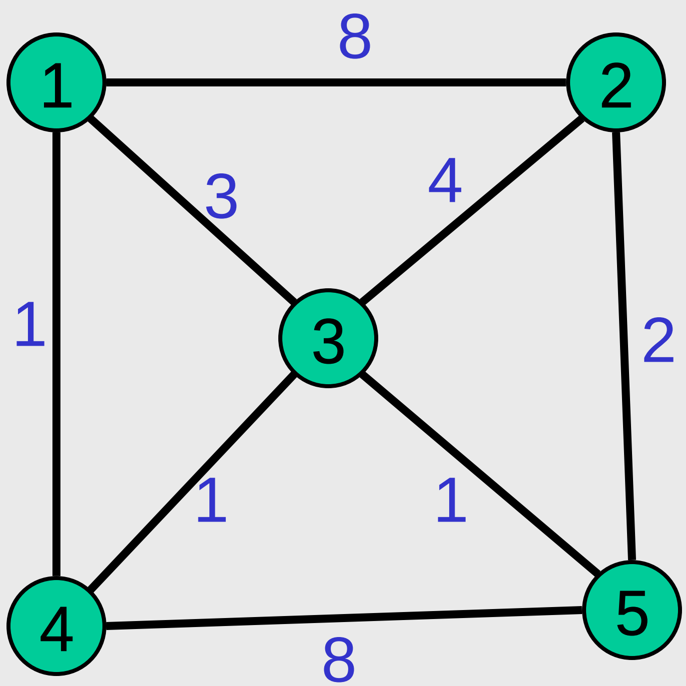


D^0

	1	2	3	4	5
1	0	8	3	1	∞
2	8	0	4	∞	2
3	3	4	0	1	1
4	1	∞	1	0	8
5	∞	2	1	8	0

Adjacency matrix

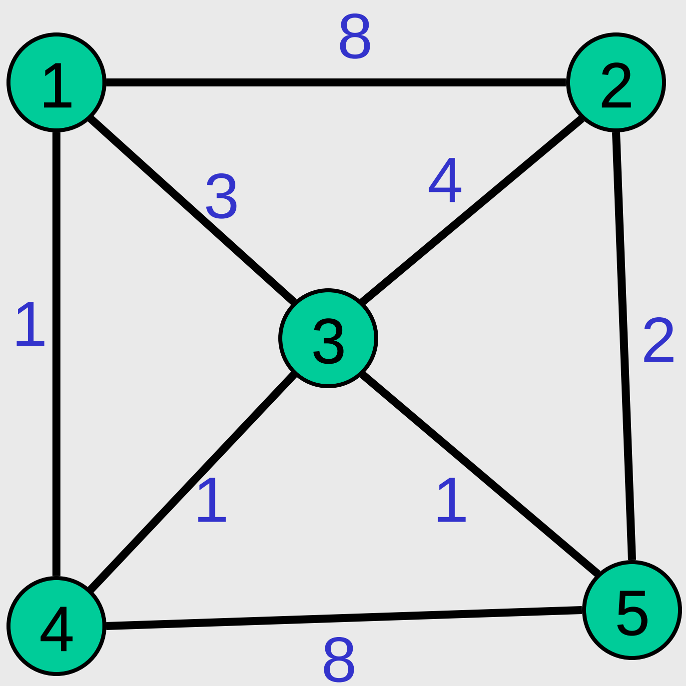
Floyd's Shortest Path Algorithm EXAMPLE (cont.)



D^1

	1	2	3	4	5
1	0	8	3	1	∞
2	8	0	4	9	2
3	3	4	0	1	1
4	1	9	1	0	8
5	∞	2	1	8	0

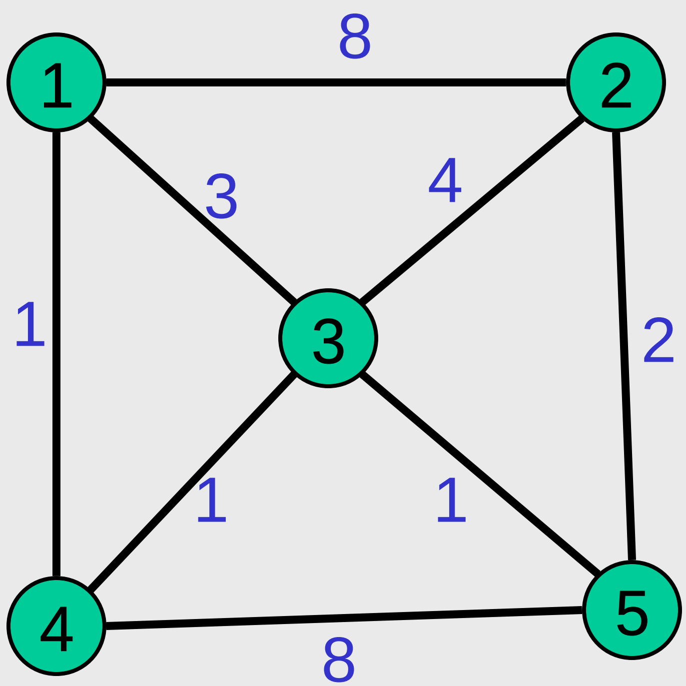
Floyd's Shortest Path Algorithm EXAMPLE (cont.)



D^2

	1	2	3	4	5
1	0	8	3	1	10
2	8	0	4	9	2
3	3	4	0	1	1
4	1	9	1	0	8
5	10	2	1	8	0

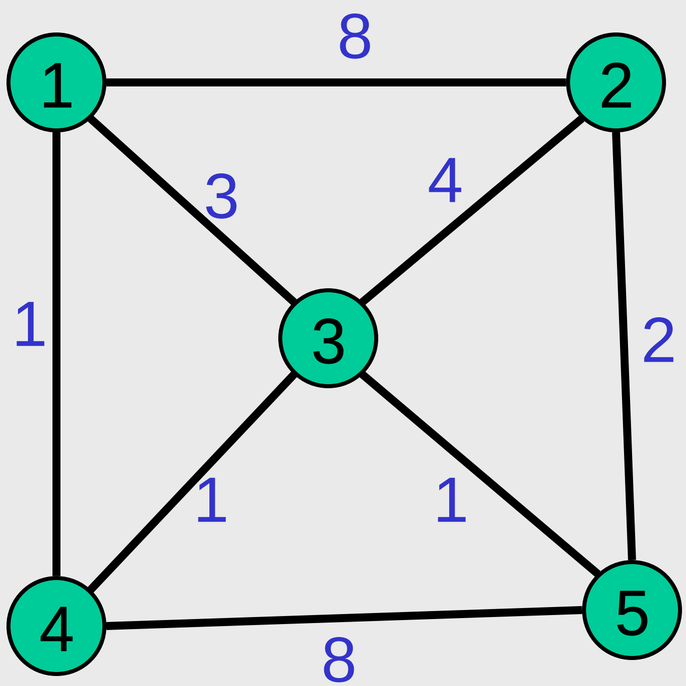
Floyd's Shortest Path Algorithm EXAMPLE (cont.)



D^3

	1	2	3	4	5
1	0	7	3	1	4
2	7	0	4	5	2
3	3	4	0	1	1
4	1	5	1	0	2
5	4	2	1	2	0

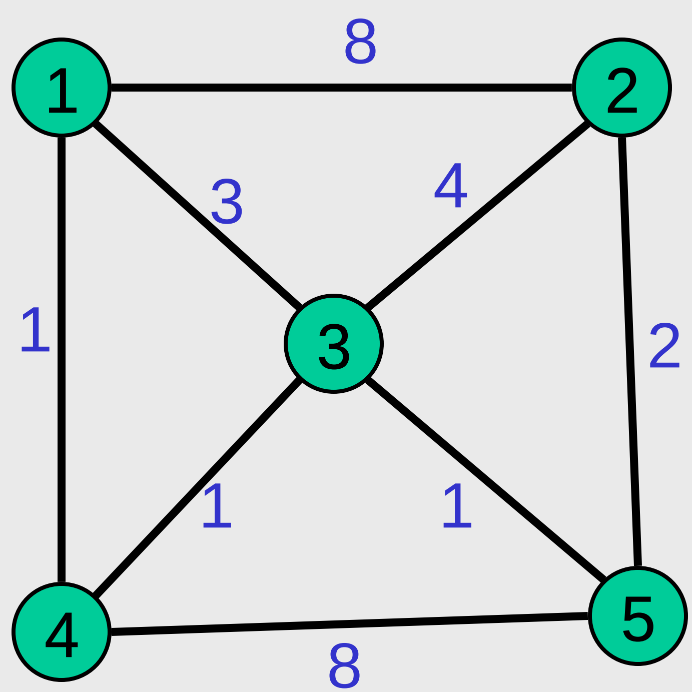
Floyd's Shortest Path Algorithm EXAMPLE (cont.)



D^4

	1	2	3	4	5
1	0	6	2	1	3
2	6	0	4	5	2
3	2	4	0	1	1
4	1	5	1	0	2
5	3	2	1	2	0

Floyd's Shortest Path Algorithm EXAMPLE (cont.)



D^5

	1	2	3	4	5
1	0	5	2	1	3
2	5	0	3	4	2
3	2	3	0	1	1
4	1	4	1	0	2
5	3	2	1	2	0

Floyd's Shortest Path Algorithm

Algorithm ***AllPair***(***G***) {assumes vertices 1,…,***n***}

for all vertex pairs (i,j)
  if $i = j$
    $d^0[i,i] \leftarrow 0$
  else if **(*i,j*) is an edge in *G***
    $d^0[i,j] \leftarrow$ ***weight of edge (i,j)***
  else $d^0[i,j] \leftarrow +\infty$

O(n+m)
or
O(n^2)

for $k \leftarrow 1$ ***to*** n ***do***
  for $i \leftarrow 1$ ***to*** n ***do***
    for $j \leftarrow 1$ ***to*** n ***do***
      $d^k[i,j] \leftarrow \min(d^{k-1}[i,j], d^{k-1}[i,k]+d^{k-1}[k,j])$

O(n^3)

Floyd's Shortest Path Algorithm

Observation:

$$\boldsymbol{d}^{\,k}\,[\boldsymbol{i,k}] = \boldsymbol{d}^{\,k-1}\,[\boldsymbol{i,k}]$$

$$\boldsymbol{d}^{\,k}\,[\boldsymbol{k,j}] = \boldsymbol{d}^{\,k-1}\,[\boldsymbol{k,j}]$$

This observation leads to the following simplification of the algorithm:

Floyd's Shortest Path Algorithm - Modified

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all  vertex pairs (i,j)
   if  i = j
      d[i,i] ← 0
   else if (i,j) is an edge in G
      d[i,j] ← weight of edge (i,j)
   else  d[i,j] ← + ∞
 for k ← 1 to n do
   for i ← 1 to n do
      for j ← 1 to n do
         d[i,j] ← min(d[i,j], d[i,k]+d[k,j])
```

Floyd's Shortest Path Algorithm - Modified

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all  vertex pairs (i,j)
  path[i,j] ← null
  if  i = j
    d[i,i] ← 0
  else if (i,j) is an edge in G
    d[i,j] ← weight of edge (i,j)
  else  d[i,j] ← + ∞
 for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
       if (d[i,k]+d[k,j] < d[i,j])
          path[i,j] ← vertex k
          d[i,j] ← d[i,k]+d[k,j]
```


Floyd’s Shortest Path Algorithm

- Can be applied to Directed Graphs
- Can accept negative weight edges as long as there are no negative weight cycles.

Transitive Closure

- A relation $\boldsymbol{R}$ on a set $\boldsymbol{A}$ is called transitive if and only if for any $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ in $\boldsymbol{A}$, whenever $\boldsymbol{<a, b>} \in \boldsymbol{R}$, and $\boldsymbol{<b, c>} \in \boldsymbol{R}$, $\boldsymbol{<a, c>} \in \boldsymbol{R}$.

- The transitive closure of a relation $\boldsymbol{R}$ is the smallest (in the sense of set containment) transitive relation containing $\boldsymbol{R}$.

- More formally: Let $\boldsymbol{R}$ be a relation on set $\boldsymbol{X}$. The transitive closure of $\boldsymbol{R}$ is a relation $\boldsymbol{R^T}$ on **X** satisfying:

1. $\boldsymbol{R \subseteq R^T}$
2. $\boldsymbol{R^T}$ is transitive
3. If $\boldsymbol{T}$ is any transitive relation on $\boldsymbol{X}$ and $\boldsymbol{R \subseteq T}$, then $\boldsymbol{R^T \subseteq T}$

Transitive Closure

- If R is a relation on set X, the transitive closure of R is the set:

$$R^T = \{(x_1, x_k) \mid (x_1, x_2), (x_2, x_3), \ldots, (x_{k-1}, x_k) \in R\}$$

Example:

Let X = {1, 2, 3, 4, 5}

Let R = {(1, 2), (2,3), (4,5), (5,4), (5,5)}

Then

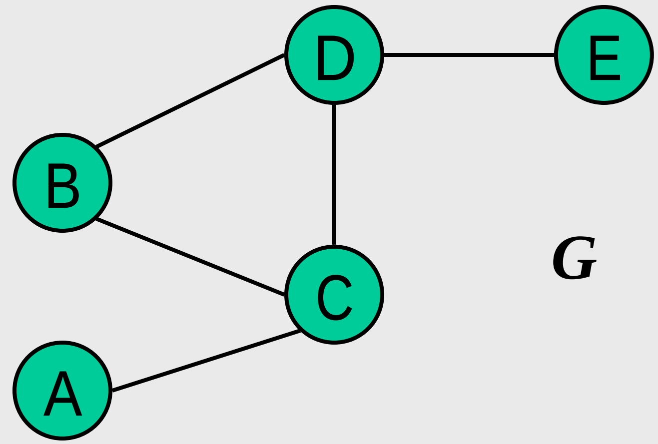
R^T = {(1,2), (1,3), (2,3), (4,4), (4,5), (5,4), (5,5)}

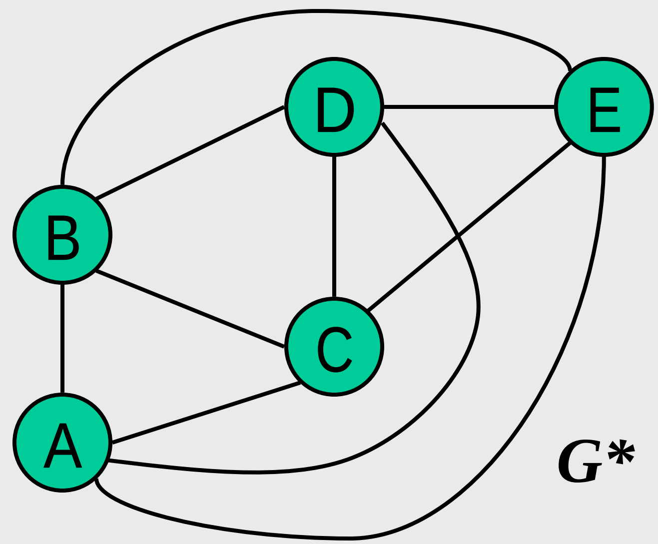
(1,2) and (2,3) $\in$ R $\Rightarrow$ (1,3) $\in R^T$, similarly

(4,5) and (5,4) $\in$ R $\Rightarrow$ (4,4) $\in R^T$

Transitive Closure For Graphs

- Given a graph ***G***, the transitive closure of ***G*** is ***G**** such that
  - ***G**** has the same vertices as ***G***
  - if ***G*** has a path from ***u*** to ***v*** ($u \neq v$), ***G**** has a edge from ***u*** to ***v***





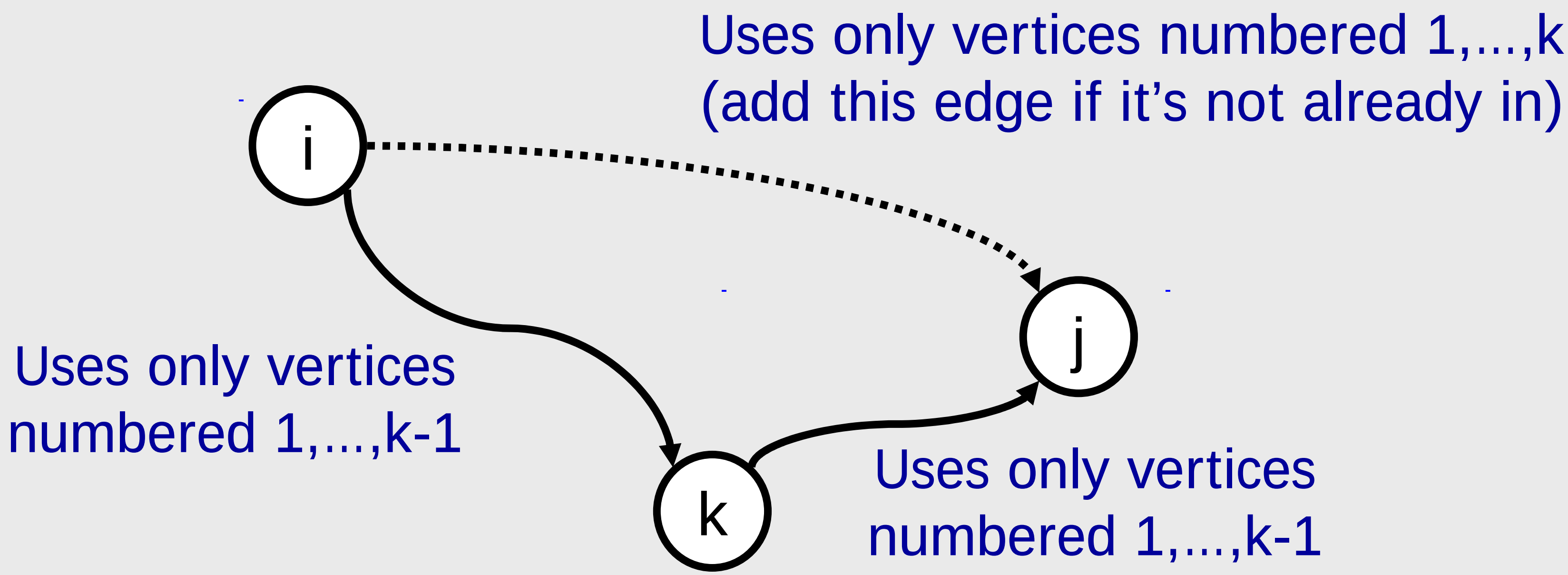
The transitive closure provides reachability information about a graph.

Computing the Transitive Closure

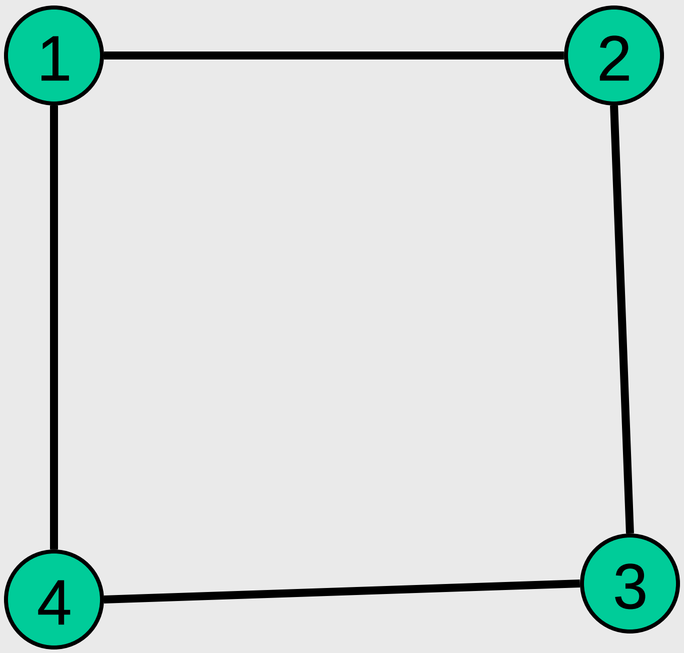
- Using the transitive closure concept, if there is a way to get from vertex A to vertex B and also a way to get from vertex B to vertex C, then there is a way to get from vertex A to vertex C.
- We can perform BFS starting at each vertex, which would give an O(n(n+m)) algorithm.
- A better way is to use Warshall's dynamic programming algorithm.

Floyd-Warshall Transitive Closure

- Idea #1: Number the vertices 1, 2, …, n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, …, k, as intermediate vertices:



Warshall's Transitive Closure - EXAMPLE

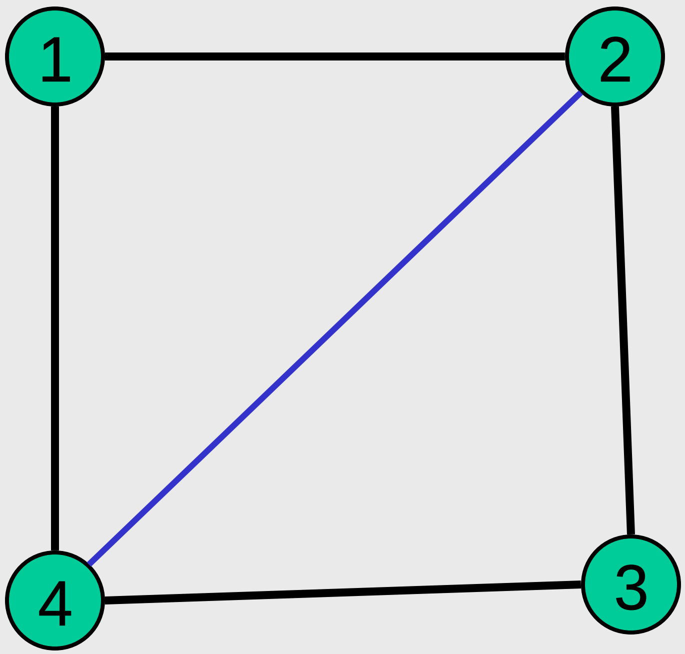


A^0

	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Initial Adjacency Matrix

Warshall's Transitive Closure - EXAMPLE

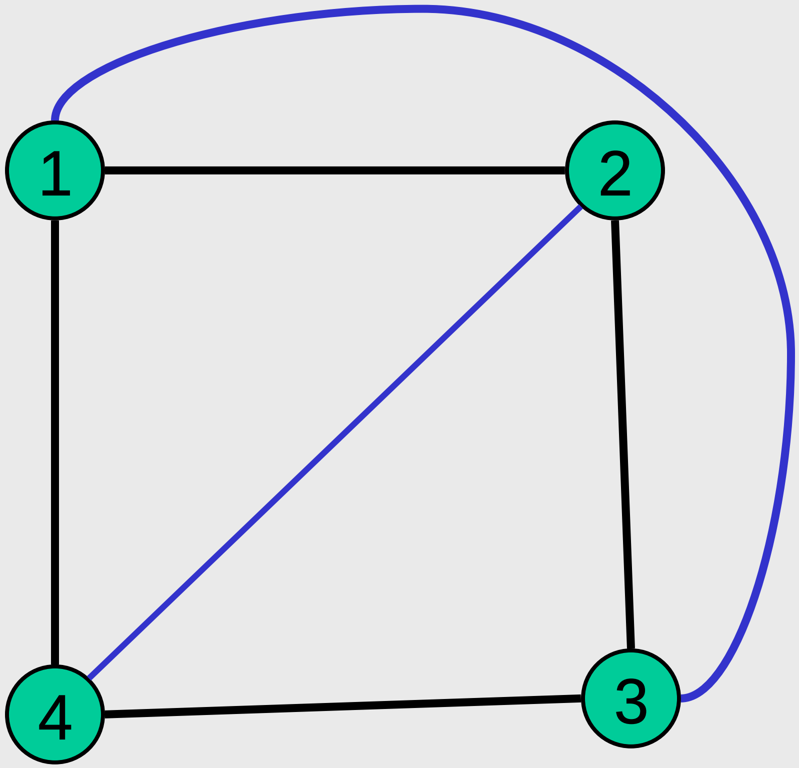


A^1

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	1
4	1	1	1	0

Edges (2,3) and (3,4) $\Rightarrow$ edge (2,4) $\in$ G*

Warshall's Transitive Closure - EXAMPLE

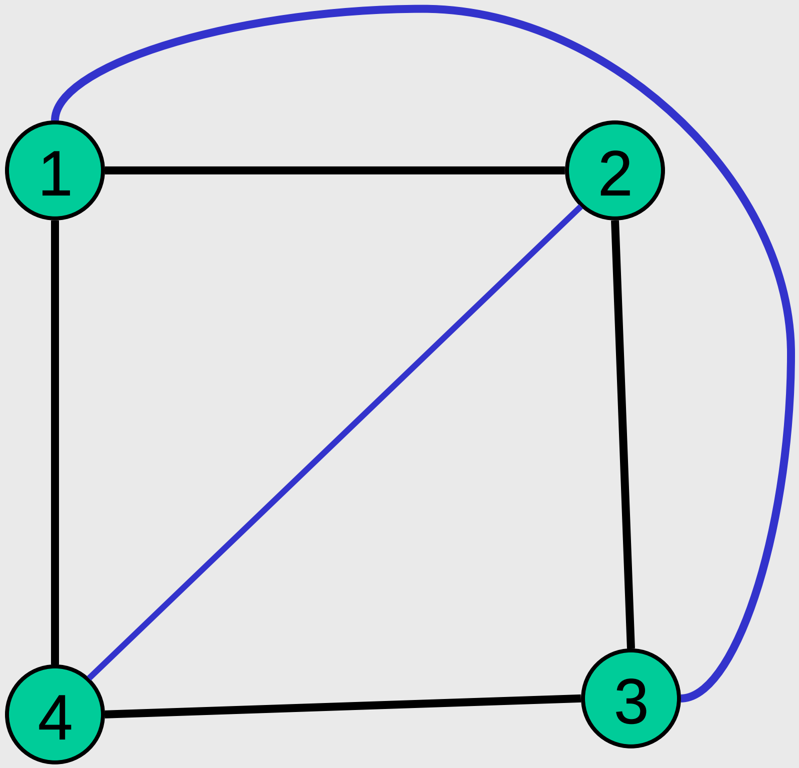


A^2

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Edges (1,2) and (2,3) $\Rightarrow$ edge (1,3) $\in$ G*

Warshall's Transitive Closure - EXAMPLE

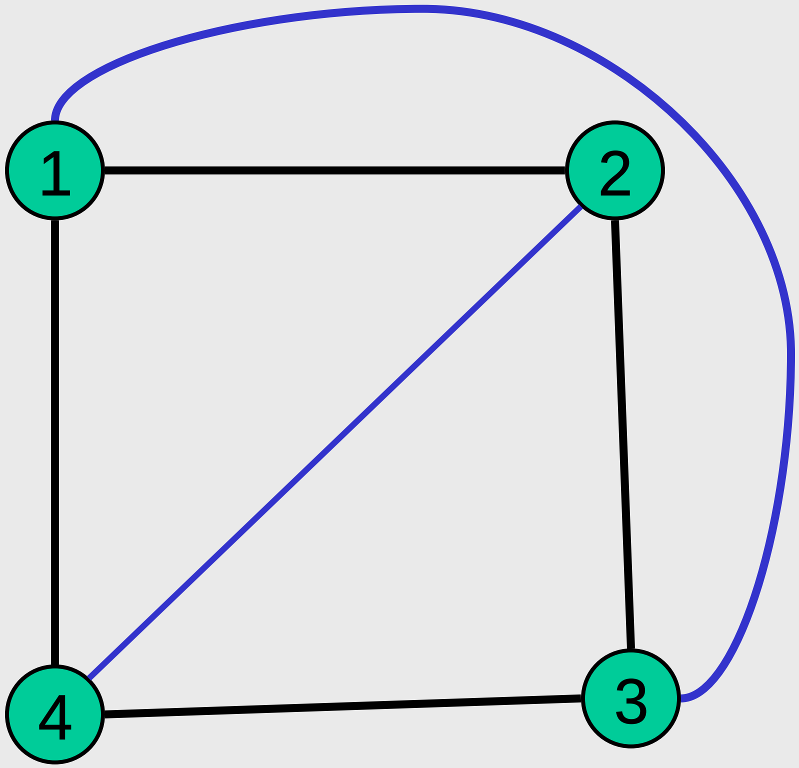


A^3

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

No additional edges added to G*

Warshall's Transitive Closure - EXAMPLE



A^4

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

No additional edges added to G*

Warshall's Algorithm

- Warshall's algorithm numbers the vertices of $\boldsymbol{G}$ as $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ and computes a series of graphs $\boldsymbol{G}_0, \ldots, \boldsymbol{G}_n$
  - $\boldsymbol{G}_0 = \boldsymbol{G}$
  - $\boldsymbol{G}_k$ has a edge $(\boldsymbol{v}_i, \boldsymbol{v}_j)$ if $\boldsymbol{G}$ has a path from $\boldsymbol{v}_i$ to $\boldsymbol{v}_j$ with intermediate vertices in the set $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$
- We have that $\boldsymbol{G}_n = \boldsymbol{G}^*$
- In phase $\boldsymbol{k}$, graph $\boldsymbol{G}_k$ is computed from $\boldsymbol{G}_{k-1}$
- Running time: $O(n^3)$, assuming areAdjacent is $O(1)$ (e.g., adjacency matrix)

```
Algorithm FloydWarshall(G)
   Input graph G
   Output transitive closure G* of G
   i ← 1
   for all v ∈ G.vertices()
      denote v as v_i
      i ← i + 1
   G_0 ← G
   for k ← 1 to n do
      G_k ← G_{k−1}
       for i ← 1 to n (i ≠ k) do
         for j ← 1 to n (j ≠ i, k) do
            if G_{k−1}.areAdjacent(v_i, v_k) ∧
                  G_{k−1}.areAdjacent(v_k, v_j)
               if ¬G_k.areAdjacent(v_i, v_j)
                  G_k.insertEdge(v_i, v_j, k)
      return G_n
```

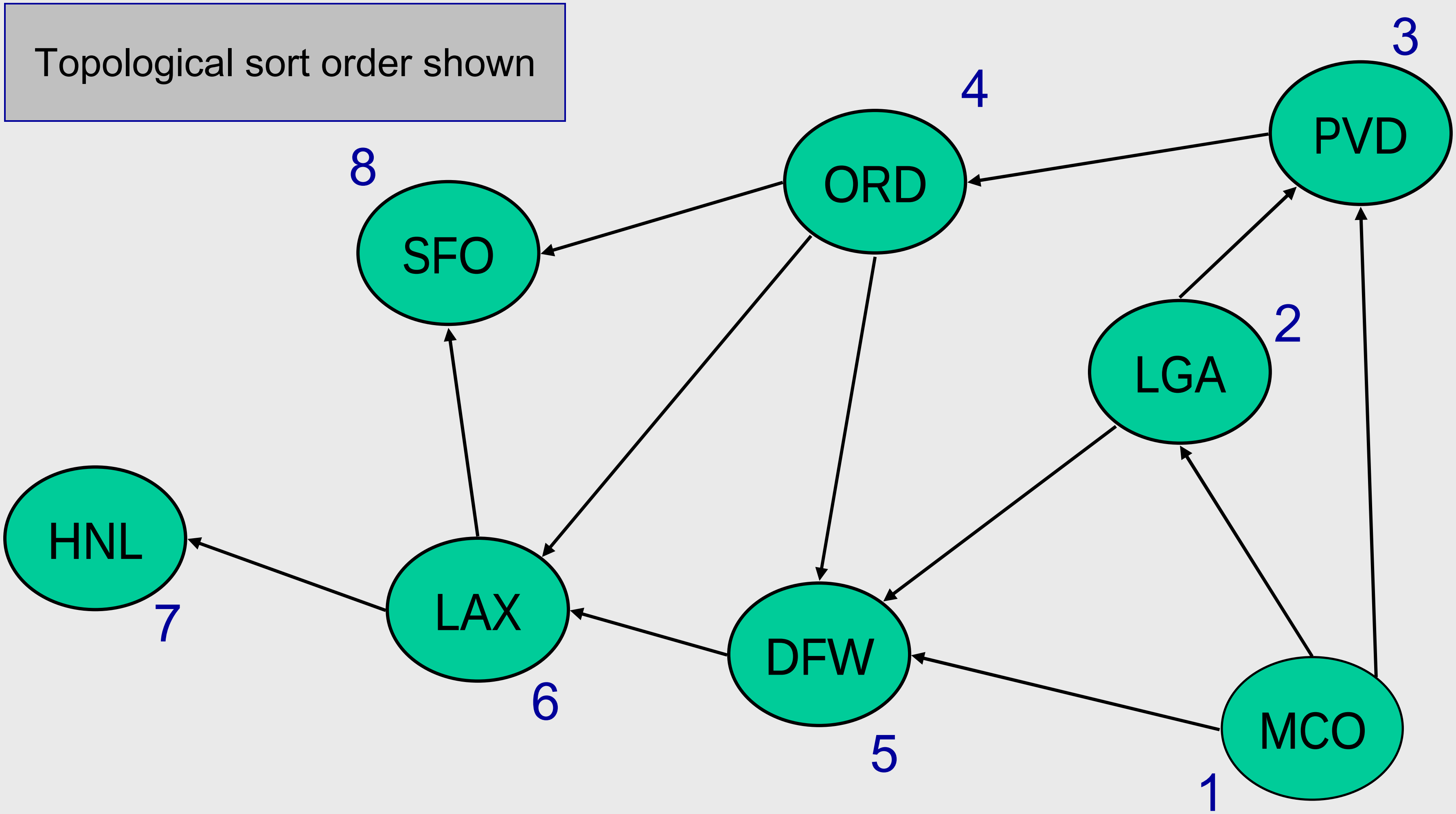
Shortest Path Algorithm for DAGs

Options:

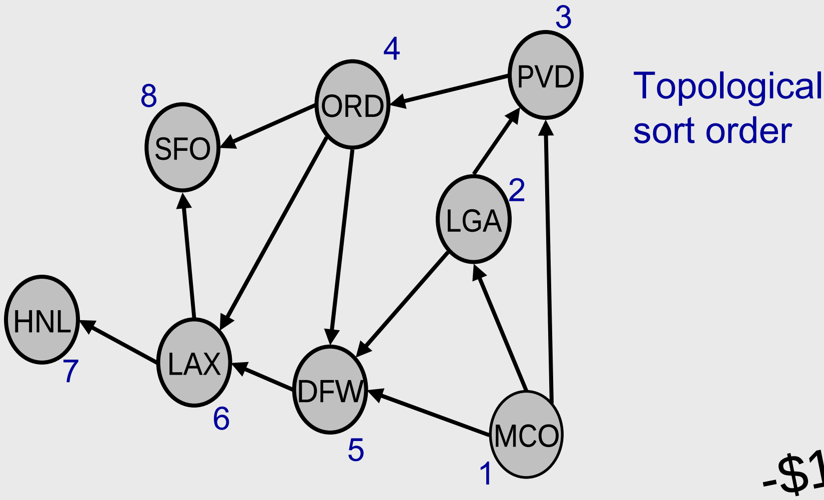
- Use Dijkstra's algorithm
  - Time: O((n+m) log n)
- Use Topological sort based algorithm
  - Time: O(n+m).

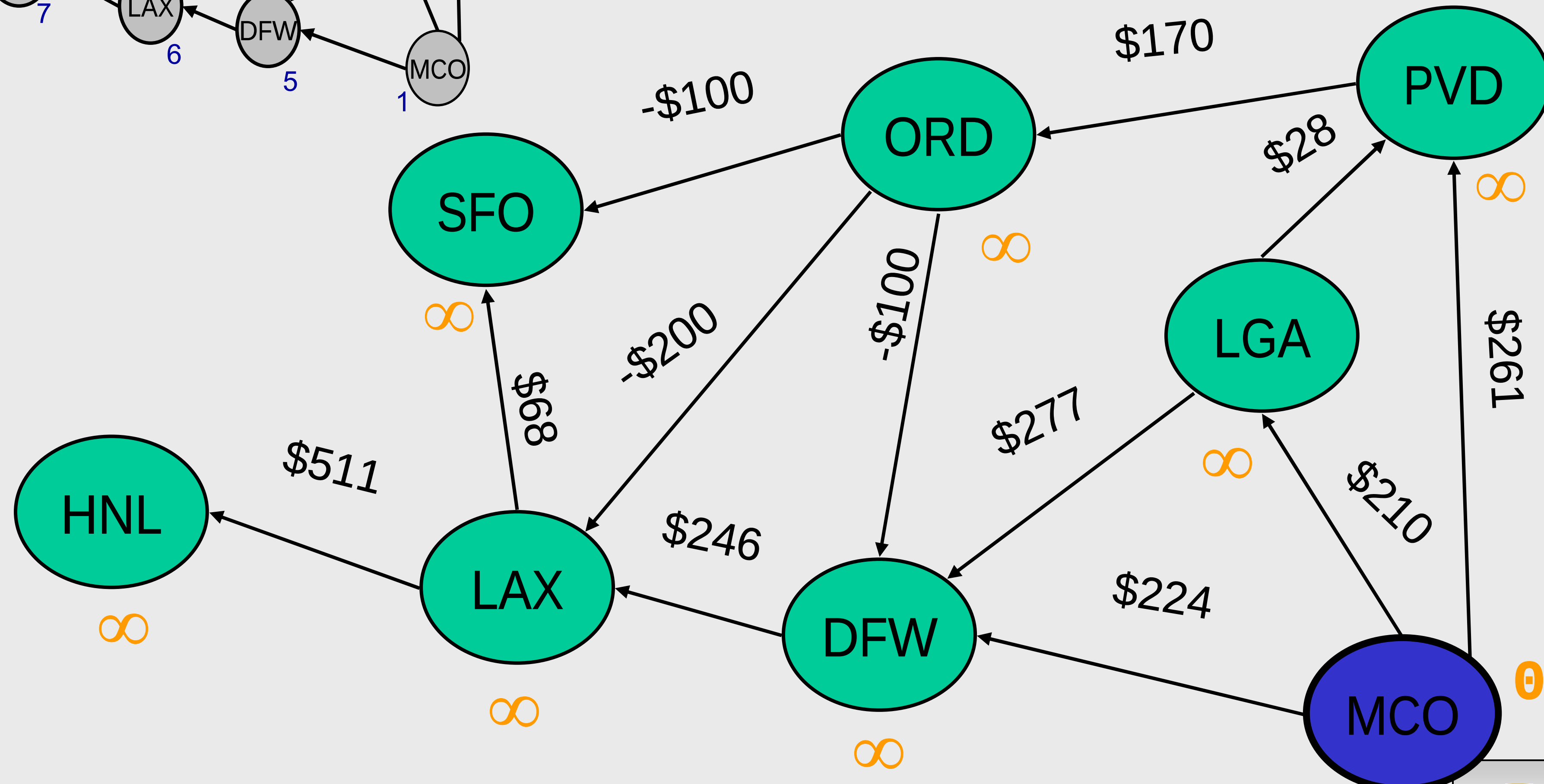
DAG With Negative Weight Edges

Topological sort order shown

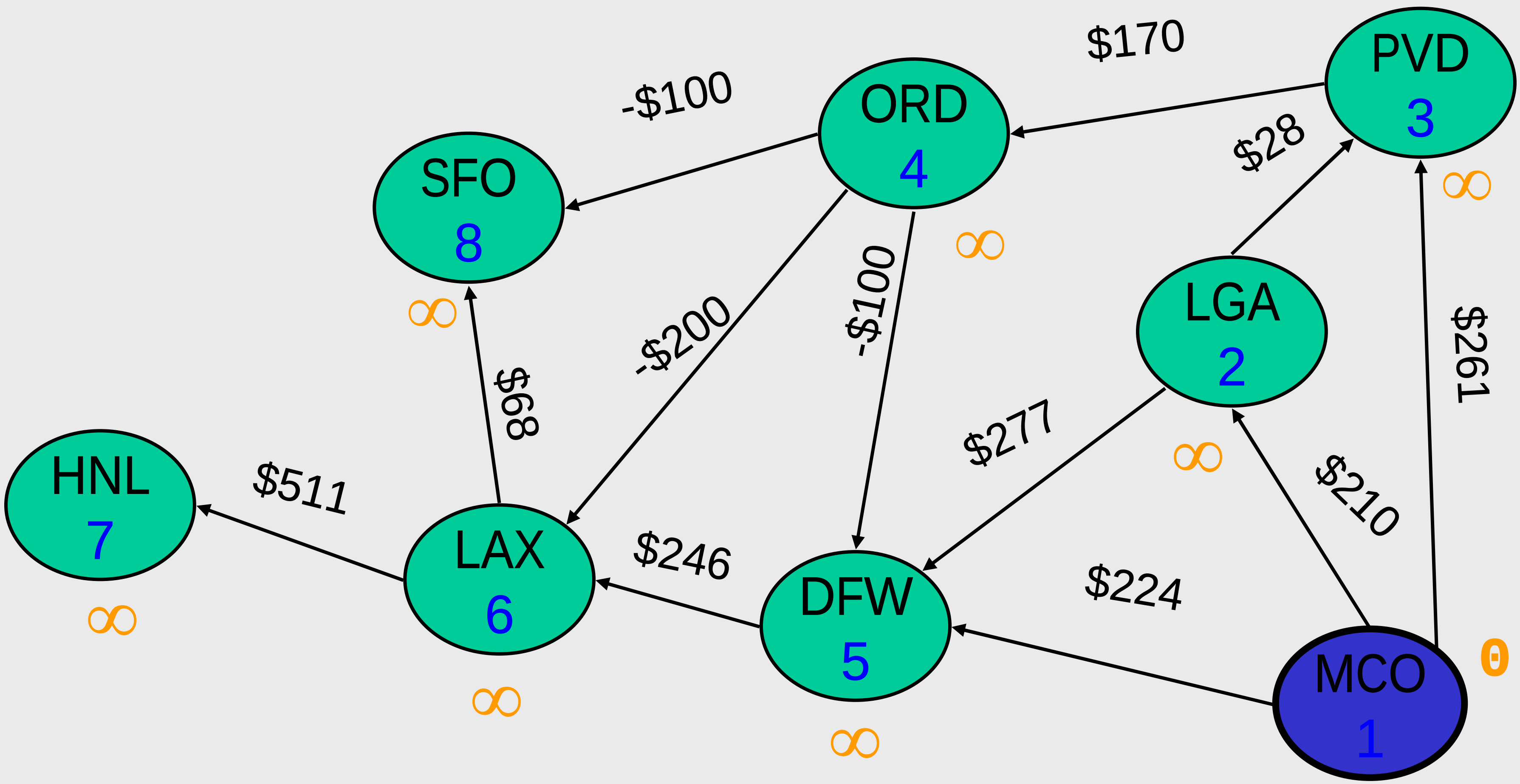


DAG With Negative Weight Edges

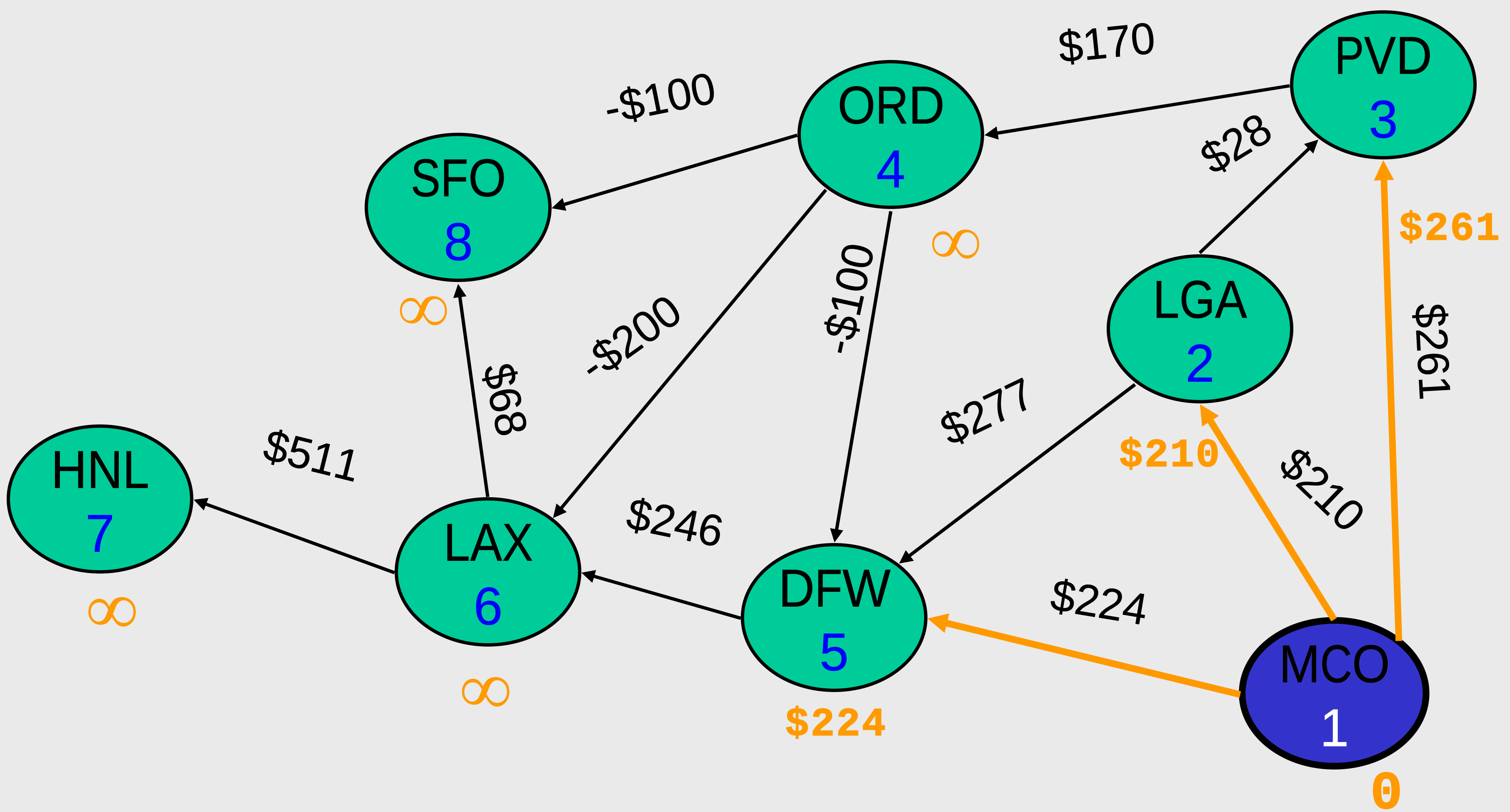




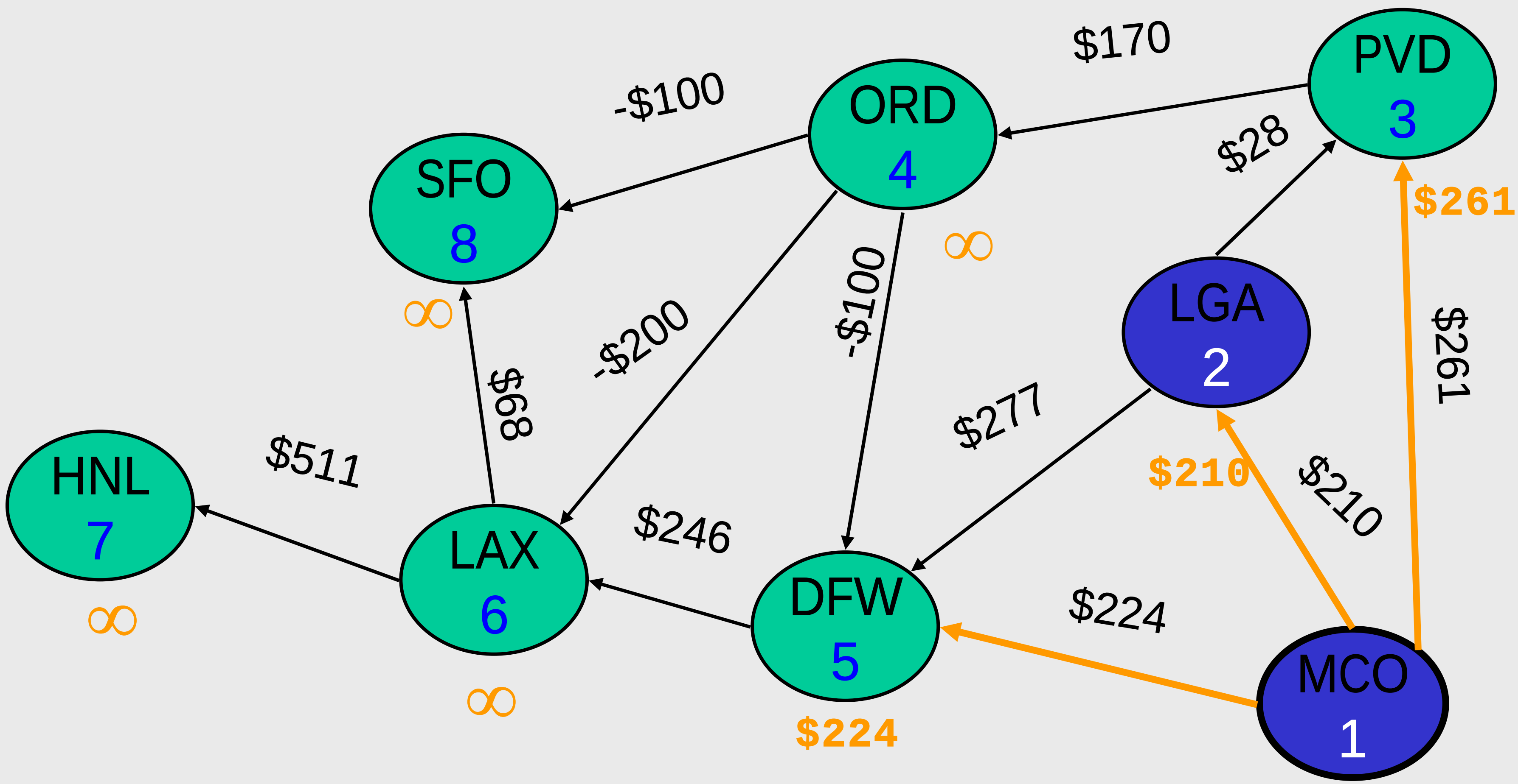
DAG With Negative Weight Edges



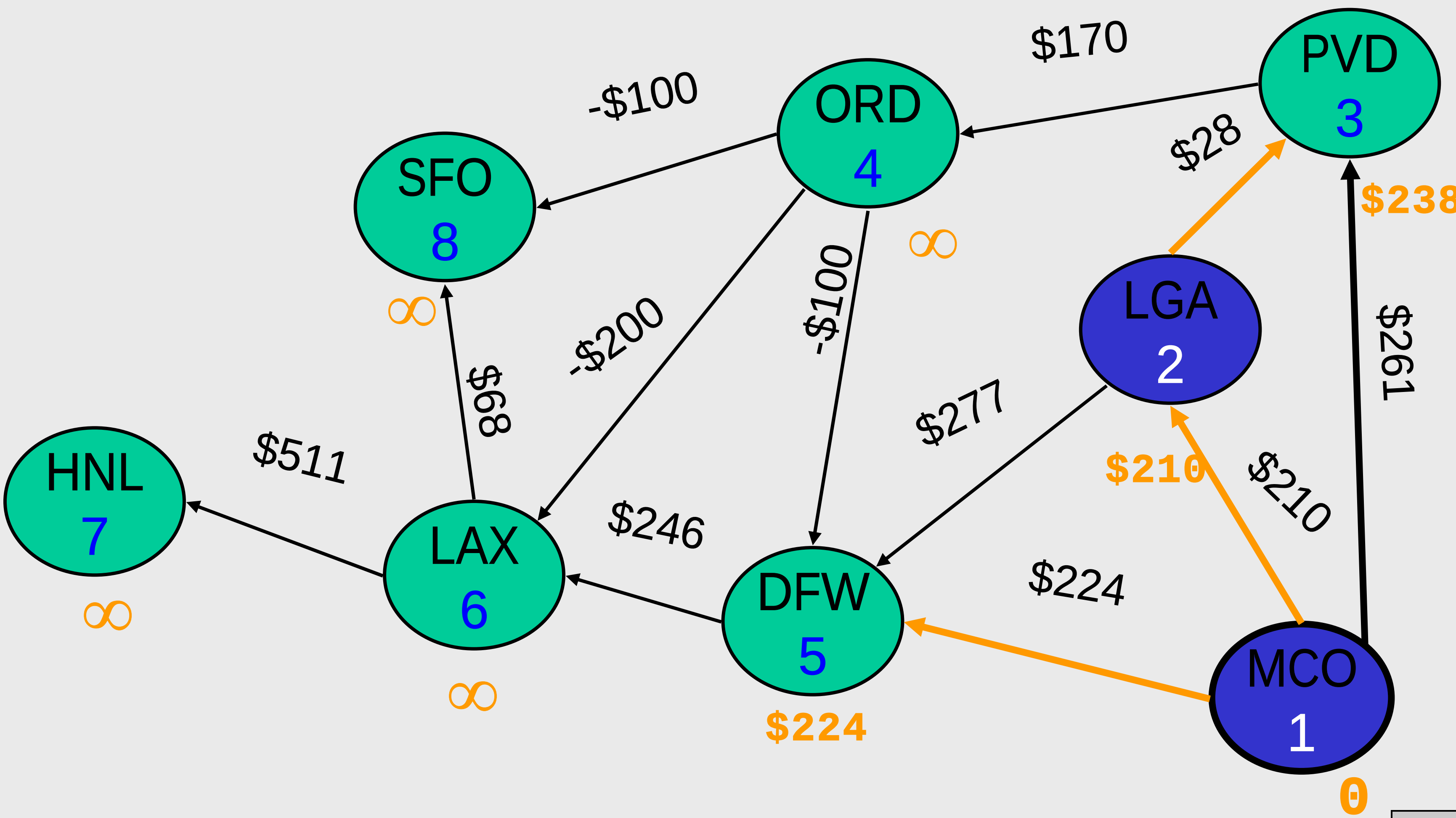
DAG With Negative Weight Edges



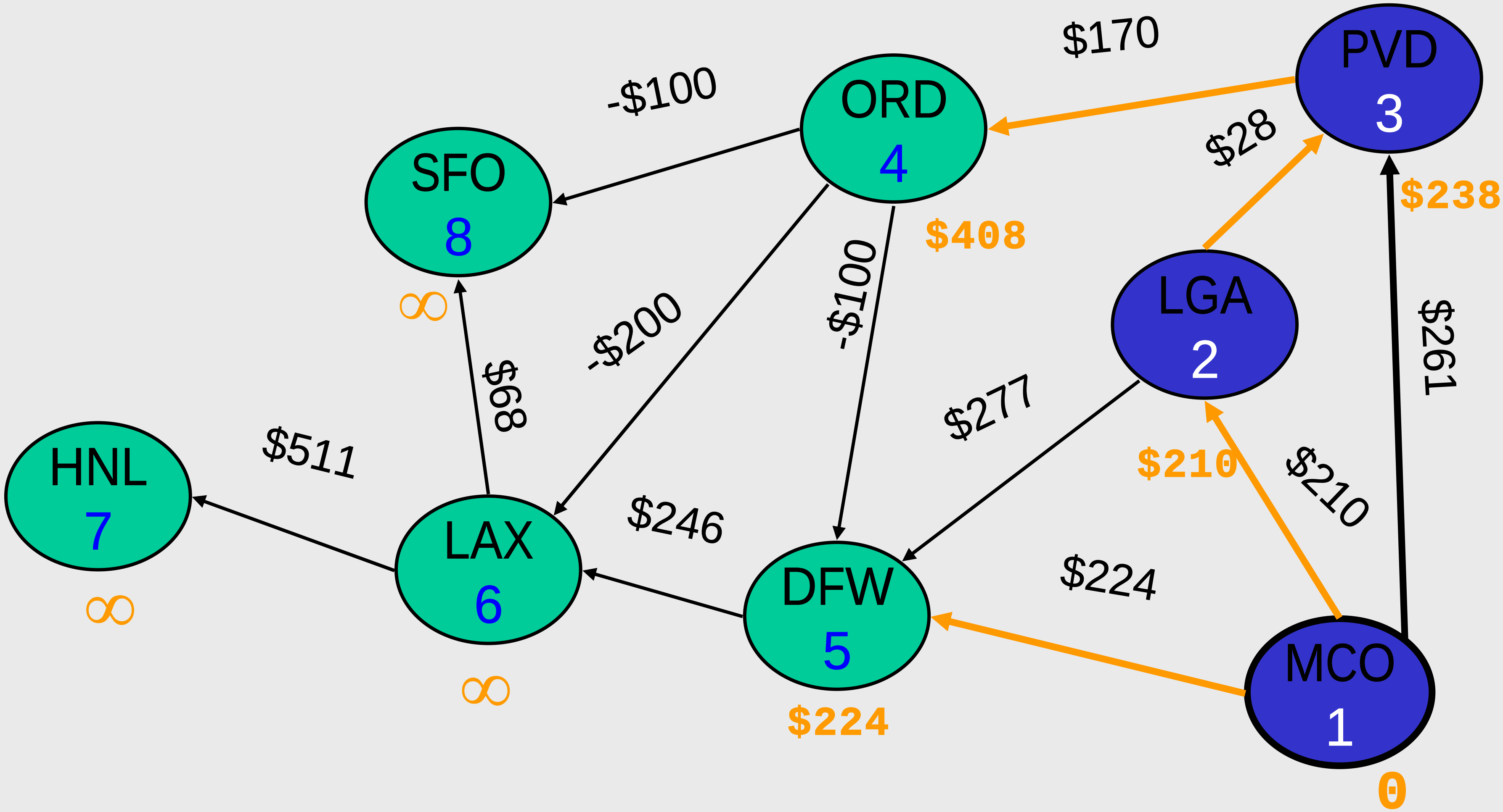
DAG With Negative Weight Edges



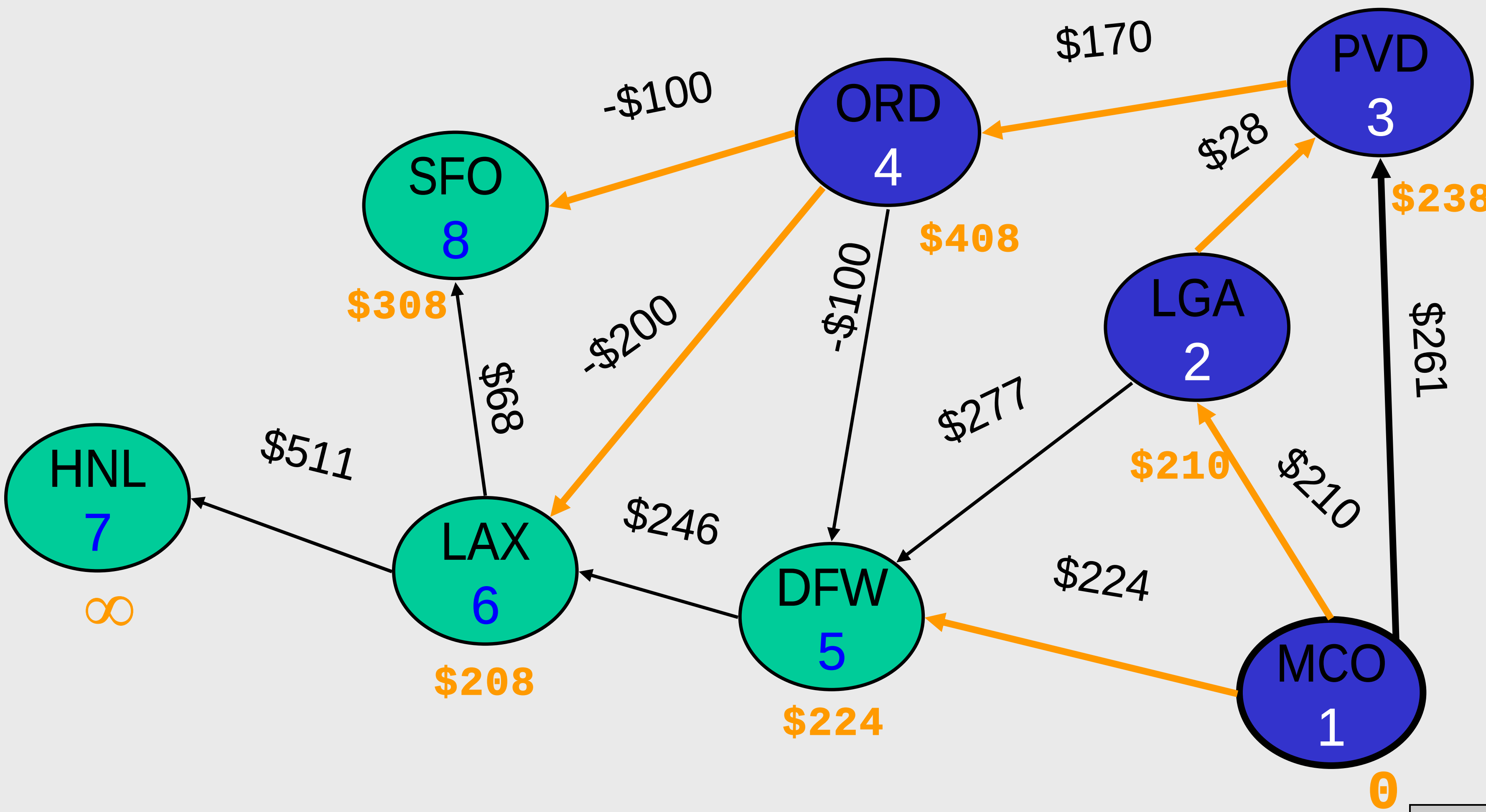
DAG With Negative Weight Edges



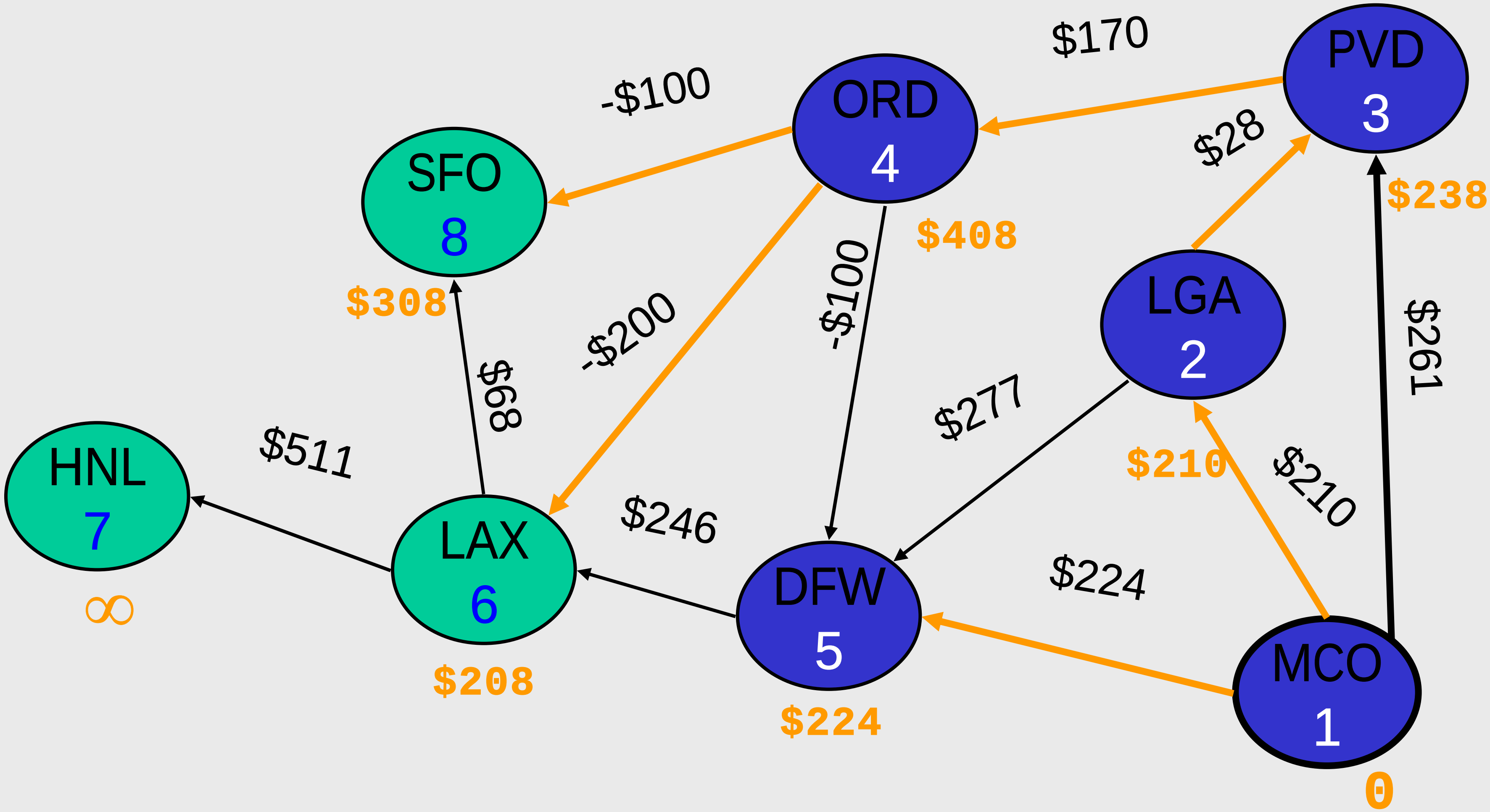
DAG With Negative Weight Edges



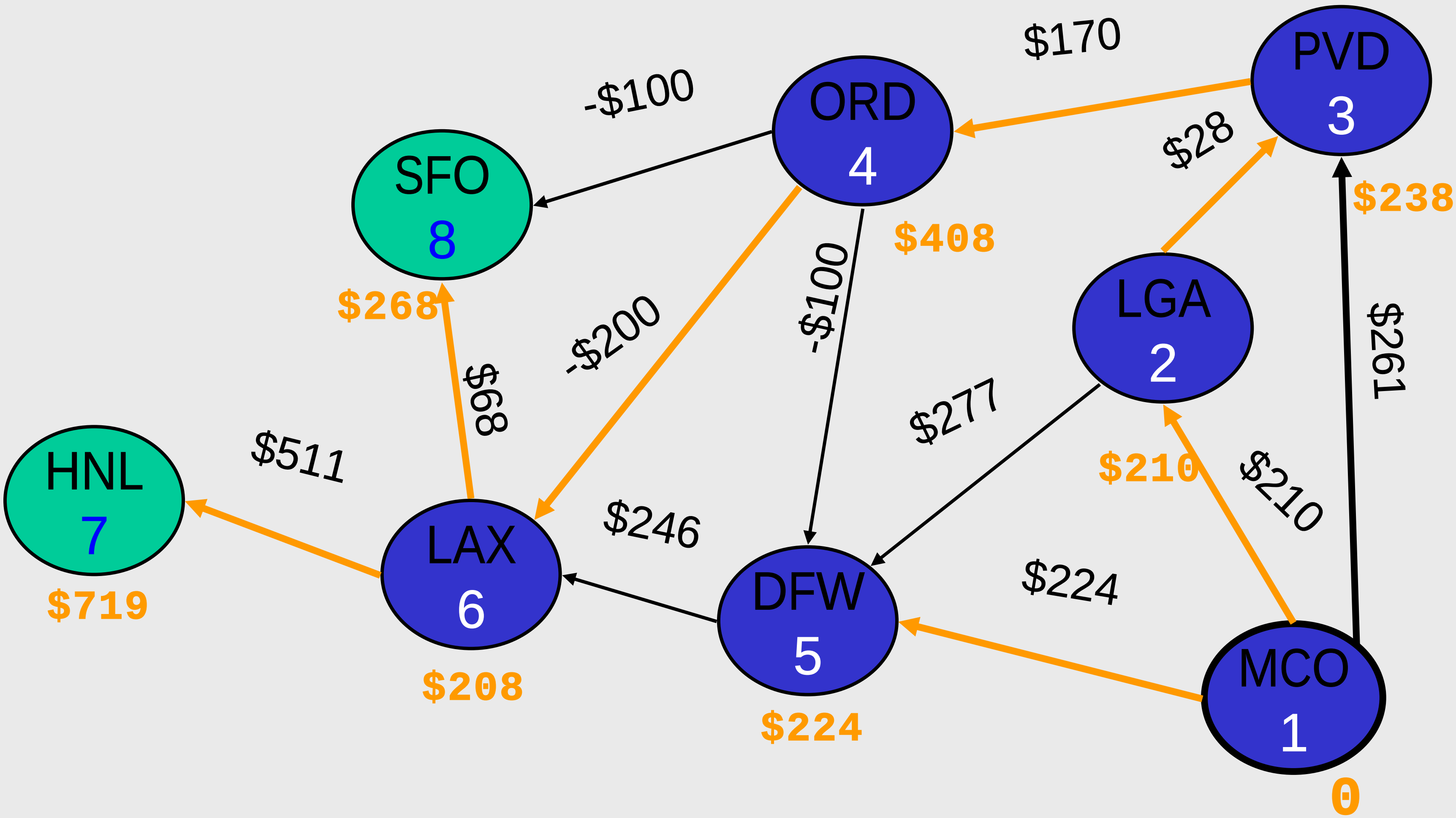
DAG With Negative Weight Edges



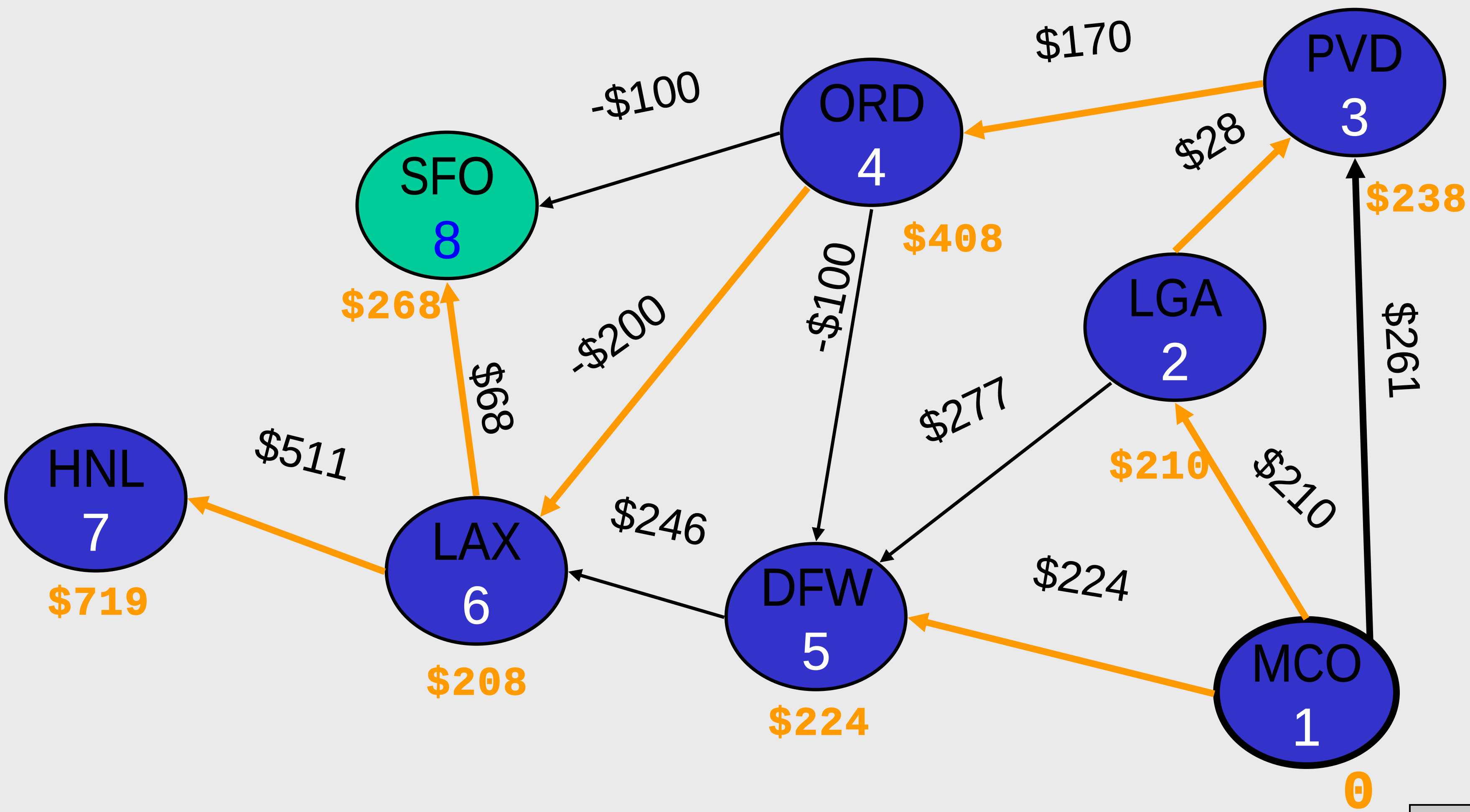
DAG With Negative Weight Edges



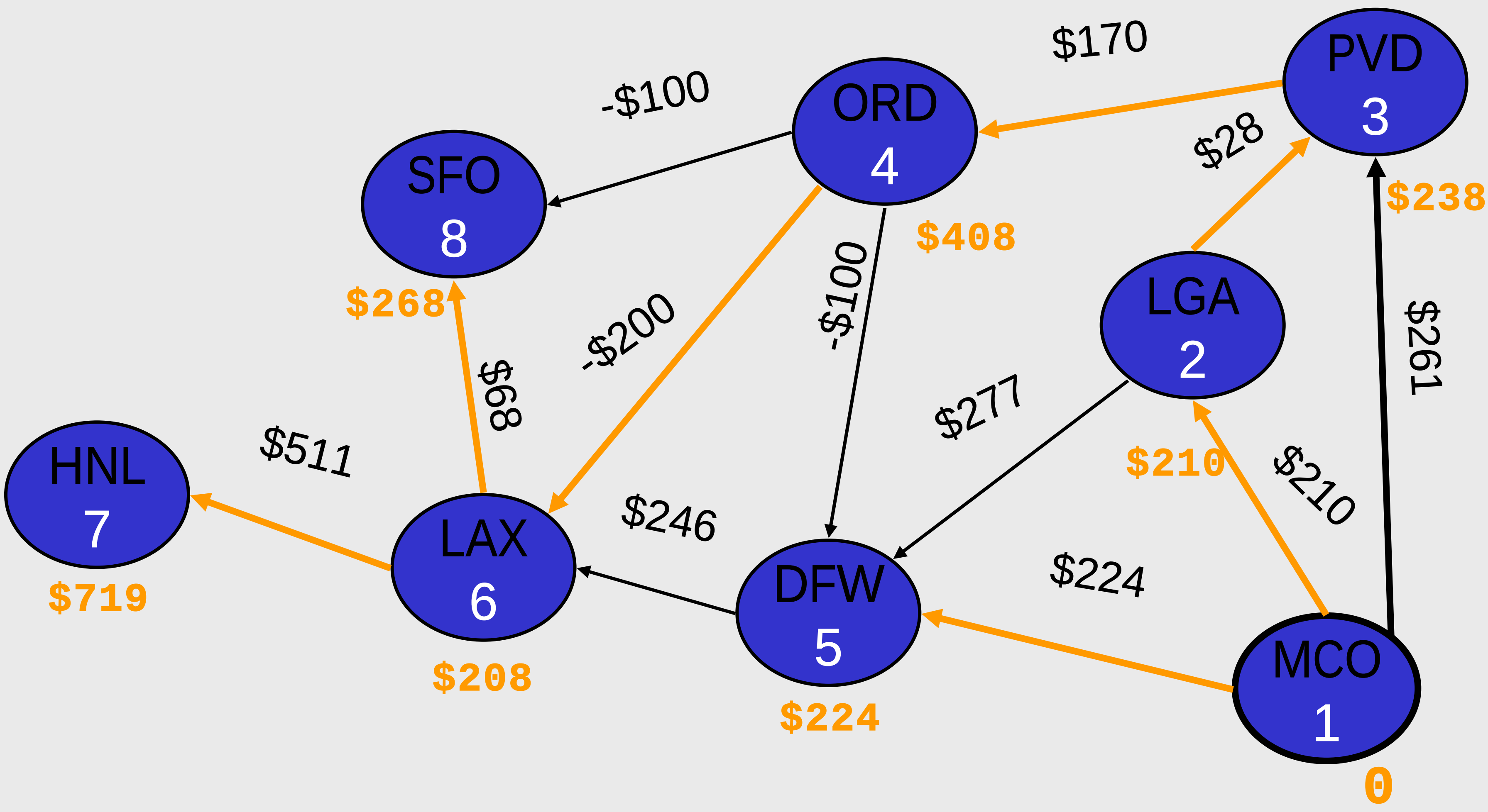
DAG With Negative Weight Edges



DAG With Negative Weight Edges



DAG With Negative Weight Edges



The DAG-based Shortest Path Algorithm

Advantages:

- Works even with negative-weight edges.
- Uses topological order.
- Doesn't use any fancy data structures.
- Is much faster than Dijkstra's algorithm.
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
   for all v ∈ G.vertices()
      v.parent ← null
      if v = s
         v.distance ← 0
      else
         v.distance ← ∞
   Perform a topological sort of the vertices
   for u ← 1 to n do    {in topological order}
      for each e ∈ G.outEdges(u)
         w ← G.opposite(u,e)
         r ← getDistance(u) + weight(e)
         if r < getDistance(w)
            w.distance ← r
            w.parent ← u
```