Shortest Paths

Dijkstra's Algorithm & Bellman-Ford Algorithm

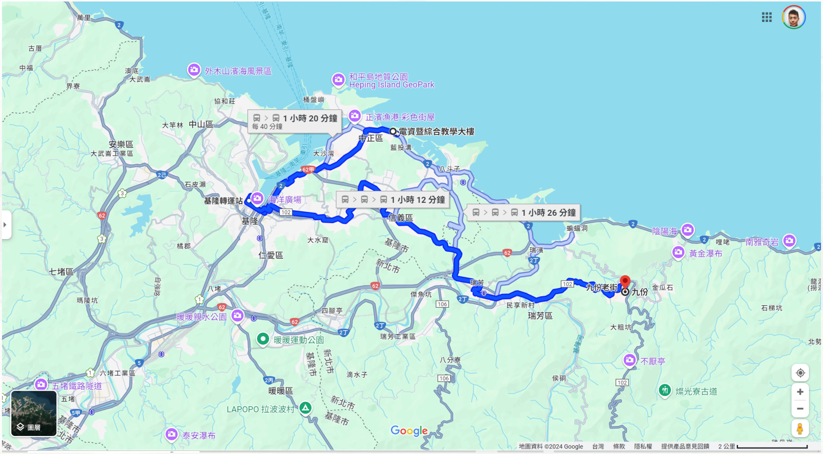
Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

Outline

Shortest path(s) from NTOU to Jiufen Old Street.



Shortest Paths

- Model the problem via a graph.
- vertices $\mapsto$ locations (e.g., stations, restaurants, gas stations, etc.)
  - Including the source and the destination.
- edges $\mapsto$ highways, railways, roads, etc.
  - edge weight: tolls, the distance, passing-through time, etc.

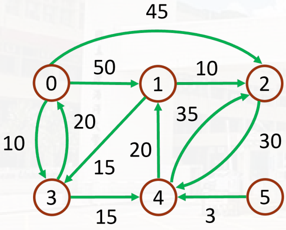
Shortest Paths

- Model the problem via a graph.
- vertices $\mapsto$ locations (e.g., stations, restaurants, gas stations, etc.)
  - Including the source and the destination.
- edges $\mapsto$ highways, railways, roads, etc.
  - edge weight: tolls, the distance, passing-through time, etc.

Questions

- Is there a path from NTOU to Jiufen?
- If it exists, which one is the shortest?

Single Source/All Destinations (Nonnegative Edge Costs)



	path	length (cost)
1	0, 3	10
2	0, 3, 4	25
3	0, 3, 4, 1	45
4	0, 2	45

Notations:

- A directed graph $G = (V, E)$; a weight function $w(e)$, $w(e) > 0$ for any edge $e \in E$.
- v_0: source vertex.
- If $(v_i, v_j) \notin E$, $w(v_i, v_j) = \infty$.

Outline

Greedy Method

- The greedy method can help here!

Greedy Method

- The greedy method can help here!
- Let S denote the set of vertices, including v_0, whose shortest paths have been found.

Greedy Method

- The greedy method can help here!
- Let S denote the set of vertices, including v_0, whose shortest paths have been found.
- For $v \notin S$, let $\text{dist}[v]$ be the length of the shortest path starting from v_0, going through vertices ONLY in S, and ending in v.

Dijkstra's Algorithm

- At the first stage, we add v_0 to S, set dist$[v_0] = 0$ and determine dist$[v]$ for each $v \notin S$.

Dijkstra's Algorithm

- At the first stage, we add v_0 to S, set $\text{dist}[v_0] = 0$ and determine $\text{dist}[v]$ for each $v \notin S$.
- Next, at each stage, vertex w is chosen so that it has the minimum distance, $\text{dist}[w]$, among all the vertices not in S.

Dijkstra's Algorithm

- At the first stage, we add v_0 to S, set $\mathrm{dist}[v_0] = 0$ and determine $\mathrm{dist}[v]$ for each $v \notin S$.
- Next, at each stage, vertex w is chosen so that it has the minimum distance, $\mathrm{dist}[w]$, among all the vertices not in S.
- Adding w to S, and updating $\mathrm{dist}[v]$ for v, where $v \notin S$ currently.

Dijkstra's Algorithm

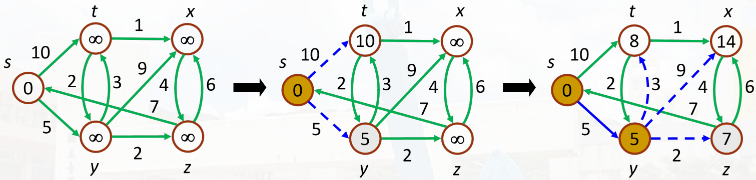
- At the first stage, we add v_0 to S, set $\text{dist}[v_0] = 0$ and determine $\text{dist}[v]$ for each $v \notin S$.
- Next, at each stage, vertex w is chosen so that it has the minimum distance, $\text{dist}[w]$, among all the vertices not in S.
- Adding w to S, and updating $\text{dist}[v]$ for v, where $v \notin S$ currently.
- Repeat the vertex addition process until $S = V(G)$

Dijkstra's Algorithm

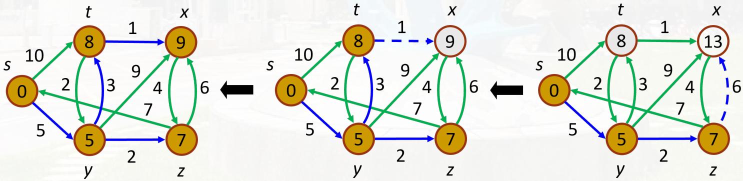
- At the first stage, we add v_0 to S, set $\text{dist}[v_0] = 0$ and determine $\text{dist}[v]$ for each $v \notin S$.
- Next, at each stage, vertex w is chosen so that it has the minimum distance, $\text{dist}[w]$, among all the vertices not in S.
- Adding w to S, and updating $\text{dist}[v]$ for v, where $v \notin S$ currently.
- Repeat the vertex addition process until $S = V(G)$

Time complexity: $O(n^2)$.

Illustration of Dijkstra's Algorithm

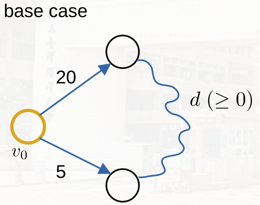


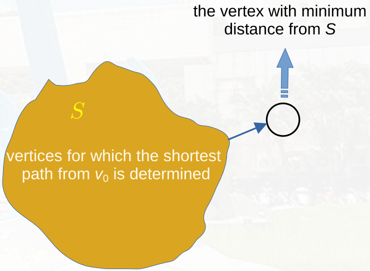
During each iteration:
1. Update the distance of the rest vertices
2. Pick the vertex with the smallest distance value



Basic Idea of Dijkstra's Algorithm

- Induction on *n*.





$20 + d \geq 5$ ($\because$ triangular inequality)

The Pseudo-code of Dijkstra's Algorithm

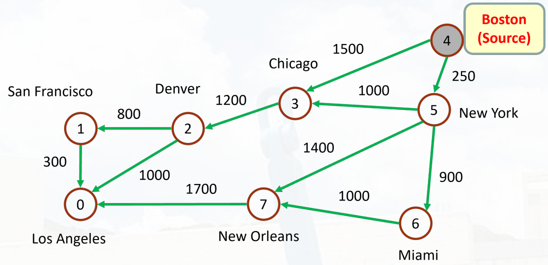
```
S = { v0 };
dist[v0] = 0;
for each v in V - {v0} do
    dist[v] = e(v0,v); // initialization
while (S != V) do
    choose a vertex w in V - S such that dist[w] is a minimum;
    add w to S; // the other nodes in S have been utilized!
    for each v in V - S do
        dist[v] = min(dist[v], dist[w]+e(w, v));
    endfor
endwhile
```

Dijkstra's Algorithm (Functions (1/2))

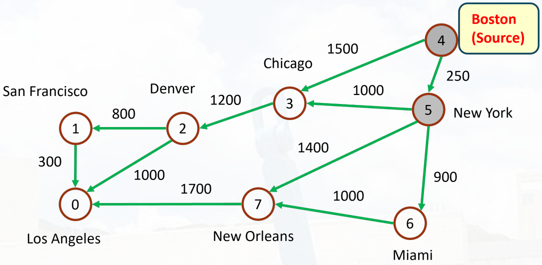
```
void shortestPath (int v, int cost[][MAX_VERTICES],
                   int distance[], int n, bool found[]) {
/* distance[i]: the shortest path from vertex v to i
   found[i]: 0 if the shortest path from vertex i has not
   been found and a 1 otherwise
   cost: the adjacency matrix */
    int i, u, w;
    for (i=0; i<n; i++) {
        found[i] = false; distance[i] = cost[v][i];
    }
    found[v] = true; //initialization
    distance[v] = 0; //initialization
    for (i=0; i<n-1; i++) {
        u = choose(distance, n, found);
        found[u] = true;
        for (w=0; w<n; w++)
        if (!found[w])
            if (distance[u] + cost[u][w] < distance[w])
                distance[w] = distance[u]+cost[u][w];
    }
}
```

Dijkstra's Algorithm (Functions (2/2))

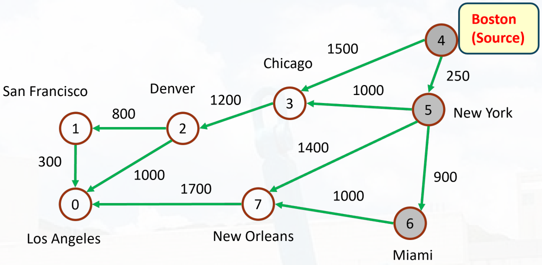
```
int choose (int distance[], int n, bool found[]) {
/* find the smallest distance not yet checked */
    int i, min, min_pos;
    min = INT_MAX;
    min_pos = -1;
    for (i=0; i<n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            min_pos = i;
        }
    return min_pos;
}
```



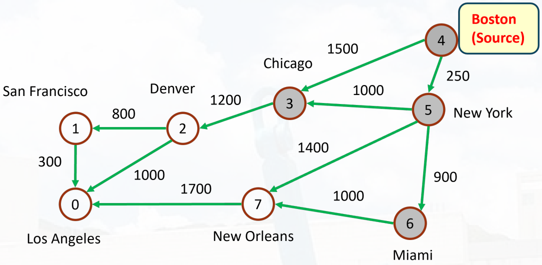
Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞



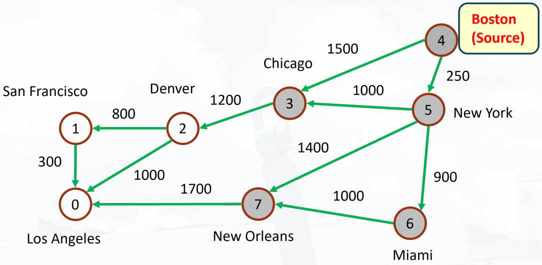
Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650



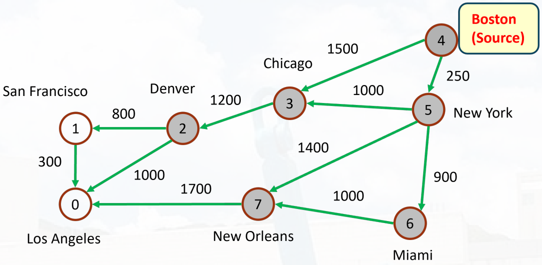
Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650



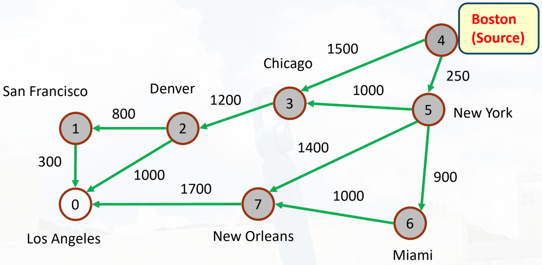
Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650



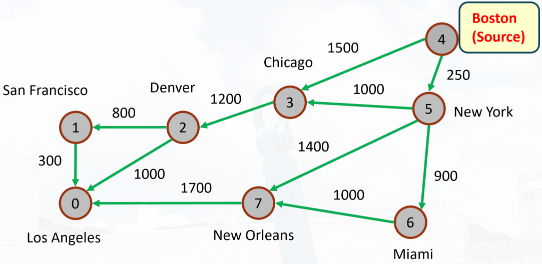
Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650



Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650



Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650
6	1	3350	3250	2450	1250	0	250	1150	1650



Iteration	Vertex Select.	Distance							
		LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MIA [6]	NO [7]
initial	—	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	2450	1250	0	250	1150	1650
4	7	3350	∞	2450	1250	0	250	1150	1650
5	2	3350	3250	2450	1250	0	250	1150	1650
6	1	3350	3250	2450	1250	0	250	1150	1650
7	0	3350	3250	2450	1250	0	250	1150	1650

Outline

Single Source/All Destinations: General Weights

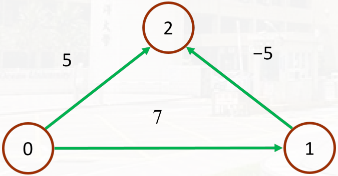
- **Focus:** Some edges of the directed graph G have negative length.

Single Source/All Destinations: General Weights

- **Focus:** Some edges of the directed graph G have negative length.
- The function `shortestPath` may NOT work!

Single Source/All Destinations: General Weights

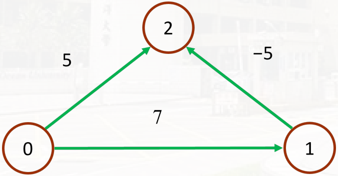
- **Focus:** Some edges of the directed graph G have negative length.
- The function `shortestPath` may NOT work!
- For example,



- $\text{dist}[1] = 7, \text{dist}[2] = 5.$

Single Source/All Destinations: General Weights

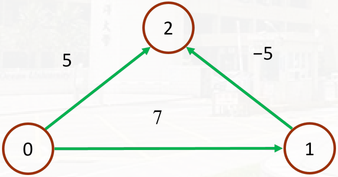
- **Focus:** Some edges of the directed graph G have negative length.
- The function `shortestPath` may NOT work!
- For example,



- $\text{dist}[1] = 7, \text{dist}[2] = 5$.
- The shortest path from 0 to 2 is: $0 \rightarrow 1 \rightarrow 2$ (length $= 2$).

Single Source/All Destinations: General Weights

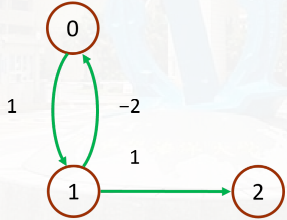
- **Focus:** Some edges of the directed graph G have negative length.
- The function `shortestPath` may NOT work!
- For example,



- $\text{dist}[1] = 7, \text{dist}[2] = 5.$
- The shortest path from 0 to 2 is: $0 \to 1 \to 2$ (length $= 2$).
- Dijkstra's "greedy" approach does not work here.

Workaround Solution: NO negative cycle is permitted!

- When negative edge lengths are permitted, we require that the graph have no cycles of negative length.
- This is necessary so as the ensure that shortest paths consist of a finite number of edges.



Observations

- When there are NO cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has $\leq n-1$ edges on it.

Observations

- When there are NO cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has $\leq n-1$ edges on it.
  - Otherwise, the path must repeat at least one vertex, and hence must contain a cycle.

Observations

- When there are NO cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has $\leq n-1$ edges on it.
  - Otherwise, the path must repeat at least one vertex, and hence must contain a cycle.
- So, eliminating the cycles from the path results in another path with the same source and destination.
  - The length of the new path should be no more than that of the original.

Dynamic Programming Approach

$\text{dist}^k[u]$: the length of a shortest path from the source v to u under the constraint that the shortest path contains $\leq k$ edges.

Dynamic Programming Approach

$\text{dist}^k[u]$: the length of a shortest path from the source v to u under the constraint that the shortest path contains $\leq k$ edges.

- Hence, $\text{dist}^1[u] =$

Dynamic Programming Approach

$\text{dist}^k[u]$: the length of a shortest path from the source v to u under the constraint that the shortest path contains $\leq k$ edges.

- Hence, $\text{dist}^1[u] = \text{length}[v][u]$, for $0 \leq u < n$.

Dynamic Programming Approach

$\text{dist}^k[u]$: the length of a shortest path from the source v to u under the constraint that the shortest path contains $\leq k$ edges.

- Hence, $\text{dist}^1[u] = \text{length}[v][u]$, for $0 \leq u < n$.
- The goal: Compute $\text{dist}^{n-1}[u]$ for all u.

Dynamic Programming Approach

$\text{dist}^k[u]$: the length of a shortest path from the source v to u under the constraint that the shortest path contains $\leq k$ edges.

- Hence, $\text{dist}^1[u] = \text{length}[v][u]$, for $0 \leq u < n$.
- The goal: Compute $\text{dist}^{n-1}[u]$ for all u.

▷ Using Dynamic Programming.

Bellman-Ford Algorithm (Sketch)

- If the shortest path from v to u with $\leq k$ edges $(k > 1)$ has no more than $k - 1$ edges, then $\text{dist}^k[u] = \text{dist}^{k-1}[u]$.

Bellman-Ford Algorithm (Sketch)

- If the shortest path from v to u with $\leq k$ edges ($k > 1$) has no more than $k-1$ edges, then $\text{dist}^k[u] = \text{dist}^{k-1}[u]$.
- If the shortest path from v to u with $\leq k$ edges ($k > 1$) has exactly k edges, there exists a vertex i such that $\text{dist}^{k-1}[i] + \text{length}[i][u]$ is minimum.
- The recurrence relation:

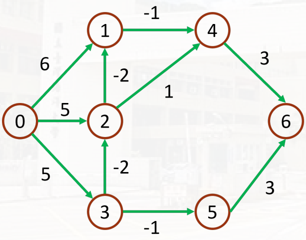
Bellman-Ford Algorithm (Sketch)

- If the shortest path from v to u with $\leq k$ edges $(k > 1)$ has no more than $k - 1$ edges, then $\text{dist}^k[u] = \text{dist}^{k-1}[u]$.
- If the shortest path from v to u with $\leq k$ edges $(k > 1)$ has exactly k edges, there exists a vertex i such that $\text{dist}^{k-1}[i] + \text{length}[i][u]$ is minimum.
- The recurrence relation:

$$\text{dist}^k[u] = \min\{\text{dist}^{k-1}[u], \min_i\{\text{dist}^{k-1}[i] + \text{length}[i][u]\}\}.$$

Shortest paths with negative edge lengths (cost)

$$\text{dist}^k[u] = \min\{\text{dist}^{k-1}[u],\ \min_i\{\text{dist}^{k-1}[i] + \text{length}[i][u]\}\}.$$



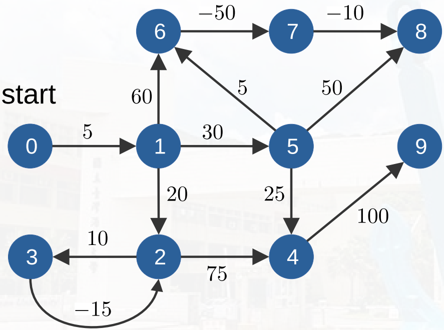
(a) A directed graph

k	$dist^k[u]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	**4**	**7**
4	**0**	**1**	**3**	**5**	**0**	**4**	**5**
5	**0**	**1**	**3**	**5**	**0**	**4**	**3**
6	**0**	**1**	**3**	**5**	**0**	**4**	**3**

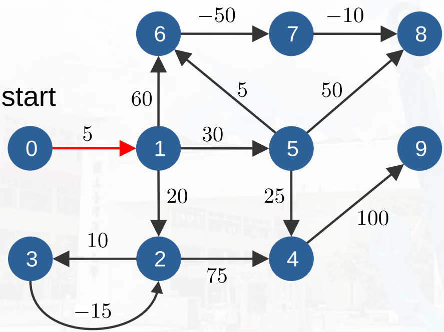
(b) $dist^k$

Bellman-Ford Algorithm (Pseudo-Code)

```
BF(int u) { // assume that the source is vertex u
    for each vertex w in V - {u}, set dist[w] = INT_MAX
    set dist[u] = 0
    for (i=0; i<n-1; i++) { // n: the number of vertices (k)
        for each edge (p,q)  in the graph {
        // we can choose p with dist[p] < INT_MAX
            if (dist[p] + length[p][q] < dist[q])
                dist[q] = dist[p] + length[p][q]
        }
    }
    // Now the distances from u to every other vertex is found.
    // Repeat the following to find nodes in a negative cycle
    for (i=0; i<n-1; i++) {
        for each edge (p,q) in the graph {
            if (dist[p] + length[p][q] < dist[q])
        }       dist[q] = -INT_MAX
    }
}
```

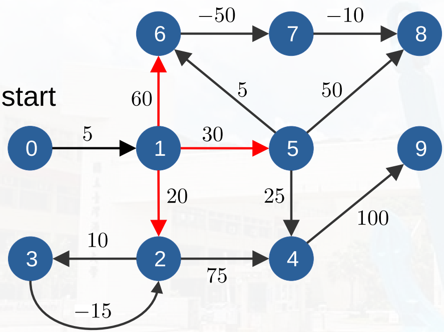


0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞



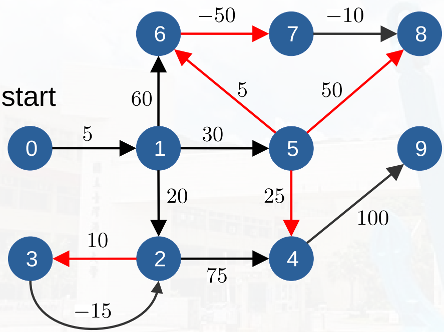
`i = 0`

0	0
1	5
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞



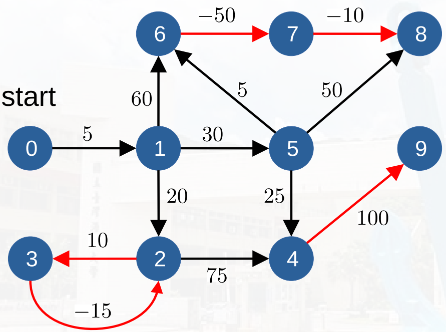
i = 1

0	0
1	5
2	25
3	∞
4	∞
5	35
6	65
7	∞
8	∞
9	∞



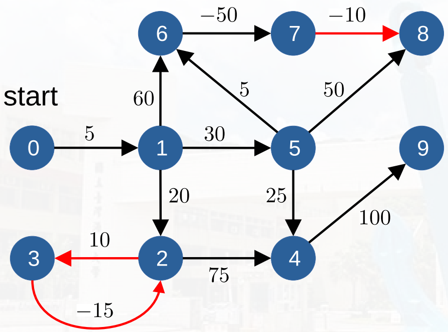
i = 2

0	0
1	5
2	25
3	35
4	60
5	35
6	40
7	15
8	85
9	∞



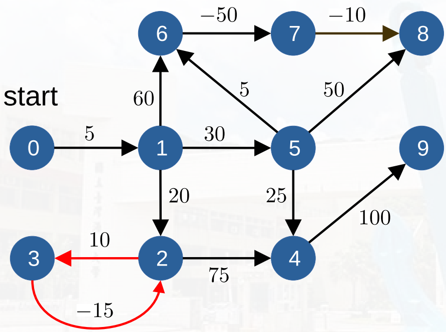
i = 3

0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	−10
8	5
9	160



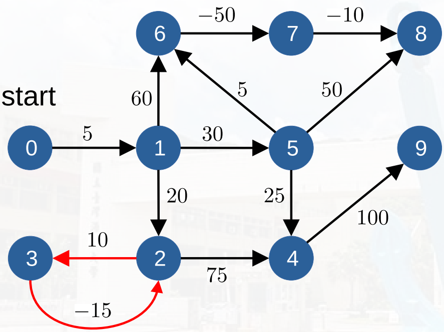
`i = 4`

0	0
1	5
2	10
3	25
4	60
5	35
6	40
7	−10
8	−20
9	160



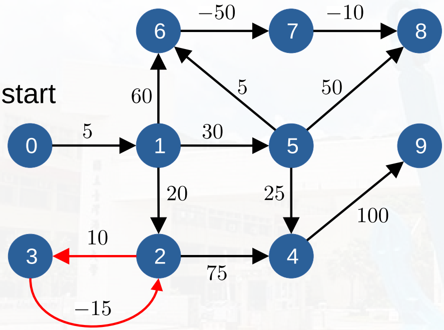
i = 5

0	0
1	5
2	5
3	20
4	60
5	35
6	40
7	−10
8	−20
9	160



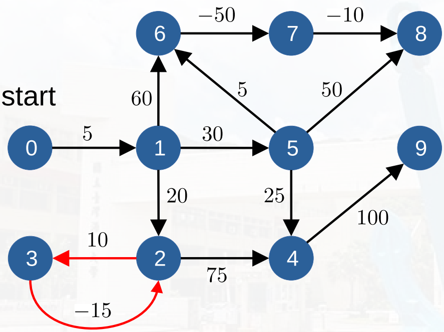
`i = 6`

0	0
1	5
2	0
3	15
4	60
5	35
6	40
7	−10
8	−20
9	160



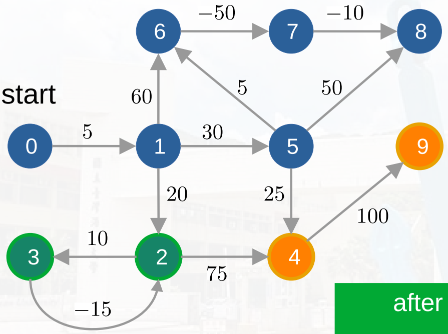
`i = 7`

0	0
1	5
2	−5
3	10
4	60
5	35
6	40
7	−10
8	−20
9	160



i = 8

0	0
1	5
2	−10
3	5
4	60
5	35
6	40
7	−10
8	−20
9	160



after negative cycle detection

0	0
1	5
2	$-\infty$
3	$-\infty$
4	$-\infty$
5	35
6	40
7	−10
8	−20
9	$-\infty$

Discussions