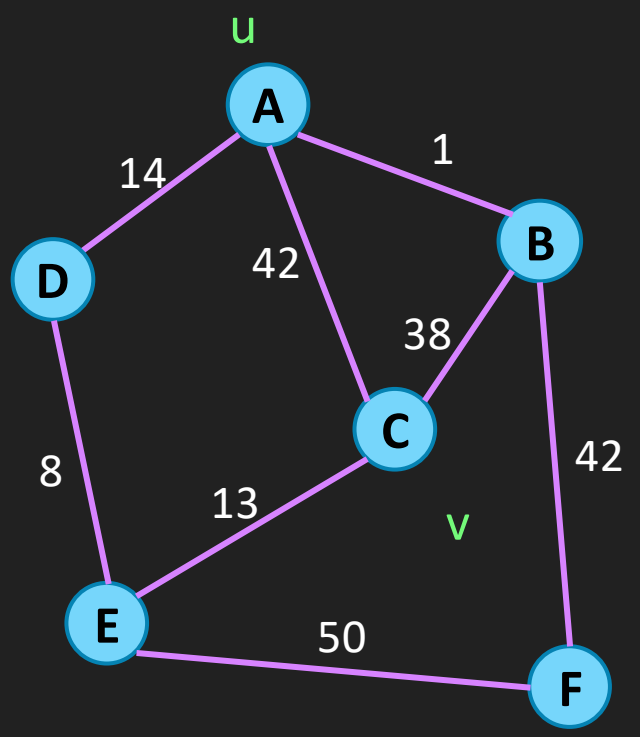
Shortest Path in Graph

Nattee Niparnan

Department of Computer Engineering
Chulalongkorn University

The Problem



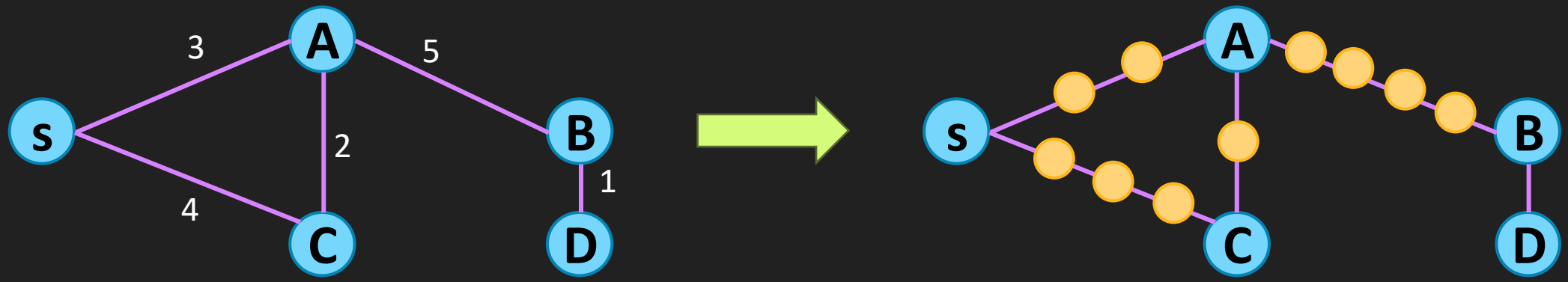
- Finding a path from u to v with minimum summation of weight of the path
  - May not be the path with minimum number of edges
- [A,C] 42
- [A,B,C] 39
- [A,D,E,C] 35

Single Source Shortest Path Problem

- Problem: Given a graph G with a starting node s, find the shortest path from s to every node in the graph
- Input:
  - A graph G = (V,E)
  - A starting node s
  - A weight function w where w((a,b)) returns a weight of an edge (a,b)
    - For unweighted graph, we can set w((a,b)) = 1 for any edge (a,b)
- Output:
  - `dist[x]` the shortest distance from s to x
    - If a path is needed, `prev[x]` is used just like MST

Direct solution using BFS

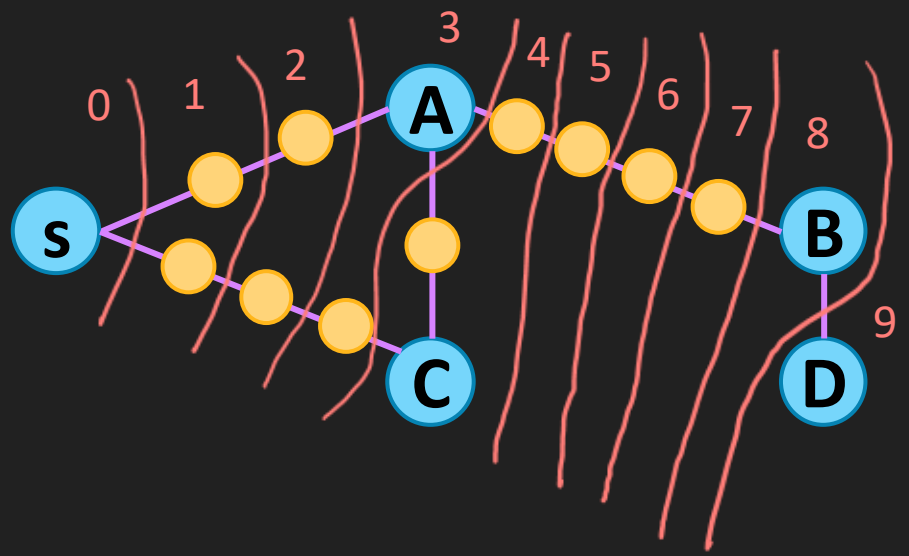
- BFS can give us the shortest path on unweighted graph
- Convert weighted to unweighted by adding auxiliary nodes



- However, this is very slow. BFS is O(n+e) but now we have n approximately as sum(w(*))
  - Imagine when a single edge has length 10^6

Alarm Clock Analogy

- No need to calculate the distance to the auxiliary node
  - We want to skip to the actual node
- Since BFS consider nodes in order of distance, we want to skip to the distance when actual node is reached



no need to calculate distance of 1, 2, 5, 6, 7 we can skip to 3,4, 8 and 9

Alarm Clock Algorithm

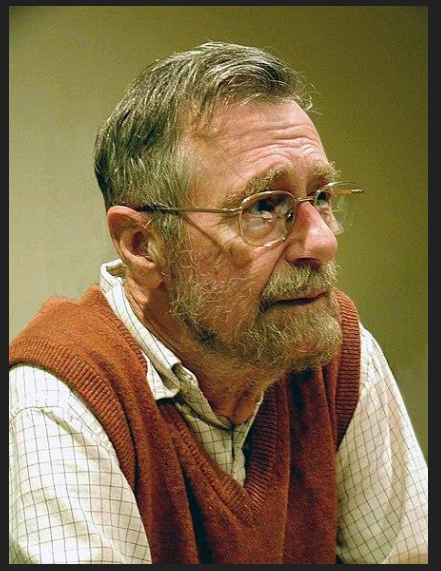
- Each node has its alarm clock, initially these alarm is not set.
- Set an alarm clock for node s at time 0
- Repeat until there are no more alarms:
  - Let the next alarm goes off at time T, at node u. Then:
    - The distance from s to u is T (i.e. set dist[u] = T)
    - For v in G.adj(u)
      - Consider the edge (u,v) having weight as w(u,v).
      - If we do BFS on this edge with auxiliary nodes, we will reach v at time T + w(u, v)
      - If there is no alarm yet for v set it to the time T + w(u, v).
      - If v's alarm is set for later than T + w(u, v), then reset it to this earlier time.

Dijkstra's Algorithm

```
def dijkstra(G,w,s)
  for u in G.V
    dist[u] = ∞
    prev[u] = -1
  end
  S = new Set
  S.insert( (0,s) )
  while S is not empty
    (T,u) = S.min
    for v in G.adj(u)
      if dist[v] > T + w(u,v)
        # change the value of v to the new cost[v]
        S.remove_if_exist( (dist[v],v) )
        dist[v] = T + w(u,v)
        prev[v] = u
        S.insert( (dist[v],v) )
      end
    end
  end
  return dist
end
```

This is the same as prim's algorithm except that we use T + w(u,v) instead of w(u,v)

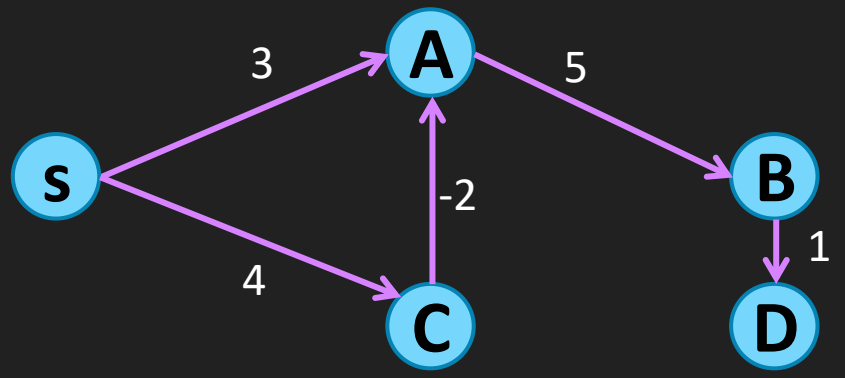
- Use set to store alarm of each node.
- The next alarm goes off at the minimum value in the set
- Adjust alarm by remove and re-insert alarm of that node



Analysis

- Exactly the same as prim's algorithm
  - O(n (lg n) + e (lg n)) = O((n+e) lg n)
  - Depends on underlying datastructure that store the alarm
    - Fibonacci Heap gives the best complexity
- Unlike minimum spanning tree, Dijkstra's algorithm works with directed graph as well
  - Because we do not care about going back from v to s, we consider only from s to other nodes

Graph with Negative Edge



Shortest path to A pass through C which is, in BFS sense, is further than A

- Disjktra's works because a shortest path to v must pass through a node u which is closer to s than v
  - Dijkstra's algorithm says that when an alarm goes off, that is when BFS has reached that node and we knows the shortest distance.
  - The alarm at the closer node goes off before v so that we can calculate the alarm at v correctly
  - This assumption is not correct when the graph has negative edges

Directed Graph and Negative Edges

- An undirected graph with a negative edge has no shortest path
  - The shortest path makes no sense because we can go through the negative edge repeatedly and the distance always decrease
- Unlike minimum spanning tree, Dijkstra's algorithm works with directed graph as well
  - Because we do not care about going back from v to s, we consider only from s to other nodes
- For directed graph, negative edges are possible if they do not make a negative cycle

Directed	Negative Edge	Shortest Path possible?	Dijkstra's Algorithm Work Correctly?
No	No	Yes	Yes
No	Yes	No	No
Yes	No	Yes	Yes
Yes	Yes	Yes (graph can't have negative cycle)	No

Bellman-Ford Algorithm

Dealing with directed graph with negative edge

Key Idea in Shortest Path

```
...
while S is not empty
  (T,u) = S.min
  for v in G.adj(u)
    if dist[v] > T + w(u,v)
      # change the value of v to the new cost[v]
      S.remove_if_exist( (dist[v],v) )
      dist[v] = T + w(u,v)
      prev[v] = u
      S.insert( (dist[v],v) )
    end
  end
end
```

T is the current dist[u]

- Update the distance

dist[u] is T, the distance of the node whose alarm just go off

```
if dist[v] > dist[u] + w(u,v)

    dist[v] = dist[u] + w(u,v)
```

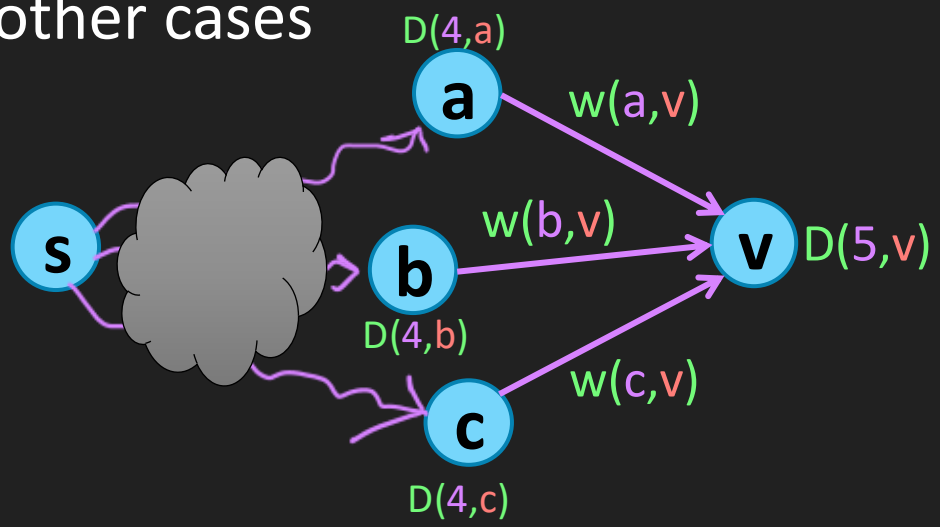
dist[v] is the alarm of neighbor of u

Another Approach to Shortest Path

- A shortest path can have at most n - 1 edges
- Writing a recurrence relation on number of edges
  - D(a,v) = shortest distance from s to v using at most a edges
- Initial condition D(*,s) = 0, D(*, *) = ∞ for any other cases
- D(a,v) = min
  - D(a – 1, v)
  - min(D(a – 1, u) + w(u,v))
    - For any u that has edges go to v

Remember the shortest path using lesser number of edges

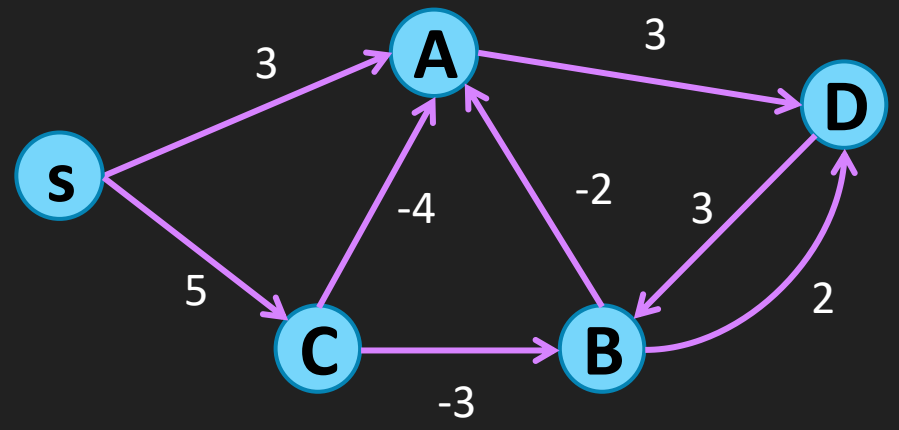
We use a-1 edges to reach a node that has a path to v and use the edge (u,v) to reach v



Bellman-Ford Algorithm

- Dynamic Programming that calculates D(*,*)

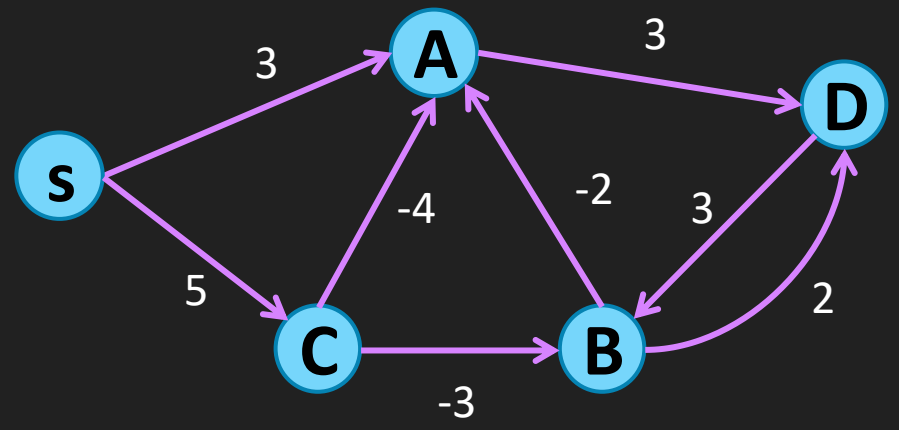
	s	A	B	C	D
0	0	∞	∞	∞	∞
1	0	3	∞	5	∞
2	0				
3	0				
4	0				



Bellman-Ford Algorithm

- Dynamic Programming that calculates D(*,*)

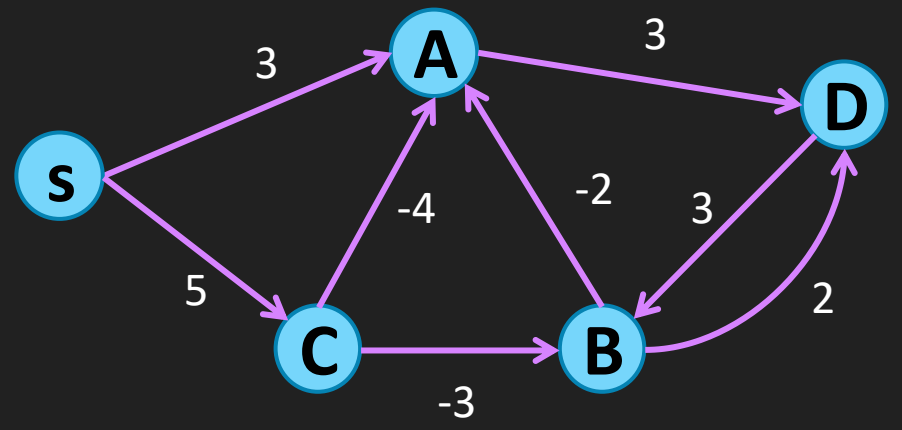
	s	A	B	C	D
0	0	∞	∞	∞	∞
1	0	3	∞	5	∞
2	0	1	2	5	6
3	0				
4	0				



Bellman-Ford Algorithm

- Dynamic Programming that calculates D(*,*)

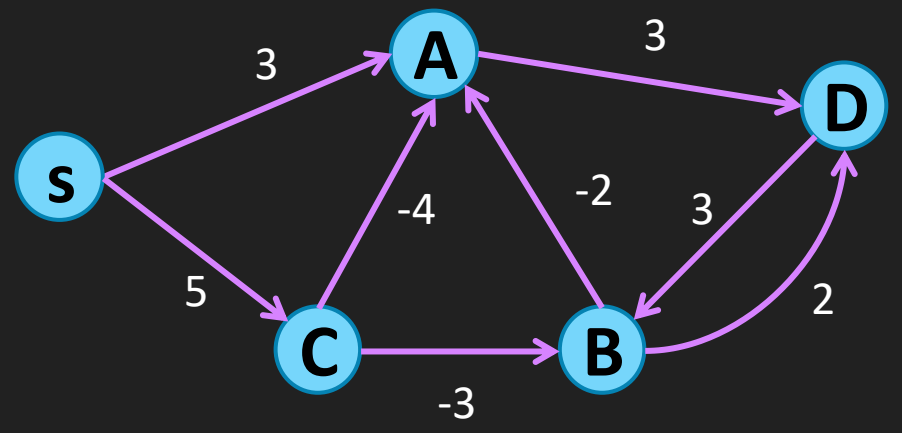
	s	A	B	C	D
0	0	∞	∞	∞	∞
1	0	3	∞	5	∞
2	0	1	2	5	6
3	0	0	2	5	4
4	0				



Bellman-Ford Algorithm

- Dynamic Programming that calculates D(*,*)

	s	A	B	C	D
0	0	∞	∞	∞	∞
1	0	3	∞	5	∞
2	0	1	2	5	6
3	0	0	2	5	4
4	0	0	2	5	3

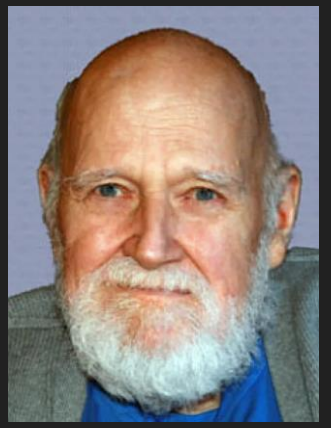


Bellman-Ford Algorithm (Direct Implementation)

```
def bellman_ford(G, w, s)
  for u in G.V
    D[0][u] = ∞
    prev[u] = -1
  end

  D[0][s] = 0
  for i from 1 to n − 1
    for u in G.V
      D[i][u] = D[i-1][u]
    for all edges (a,b) in G.E
      if D[i][b] > D[i-1][a] + w(a,b)
         D[i][b] = D[i-1][a] + w(a,b)
         prev[b] = a
      end
    end
  end

  for u in G.V
    dist[u] = D[n-1][u]
end
```



Bellman-Ford Algorithm (Actual)

- Observe that D[a][*] uses D[a-1][*]
  - When computing D[a], we don't need D[a-2][], D[a-3][], …
  - No need to keep every row of D, just remember the previous row
- In fact, Bellman-ford proposed that we just use only one row and compute the recurrence relation directly on that row
  - This does not strictly compute the recurrence relation faithfully in each iteration (it might "compute ahead") but at the final round, it is guaranteed that D[*] is the same as the recurrence relation, or lower because no path has more than n-1 edges

```
def bellman_ford(G, w, s)
  for u in G.V
    dist[u] = ∞
    prev[u] = -1
  end

  dist[s] = 0
  for i from 1 to n – 1
    for all edges (a,b) in G.E
      if dist[b] > dist[a] + w(a,b)
         dist[b] = dist[a] + w(a,b)
         prev[b] = a
      end
    end
  end
  return dist
end
```

Detecting Negative Cycle

- After repeating the loop n-1 times
- If there is a case when

  `dist[v] > dist[u] + w(u,v)`

- Then, there is a negative cycle
  - Because there is a shorter path that use n edges which indicates that some node are repeated in the path, hence, a cycle

```
def bellman_ford(G, w, s)
  for u in G.V
    dist[u] = ∞
    prev[u] = -1
  end
  dist[s] = 0
  for i from 1 to n - 1
    for all edges (a,b) in G.E
      if dist[b] > dist[a] + w(a,b)
         dist[b] = dist[a] + w(a,b)
         prev[b] = a
      end
    end
  end
  for all edges (a,b) in G.E
    if dist[b] > dist[a] + w(a,b)
      return “negative cycle”
  return dist
end
```

Analysis

- Very simple
- Loop n-1 times (plus another round for detect a negative cycle)
  - Each loop takes e iterations
- O(ne)
  - Comparing to Dijkstra's O(e lg n)
- Dense graph
  - Bellman-Ford $O(n^3)$
  - Dijkstra's $O(n^2 \lg n)$
- Bellman-Ford is slower but can work with negative edge

All Pair Shortest Path

Shortest path between every pair of nodes

All Pair Shortest Path Problem

- Problem: Given a graph G find the shortest path for every pair of nodes
- Input:
  - A graph G = (V,E)
  - A weight function w where w((a,b)) returns a weight of an edge (a,b)
    - For unweighted graph, we can set w((a,b)) = 1 for any edge (a,b)
- Output:
  - `dist[a][b]` the shortest distance from a to b
    - If a path is needed, `prev[a][b]` is used just like MST, starting with a

Approach

- Standard shortest path gives shortest path from a given vertex s to every vertex
  - Repeat this for every starting vertex s
  - Dijkstra O(n (e lg n))
  - Bellman-Ford O(n (e n))
- Floyd-Warshall use Dynamic Programming Approach that gives $O(n^3)$

Floyd-Warshall

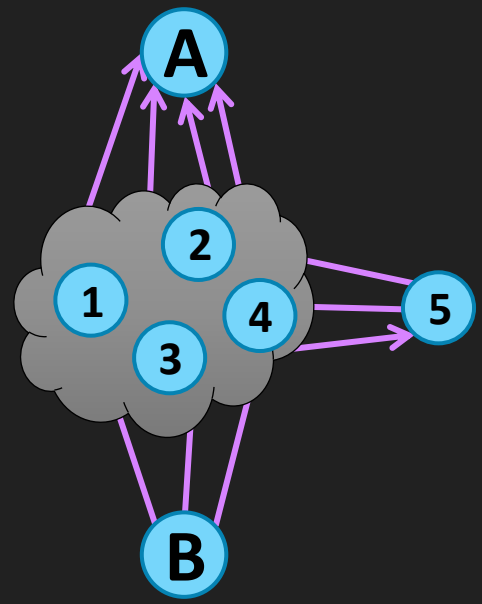
- Use another recurrent
- $d_k(a,b)$ is a shortest distance from a to b that the path can visit node only in the set {1,2,3,…,k}
  - I.e., the shortest path can be [a,p1,p2,…,pm,b] where 1 <= pi <= k and we can use any number of nodes in p1,…,pm
- $d_0(a,b) = w(a,b)$ because it can not go through any nodes

Recurrent Relation

- $d_k(a,b)$ = min of
  - $d_{k-1}(a,b)$
  - $d_{k-1}(a,k) + d_{k-1}(k,b)$
- Initial condition
  - $d_0(a,b) = w(a,b)$
  - $d_0(a,a) = 0$
- The answer to the problem is $d_n(*,*)$

There are exactly three dependency (comparing to Bellman-Ford where each value depends on P other entries where P is the in-degree)

Even though D is now 3 dimension, d_k depends on d_{k-1}
There is no need to remember multiple layer. So, 2D dimension array can be used.



Floyd-Warshall Implementation

```
def floyd_warshall(G,w)
  for i = 1 to n
    for j = 1 to n
      dist[i][j] = w(i,j)

  for k = 1 to n
    for i = 1 to n
      for j = 1 to n
        dist[i][j] =
          min (dist[i][j],
               dist[i][k] + dist[k][j])
  for i = 1 to n
    if dist[i][i] < 0
     return “negative cycle”
  return dist
end
```

Also assume that w(a,a) is 0

The node k must be the outermost loop

- Analysis
  - Very simple
  - 3 nested loops of n
  - $O(n^3)$
- Detecting negative cycle by checking if dist[a][a] is negative

Comparing with Bellman-Ford

- For dense graph (where e approach n^2)
  - Both Bellman-Ford and Floyd-Warshall has the same time complexity $O(n^3)$
    - May choose Floyd-Warshall over Bellman-Ford
- For sparse graph
  - Bellman-ford is $O(n\ e)$
  - Floyd-Warshall is still $O(n^3)$