EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

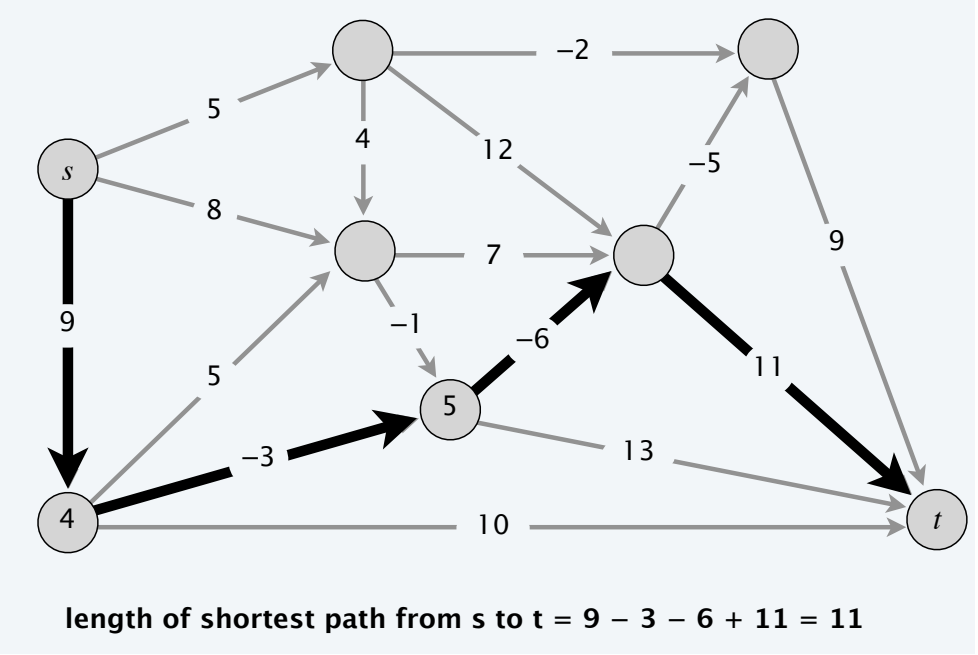
GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

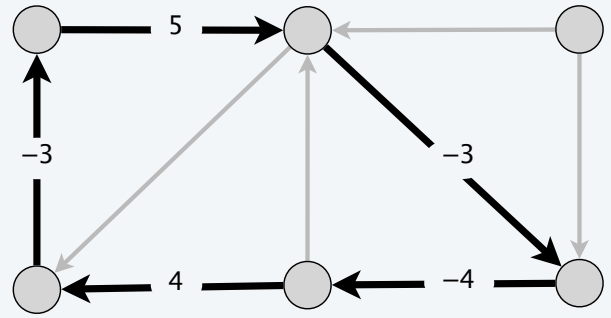
Shortest paths: Graphs with Negative Weights

- Need different algorithm
- Dijkstra's algorithm: "label-setting"
  - Once vertex exits priority queue, its distance label never changes
- Different class of algorithms: "label-correcting"
  - Exploit Bellman's principle of optimality
  - Will handle some cases with negative weights



Limitations: Negative Cycles

- A negative cycle is a directed cycle where the sum of the weights on the edges is negative
- Lemma: if some path from s to t contains a negative cycle, there does not exist a shortest path from s to t
  - Can go around the negative cycle an infinite amount of time…



- Lemma: if there exist no negative cycles in directed graph, then there exists a shortest path from s to t that is simple (no cycles) and has < |V| edges
  - Consider minimum number of edges among all shortest paths from s to t
  - If numbers edges $\geq |V|$, then it must have a cycle
  - If this path has a cycle, can remove cycle and get one with less edges and same total weight

Dynamic Programming

- Dynamic programming: an algorithmic technique for solving an optimization problem involving sequences of decisions by
  - Breaking it down into simpler subproblems
  - Exploiting that the optimal solution to the overall problem depends upon the optimal solution to its subproblems
  - Bellman's principle of optimality: shortest path from s to t is composed of shortest paths to and from any intermediate vertices
- Idea: let's solve subproblem of finding shortest path with a maximum number of edges
  - Easy if we limit it to 1; if we extend it to |V|-1, we solved the full problem

Bellman-Ford Algorithm

- Notation: D(k,v) is shortest distance from vertex s to vertex v on paths with less than or equal to k edges

  Initial value: $D(0,v) = \infty,\ v \neq s;\ 0,\ v = s$

  Bellman equation $D(k,v) = \min\left\{ D(k-1,v),\ \min_{(u,v)\in E} [D(k-1,u) + w(u,v)] \right\}$ (DP)

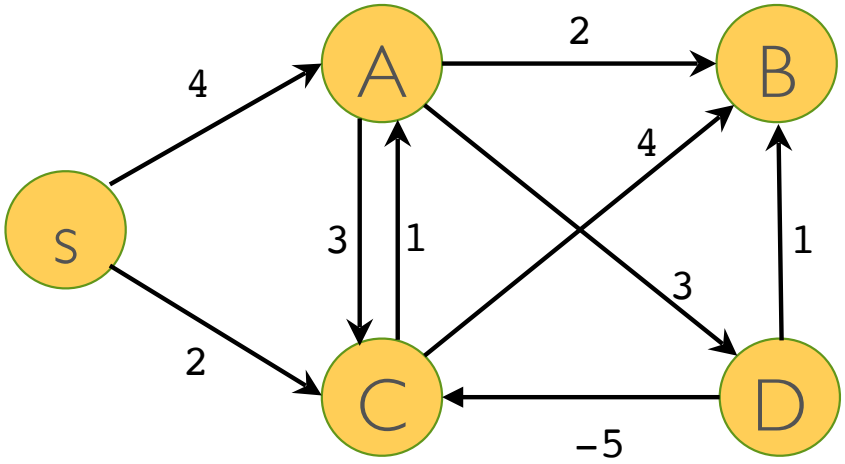
- Bellman-Ford Algorithm
  - For each $v \in V$, set $D(0,v) = \infty,\ v \neq s;\ 0,\ v = s$; set pred[v] = null
  - For k = 1 to |V| - 1:
    - For each $v \in V$, set $D(k,v) = D(k-1,v)$
    - For each $(u,v) \in E$:
      - if $D(k,v) > D(k-1,u) + w(u,v)$:
        - Set $D(k,v) = D(k-1,u) + w(u,v)$, set pred[v] = u

Bellman-Ford Algorithm Properties

- D(k,v) is the shortest distance for a path of k or less edges from s to v
  - Prove by induction: obviously true at k = 0. Assume true for k-1. Assume $D(k,v) = D(k-1,v)$. Then, from the iteration, no path with k vertices is shorter than D(k,v). Note that $D(k-1,u) + w(u,v)$ is the shortest distance of all paths of length k or less that end in v and have next to last vertex u. This proves the result, because $D(k,v) = \min_{(u,v)\in E} D(k-1,u) + w(u,v)$
- At each iteration, D(k,v) is upper bound on the shortest path distance from s to v
- D(k,v) is non-increasing in k
- If $D(k-2,v) = D(k-1,v)$, one does not have to scan edges from node v for D(k,v)
  - Best path to v is less than k-1 edges
- Unlike Dijkstra's algorithm, each edge is examined more than once; once per k
- Brute force complexity: |V| iterations, each of which is O(|E|) —> complexity O(|V||E|)

Bellman-Ford Algorithm Properties

- Algorithm converges to shortest paths provided no negative cycles exist in graph
- If negative cycles exist, predecessor graph will have cycles!
- Distances continue decreasing after |V|-1 iterations (neg. cycle detection)



	D(s)	D(A)	D(B)	D(C)	D(D)
k=1	0	4	Inf	2	Inf
k=2	0	3	6	2	7
k=3	0	3	5	2	6
k=4	0	3	5	1	6

Final pred array: pred[s] = null, pred[A] = C, pred[B] = A, pred[C] = D, pred[D] = A

Better Bellman-Ford Implementation

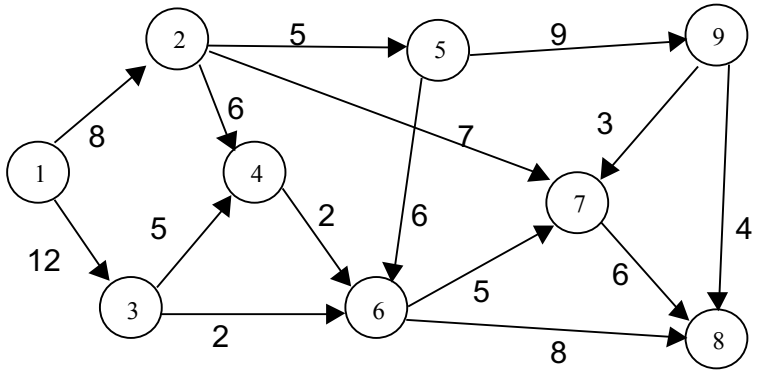
- Use a queue of vertices where distances have changed
  - Avoid looking at edges out of vertices where distances have not changed
- Algorithm
  - Initialize $D[s] = 0;\ D[v] = \infty, v \neq s;$ pred$[v] = null$
  - Insert s into queue Q; mark inqueue[s] = true, mark inqueue[v] = false, $v \neq s$
  - While Q is not empty:
    - Select u out of queue, mark inqueue[u] = false
    - For each edge (u,v) in E:
      - If $D[v] > D[u] + w(u, v)$:
        - Set $D[v] = D[u] + w(u, v)$, set pred[v] = u
        - if inqueue[v] = false, add v to Q, mark inqueue[v] = true

Better Bellman-Ford Implementation

- Many variations possible
  - Have not specified in what order to insert vertices into queue Q
  - Have not specified in what order to extract vertices from Q
  - Algorithm works anyway, as long as there are no negative weight cycles
- Detecting negative cycles
  - A negative cycle exists if and only if a vertex enters the Q |V| times
- Simplest implementation: use queue in FIFO order
- More interesting implementations: insert vertices with shorter distances D[v] in the front of the queue
- Convergence happens because distance labels D[v] decrease monotonically, and are bounded below
- Bellman-Ford and variations are known as label-correcting algorithms: distance labels D[v] are not set when v leaves the Queue
  - Less overhead than Dijkstra; can be empirically faster

Example

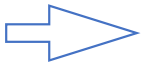
- Source node 1



If $D[v] > D[u] + w(u, v)$, set $D[v] = D[u] + w(u, v)$

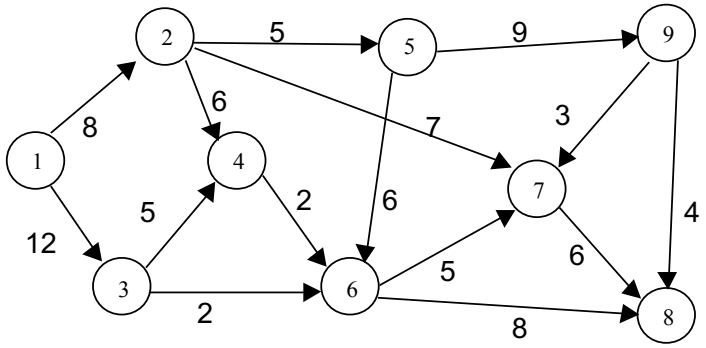
	D[ ]	pred[ ]	Q [ ]
1	0	Null	1
2	Inf	Null	
3	Inf	Null	
4	Inf	Null	
5	Inf	Null	
6	Inf	Null	
7	Inf	Null	
8	Inf	Null	
9	Inf	Null	

Pop 1



	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	1
3	12	1	2
4	Inf	Null	
5	Inf	Null	
6	Inf	Null	
7	Inf	Null	
8	Inf	Null	
9	Inf	Null	

Example - 2



If $D[v] > D[u] + w(u, v)$, set $D[v] = D[u] + w(u, v)$

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	1
3	12	1	2
4	Inf	Null	
5	Inf	Null	
6	Inf	Null	
7	Inf	Null	
8	Inf	Null	
9	Inf	Null	

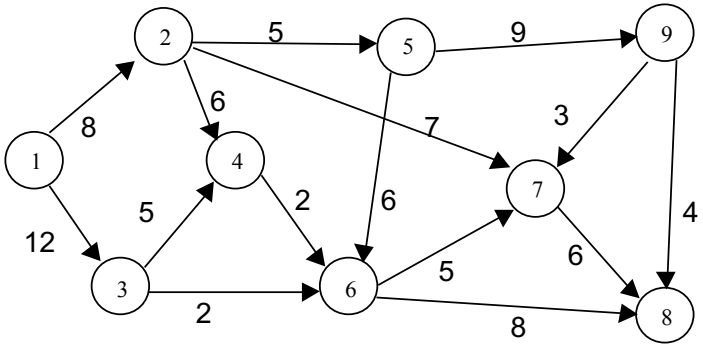
Pop 2

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	1
4	14	2	2
5	13	2	3
6	Inf	Null	
7	15	2	4
8	Inf	Null	
9	Inf	Null	

Pop 3

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	1
5	13	2	2
6	14	3	4
7	15	2	3
8	Inf	Null	
9	Inf	Null	

Example - 3



If $D[v] > D[u] + w(u, v)$, set $D[v] = D[u] + w(u, v)$

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	1
5	13	2	2
6	14	3	4
7	15	2	3
8	Inf	Null	
9	Inf	Null	

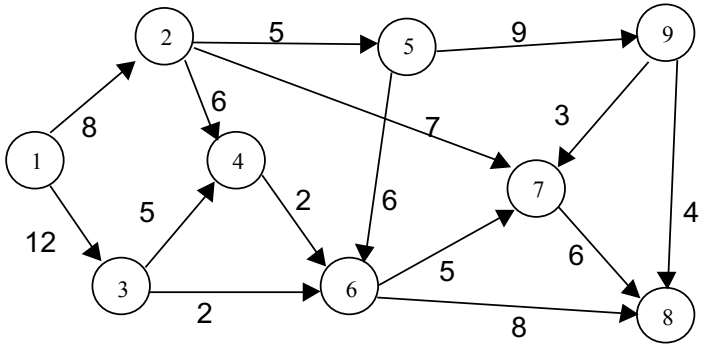
Pop 4

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	1
6	14	3	3
7	15	2	2
8	Inf	Null	
9	Inf	Null	

Pop 5

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	2
7	15	2	1
8	Inf	Null	
9	22	5	3

Example - 4



If $D[v] > D[u] + w(u, v)$, set $D[v] = D[u] + w(u, v)$

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	2
7	15	2	1
8	Inf	Null	
9	22	5	3

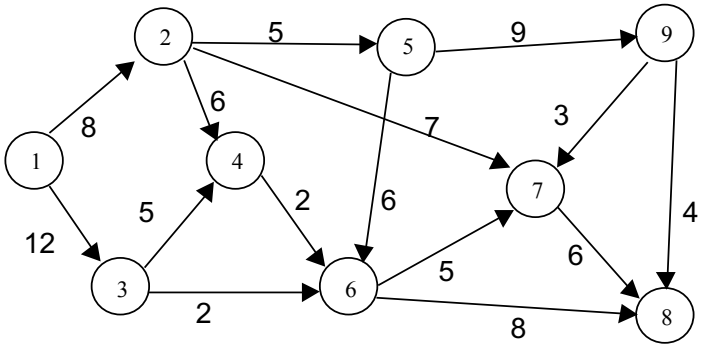
Pop 7

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	1
7	15	2	
8	21	7	3
9	22	5	2

Pop 6

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	2
9	22	5	1

Example - 5



If $D[v] > D[u] + w(u, v)$, set $D[v] = D[u] + w(u, v)$

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	2
9	22	5	1

Pop 9

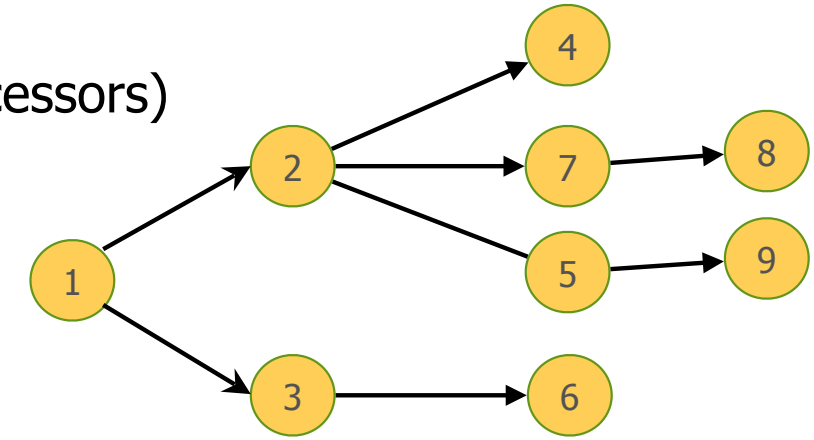
	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	1
9	22	5	

Pop 8

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	
9	22	5	

Example - 6

- Observations
  - Each vertex left queue only once, so edges out of each vertex were scanned only once
  - With simple data structure, this is much faster than Dijkstra
  - Compared with full Bellman-Ford, each edge would be scanned 8 times (|V| -1)
- Final shortest path tree (use predecessors)



Bellman-Ford vs Dijkstra

- Dijkstra's algorithm
  - At each step, finds the next vertices that is closest to the start vertex
  - Grows the shortest path tree in order of shortest distance
  - Greedy: no backtracking
  - O(|E| + |V| log(|V|)) with complex data structures
  - Requires non-negative weights
- Bellman-Ford
  - At each step, finds shortest distance to vertices over paths with less than k edges
  - Grows the shortest path tree in order of minimum number of edges from start vertex
  - Backtracks, can scan the same vertex multiple times
  - O(|E||V|) worst case with very simple data structures
  - Can handle negative weights, but no negative cycles
  - Will work asynchronously, perfect for parallel applications

New Variation: All Pairs Shortest Paths

Input: Directed graph G = (V, E), where V = {1, 2, ..., n}, with edge-weight function w : E → R

- Weights may be negative, but no negative cycles

Output: n × n matrix of shortest-path lengths D(i, j) for all i, j ∈ V (and routes)

Possible solution:

- Run Bellman-Ford once from each vertex as starting node
- Time $O(|V|^2 E)$
- Dense graph ($\Theta(n^2)$ edges) ⇒ $O(n^4)$ time in the worst case

New Variation: All Pairs Shortest Paths

Input: Directed graph G = (V, E), where V = {1, 2, ..., n}, with edge-weight function w : E → R

- Assume al weights are non-negative

Possible solution:

- Run Dijkstra once from each vertex as starting node
- Time $O\big(|V||E| + |V|^2 \log(|V|)\big)$
- Dense graph ($\Theta(n^2)$) ⇒ $O(n^3)$ time in the worst case with array priority queue

Better Approach: Use Dynamic Programming

- Concept
  - Let D(i,j) be the shortest distance from vertex i to vertex j.
  - Assume D(j,j) = 0
  - Then, shortest path from i to j must have at least one vertex before j (could be i)

$$D(i,j) = \min_{k:\ (k,j)\in E} [D[i,k] + w(k,j)]$$

  - Bellman's principle of optimality: shortest path is composed of shortest paths

$$D(i,j) = \min_{k:\ k\in V} [D[i,k] + D[k,j]]$$

Floyd-Warshall Algorithm

- Initialize $n \times n$ weighted adjacency matrix A and initial distance matrix $D^{(0)}$ as

$$A(i,j) = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E, \\ \infty & \text{if } (i,j) \notin E \end{cases} \qquad D^{(0)}(i,j) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

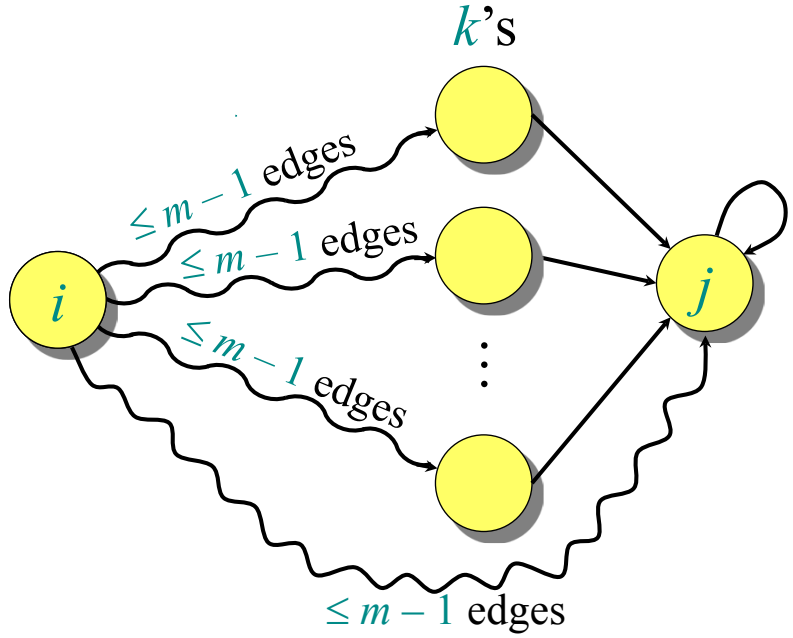
- Note that $D^{(0)}$ is the matrix of shortest distances among paths with zero edges!
- A simple algorithm
  - For m = 1, 2, …, n-1:

$$D^{(m)}(i,j) = \min_{k \in V}[D^{(m-1)}(i,k) + A(k,j)]$$

  - Note that $A(j,j) = 0$, so that $D^{(m)}(i,j) = D^{(m-1)}(i,j)$ is possible, and thus $D^{(m)}(i,j)$ is non-increasing

Floyd-Warshall Algorithm

- Claim: $D^{(m)}(i, j)$ is shortest distance from i to j among paths with m or less edges
  - Any path with less than or equal to m edges must have next to last vertex with shortest path less than m-1 edges
  - Principle of optimality



Floyd-Warshall Algorithm

- Variation: faster algorithm with less memory

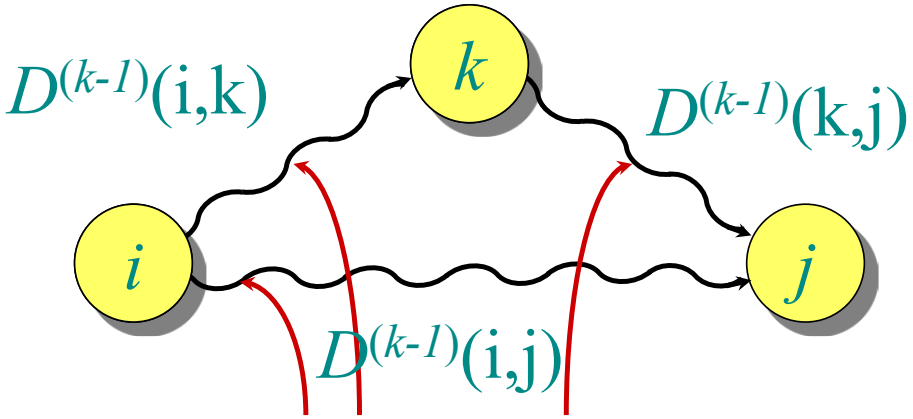
$$\text{Initially } D(i,j) = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$$

- For k in V:
  - For i in V:
    - For j in V:

      If $D(i,j) > D(i,k) + D(k,j)$:

      set $D(i,j) = D(i,k) + D(k,j)$

  - Complexity: $O(|V|^3)$

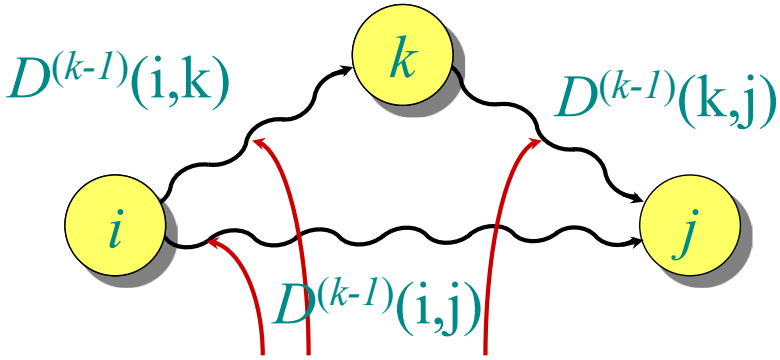


Floyd-Warshall Algorithm

- Keeping track of paths

Initially $pred(i,j) = \begin{cases} i & (i,j) \in E \\ -1 & (i,j) \notin E \end{cases}$ $D(i,j) = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$

- For k in V:
  - For i in V:
    - For j in V:



If $D(i,j) > D(i,k) + D(k,j)$:

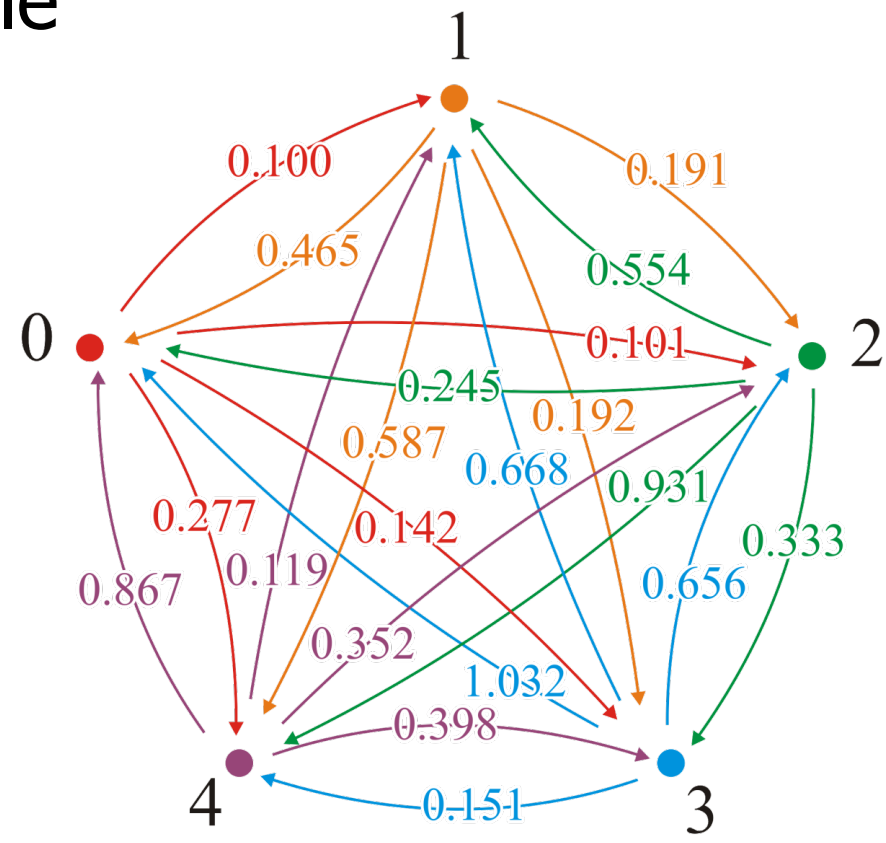
set $D(i,j) = D(i,k) + D(k,j)$, $pred(i,j) = pred(k,j)$

Example

- Weighted adjacency matrix

- A=

0	0.1	0.101	0.142	0.277
0.465	0	0.191	0.192	0.587
0.245	0.554	0	0.333	0.931
1.032	0.668	0.656	0	0.151
0.867	0.119	0.352	0.398	0



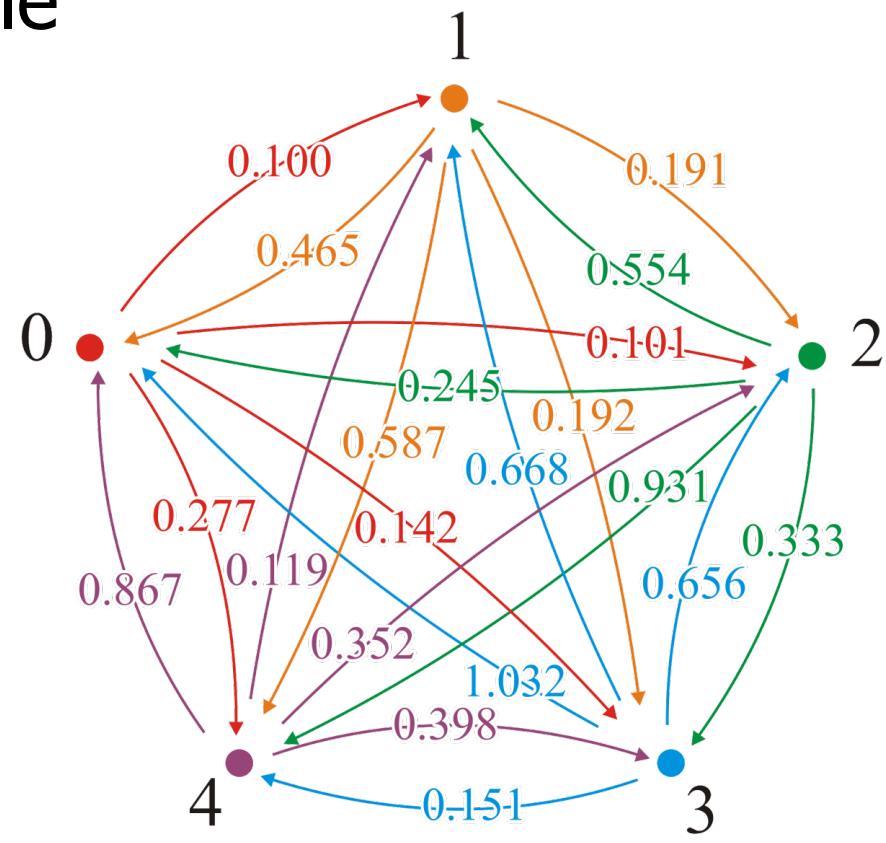
Initially, D = A (fully connected)

Example

- Iteration for k = 0
  - D=

0	0.1	0.101	0.142	0.277
0.465	0	0.191	0.192	0.587
0.245	0.345	0	0.333	0.522
1.032	0.668	0.656	0	0.151
0.867	0.119	0.352	0.398	0

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

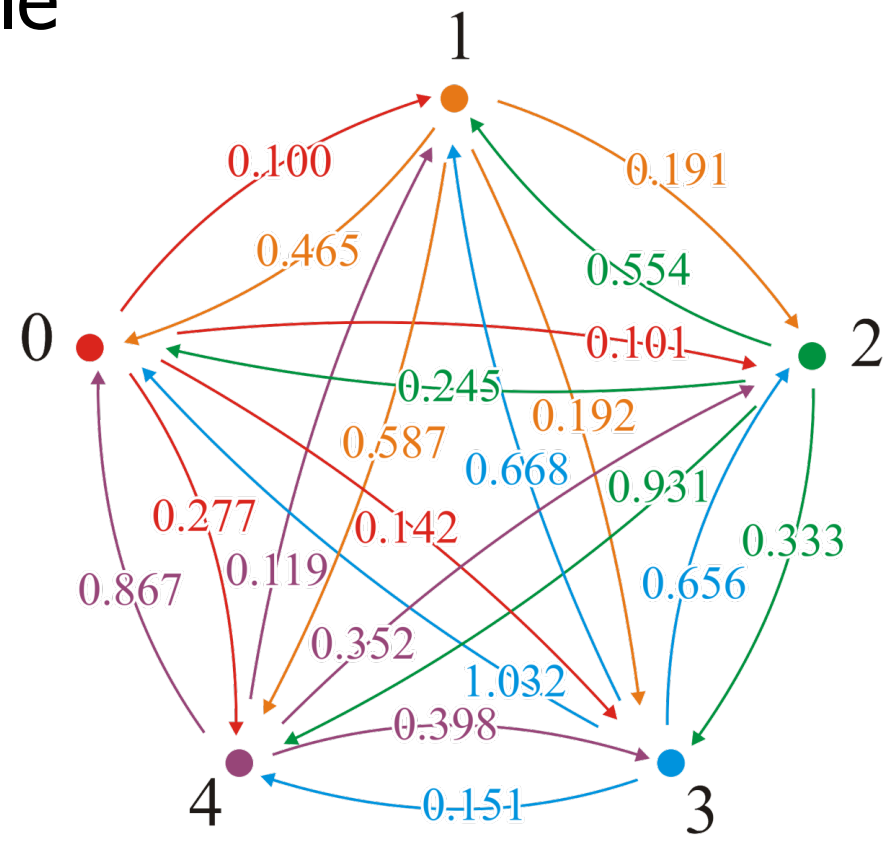


Example

- Iteration for k = 1
  - D=

0	0.1	0.101	0.142	0.277
0.465	0	0.191	0.192	0.587
0.245	0.345	0	0.333	0.522
1.032	0.668	0.656	0	0.151
0.584	0.119	0.31	0.311	0

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

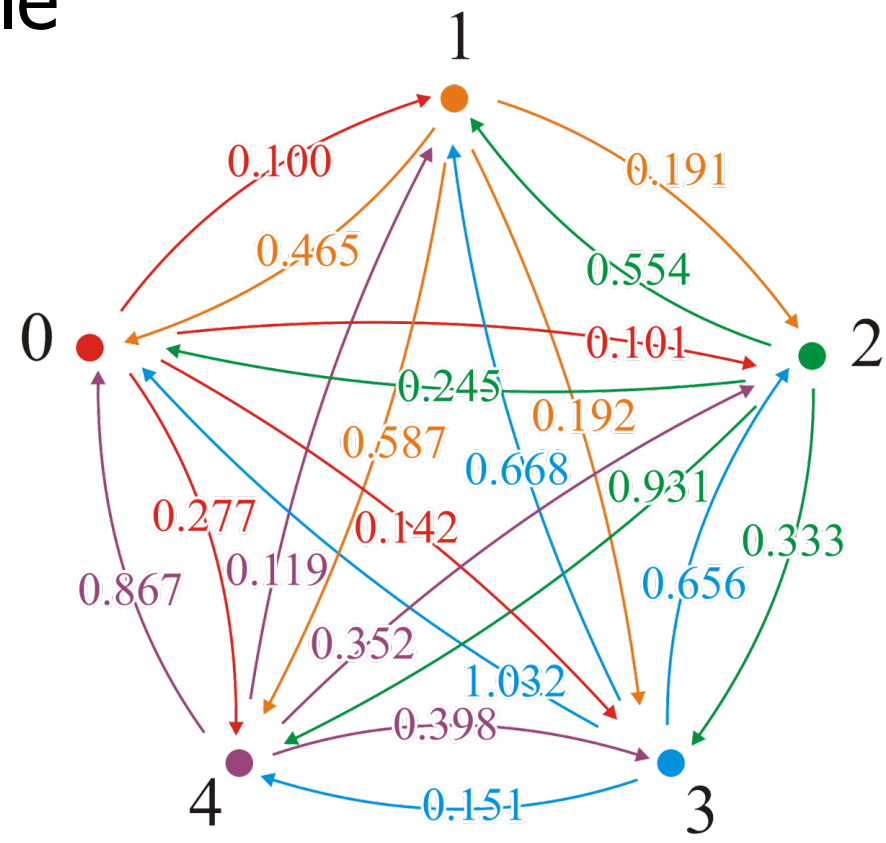


Example

- Iteration for k = 2
  - D=

0	0.1	0.101	0.142	0.277
0.436	0	0.191	0.192	0.587
0.245	0.345	0	0.333	0.522
0.901	0.668	0.656	0	0.151
0.555	0.119	0.31	0.311	0

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

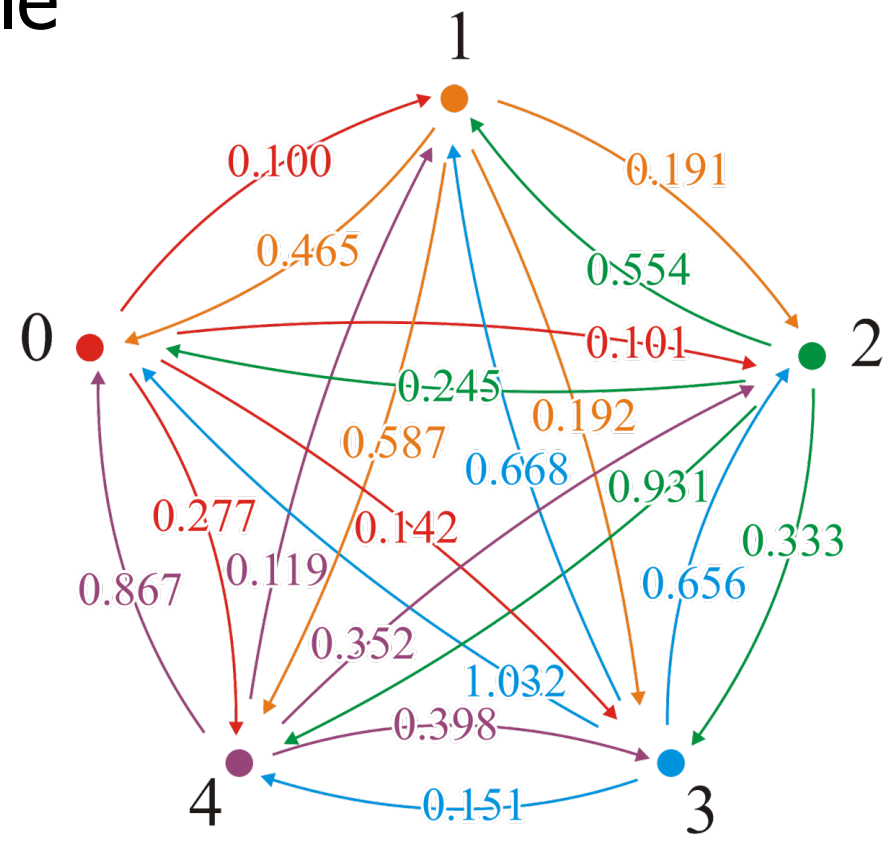


Example

- Iteration for k = 3
  - D=

0	0.1	0.101	0.142	0.277
0.436	0	0.191	0.192	0.343
0.245	0.345	0	0.333	0.484
0.901	0.668	0.656	0	0.151
0.555	0.119	0.31	0.311	0

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

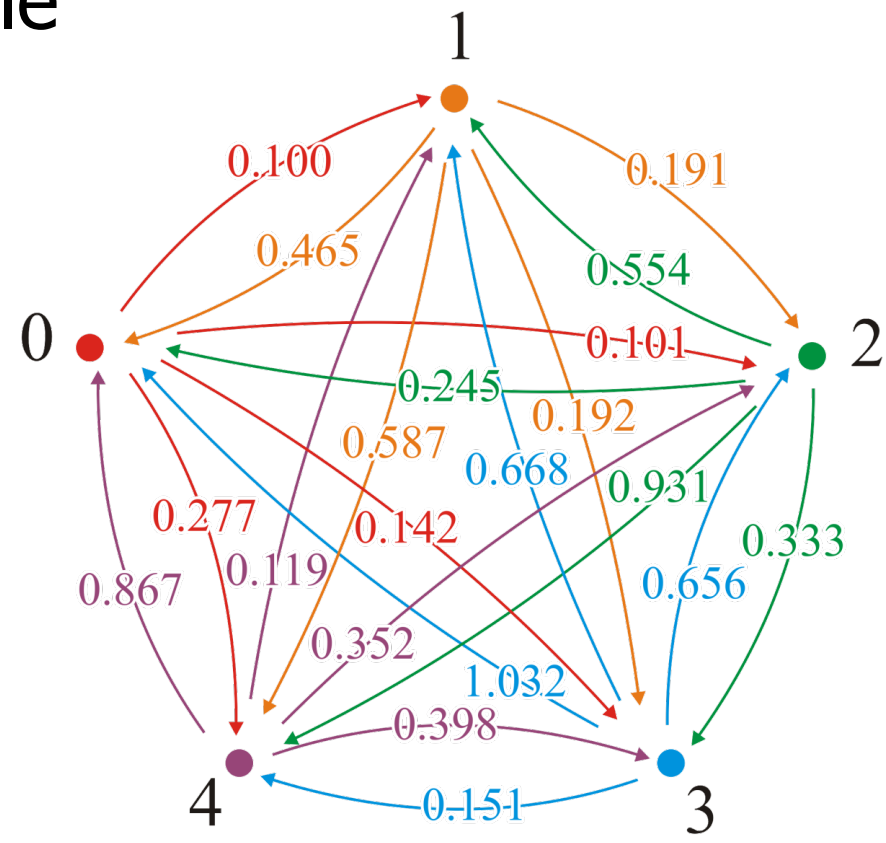


Example

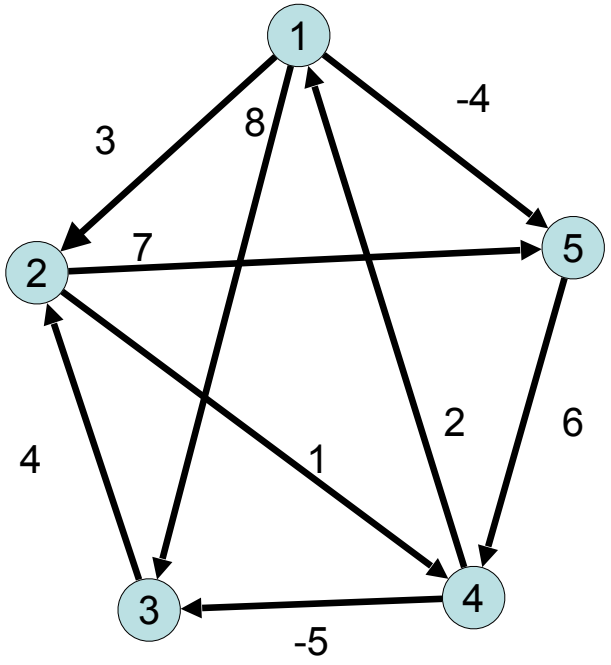
- Iteration for k = 4
  - D=

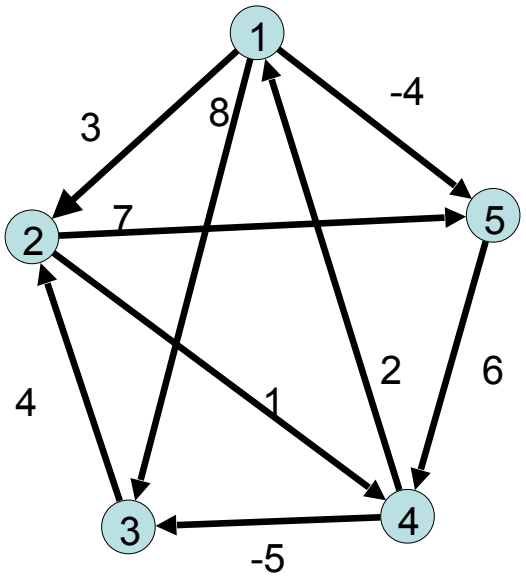
0	0.1	0.101	0.142	0.277
0.436	0	0.191	0.192	0.343
0.245	0.345	0	0.333	0.484
0.706	0.27	0.461	0	0.151
0.555	0.119	0.31	0.311	0

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



Floyd-Warshall: Example 2

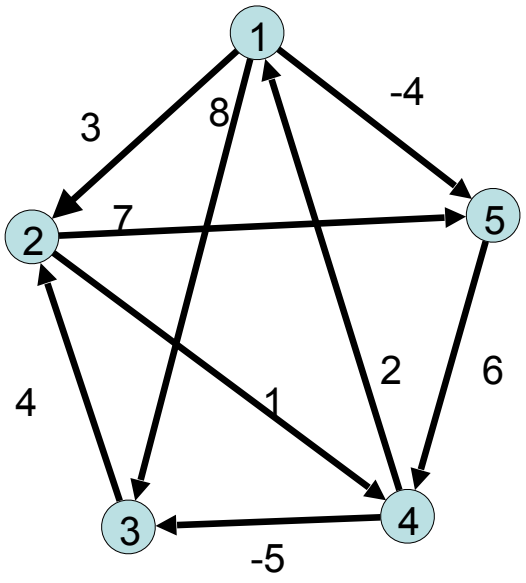




Floyd-Warshall

Initial distances and predecessors

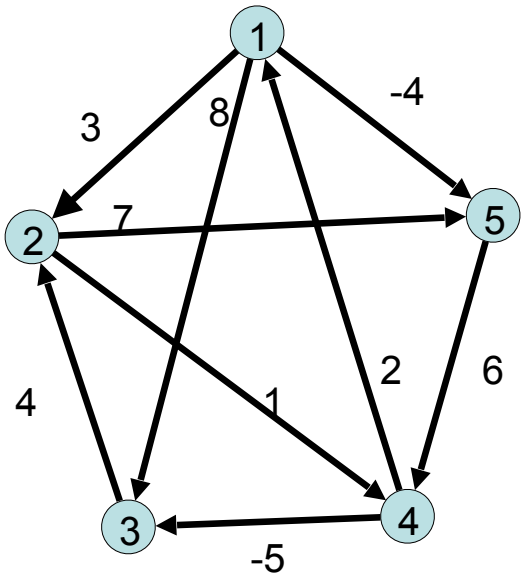
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 1

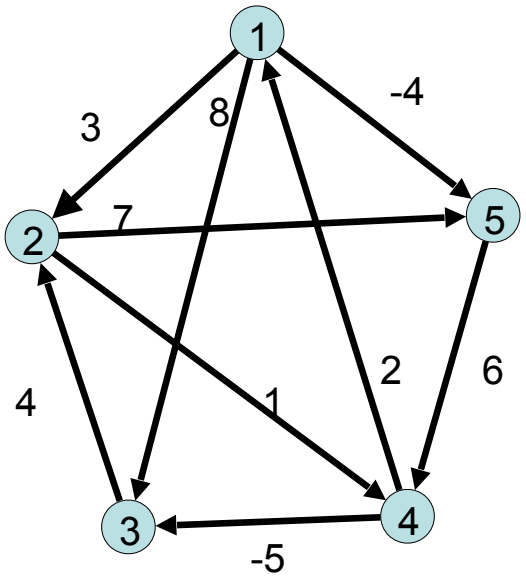
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 2

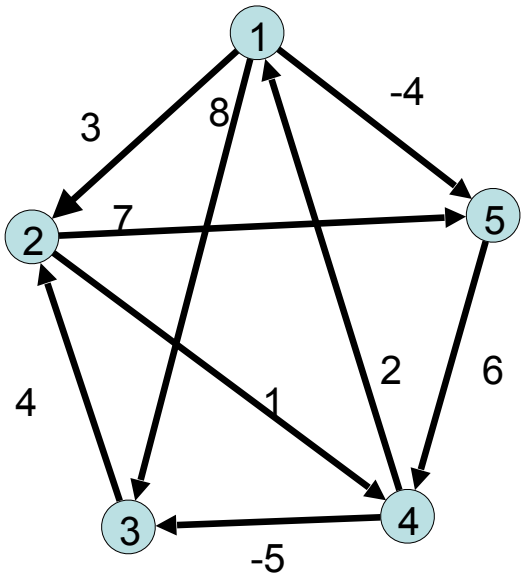
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 3

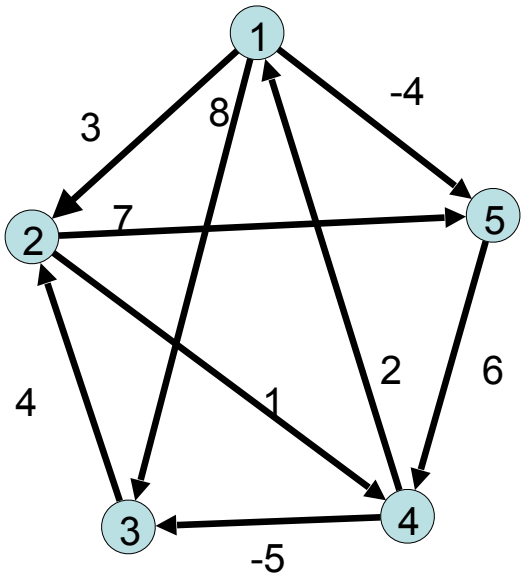
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 4

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$



Floyd-Warshall

After k = 5 (final)

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Can We Do Better for Sparse Graphs?

- If graph is sparse, we can do better with Dijkstra from every vertex, provided weights are non-negative
  - Dijkstra: $O(|V||E| + |V|^2 \log(|V|))$ vs Floyd-Warshall $O(|V|^3)$

- Can we apply Dijkstra when graph has negative weight edges, but not negative cycles?
  - Graph reweighing

Graph Reweighing

Theorem. Given a function h : V → R, **reweigh** each edge (u,v) ∈ E by $w_h(u,v)=w(u,v)+h(u)–h(v)$.

Then, for any two vertices, all paths between them are reweighed by the same amount
==> shortest path in reweighed graph is also shortest in original path

Proof. Let $p = v_1 \to v_2 \to \cdots v_k$ be a path in the original graph G. We have

$$w(p) = w(v_1, v_2) + w(v_2, v_3) + \cdots + w(v_{k-1}, v_k)$$

On the reweighed graph G_h, we have

$$\begin{aligned} w_h(p) &= w_h(v_1, v_2) + w_h(v_2, v_3) + \cdots + w_h(v_{k-1}, v_k) \\ &= w(v_1, v_2) + h(v_1) - h(v_2) + w(v_2, v_3) + h(v_2) - h(v_3) + \cdots + w(v_{k-1}, v_k) + h(v_{k-1}) - h(v_k) \\ &= w(p) + h(v_1) - h(v_k) \end{aligned}$$

Same amount for every path that starts at v_1, ends at v_k!

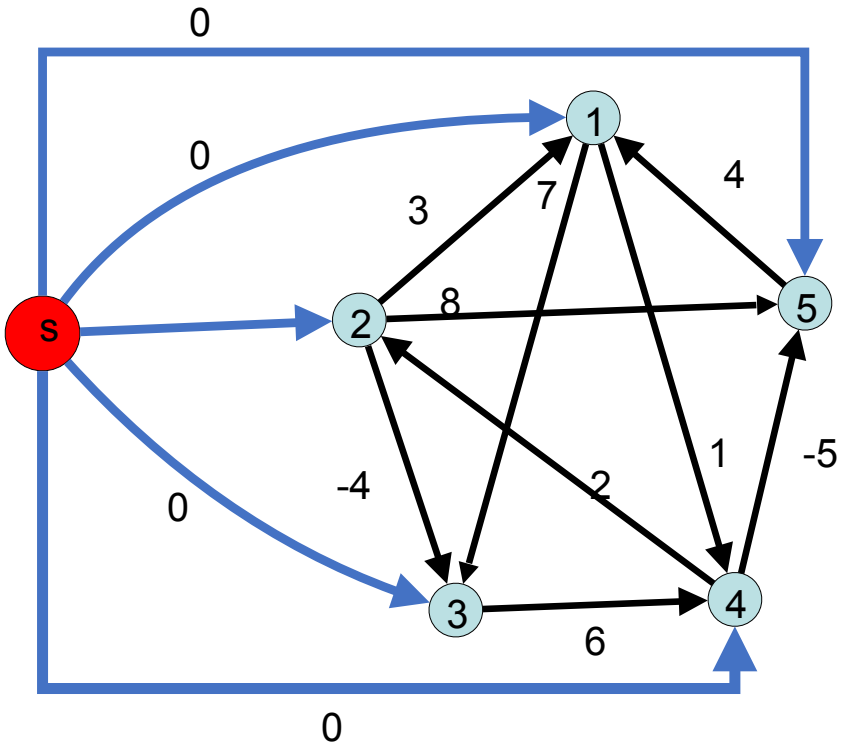
Can We Do Better for Sparse Graphs?

- So, want to find a reweighing function such that $w_h(u, v) \geq 0$ for all edges in E
  - Then, we could solve using Dijkstra's algorithm starting from each vertex
- **Johnson's** algorithm
  - Use an auxiliary shortest path problem solved with Bellman-Ford to using Bellman-Ford to find h(v) such that h(v) – h(u) ≤ w(u, v), or determine that a negative-weight cycle exists.
    • Time = O(|V| |E|)
  - Resulting problem with $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$

Johnson's algorithm

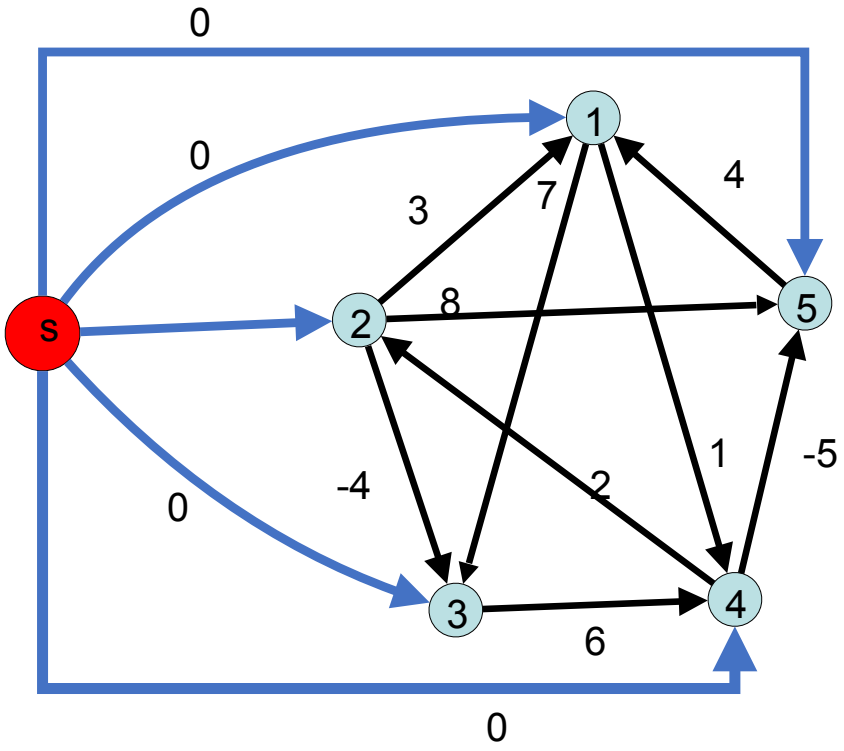
```
Johnson(G)
{   compute G', where V[G']=V[G]∪{s} and
                        E[G']=E[G]∪{(s,v):v∈V[G]
    if Bellman-Ford(G',w,s)=False
       then print "There is a neg-weight cycle"
    else for each vertex v ∈ V[G']
              set h(v)=δ(s,v) computed by Bellman-Ford algo.
              for each edge (u,v)∈E[G']:
                  w'(u,v)=w(u,v)+h(u)-h(v)
              for each vertex u ∈V[G']
                   run Dijkstra(G,w',u) to compute δ'(u,v)
              for each v ∈V[G]
                   d_uv=δ'(u,v)-h(u)+h(v)
    return D
}
```

Johnson's algorithm



Main idea: Add a dummy vertex with a directed edge of weight 0 to all other vertices

Johnson's algorithm

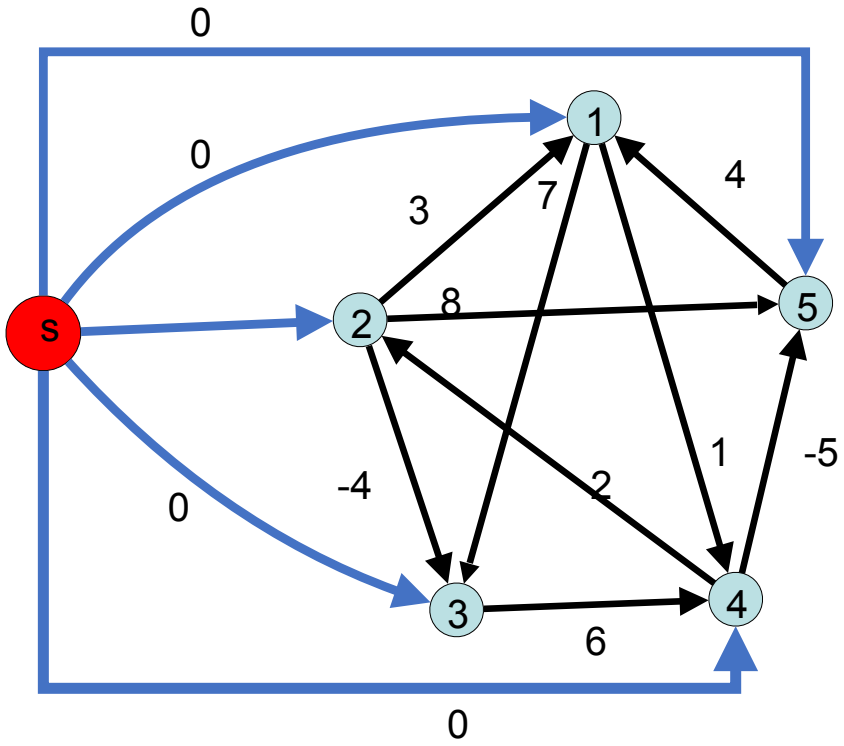


Use Bellman-Ford from s to get shortest distance to all other nodes:

Must be negative

	D[ ]	pred[ ]	Q [ ]
1	0	S	1
2	0	S	2
3	0	S	3
4	0	S	4
5	0	S	5

Johnson's algorithm



Pop 1

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	1
3	0	S	2
4	0	S	3
5	0	S	4

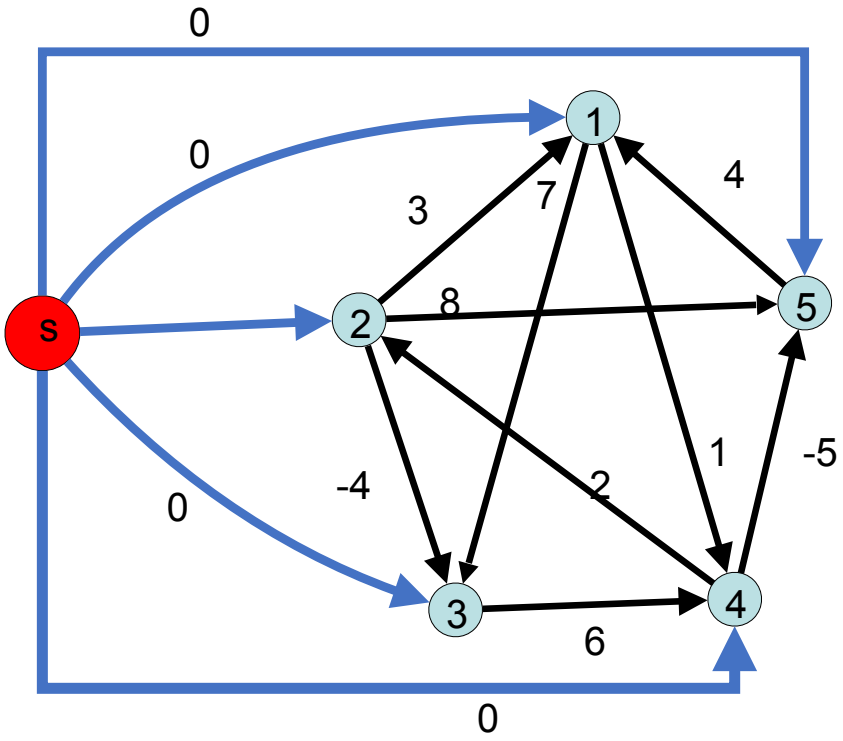
Pop 2

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	1
4	0	S	2
5	0	S	3

. . .

All-Pairs Shortest Paths

Johnson's algorithm



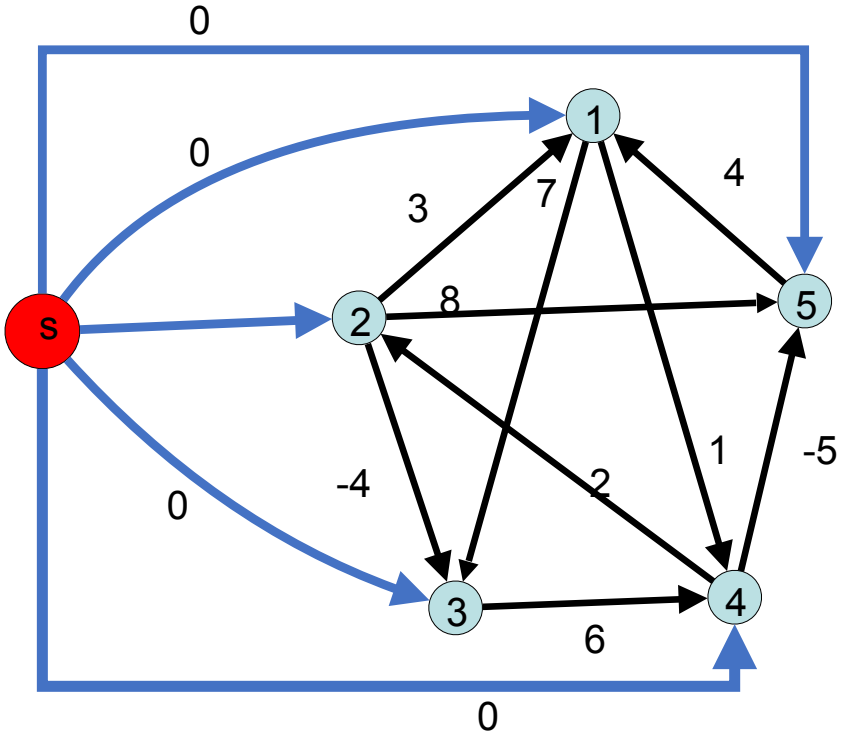
Pop 3

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	
4	0	S	1
5	0	S	2

Pop 4

	D[ ]	pred[ ]	Q [ ]
1	0	S	
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	1

Johnson's algorithm



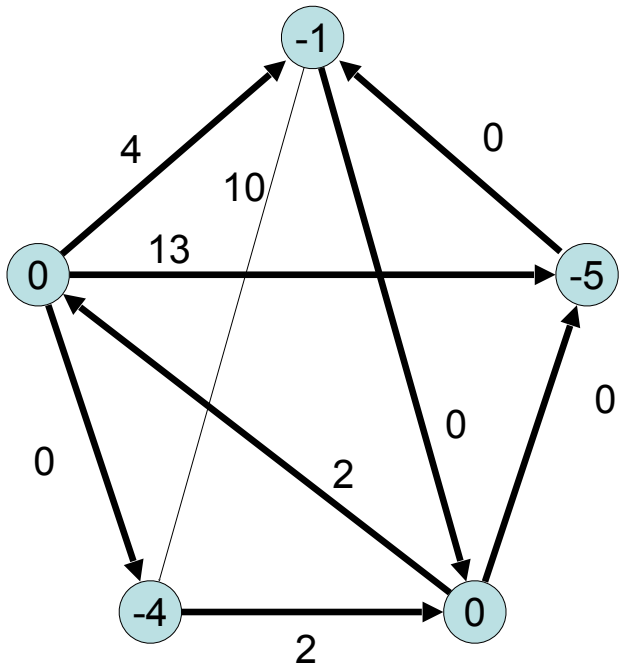
Pop 5

	D[ ]	pred[ ]	Q [ ]
1	-1	5	1
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	

Pop 1

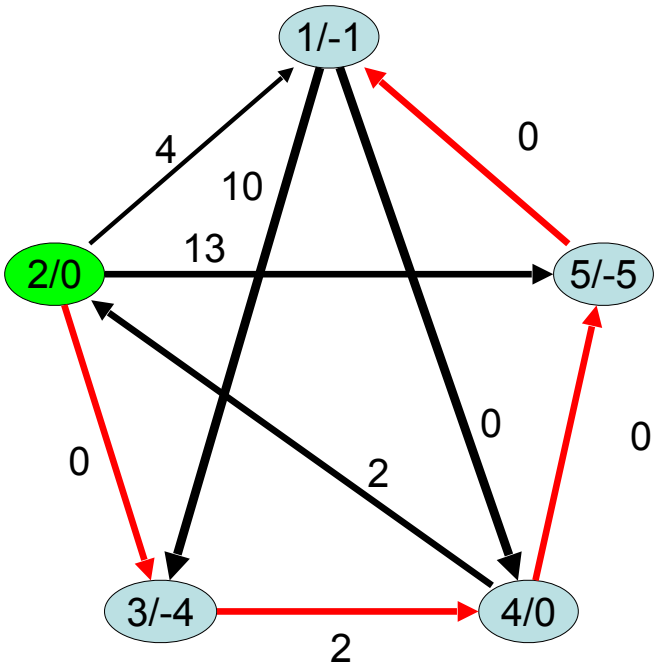
	D[ ]	pred[ ]	Q [ ]
1	-1	5	
2	0	S	
3	-4	2	
4	0	S	
5	-5	4	

Johnson's algorithm



Use Distance from previous problem for reweighing

Johnson's algorithm



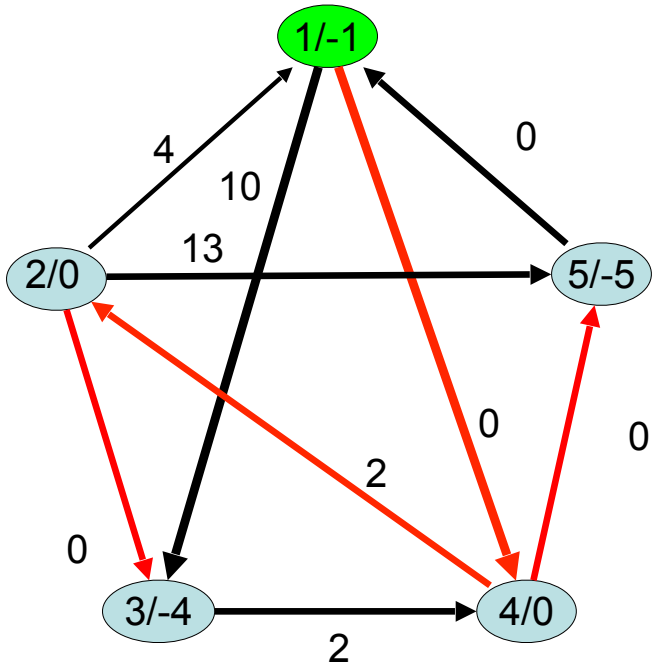
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	0
3	Inf	Null	
4	Inf	Null	
5	inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	0
4	Inf	Null	
5	13	2	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	0
5	13	2	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	
5	2	4	1

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	0	null	
3	0	2	
4	2	3	
5	2	4	1



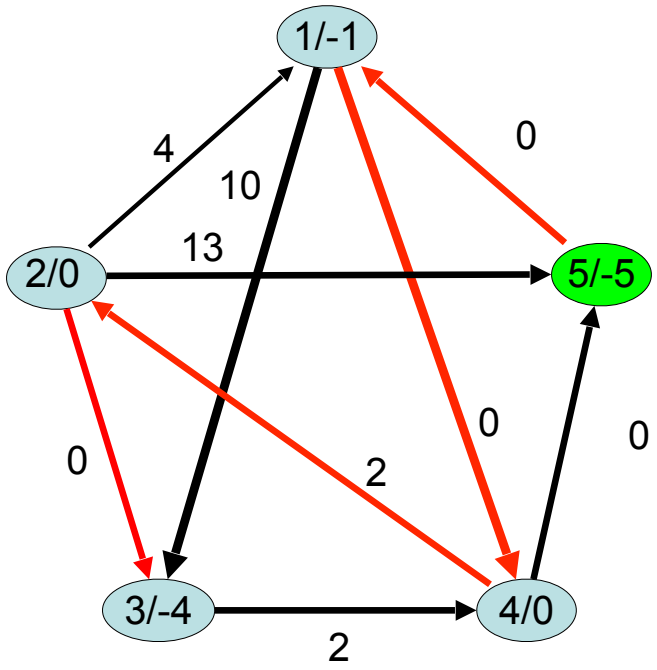
	D[ ]	pred[ ]	Q [ ]
1	0	Null	0
2	Inf	null	
3	Inf	Null	
4	Inf	Null	
5	inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	2	null	2
3	10	1	1
4	0	1	
5	0	5	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	null	
3	0	2	0
4	0	1	
5	0	5	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	Inf	null	
3	10	1	1
4	0	1	0
5	Inf	null	

	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	2	null	0
3	10	1	1
4	0	1	
5	0	5	



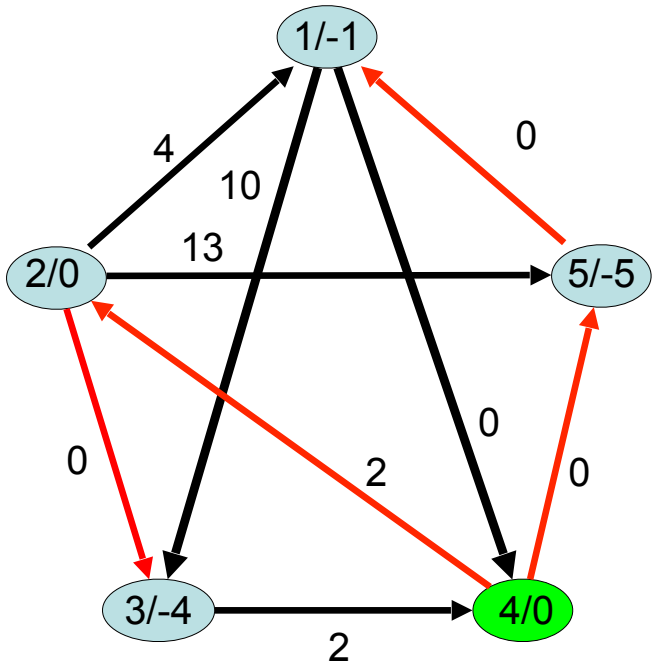
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	Inf	Null	
4	Inf	Null	
5	0	Null	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	0
2	Inf	null	
3	inf	Null	
4	inf	Null	
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	Inf	null	
3	10	1	1
4	0	1	0
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	0
3	10	1	1
4	0	1	
5	0	null	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	
3	2	2	0
4	0	1	
5	0	null	



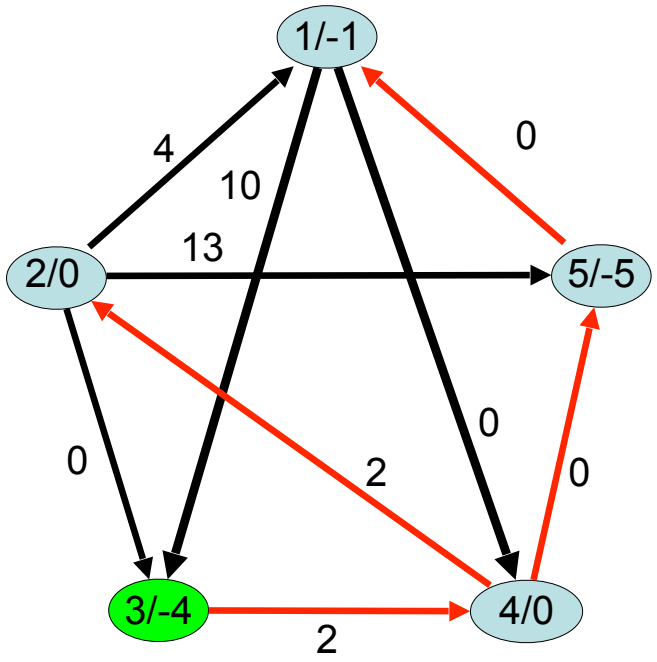
	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	Inf	Null	
4	0	Null	0
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	2	4	1
3	Inf	Null	
4	0	Null	
5	0	4	0

	D[ ]	pred[ ]	Q [ ]
1	0	5	0
2	2	4	1
3	Inf	Null	
4	0	Null	
5	0	4	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	0
3	10	1	1
4	0	Null	
5	0	4	

	D[ ]	pred[ ]	Q [ ]
1	0	5	
2	2	4	
3	2	3	0
4	0	Null	
5	0	4	



	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	0	Null	0
4	inf	Null	
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	Inf	null	
3	0	Null	
4	2	3	0
5	Inf	Null	

	D[ ]	pred[ ]	Q [ ]
1	Inf	Null	
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	0

	D[ ]	pred[ ]	Q [ ]
1	2	5	0
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	

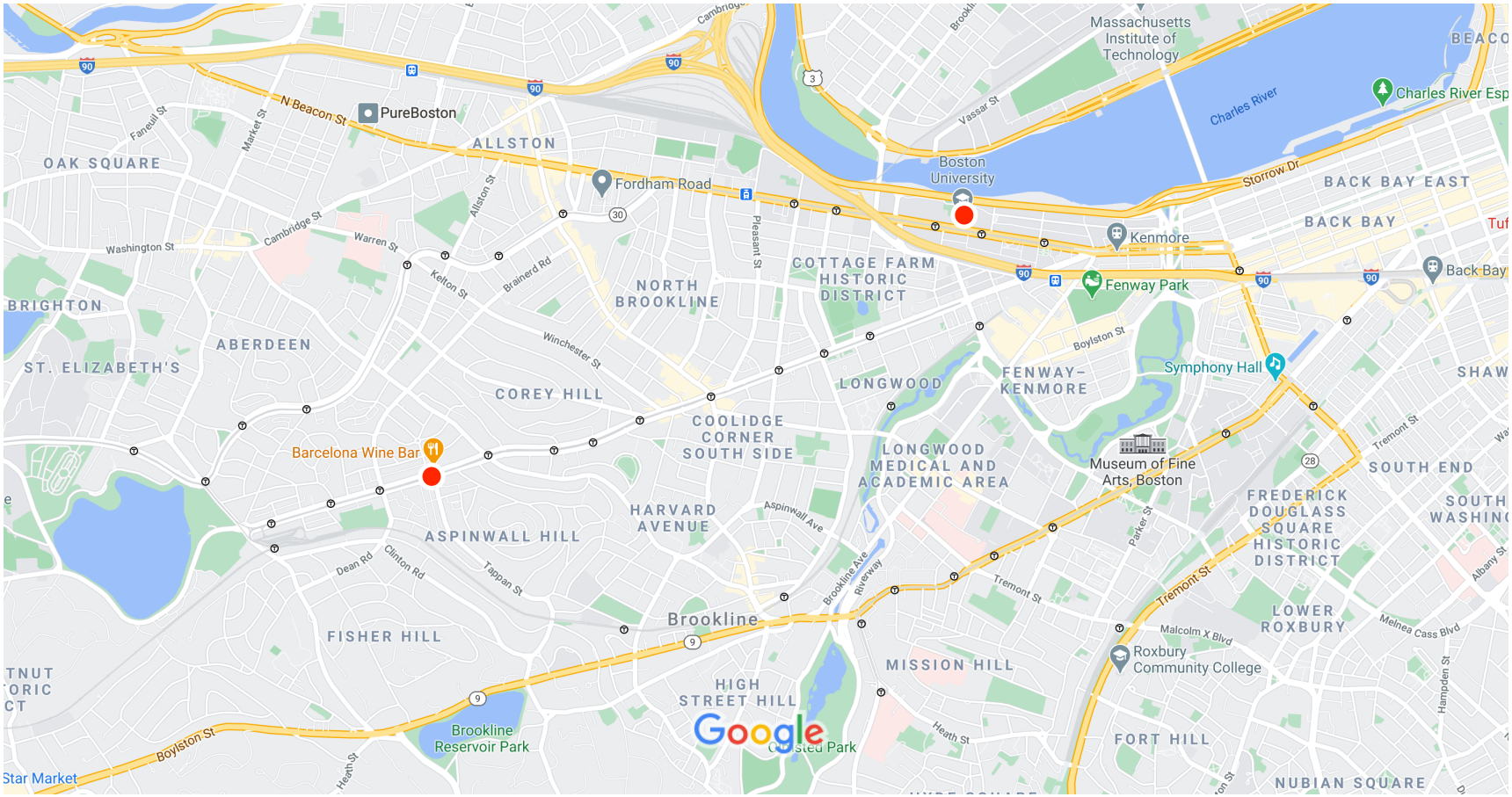
	D[ ]	pred[ ]	Q [ ]
1	2	5	
2	4	4	1
3	0	Null	
4	2	3	
5	2	4	

Problem: Shortest Path from s to a single t

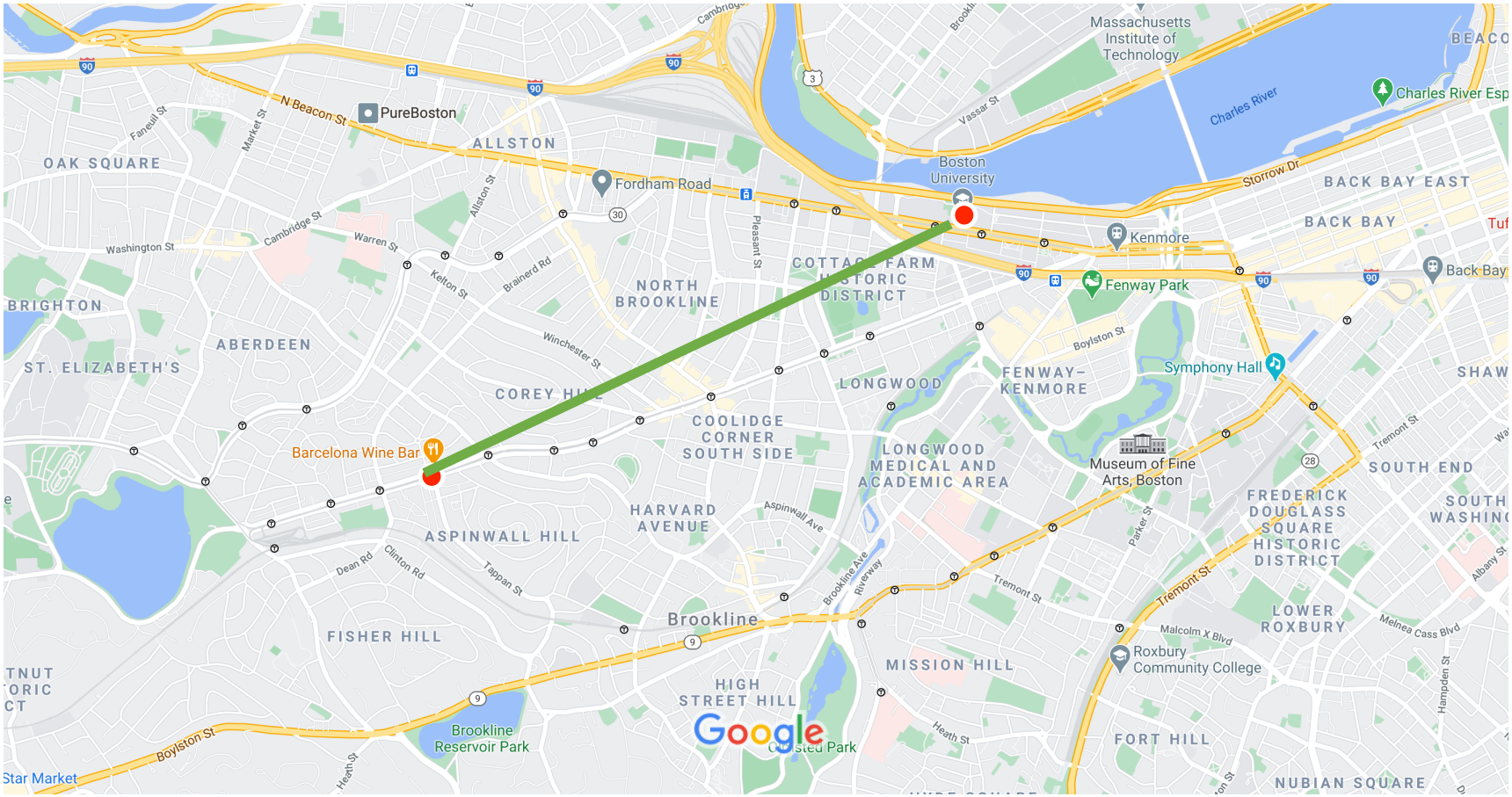
- We only want a path to a destination
  - But all algorithms compute paths from s to every destination
    - When can we stop?
    - Can we focus on getting only to where we want to go?
    - This is the Google maps or Waymo problem

- New algorithm: $A*$ search
  - Will search in direction of t for shortest path
  - A modification of Dijkstra to bias the search
  - Guaranteed optimal under certain conditions

Example


Massachusetts Institute of Technology
Charles River
Charles River Esp
PureBoston
N Beacon St
ALLSTON
OAK SQUARE
Fordham Road
Boston University
Storrow Dr
BACK BAY EAST
BACK BAY
Kenmore
Fenway Park
Back Bay
COTTAGE FARM HISTORIC DISTRICT
NORTH BROOKLINE
BRIGHTON
ABERDEEN
ST. ELIZABETH'S
COREY HILL
Winchester St
Boylston St
Symphony Hall
FENWAY–KENMORE
LONGWOOD
SHAW
Barcelona Wine Bar
COOLIDGE CORNER SOUTH SIDE
LONGWOOD MEDICAL AND ACADEMIC AREA
Museum of Fine Arts, Boston
SOUTH END
FREDERICK DOUGLASS SQUARE HISTORIC DISTRICT
HARVARD AVENUE
Aspinwall Ave
ASPINWALL HILL
Tremont St
Brookline
FISHER HILL
LOWER ROXBURY
Malcolm X Blvd
Roxbury Community College
MISSION HILL
HIGH STREET HILL
Brookline Reservoir Park
FORT HILL
NUBIAN SQUARE
Star Market
Google

Distance lower bound: Straight Path



Heuristic Distance Estimates

- Let $h : V \rightarrow \Re$ be a heuristic function that approximates the distance from vertex v to target t
- We call h **admissible** if: for each vertex v: h(v) ≤ D(v,t), where D(v,t) is the true shortest path distance
- We call h **consistent** if: For each (u,v) in E: h(u) ≤ w(u,v) + h(v).
  - For example, Euclidean distance to target on the plane
    - Euclidean distance is admissible and consistent

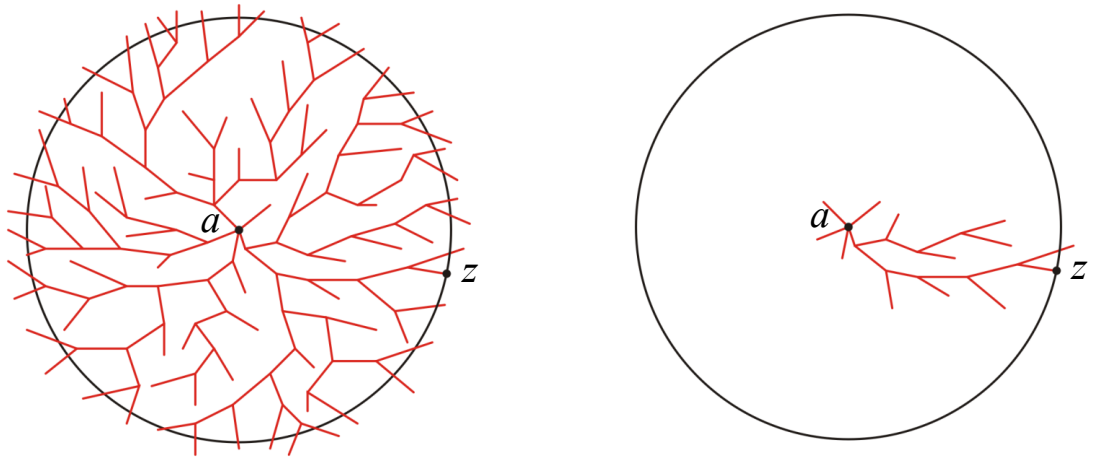
A* Search

- Simple idea: use heuristic h() to reweigh edges
  - $w_h(u, v) = w(u, v) - h(u) + h(v)$
    - Measures deviation from ideal distance estimate by h
    - Consistent heuristics guarantee non-negative weights
- Solve using Dijkstra's algorithm
- Result: If heuristic is consistent and admissible, we get an optimal shortest path
  - Dijkstra's distance is biased towards vertices that have shorter distance estimates

Comparison with Dijkstra's Algorithm

Graphically, we can suggest the behavior of the two algorithms as follows:

- Suppose we are moving from a to z:

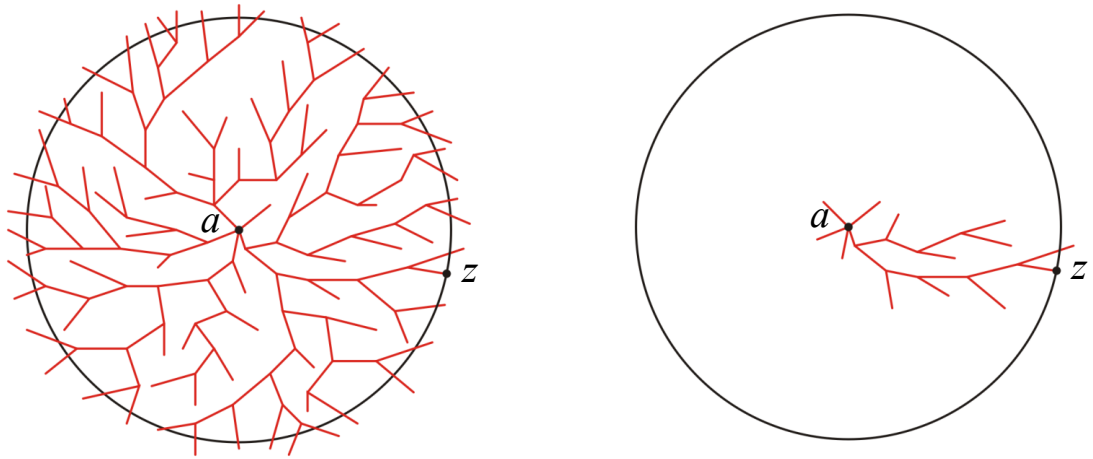


Representative search patterns for Dijkstra's and the A* search algorithms

Comparison with Dijkstra's Algorithm

Dijkstra's algorithm is the A* search algorithm when using the discrete distance $h(u,v) = \begin{cases} 0 & u = v \\ 1 & u \neq v \end{cases}$

- No vertex is better than any other vertex



Representative search patterns for Dijkstra's and the A* search algorithms