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

MHT Algorithms are Greedy Algorithms

- Greedy algorithm: an algorithm that builds a solution adding an element a a time that is the locally optimal choice at that time
  - Uses a simple rule e.g. MST adds edge with minimum weight across a cut
  - No backtracking (!!!)
  - Greedy algorithms are not always optimal
  - Special classes of problems can be solved to optimality by greedy algorithms

Other Scheduling Problems - 1

1. Single server, N jobs

- Job j has processing times t(j)
- Select start time s(j) for j, when server is idle; finish time f(j) = s(j) + t(j)
- Objective: Schedule jobs to minimize total finishing time summed over jobs

$$\text{Min} \sum_{j=1}^{N} f(j)$$

- Optimal solution: Greedy algorithm
  - Sort jobs so $t(1) \leq t(2) \leq \cdots \leq t(N)$ and schedule back-to-back in that order

Other Scheduling Problems - 2

- Proof by contradiction:

  Let P is an optimal schedule. Let p_j be job in position j in P

  Assume P is not ordered according to increasing processing time

  Find first j such that $t_{p_j} > t_{p_{j+1}}$ (must exist because of P ordering)

  Swap these in schedule to get schedule P′, and show P′ is better

  Note: finishing time of $p_1, \ldots, p_{j-1}$ and $p_{j+1}, \ldots, p_N$ is the same under P, P′

  Let T_{j-1} be finishing time p_{j-1}. Then,

  Under P, finishing times of p_j and p_{j+1} are: $T_{j-1} + t_{p_j}$, $T_{j-1} + t_{p_j} + t_{p_{j+1}}$

  Under P′, finishing times of p_j and p_{j+1} are: $T_{j-1} + t_{p_{j+1}}$, $T_{j-1} + t_{p_j} + t_{p_{j+1}}$

  Since $t_{p_j} > t_{p_{j+1}}$, P′ is better! So P cannot be optimal, contradiction.

Example

- Jobs: 1, 2, 3, 4, 5;
- Processing times: 5, 4, 3, 2, 3
- Algorithm: Sort jobs by processing time: 4, 3, 5, 2, 1, schedule in order

- Finishing times: 2 + 5 + 8 + 12 + 17 = 44
- Alternative schedule: schedule by number: 1, 2, 3, 4, 5

- Finishing times: 5 + 9 + 12 + 14 + 17 = 57

Other Scheduling Problems - 3

2. Single server, N jobs

- Job j has unit processing time, hard deadline d(j), Value V(j) if job completed on time
- Objective: Find sequence of jobs that will be scheduled to maximize value completed on time
- Concept: a set of jobs $\{j_1, \ldots, j_k\}$ is feasible $\Longleftrightarrow$ there is an ordering of these jobs where each job is completed before its deadline
  - If a set of jobs is feasible, scheduling them by earliest deadline first is sufficient —> simple to show
  - Can also schedule them just in time — schedule job j as close as possible to its deadline d(j)

Other Scheduling Problems

- Determining the best feasible set is the hard problem
- Algorithm:
  - Initialize feasible set S as empty; sort the jobs by decreasing value V(j), into order $j_1, \ldots, j_N$
  - Initialize step k = 1
  - At step k, add job j_k to S, and see if S is still a feasible set
    - If S is not feasible, remove j_k from S
    - If k < N, increment k = k+1 and repeat
- Claim: this will be an optimal feasible set
  - Important feature: every job takes the same amount of time

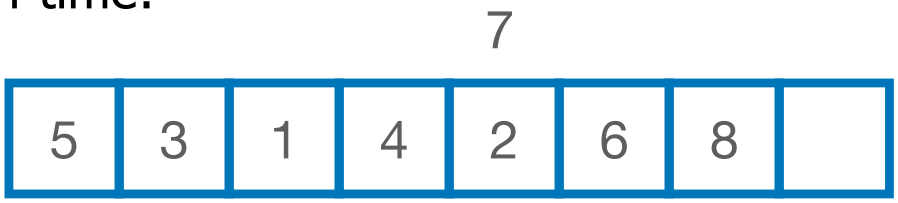
Other Scheduling Problems

- Proof of optimality
  - Let S: feasible set constructed by algorithm. T: optimal feasible set that has the most overlap with S
  - Assume T has more value than S, and assume the jobs in T are indexed according to decreasing value
  - Assume that all jobs that are **both** in S and T are scheduled at the same times
    - This can always be done, by scheduling jobs as late as possible
  - Find the highest value job in S that is not in T, denoted j*. Swap that job with whatever job is scheduled at the same time in T. You get a no worse T*, so T* is as optimal as T or more so
    - Why is T* feasible? Because j* is scheduled before its deadline in S, hence also in T*
    - Why is T* better? Because S selected feasible sets by decreasing value, so value of j* at least as large as value of any job in T that is not in S
    - Why do we get a contradiction? Because T* is at least optimal, and has more overlap with S than T; but T had the highest overlap with S among optimal sequences

Example

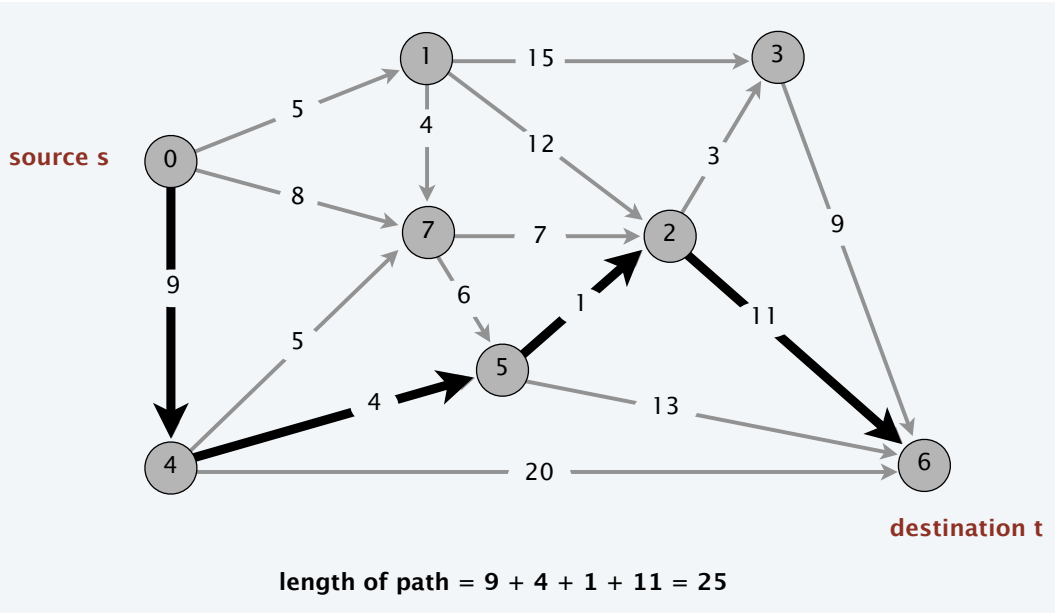
Jobs	1	2	3	4	5	6	7	8
Values	5	4	3	2	3	4	2	1
Deadlines	3	5	2	4	3	6	5	7

- Algorithm: Sort by value: 1, 2, 6, 3, 5, 4, 7, 8
- Find feasible set by greedy algorithm:
  - Just-in-time:



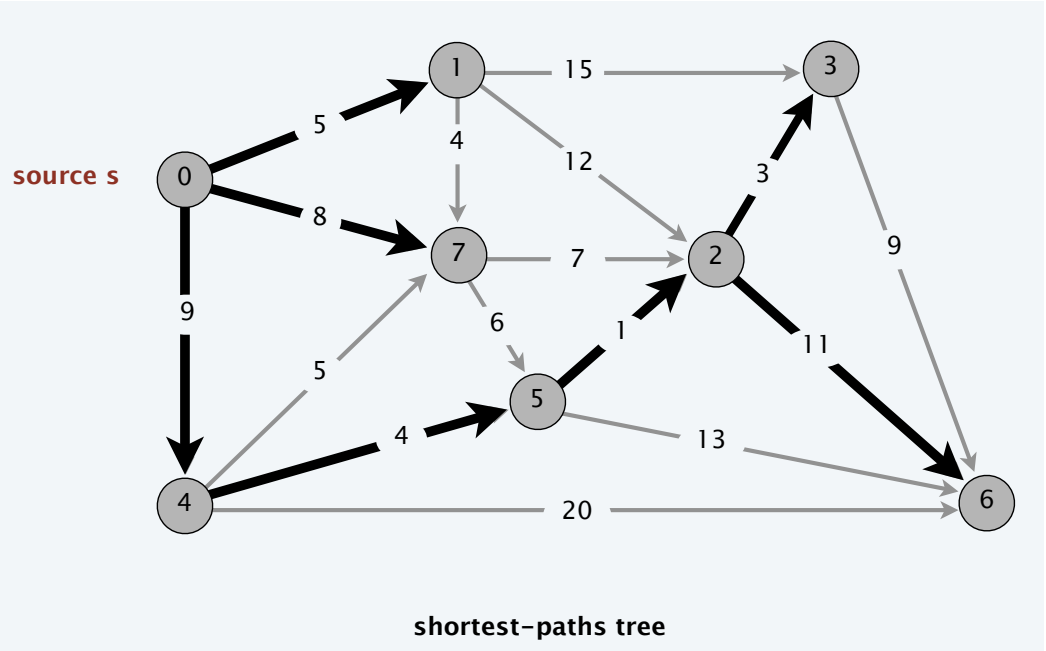
What is a Shortest Path?

- Given weighted, directed graph G(V,E) (weights on edges)…
  - …what is shortest path from vertex s to t?
  - Distance is additive
- Applications
  - Google maps
  - Routing packets on the Internet
  - Social networks



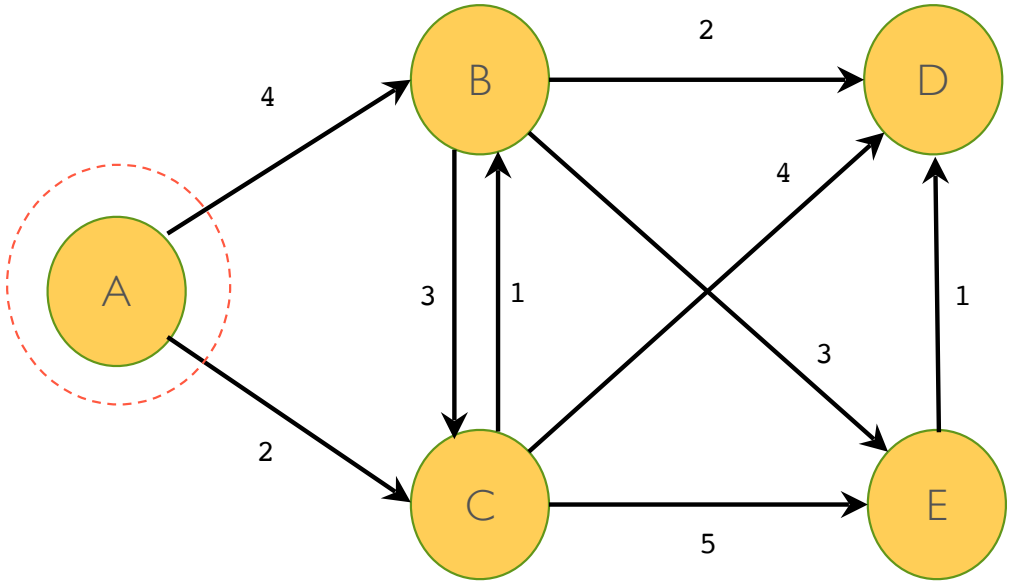
Single Source Shortest Paths (SSSP)

- Given a graph and a source vertex
  - find the shortest paths to all other vertices
  - results in a shortest-path tree
  - Single directed path to every other vertex



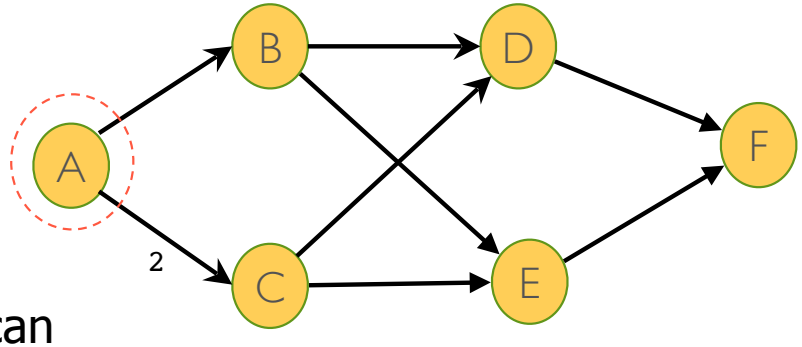
Quick Insight Quiz

- Suppose that you change the length of every edge of G as follows. For which is every shortest path in G a shortest path in G'?
  - Add 17
  - Multiply by 17
  - Either A or B
  - Neither A nor B



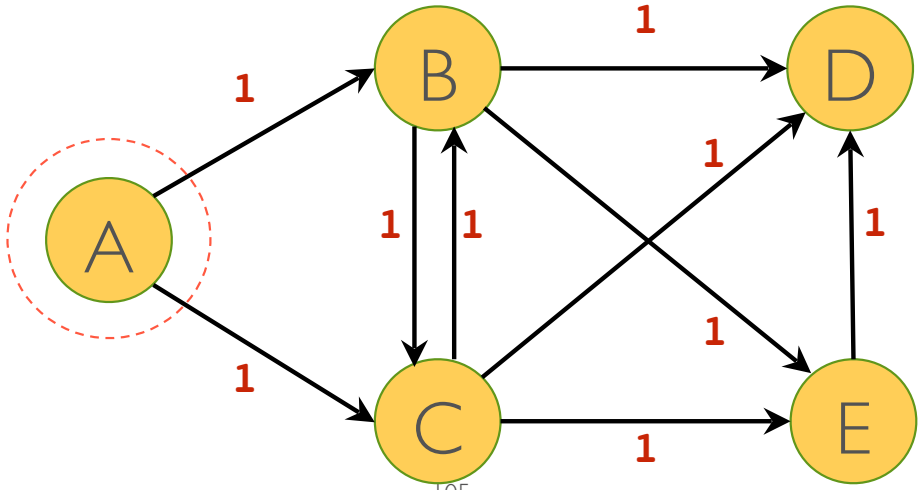
Important Property of Shortest Paths

- Lemma: The shortest path from vertex s to a vertex t is composed of shortest paths to and from any intermediate vertices
  - Proof by contradiction on path from s to t
  - Consider intermediate vertex v
  - If better path from s to v was possible, then can replace original path to v and get shorter
  - If better path from v to t was possible, then can original path from v to t and get shorter
  - Hence, original path was not shortest path
- Bellman's Principle of Optimality



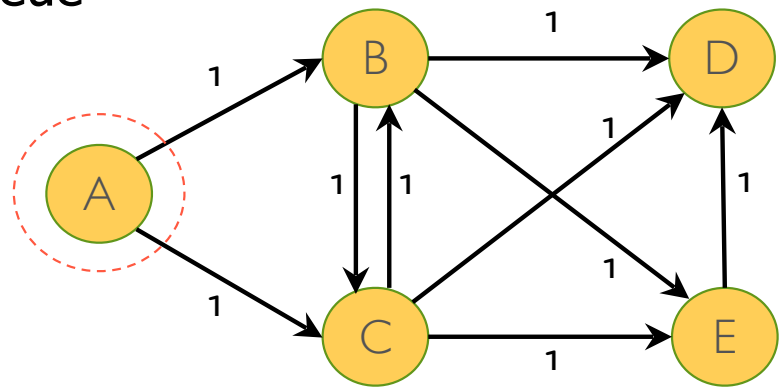
Simpler Problem: Unit Edges

- Let's start with simpler problem
  - On graph where every edge has unit cost
  - Shortest path = minimum number of hops

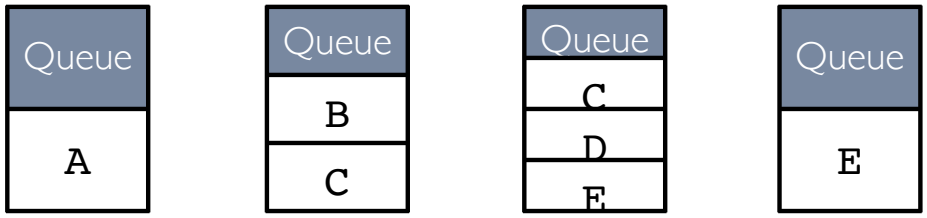


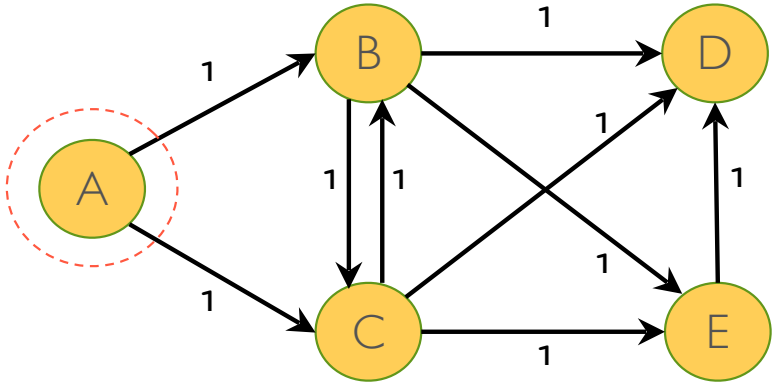
Solution: Breadth-First Search

- Use BFS to find shortest path from A to E.
  - Consider all steps of adding/removing nodes from queue …
  - … updating each vertex's predecessor
    - predecessor: id of vertex that adds current vertex to queue
  - … updating distances when added to queue
    - Distance to vertex is 1 more than distance to vertex that added current vertex to queue

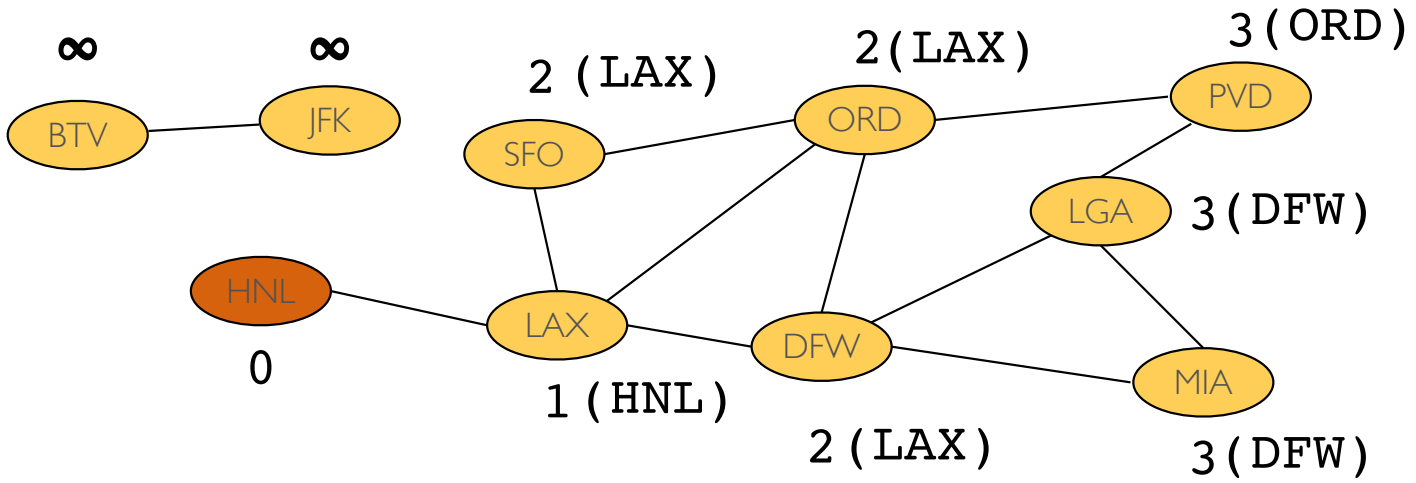


Example





numStops (with BFS)

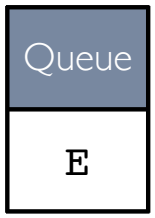


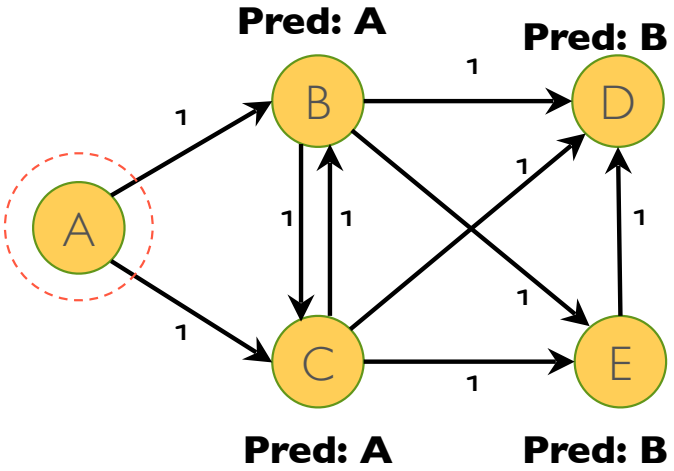
Have *distance*
Want a *path*

Follow 'predecessor' pointers to get a path

Breadth First Search

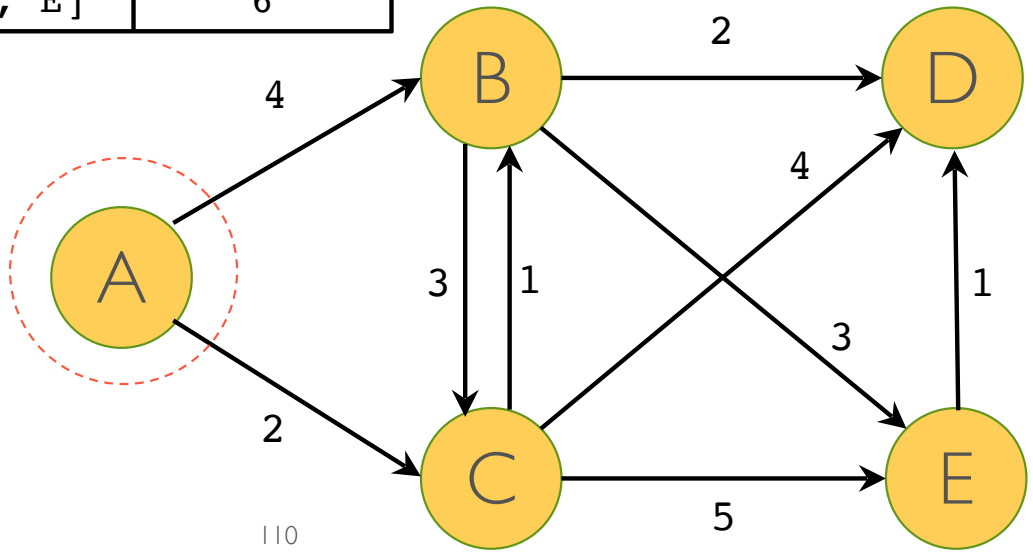
- When we dequeue E…
- …we traverse the pred to return paths
  - shortest path to E: [A,B,E]



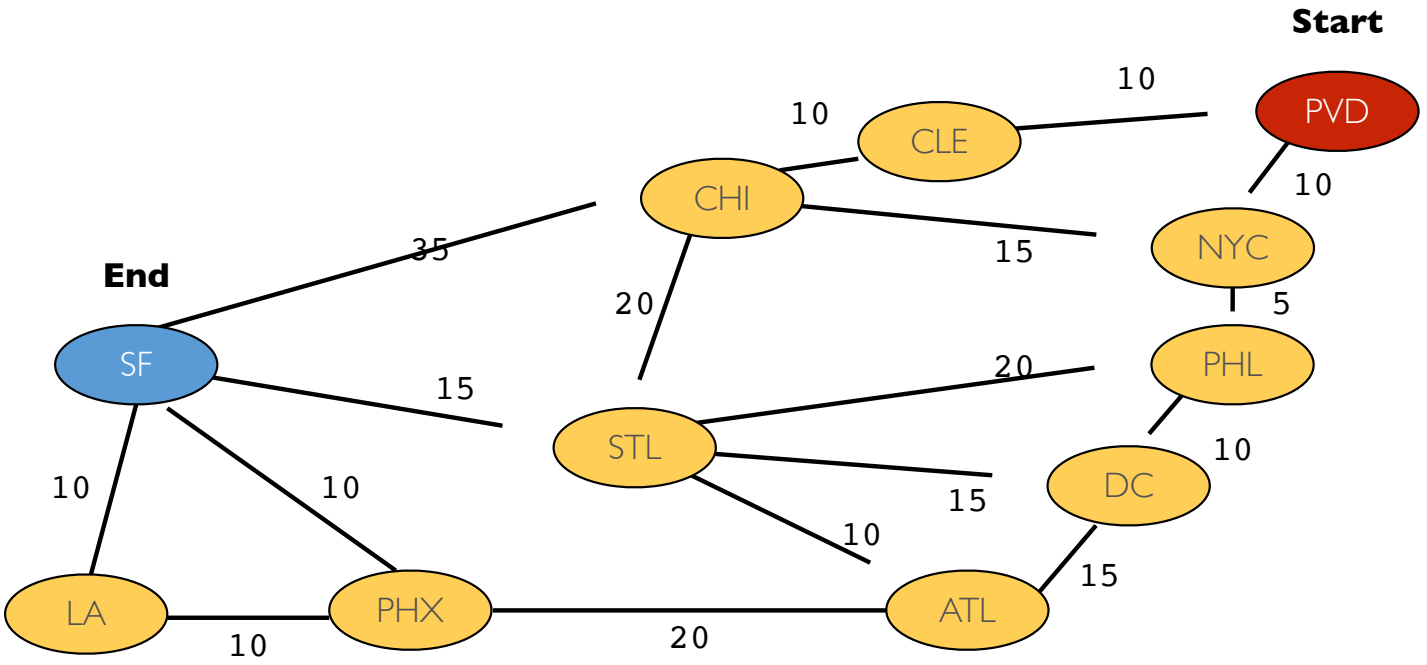


Non-unit Edge Weights

Goal Node	Shortest Path	Shortest Distance
B	[A, C, B]	3
C	[A, C]	2
D	[A, C, B, D]	5
E	[A, C, B, E]	6



Our Graph


Start
End
PVD
CLE
CHI
NYC
PHL
DC
ATL
STL
PHX
LA
SF
10
10
10
10
35
15
20
5
20
15
10
10
15
10
20

Dijkstra's Algorithm

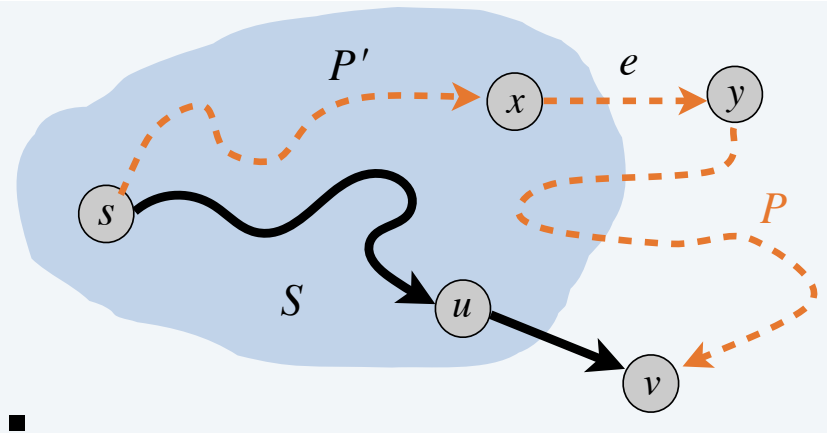
- Greedy Algorithm: Assumes all edges have **nonnegative** weights
  - Maintain a set of **explored** nodes S for which algorithm has determined D[u] = length of a shortest s to u path
  - Initialize S = {s}, D[s] = 0; D[v] = infinity
  - Choose unexplored node $v \notin S$ which minimizes $D[u] + w(u, v), u \in S$

    Set $D[v] = \min_{\{(u,v) \in E : u \in S\}} D[u] + w(u, v)$, and add v to S

    Set pred[v] = vertex u in S that achieves d[v]
  - Repeat until all vertices are explored, so S = V
    - Path to any vertex can be found by using pred[] labels

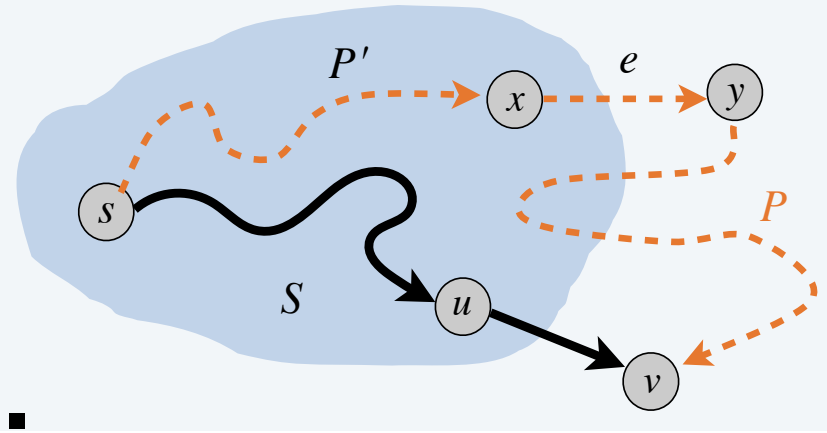
Dijkstra's Algorithm: Correctness

- Claim: For every vertex v in S, D[v] is shortest path distance from s to v
  - Prove by induction: clearly true when S = {s}, D[s] = 0
  - Assume true for S of size k. Let v be the next vertex added to S, and let (u,v) be the final edge, so D[v] = D[u] + w(u,v), where u in S
  - Assume there is another path P' from s to v that is shorter than D[v]
  - P' starts in S, and must leave S to go to some vertex $x \in S$, then use an edge e that leaves S for a vertex y on the way to v
  - Note: y can be v, x can be u, but not both!



Dijkstra's Algorithm: Correctness - 2

- Claim: For every vertex v in S, D[v] is shortest path distance from s to v
  - Let L(P') be the length of P', and let P be the path in P' from y to v Then, $L(P') = D[x] + w(e) + L(P) \geq D[x] + w(e)$ because all weights are nonnegative, so $L(P) \geq 0$
  - Note: $D[x] + w(e) \geq D[u] + w(u, v)$ because we chose (u,v) such that $D[v] = \min_{\{(u,v) \in E : u \in S\}} D[u] + w(u, v)$
  - Therefore, D[v] is no longer than L(P'), which was shortest path distance, so D[v] is the shortest path distance



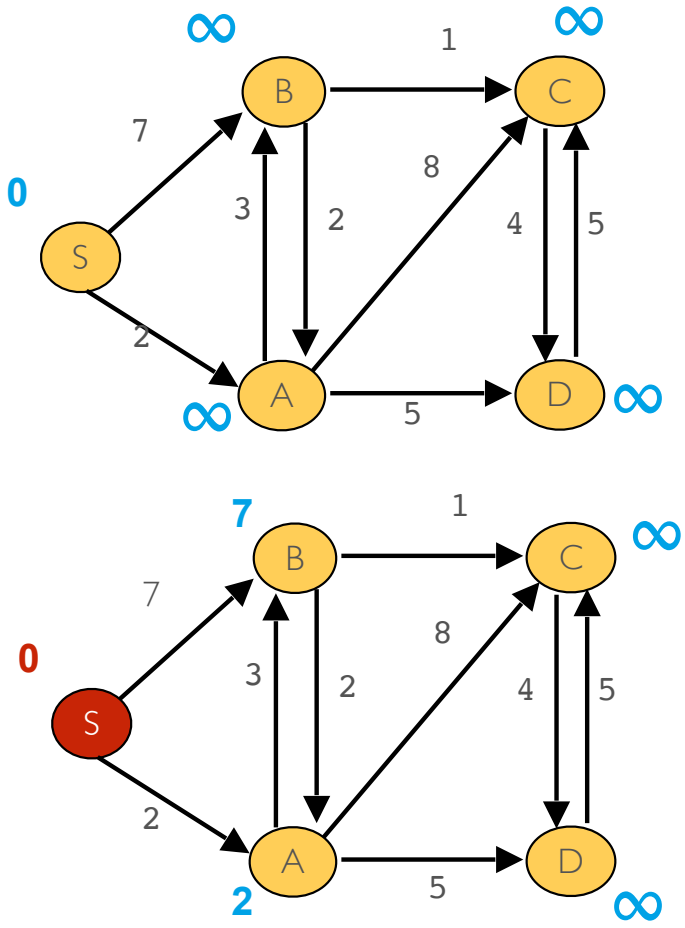
Implementing Dijkstra's Algorithm

- Previous description is hard to implement
- Better approach: Keep a priority heap of vertices v not in S, with keys D[v] that are shortest distance to v of all paths through S with one edge leaving S for v
  - D[v] is an upper bound on shortest distance to v if v not in S
  - Best node v not in S is at top of priority queue
- Can maintain D[v] to all v not explored recursively as we increase S

Dijkstra: Pseudo code

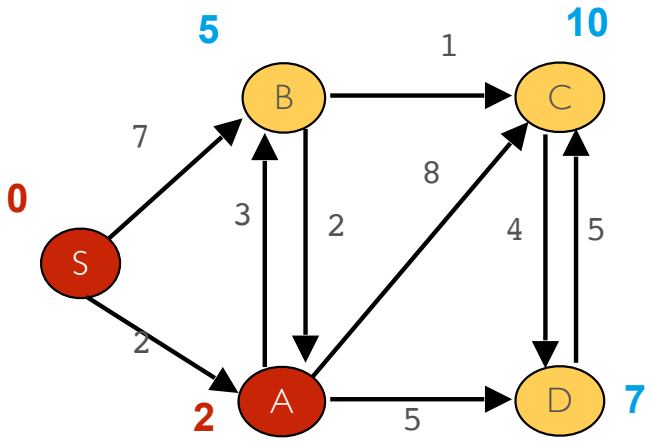
- Initially, $S = \emptyset$, $D(s) = 0$, $D[v] = \infty$, $v \neq s$, and pred[v] = null for all $v \in V$
- Insert all $v \in V$ into priority queue PQ indexed by $D[v]$
- While PQ is not empty,
  - Delete_min of PQ, denoted by u, and add u to S
  - For each edge (u,v) leaving S:
    - If $D[v] > D[u] + w(u, v)$:
      - Decrease_key of v in PQ to $D[u] + w(u, v)$
      - Set pred[v] = u, $D[v] = D[u] + w(u, v)$

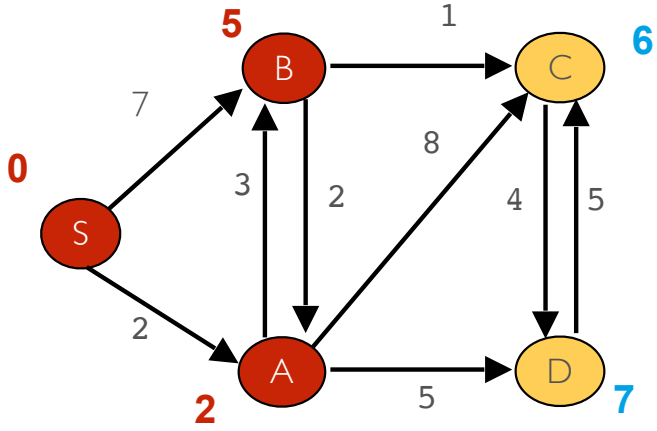
Dijkstra's Algorithm Example



- Step 1
  - Label source s w/ distance 0
  - Label other vertices w/ distance ∞
  - Add all nodes to Q
- Step 2
  - Remove vertex n with min. priority from Q (n=s first)
  - Calculate distance from source to removed node's neighbors…
  - …by adding adjacent edge weights to distance D[n]
  - Reduce keys if D[v] > D[n] + w(n,v), set new D[v] = D[n] + w(n,v), update pred(v) = n

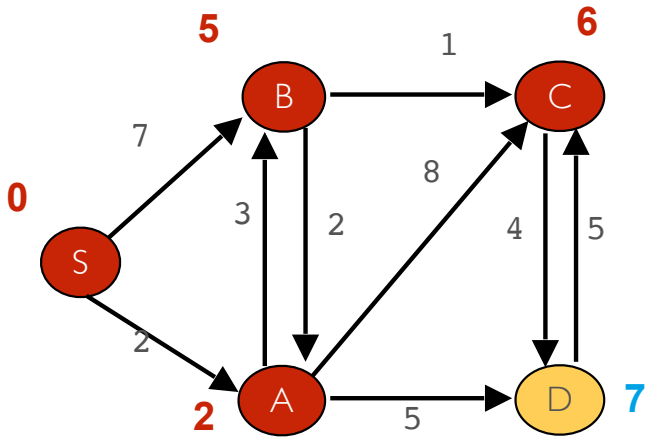
Example - 2

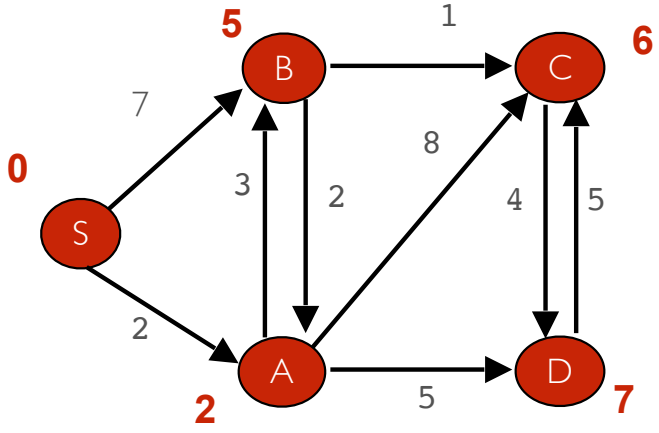




- Step 3
  - While Q isn't empty,
    - repeat previous step
    - removing A this time
  - Priorities of nodes in Q may have to be updated
    - ex: B' s priority
- Step 4
  - Repeat again by removing vertex B
  - Update distances that are shorter using this path than before
  - ex: C now has a distance 6 not 10

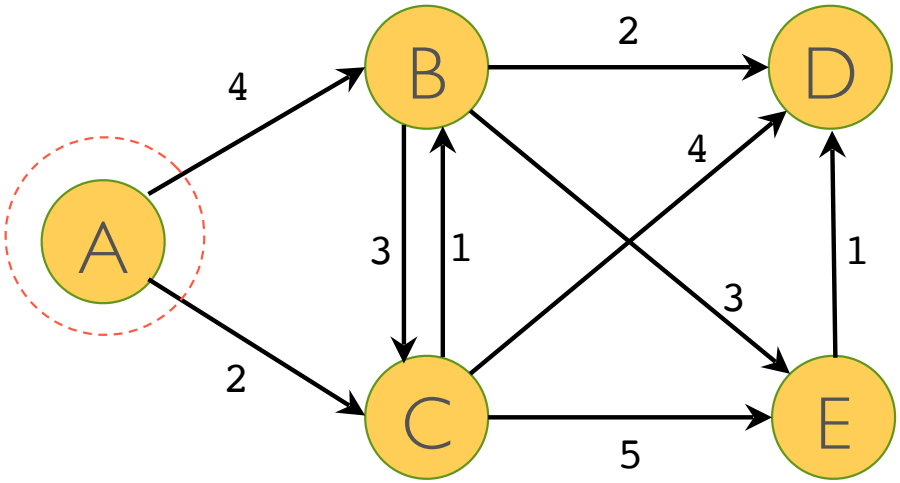
Example - 3





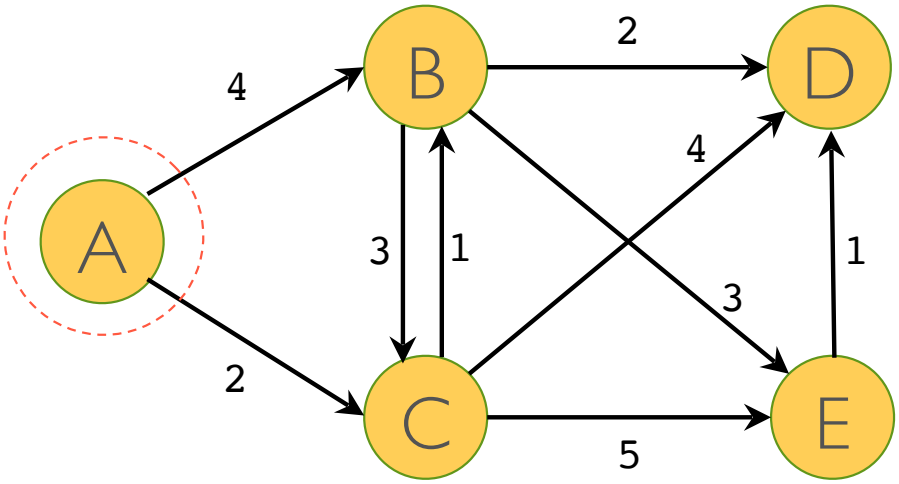
- Step 5
  - Repeat
    - this time removing C
- Step 6
  - After removing D…
  - …every node has been visited…
  - …and marked with shortest distance to source

Another Example



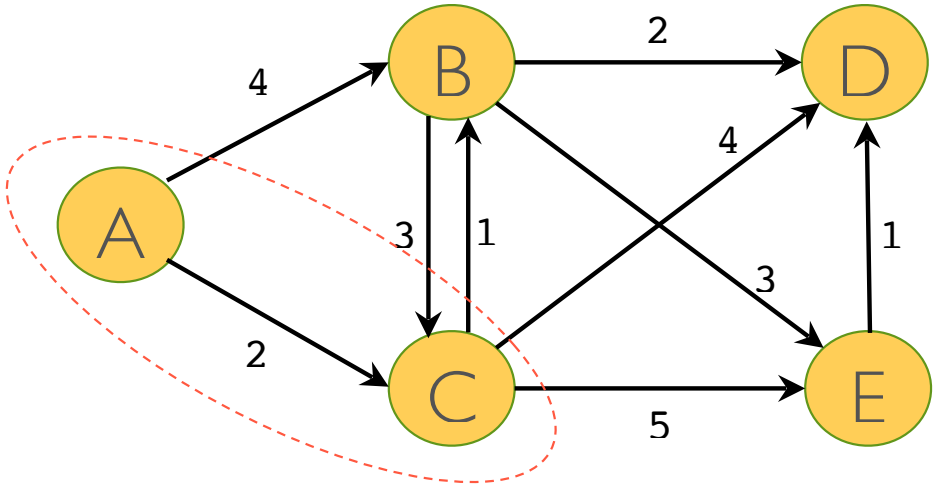
	A	B	C	D	E
D[ ]	0	∞	∞	∞	∞
pred[ ]					

Another Example



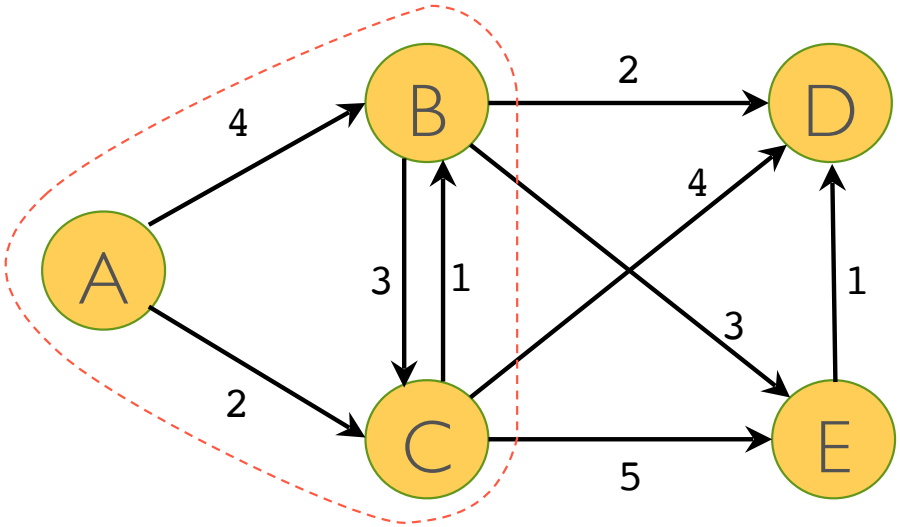
	A	B	C	D	E
D[ ]	0	4	2	∞	∞
pred[ ]	Null	A	A	Null	Null

Another Example



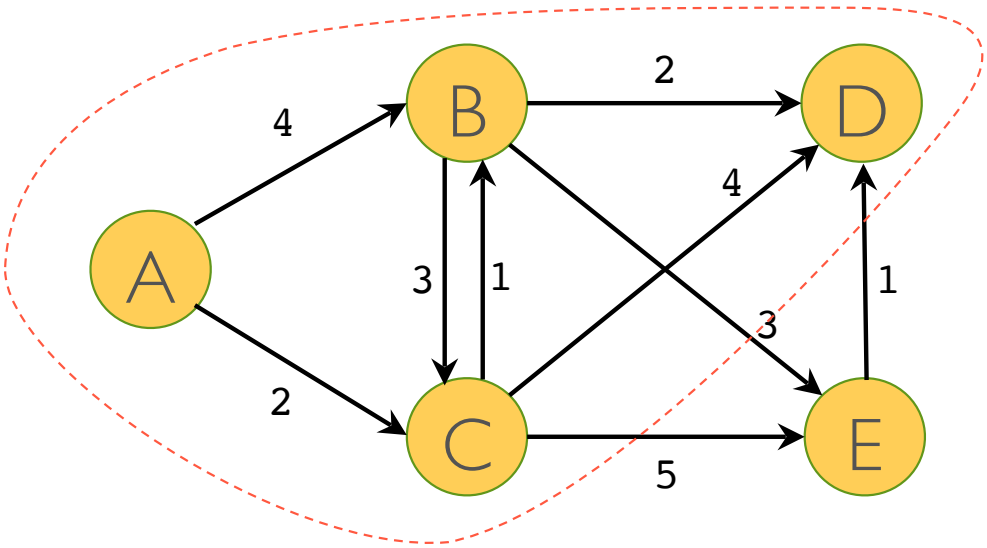
	A	B	C	D	E
D[]	0	3	2	6	7
pred[]	Null	C	A	C	C

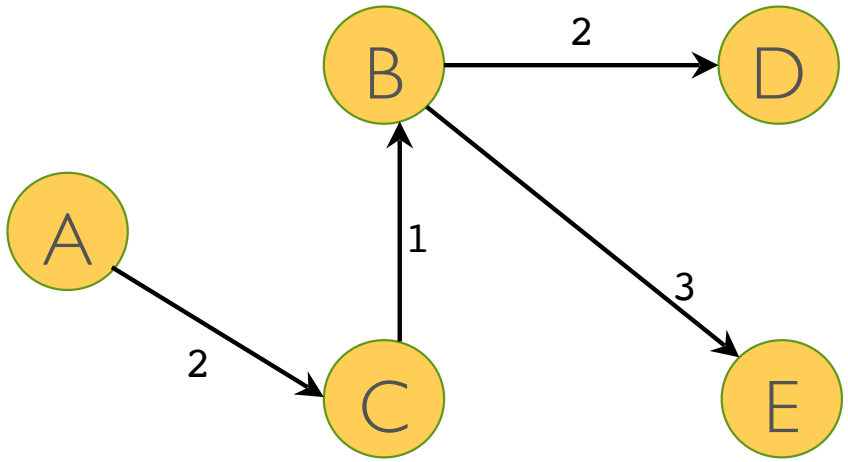
Another Example



	A	B	C	D	E
D[]	0	3	2	5	6
pred[]	Null	C	A	B	B

Another Example





	A	B	C	D	E
D[]	0	3	2	5	6
pred[]	Null	C	A	B	B

Dijkstra vs Prim

```
function dijkstra(G, s):
   // Input: weighted, dir. graph G, vertices V, source s
   // Output: Nothing
   // Purpose: compute for all V shortest distance from s
   for v in V:
      v.dist = infinity  // Initialize distance
      v.prev = null      // Initialize pred pointers to
null
   s.dist = 0         // Set distance to start to 0

   PQ = PriorityQueue(V)    // Use v.dist as priorities
   while PQ not empty:
      u = PQ.removeMin()
      for all edges (u, v):    //each edge coming out of u
         if v.dist > u.dist + weight(u, v): //
            v.dist = u.dist + weight(u,v)   // Replace
            v.prev = u     // Maintain pointers for path
            PQ.decreaseKey(v, v.dist)
```

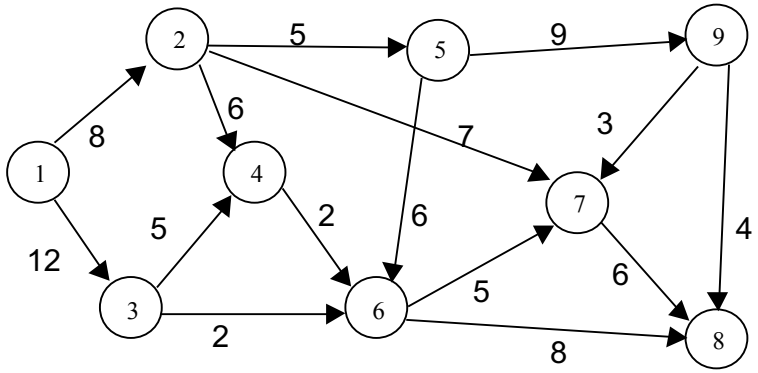
```
function prim(G):
   // Input: weighted undir. graph G with vertices V
   // Output: list of edges in MST
   for all v in V:
      v.dist = infinity
      v.prev = null
   s = any v in V // pick a random source s
   s.dist = 0
   MST = []
   PQ = PriorityQueue(V) // priorities will be v.cost
   while PQ is not empty:
      U = PQ.removeMin()
      if u.prev != null:
         MST.append((u, u.prev))
      for all edges (u,v) such that v is in PQ:
         if v.dist > weight(u,v):
            v.dist = weight(u,v)
            u.prev = v
            PQ.decreaseKey(v, v.dist)
  return MST
```

Dijkstra: Which Priority Queue?

- Algorithm complexity
  - Priority queue with |V| elements
  - Examine every edge once
  - May Decrease_key for every edge
  - |V| Delete_min
- Total run time with array as priority queue: $O\left(|E|+|V|^2\right)$
- Total run time with binary heap as priority queue: $O\left((|E|+|V|)\log(|V|)\right)$
- Total run time with Fibonacci heap or rank-pairing heap as priority queue: $O\left(|E|+|V|\log(|V|)\right)$

Example

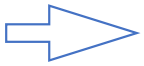
- Source node 1, use binary heap
  - Q[ ] is position in heap



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

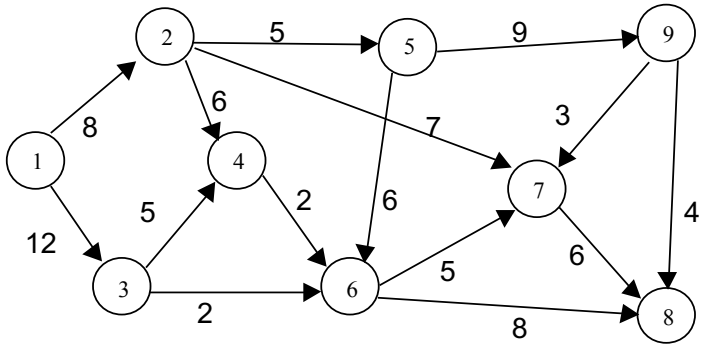
	D[ ]	pred[ ]	Q [ ]
1	0	Null	0
2	Inf	Null	
3	Inf	Null	
4	Inf	Null	
5	Inf	Null	
6	Inf	Null	
7	Inf	Null	
8	Inf	Null	
9	Inf	Null	

Pop 1



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

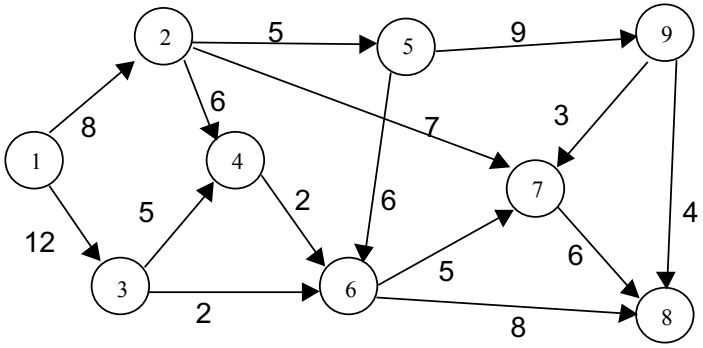
Pop 2

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

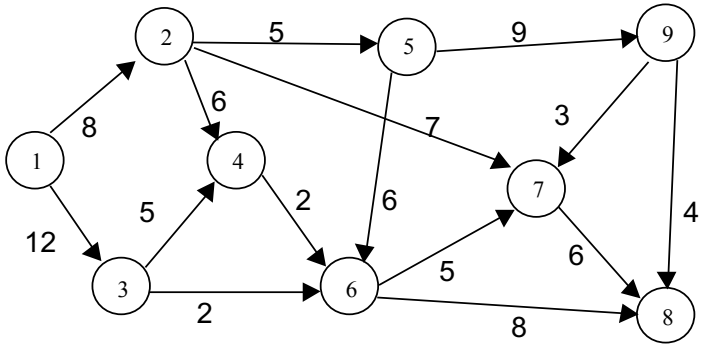
Pop 5

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

Pop 6

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

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	0
5	13	2	
6	14	3	
7	15	2	2
8	22	6	3
9	22	5	1

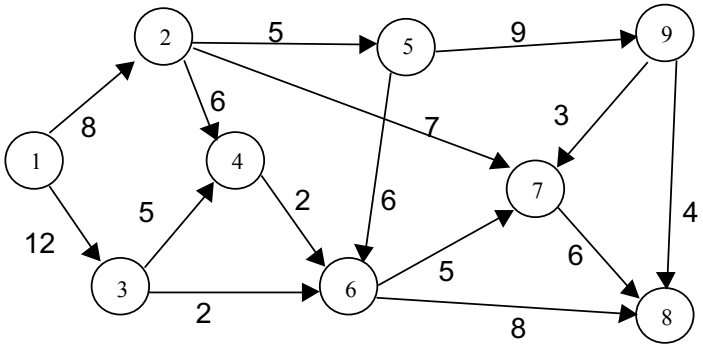
Pop 4

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

Pop 7

	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	0
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	0
9	22	5	1

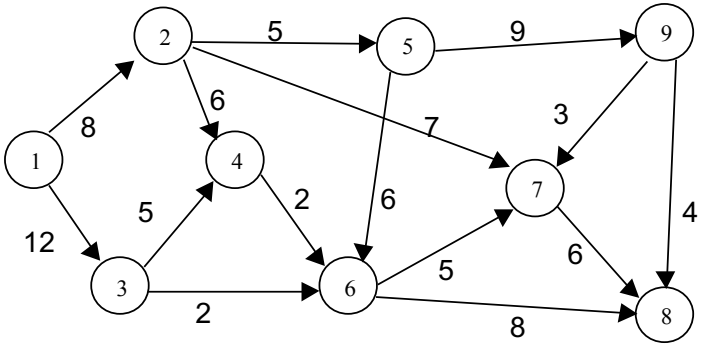
Pop 8

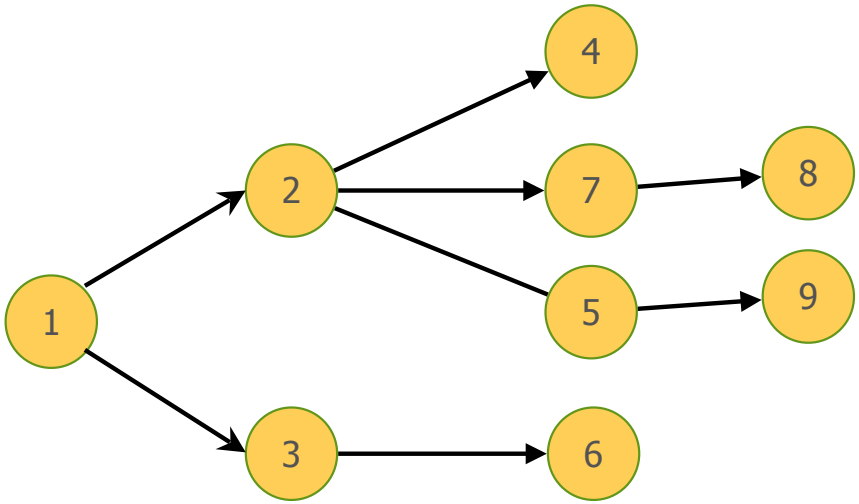
	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	
9	22	5	0

Pop 9

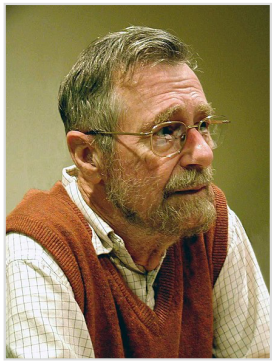
	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	
9	22	5	

Example - 6





Historical Aside

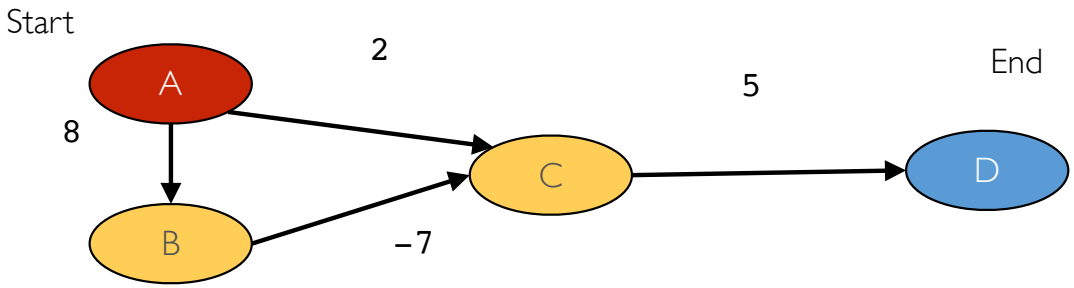


- Edsger Dijkstra (1930-2002): Dutch computer scientist
  - "What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning (1956) I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path…
    One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities." (Communications of the ACM 53 (8), 2001)
- Won the Turing award in 1972
- Dijkstra's algorithm used in internet protocols: OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System) protocols
- Many other contributions…e.g. concurrent programming and smoothsort

Dijkstra Limitation

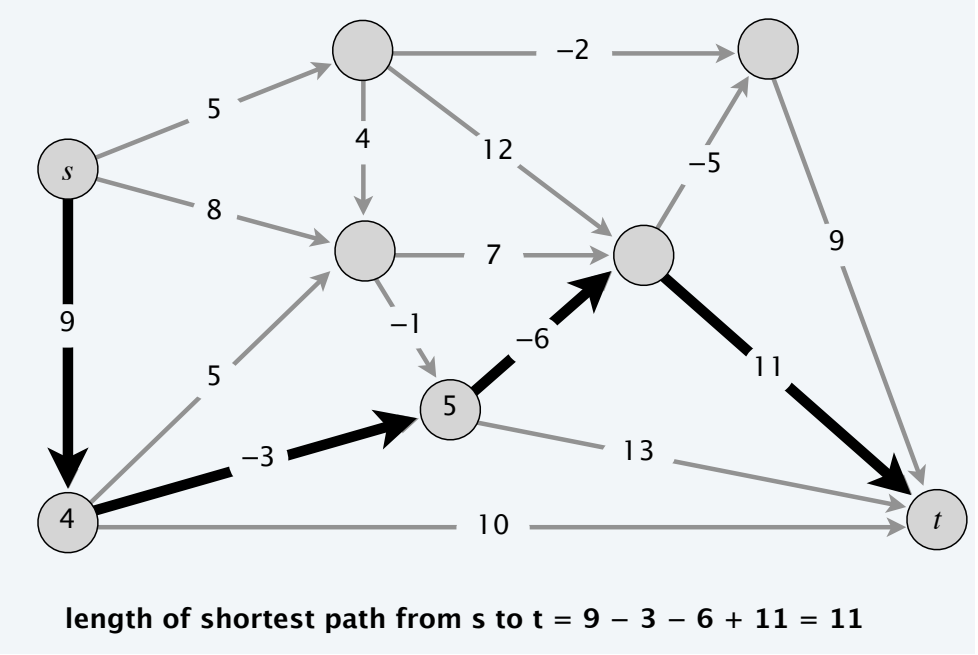
- Edges must have non-negative weight
  - Otherwise, proof of correctness fails (continuation of path can have negative length)

- For graph below, Dijkstra computes shortest A-D of 7



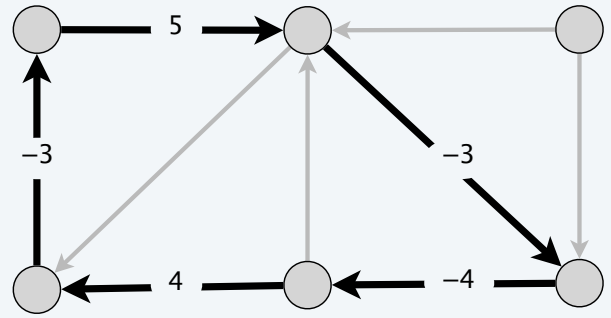
Shortest paths: Graphs with Negative Weights

- 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
  - Breaking it down into simpler subproblems
  - Exploit 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

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)

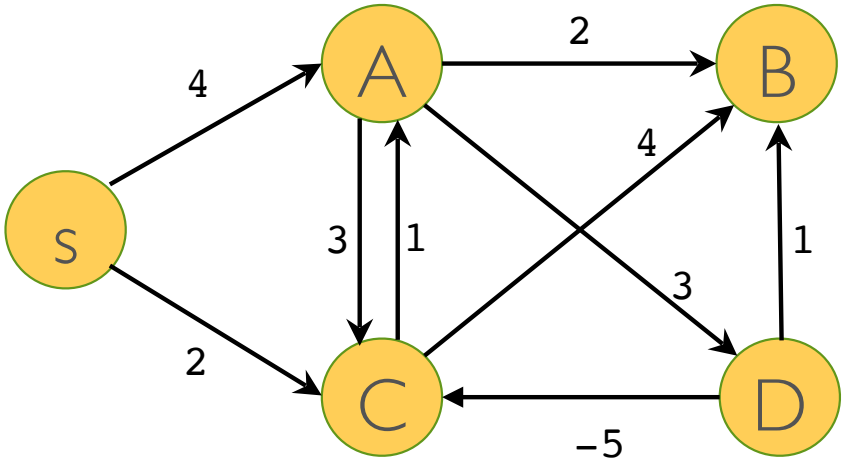
- Bellman-Ford Algorithm
  - For each $v \in V$, set $D(0,v) = \infty,\ v \neq s;\ 0,\ v = s$; set pred[v] = null
  - For k = 1 to |V| - 1:
    - For each $v \in V$, set $D(k,v) = D(k-1,v)$
    - For each $(u,v) \in E$:
      - if $D(k,v) > D(k-1,u) + w(u,v)$:
        - Set $D(k,v) = D(k-1,u) + w(u,v)$,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

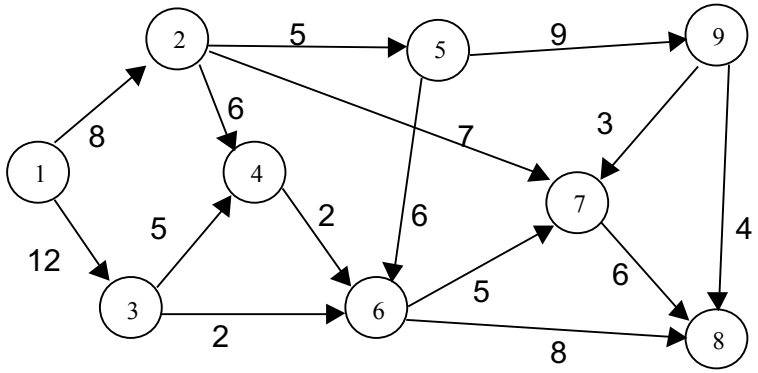
- Use a queue of vertices where distances have changed
  - Avoid looking at edges out of vertices where distances have not changed
- Algorithm
  - Initialize $D[s] = 0;\ D[v] = \infty, v \neq s;$ pred$[v] = null$
  - Insert s into queue Q; mark inqueue[s] = true, mark inqueue[v] = false, $v \neq s$
  - 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

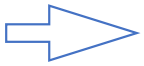
- Source node 1



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

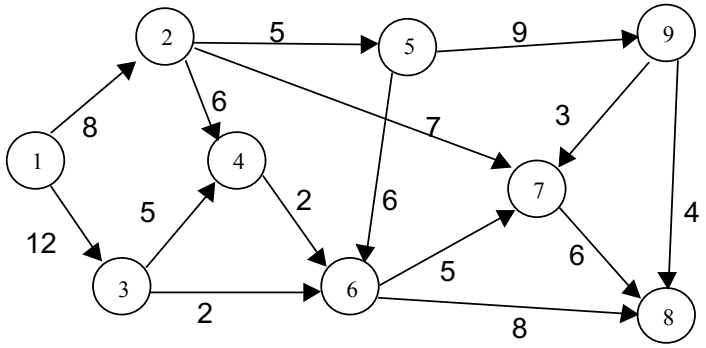
	D[ ]	pred[ ]	Q [ ]
1	0	Null	1
2	Inf	Null	
3	Inf	Null	
4	Inf	Null	
5	Inf	Null	
6	Inf	Null	
7	Inf	Null	
8	Inf	Null	
9	Inf	Null	

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	

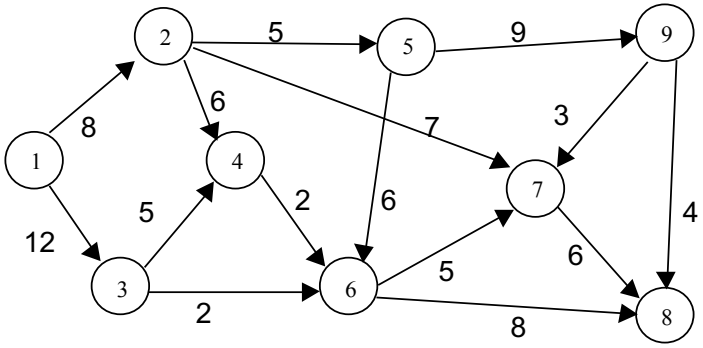
Pop 2

	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	

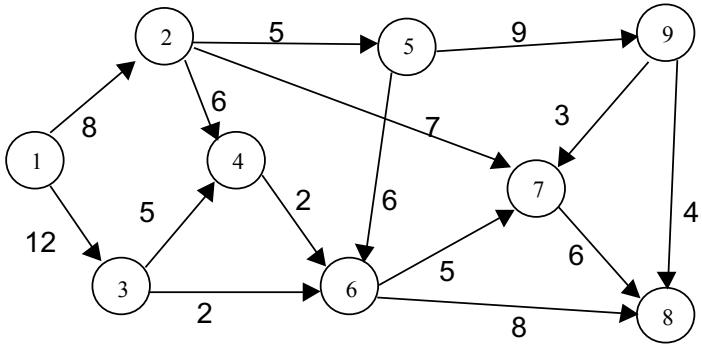
Pop 4

	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

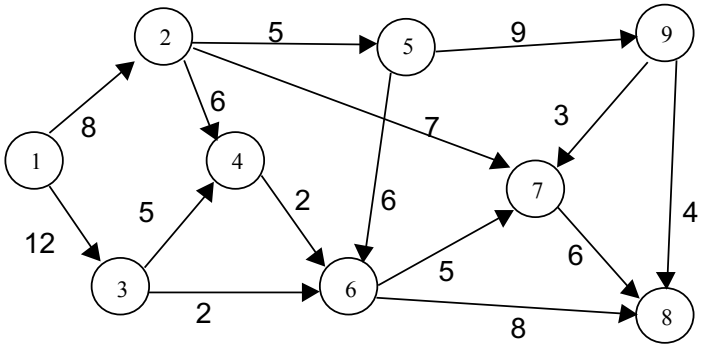
Pop 7

	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

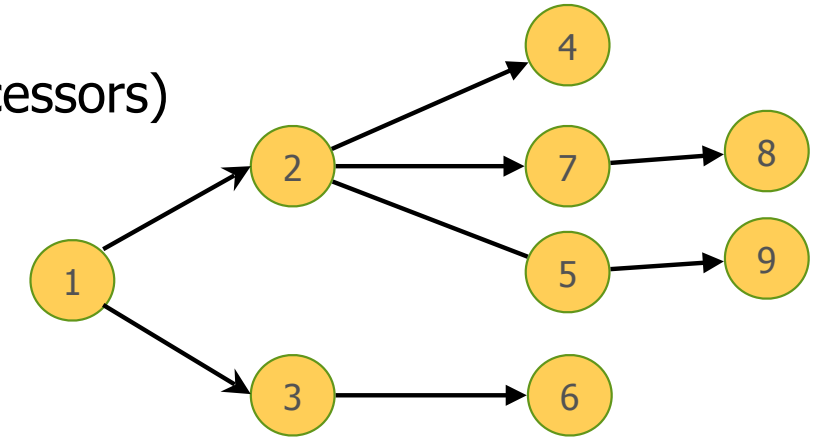
	D[ ]	pred[ ]	Q [ ]
1	0	Null	
2	8	1	
3	12	1	
4	14	2	
5	13	2	
6	14	3	
7	15	2	
8	21	7	1
9	22	5	

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)



New Variation: All Pairs Shortest Paths

- 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)