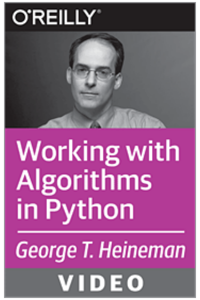
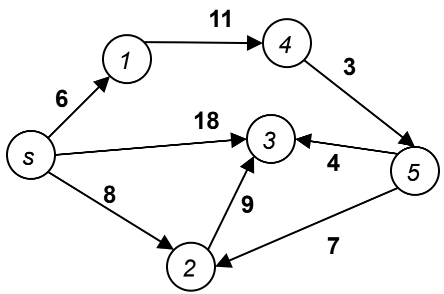
O'REILLY®

Single-Source Shortest Path



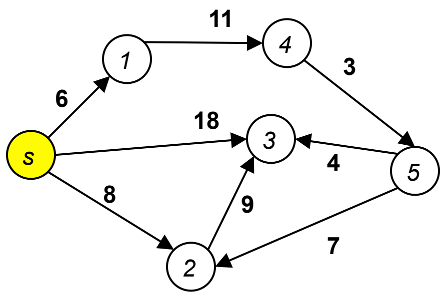
Single-Source Shortest Path



- Given directed graph G = (V, E)
  - Compute path from vertex *s* to every other vertex whose accumulated edge weight is smallest
  - Each edge weight is positive (i.e., > 0)
  - Graph is simple (no self edges)

Single-Source
Shortest Path Demonstration

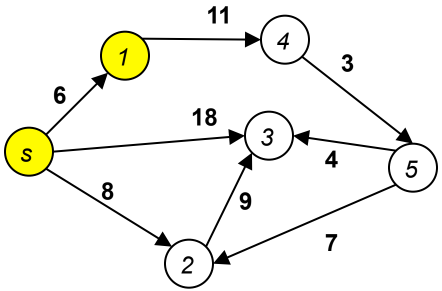
- Compute array dist[u]
  - Records best shortest path from *s*
  - Initially contains edge information
- *s* is the only visited vertex for now
- Find unvisited vertex that is shortest distance from *s*
  - Expand search in greedy fashion



	s	*1*	*2*	*3*	*4*	*5*
dist[u]	0	6	8	18	∞	∞

Single-Source Shortest Path Demonstration

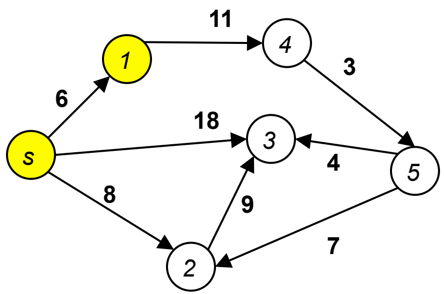
- Observe that vertex v_1 is closest
  - Expand in that direction by seeing if neighbors (v_4) are now closer



	s	*1*	*2*	*3*	*4*	*5*
dist[u]	0	6	8	18	∞	∞

Single-Source Shortest Path Demonstration

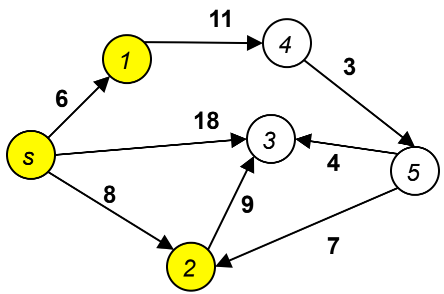
- Observe that vertex v_1 is closest
  - Expand in that direction by seeing if neighbors (v_4) are now closer
  - It is ($17 < \infty$) so update dist[v_4]
- Locate unvisited closest vertex
  - That would be v_2



	s	1	2	3	4	5
dist[u]	0	6	8	18	17	∞

Single-Source Shortest Path Demonstration

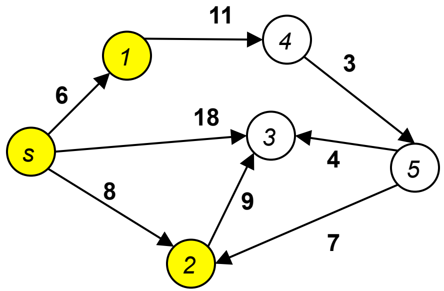
- Observe that vertex v_2 is closest
  - Expand in that direction by seeing if neighbors (v_3) are now closer



	s	1	2	3	4	5
dist[u]	0	6	8	18	17	∞

Single-Source Shortest Path Demonstration

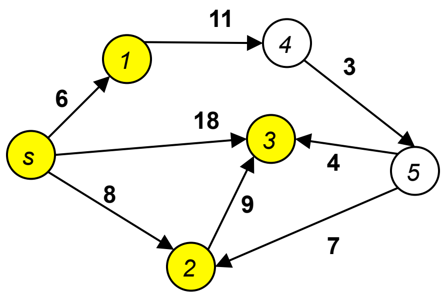
- Observe that vertex v_2 is closest
  - Expand in that direction by seeing if neighbors (v_3) are now closer
  - It is ($17 < 18$) so update dist[v_3]
- Locate next unvisited closest vertex
  - Either v_3 or v_4 – Choose v_3



	s	1	2	3	4	5
dist[u]	0	6	8	17	17	∞

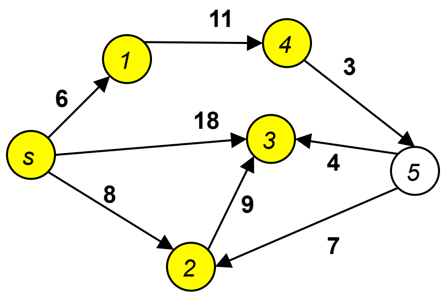
Single-Source Shortest Path Demonstration

- Observe that vertex v_3 is closest
  - No neighbors to expand
- Locate next unvisited closest vertex
  - Choose v_4



	s	1	2	3	4	5
dist[u]	0	6	8	17	17	∞

Single-Source Shortest Path Demonstration



	s	1	2	3	4	5
dist[u]	0	6	8	17	17	20

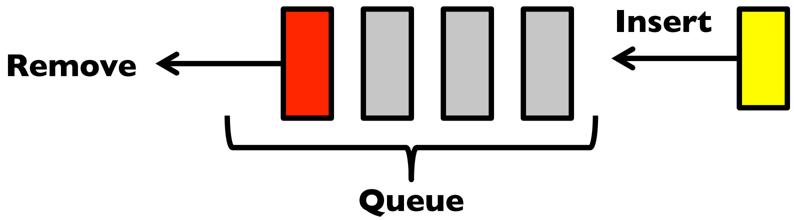
- Observe that vertex v_4 is closest
  - Expand in that direction by seeing if neighbors (v_5) are now closer
  - It is ($20 < \infty$)
- No more vertices remaining
  - Computed paths
  - Modify to store pred[u] to recover actual paths

Single-Source Shortest Path Challenge

- Two challenges in implementation
  – How to locate next unvisited closest vertex
  – Impact of reduced dist[u] values
- Solve using a specially constructed Priority Queue

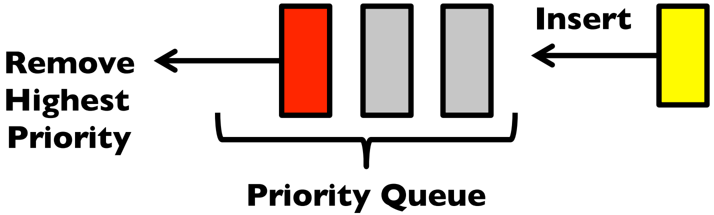
Queue Basics

- A Queue is a collection that supports specific behavior
  - **Insert** an element to the tail of the collection
  - **Remove** an element from the head of the collection
- “First-in, First-out”



Priority Queue

- Assume each element has associated priority
- In a **Priority Queue** one can insert an element
  - **Remove** now removes element with highest priority (i.e., not just the oldest-one inserted)
- Could implement by sorting
  - Too costly
  - Consider Heap data structure



Why Heap?

- Heap returns maximum element in O(1)
- Size of the heap is limited to number of vertices
- Special Priority Queue structure
  - Most PQ implementations do not allow you to adjust the priority of an element already in the PQ
  - This capability is required by Dijkstra’s Algorithm
  - I’ve provided Binary Heap implementation (BHeap)

SingleSource ShortestPath

- Instead of $O(n^2)$ algorithm
  - Not quite O(n log n) where n is number of vertices
- Use PriorityQueue
  - Contains vertices v with priority equal to computed dist[v]
  - decreaseKey (official name) updates information for *v* in PQ to reduce dist[v] and thereby increase priority to be selected

Dijkstra's Algorithm PQ

Best	Average	Worst
O((V+E)*log V)	*same*	*same*

Weighted Directed Graph

Priority queue

Array

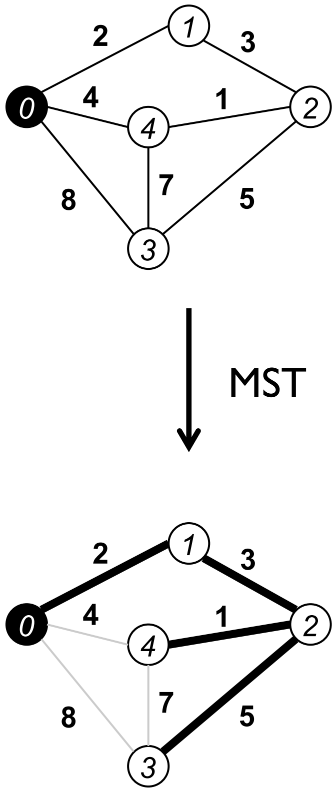
Overflow

```
def singleSourceShortest (G, s):
  PQ = new Priority Queue
  set dist[v] to ∞ for all v∈G
  set pred[v] to –1 for all v∈G
  dist[s] = 0
  foreach v∈G do
    PQ.insert (v, dist[v])

  while (PQ is not empty) do
    u = getMin (PQ)
    foreach neighbor v of u do
      w = weight of edge (u,v)
      newLen = dist[u] + w
      if (newLen < dist[v]) then
        decreaseKey (PQ, v, newLen)
        dist[v] = newLen
        pred[v] = u
```

BinaryHeap Problem

- Minimum Spanning Tree (MST)
  - Given undirected, connected graph
  - Subset of edges that retain connected property of graph with smallest accumulated edge weights
- Prim's Algorithm uses BHeap structure



Single-Source Shortest Path Summary

- Assumptions
  - Edges all have positive weight
  - Priority Queue implementation supports decreaseKey
- Demonstrates greedy strategy
  - When solving sub-problems, pursue best local strategy, which ultimately solves global problem