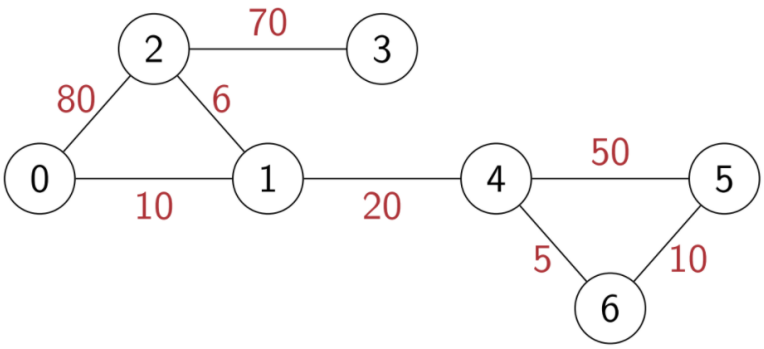
Shortest Paths in Weighted Graphs

Weighted Graphs

- Recall that BFS explores a graph level-by-level
- BFS computes the shortest path, in terms on number of edges, to every reachable vertex
- May assign values to edges
  - Cost, time, distance, ...
  - Weighted graph
- G = (V, E), W: E→R, where R represents the set of real numbers



- **Adjacency matrix**
  - Record weights along with edge information -- weight is always 0 if there is no edge

	0	1	2	3	4	5	6
0	(0,0)	(1,10)	(1,80)	(0,0)	(0,0)	(0,0)	(0,0)
1	(1,10)	(0,0)	(1,6)	(0,0)	(1,20)	(0,0)	(0,0)
2	(1,80)	(1,6)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)
3	(0,0)	(0,0)	(1,70)	(0,0)	(0,0)	(0,0)	(0,0)
4	(0,0)	(1,20)	(0,0)	(0,0)	(0,0)	(1,50)	(1,5)
5	(0,0)	(0,0)	(0,0)	(0,0)	(1,50)	(0,0)	(1,10)
6	(0,0)	(0,0)	(0,0)	(0,0)	(1,5)	(1,10)	(0,0)

- **Adjacency list**
  - Record weights along with edge information

0	[(1,10),(2,80)]
1	[(0,10),(2,6),(4,20)]
2	[(0,80),(1,6),(3,70)]
3	[(2,70)]
4	[(1,20),(5,50),(6,5)]
5	[(4,50),(6,10)]
6	[(4,5),(5,10)]

```python
#Weighted directed graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in dedges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in dedges:
    WL[i].append((j,d))
print(WL)
```

```python
#Weighted undirected graph
#Adjacency matrix representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
import numpy as np
W = np.zeros(shape=(size,size,2))
for (i,j,w) in edges:
    W[i,j,0] = 1
    W[i,j,1] = w
print(W)

#Adjacency list representation in Python
dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),
          (2,3,70),(4,5,50),(4,6,5),(5,6,10)]
edges = dedges + [(j,i,w) for (i,j,w) in dedges]
size = 7
WL = {}
for i in range(size):
    WL[i] = []
for (i,j,d) in edges:
    WL[i].append((j,d))
print(WL)
```

Shortest paths in weighted graphs

- BFS computes shortest path, in terms of number of edges, to every reachable vertex
- In a weighted graph, add up the weights along the path
- Weighted shortest path need not have minimum number of edges
    - Shortest path from 0 to 2 is via 1

Single source shortest paths

- Find shortest paths from a fixed vertex to every other vertex
- Transport finished product from factory (single source) to all retail outlets
- Courier company delivers items from distribution centre (single source) to addresses

All pairs shortest paths

- Find shortest paths between every pair of vertices i and j
- Optimal airline, railway, road routes between cities

Negative edge weights

- Can negative edge weights be meaningful?
- Taxi driver trying to head home at the end of the day
    - Roads with few customers, drive empty (positive weight)
    - Roads with many customers, make profit (negative weight)
    - Find a route toward home that minimizes the cost

Negative cycles

- A negative cycle is one whose weight is negative
    - Sum of the weights of edges that make up the cycle

- By repeatedly traversing a negative cycle, total cost keeps on decreasing
- If a graph has a negative cycle, shortest paths are not defined
- Without negative cycles, we can compute shortest paths even if some weights are negative

Summary

- In a weighted graph, each edge has a cost
  - Entries in adjacency matrix capture edge weights
- Length of a path is the sum of the weights
  - Shortest path in a weighted graph need not be the minimum in terms of number of edges
- Different shortest path problems
  - Single source: four one designated vertex to all others
  - All-pairs: between every pair of vertices
- Negative edge weights
  - Should not have negative cycles
  - Without negative cycles, shortest paths still well defined

Dijkstra's Algorithm : Single Source Shortest Path

- Dijkstra’s algorithm works for both directed and undirected graphs.
- Dijkstra’s algorithm doesn’t work for graphs with negative weights or negative weight cycles.
- This algorithm returns the shortest distance from the source to all other nodes, but after some modification like maintaining parent information of each node we can find out the shortest path.

Implementation

- Maintain two dictionaries with vertices as they keys
  - visited, initially False for all v (burnt vertices)
  - distance, initially infinity for all v (expected burn time)
- Set distance[s] to 0
- Repeat, until all reachable vertices are visited
  - Find unvisited vertex nextv with minimum distance
  - Set visited[nextv] to True
  - Recompute distance[v] for every neighbour v of nextv

Summary

- Dijkstra's algorithm computes single source shortest paths
- Use a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Need to prove that such a strategy is correct
  - Correctness requires edge weights to be non-negative

Complexity is $O(n^2)$

- Even with adjacency lists
- Bottleneck is identifying unvisited vertex with minimum distance
- Need a better data structure to identify and remove minimum (or max) from a collection.

```python
# Adjacency matrix implementation
def dijkstra(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows + 1
  (visited, distance) = ({}, {})

  for v in range(rows):
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                     if not visited[v]])
    nextvlist = [v for v in range(rows)
                   if (not visited[v]) and
                       distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for v in range(cols):
      if WMat[nextv, v, 0] == 1 and (not visited[v]):
        distance[v] = min(distance[v], distance[nextv]
                          + WMat[nextv, v, 1])

  return distance
```

```python
# Adjacency list implementation
def dijkstra(WList, s):
  infinity = 1 + len(WList.keys()) * max([d for u in WList.keys()
                                          for (v, d) in WList[u]])
  (visited, distance) = ({}, {})

  for v in WList.keys():
    (visited[v], distance[v]) = (False, infinity)

  distance[s] = 0

  for u in WList.keys():
    nextd = min([distance[v] for v in WList.keys()
                     if not visited[v]])
    nextvlist = [v for v in WList.keys()
                   if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v], distance[nextv] + d)

  return distance
```

Bellman Ford algorithm : Single Source Shortest with Negative Weights

- Bellman-Ford works for both directed and undirected graphs with non-negative edges weights.
- Bellman-Ford does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Bellman-Ford works for a directed graph with negative edge weight, but not with negative weight cycle.

Complexity

- **O(n^3)** for adjacency matrix.
- **O(mn)** for adjacency list : where m is number of edges and n is number of vertices.

Summary

- Dijkstra's algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex "burns"
  - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find the shortest paths of length 1,2,...,n−1
- Update distance of each vertex with every iteration -- **Bellman-Ford algorithm**
- $$ time with adjacency matrix, **O(mn)** time with adjacency list
- if Bellman-Ford algorithm does not converge after n−1 iterations, there is a negative cycle

```
def bellman_ford(WMat, s):
  (rows, cols, x) = WMat.shape
  infinity = np.max(Wmat) * rows + 1
  distance = {}

  for v in range(rows):
    distance[v] = infinity

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u, v, 0] == 1:
          distance[v] = min(distance[v],
                distance[u] + WMat[u, v, 1])

  return distance
```

```
def bellman_ford_list(WList, s):
  infinity = 1 + len(WList.keys()) * max
  ([d for u in WList.keys()for (v, d) in WList[u]])
  distance = {}

  for v in WList.keys():
    distance[v] = infinity

  distance[s] = 0

  for i in WList.keys():
    for u in WList.keys():
      for (v, d) in WList[u]:
        distance[v] = min(distance[v],
                          distance[u] + d)

  return distance
```

All pair of shortest path

- Find the shortest paths between every pair of vertices i and j.
- It is equivalent to if run Dijkstra or Bellman-Ford from each vertex.

Floyd-Warshall algorithm

- Floyd-Warshall's works for both directed and undirected graphs with non-negative edges weights.
- Floyd-Warshall's does not work with an undirected graph with negative edges weight, as it will be declared as a negative weight cycle.
- Floyd-Warshall's algorithm is an alternative way to compute transitive closure **B k [i, j] = 1** if we can reach j from i using vertices in {0, 1, . . . , k−1}
- Floyd-Warshall works for a directed graph with negative edge weight, but not with a negative weight cycle.
- Formula for Floyd-Warshall algorithm is given below:-
- $$SP^k[i,j] = min[SP^{k-1}[i,j],\ SP^{k-1}[i,k] + SP^{k-1}[k,j]]$$

Summary

- Warshall's algorithm is an alternative way to compute transitive closure
  - **B^k[i,j]=1** if we can reach j from i using the vertices in {0,1,...,k−1}

- Adapt Warshall's algorithm to compute all pairs of shortest paths
  - **SP^k[i,j]** if the length of the shortest path from i to j using vertices in {0,1,...,k−1}
  - **SP^n[i,j]** is the length of the overall shortest path

- **Works with negative edge weights assuming no negative cycles**
- **Simple nested loop implementation, time $O(n^3)$**
- Space can be limited to **$O(n^2)$** by re-using 2 "slices" SP and SP'

```python
#For adjacency matrix
def floyd_warshall(WMat):
  (rows, cols, x) = WMat.shape
  infinity = np.max(WMat) * rows * rows + 1
  Sp = np.zeros(shape=(rows, cols, cols + 1))

  for i in range(rows):
    for j in range(cols):
      SP[i, j, 0] = infinity

  for i in range(rows):
    for j in range(cols):
      if WMat[i, j, 0] == 1:
        SP[i, j, 0] = WMat[i, j, 1]

  for k in range(1, cols + 1):
    for i in range(rows):
      for j in range(cols):
        SP[i, j, k] = min(SP[i, j, k - 1]),
        SP[i, k - 1, k - 1] + SP[k - 1, j, k - 1])

  return SP[:, :, cols]
```

Minimum Cost Spanning Tree(MCST)

Spanning Tree(ST)

- Retain a minimal set of edges so that graph remains connected
- Recall that a minimally connected graph is a tree
- Adding an edge to a tree creates a loop
- Removing an edge disconnects the graph
- Want a tree that connects all the vertices — **spanning tree**
- More than one spanning tree, in general

Spanning trees with cost

- Restoring a road or laying a fiber optic cable has a cost
- Minimum cost spanning tree
  - Add the cost of all the edges in the tree
  - Among the different spanning trees, choose one with the minimum cost
- Some facts about trees
  - **A tree on n vertices has exactly n − 1 edges**
  - **Adding an edge to a tree must create a cycle.**
  - **In a tree, every pair of vertices is connected by a unique path**

Building minimum cost spanning trees

- We will use these facts about trees to build minimum cost spanning trees
- Two natural strategies
- Start with the smallest edge and "grow" a tree
  - **Prim's Algorithm**
- Scan the edges in ascending order of weight to connect components without forming cycles
  - **Kruskal's Algorithm**

Summary

- **Prim's algorithm grows an MCST starting with any vertex**
- Implementation similar to Dijkstra's algorithms : Update rule for distance is different

- At each step, connect one more vertex to the tree using minimum cost edge from inside the tree to outside the tree

- Complexity is **O(n^2)**
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
  - Need a better data structure to identify and remove minimum (or max) from a collection

- **Kruskal's algorithm builds an MCST bottom up**
  - Start with n components, each an isolated vertex
  - Scan the edges in ascending order of cost
  - Whenever an edge merges disjoint components, add it to the MCST
- Correctness follows from Minimum Separator Lemma
- Complexity is **O(n^2)** due to naive handling of components
  - We will see how to improve to **O(m.logn)**
- If edge weights repeat, MCST is not unique
- "Choose minimum cost edge" will allow choices
  - Consinder a triangle on 3 vertices with all edges equal
- Different choices lead to different spanning trees
- In general, there may be a very large number of minimum cost spanning trees

```
#Prim's Algorithm using List
def prim_list2(WList):
  infinity = 1 + max([d for u in WList.keys()
                        for (v, d) in WList[u]])
  (visited, distance, nbr) = ({}, {}, {})

  for v in WList.keys():
    (visited[v], distance[v], nbr[v]) =
                    (False, infinity, -1)

  visited[0] = True

  for (v, d) in WList[0]:
    (distance[v], nbr[v]) = (d, 0)

  for i in range(1, len(WList.keys())):
    nextd = min([distance[v] for v in WList.keys()
                    if not visited[v]])
    nextvlist = [v for v in WList.keys()
                    if (not visited[v]) and
                          distance[v] == nextd]

    if nextvlist == []:
      break

    nextv = min(nextvlist)
    visited[nextv] = True

    for (v, d) in WList[nextv]:
      if not visited[v]:
        (distance[v], nbr[v]) = (min(distance[v], d),
                                 nextv)
  return nbr
```

```
#Kruskal's Algorithm using List
def kruskal(WList):
  (edges, components, TE) = ([], {}, [])

  for u in WList.keys():
    edges.extend([(d,u,v) for (v,d) in WList[u]])
    component[u] = u
  edges.sort()

  for (d, u, v) in edges:
    if component[u] != component[v]:
      TE.append((u, v))
      c = component[u]

      for w in WList.keys():
        if component[w] == c:
          component[w] = component[v]

  return TE
```