Scientific Computation

Spring, 2019

Lecture 8

Today

- Depth-first search
- Dijkstra's algorithm for weighted networks

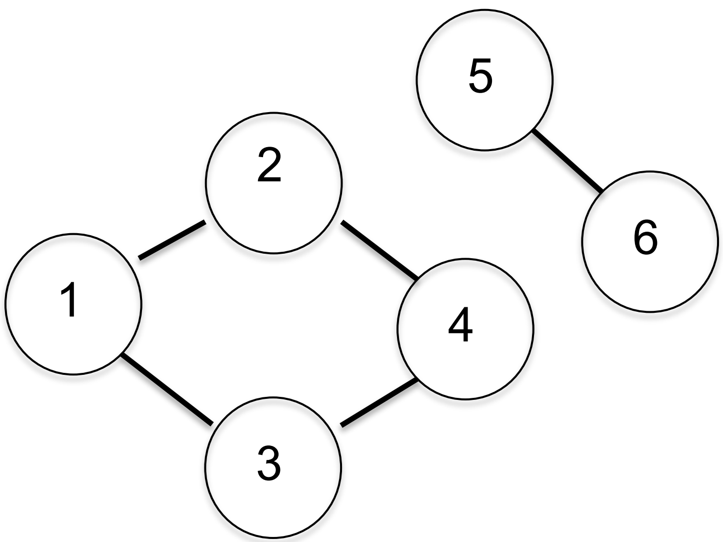
Graph search

- **Basic idea: Given a graph, G, and a source node, s, find all other nodes that can be reached from s**

- **Basic approach:**
  - **Label s as "explored" and all other nodes as "unexplored"**

  While there is at least one edge between an explored and unexplored node:

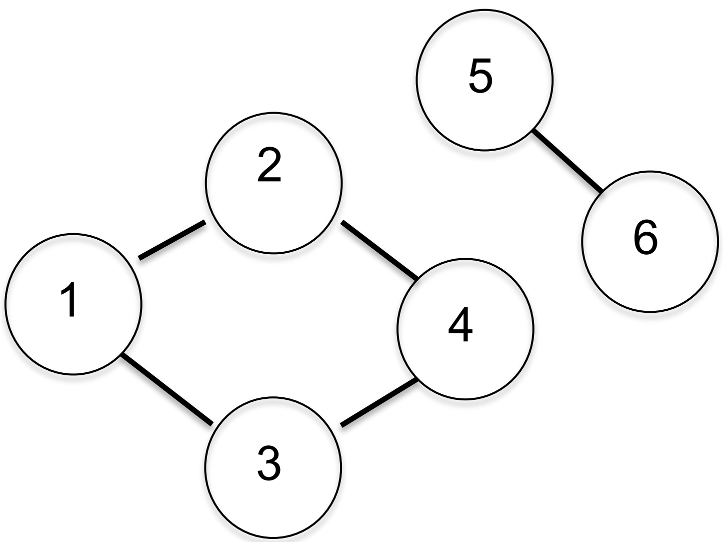
  Select one such edge and re-label the unexplored node as explored



Graph search

- **Basic approach:**
  - **Label** s **as "explored" and all other nodes as "unexplored"**

While there is at least one edge between an explored and unexplored node:
Select one such edge and re-label the unexplored node as explored



Implementation: Depends on how edges are selected each iteration

- **Depth-first search – aggressively move into the graph (1-2, 2–4)**
- **Breadth-first search – consider one "layer" at a time (1-2, 1-3)**
- **Today: DFS**

Breadth-first search

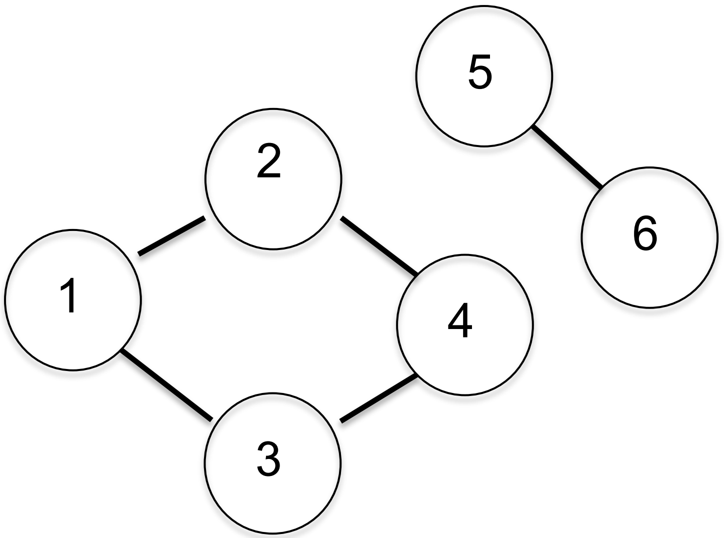
- **Python implementation**

  - **Specify: Graph and source node**
  - **Maintain: 1) list of nodes, 2) list of labels for nodes and 3) *queue* of nodes to-be explored**
    - **Initialize the queue with the source node and mark it as explored**
    - **Remove nodes from the queue in the order they were added (first in, first out)**
    - **Search through edges of removed node and add unexplored nodes to queue**
      - **Label added nodes as explored**
    - **Terminate search when queue is empty**

Breadth-first search

- **Specify: Graph and source node**

```
G = nx.Graph()
edges = [[1,2],[1,3],[1,2],[2,4],[3,4],[5,6]]
G.add_edges_from(edges)
s = 1
Q = [s] #Nodes to be explored
```



- **Create list of nodes and labels**

```
a_list = G.adjacency_list()
nodes = G.nodes()
z = [0 for i in nodes] #labels
L = [nodes,z]
L[1][s-1]=1 #mark source node as explored
```

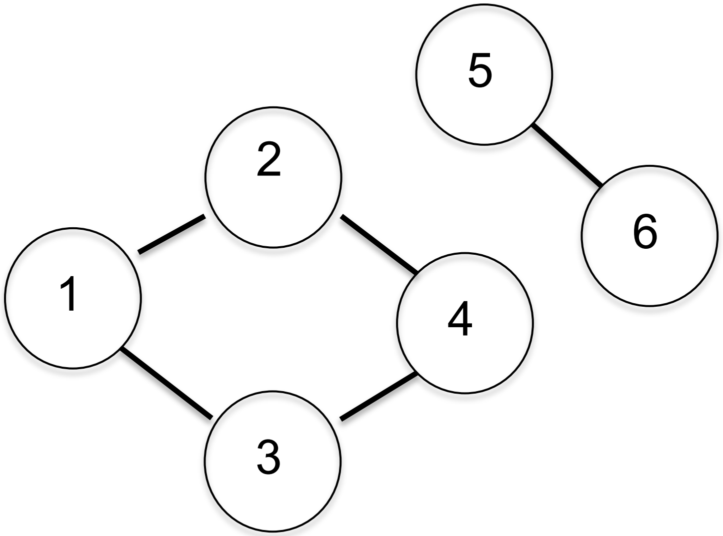
- **Iterate through nodes in queue (updating Q as appropriate)**

```
while len(Q)>0:
    n = Q.pop(0)
    for v in a_list[n-1]: #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            Q.append(v)
```

Breadth-first search

- Adding `print("n=%d,Q=" %(n),Q)` to the while loop and running the code:

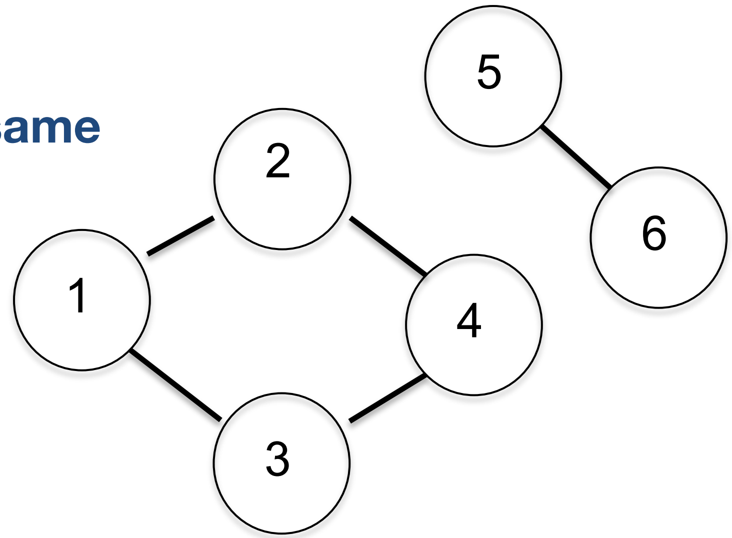
```
n=1,Q= [2, 3]
n=2,Q= [3, 4]
n=3,Q= [4]
n=4,Q= []
```



- n is the node removed from the queue and Q is the queue after edges from n have been added (if n is unexplored

Depth-first search

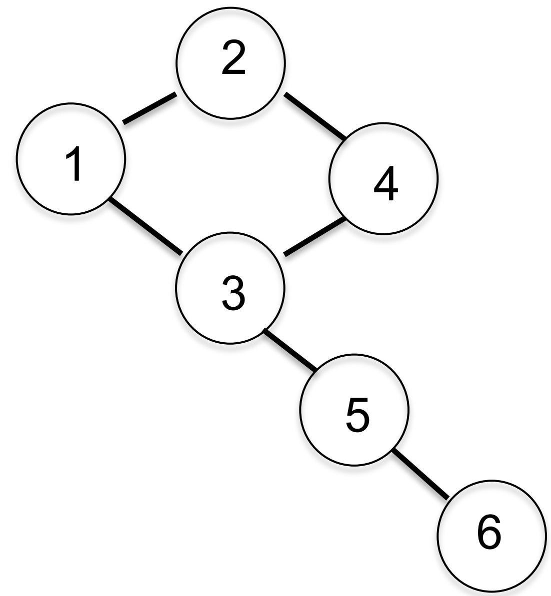
- What changes for DFS?
- How much does the code change?
- The problem setup and initialization will be the same
- Can focus on the search through the graph
- BFS: Remove node from front of queue, append unexplored neighbors to back of queue
- DFS: Remove node from end of *stack*, append unexplored neighbors to end of stack




```
#Depth-first search
while len(Q)>0:
    n = Q.pop() #Only change from BFS
    for v in a_list[n-1]: #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            Q.append(v)
```

Depth-first search

- **Let's compare DFS and BFS on this modified graph**
- **Output from codes is provided below**
- **Compare the order in which the nodes are visited**




```
In [16]: run bfs1
n=1,Q= [2, 3]
n=2,Q= [3, 4]
n=3,Q= [4, 5]
n=4,Q= [5]
n=5,Q= [6]
n=6,Q= []
```

```
In [17]: run dfs1
n=1,Q= [2, 3]
n=3,Q= [2, 4, 5]
n=5,Q= [2, 4, 6]
n=6,Q= [2, 4]
n=4,Q= [2]
n=2,Q= []
```

- **We have left the length calculation in the DFS code the same as in BFS – is this correct?**

Weighted networks

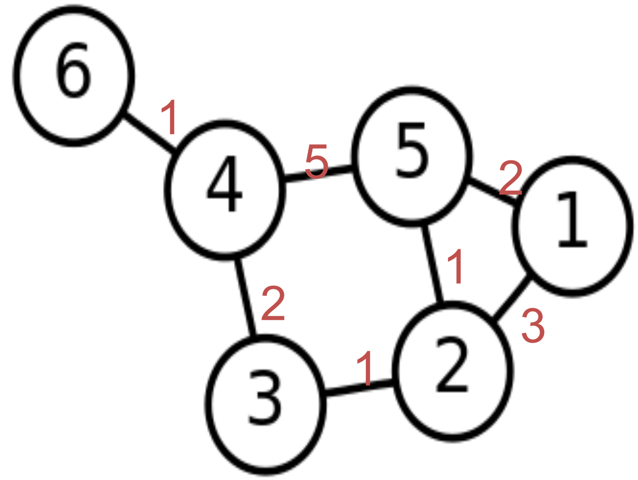
- **Have considered unweighted networks so far**

- **Often have a weight associated with each edge**
  - **E.g. distance between nodes, time to travel between nodes**
  - **More generally, nodes can represent states in a state space, and weights can represent the cost of moving between states**

- **What is the best way to compute shortest paths in a weighted network?**

- **Can we use BFS or DFS? If weights are small, can replace a weight-n link with n weight-1 links**
  - **But weights can be arbitrarily large**

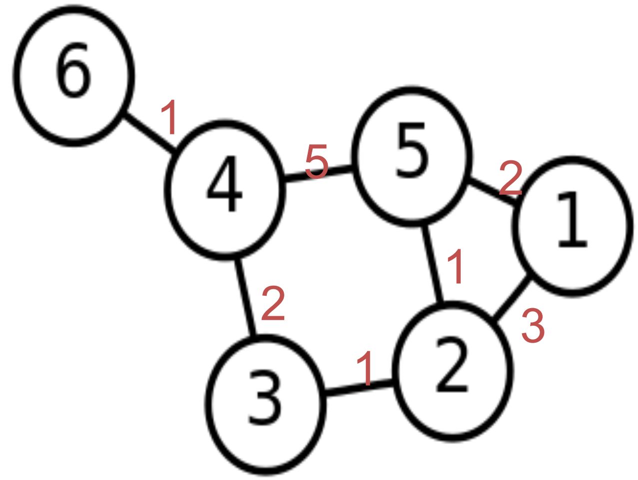
- **Dijkstra's algorithm is a graph-search method for networks with positive weights**



Shortest paths:
1→2: 3
1→4: 6

Dijkstra’s algorithm

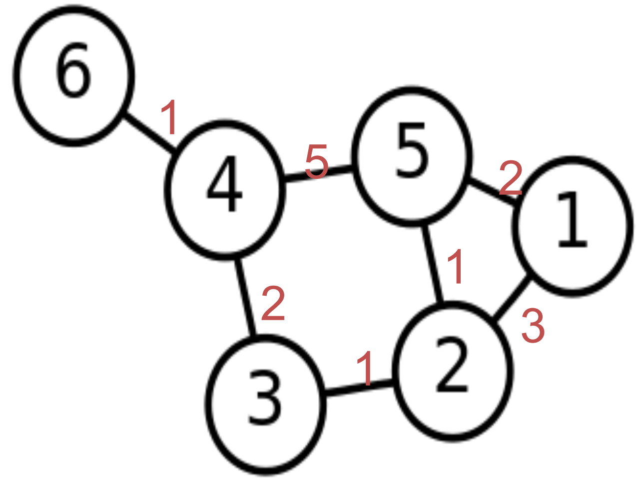
- **Dijkstra’s algorithm is a graph-search method for networks with positive weights**
- **Input: Graph (nodes, edges, weights) and source node**
- **Output: Shortest paths from source node to all nodes reachable from source**



Shortest paths:
1→2: 3
1→4: 6

Dijkstra's algorithm

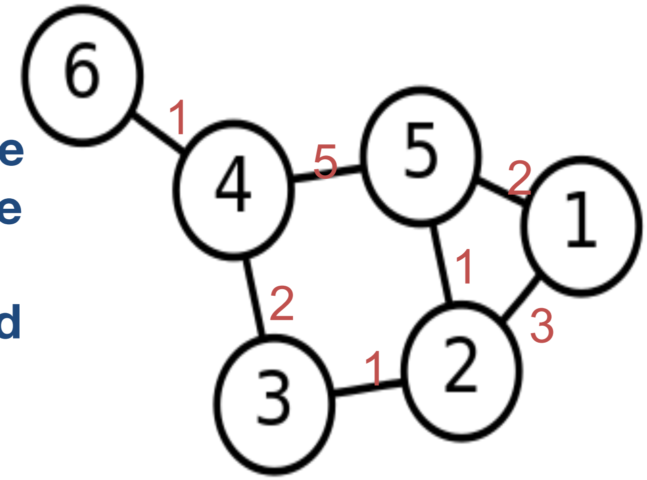
- **Dijkstra's algorithm is a graph-search method for networks with positive weights**
- **Input: Graph (nodes, edges, weights) and source node**
- **Output: Shortest paths from source node to all nodes reachable from source**
- **Basic idea:**
  - **Maintain lists of explored and unexplored nodes (E and U)**
  - **For each explored node, assume that we have calculated the shortest distance to the source**
  - **Each iteration, move one node from U to E**
    - **Which node? The neighbor of an explored node that is closest to the source.**



Shortest paths:
1→2: 3
1→4: 6

Dijkstra's algorithm

- **Basic idea:**
  - **Maintain lists of explored and unexplored nodes (E and U)**
  - **For each explored node, assume that we have calculated the shortest distance to the source**
  - **Each iteration, move one node from U to E**
    - **Which node? The neighbor of an explored node that is closest to the source,** s



Shortest paths:
1→2: 3
1→4: 6

- **Why is this the right node to move?**
  - **Let** w_{ij} **be the weight of the edge between nodes** i **and** j
  - **Let** d_i **be the length of the shortest path between node** i **and the source,** s
  - **Then** d_i+w_{ij} **is the length of the path between the source and node** j **which includes edge** i,j
  - **If we choose the "unexplored neighbor" with the minimum value of (**d_i+w_{ij}**), then all other paths from s to j will be longer (why?)**

Dijkstra's algorithm

Algorithm (sketch)

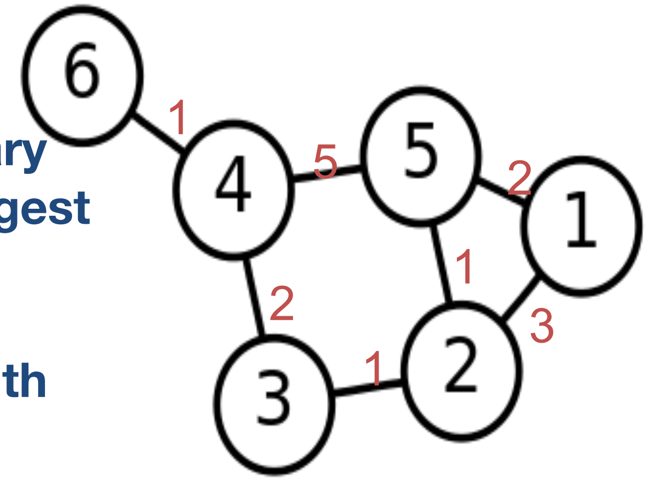
1) **Set the distance for the source to be zero**
2) **Set the distances for all other nodes to an arbitrary large number (much larger than the expected largest path lengths)**
3) **Label source as explored**
4) **Set provisional distances for all neighbors of** s **with weights of edges between** s **and neighbors**

5) **Each iteration, find the unexplored node (node** i**) that is closest to the source (smallest provisional distance)**

6) **Label this node as explored**

7) **Update the provisional distance of all neighbors of** i **with min(**$d_j, d_i + w_{ij}$**)**

8) **Continue "exploring" until no reachable nodes remain unexplored.**

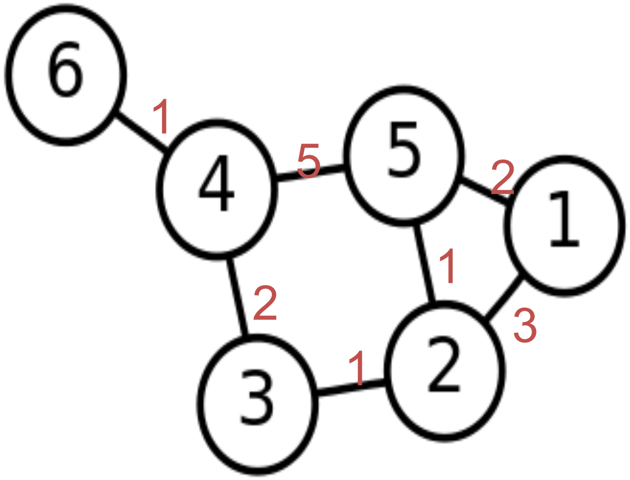


Shortest paths:
1→2: 3
1→4: 6

Dijkstra’s algorithm

Python implementation

- What data type(s) should we use?
  - list, array, dictionary, something else?
- Basic operations needed:
  - find min
  - insert/delete
- Let’s try dictionaries

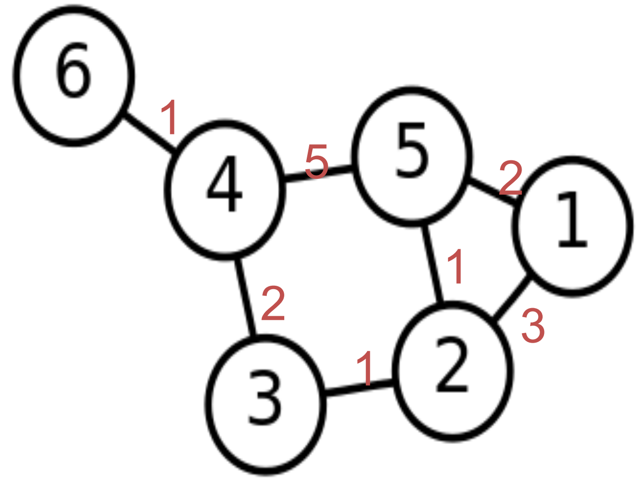


Shortest paths:
1→2: 3
1→4: 6

Dijkstra’s algorithm

Python implementation

- What data type(s) should we use?
    - list, array, dictionary, something else?
- Basic operations needed:
    - find min
    - insert/delete
- Let’s try dictionaries
- We’ll work with two dictionaries – one for explored nodes (where the shortest path length has been determined) and one for unexplored neighbors of explored nodes (where provisional shortest paths have been specified)
- And we’ll also have the adjacency list of the graph containing the edges and their weights



Shortest paths:
1→2: 3
1→4: 6

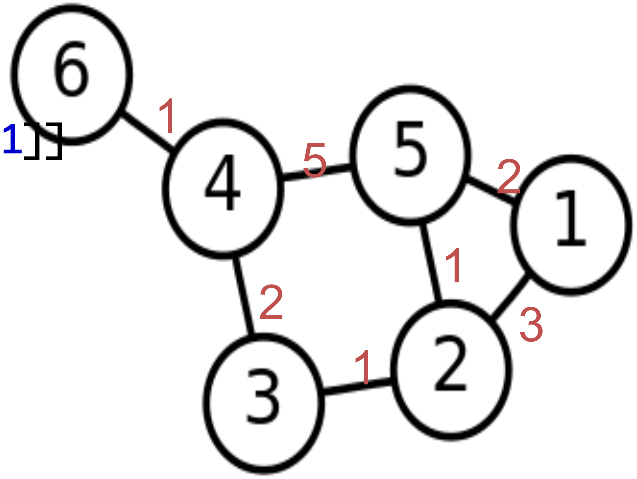
Dijkstra's algorithm

Python implementation

1. Create the graph, initialize the dicts

```
e=[[1,2,3],[1,5,2],[2,5,1],[2,3,1],[3,4,2],[4,5,5],[4,6,1]]
G = nx.Graph()
G.add_weighted_edges_from(e)

Udict = {} #Unexplored neighbor nodes
Udict[5]=0
Edict={} #Explored nodes
```



Shortest paths:
1→2: 3
1→4: 6

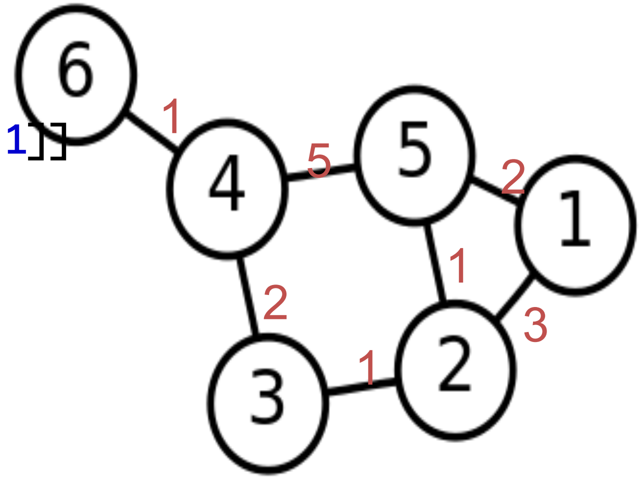
Dijkstra's algorithm

Python implementation

1. Create the graph, initialize the dicts

```
e=[[1,2,3],[1,5,2],[2,5,1],[2,3,1],[3,4,2],[4,5,5],[4,6,1]]
G = nx.Graph()
G.add_weighted_edges_from(e)

Udict = {} #Unexplored neighbor nodes
Udict[5]=0
Edict={} #Explored nodes
```



2. Find node in Udict with smallest distance

```
while len(Udict)>0:
    dmin=2000000
    for k,v in Udict.items():
        if v<dmin:
            dmin=v #min distance
            n=k #corresponding node
```

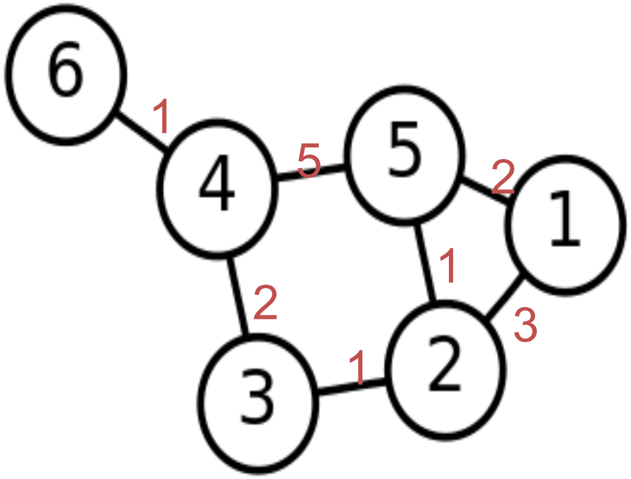
Shortest paths:
1→2: 3
1→4: 6

3. Remove 'smallest-distance' node and update provisional distances of its neighbors

Dijkstra's algorithm

Python implementation
2. Find node in Udict with smallest distance

```
while len(Udict)>0:
    dmin=2000000
    for k,v in Udict.items():
        if v<dmin:
            dmin=v #min distance
            n=k #corresponding node
```

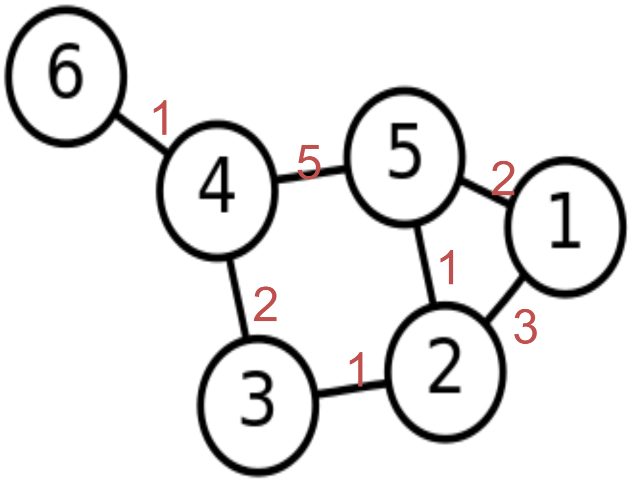


Shortest paths:
1→2: 3
1→4: 6

3. Remove 'smallest-distance' node and update provisional distances of its unexplored neighbors

```
     for m,en,wn in G.edges(n,data='weight'):
            ddist = dmin+wn
            if en in Edict:
                pass
            elif en in Udict:
                Udict[en] =
    min(Udict[en],ddist)
            else:
                Udict[en] = ddist
        Edict[n] = Udict.pop(n)
```

Dijkstra's algorithm



Setting the source to be node 5 and running the code:

```
In [56]: Edict
Out[56]: {1: 2, 2: 1, 3: 2, 4: 4, 5: 0, 6: 5}
```

The key-value pairs are node:distance

Dijkstra's algorithm

What is the cost for a graph with N nodes and M edges?

- **Estimate cost of setting up graph to be O(N+M)**
- **Each edge is only considered maximum of two times → O(M)**
- **dmin calculation → O(N^2) operations**

- **So it is the minimum calculation which is the bottleneck**

- **Can we do better?**