```
import numpy as np
import matplotlib.pyplot as plt
import networkx
```

```
%matplotlib inline
```

Dijkstra's algorithm

Here we will assume that a graph is given by a dictionary of weighted edges. The keys are the `(start, end)` pair for the edge, and the values are the weights. For example,

```
G = {
    (0, 1): 2,
    (0, 3): 3,
    (1, 2): 7,
    (1, 3): 4,
    (2, 1): 4,
    (2, 3): 2,
    (2, 4): 2,
    (2, 5): 1,
    (3, 1): 7,
    (3, 2): 1,
    (3, 4): 8,
    (4, 2): 7,
    (4, 3): 6,
    (4, 5): 2,
    (5, 2): 8,
    }
```

Utilities

First write functions to convert from the graph-as-dictionary to get the nodes list, adjacency list, and adjacency matrix.

```
def get_nodes(G):
    """
    Get the list of nodes of a graph from the dictionary.

    Parameters
    ----------
    G : dictionary
        Describes the graph as edge-weight pair.

    Returns
    -------
    V : list
        The nodes
    """
    nodes = []
    for (s, e) in G.keys():
        nodes.append(s)
        nodes.append(e)
    # Items in sets are unique. So, convert to set then to list, then
sort.
    V = sorted(list(set(nodes)))
    return V
```

```
get_nodes(G)
```

```
[0, 1, 2, 3, 4, 5]
```

Now get the adjacency list. This is a list of lists. The `i` th entry gives the nodes that can be reached by a single edge starting from `i`.

```
def adjacency_list(G):
    """
    Get the adjacency list of a graph from the dictionary.

    Parameters
    ----------
    G : dictionary
        Describes the graph as edge-weight pair.

    Returns
    -------
    adjacency_list : list
        The list as described
    """
    V = get_nodes(G)
    adjacency_list = []
    for v in V:
        nodes = []
        for w in V:
            if (v, w) in G:
                nodes.append(w)
        adjacency_list.append(nodes)
    return adjacency_list
```

```
adjacency_list(G)
```

```
[[1, 3], [2, 3], [1, 3, 4, 5], [1, 2, 4], [2, 3, 5], [2]]
```

Now get the adjacency matrix. For row `i` and column `j` this gives has $A_{i,j} = w$, where `w` is the weight of the edge between `i` and `j`. If there is no such edge the value is set to infinity.

```
def adjacency_matrix(G):
    """
    Get the adjacency list of a graph from the dictionary.

    Parameters
    ----------
    G : dictionary
        Describes the graph as edge-weight pair.

    Returns
    -------
    adjacency_matrix : array
        The matrix as described
    """
    V = get_nodes(G)
    n = len(V)
    adjacency_matrix = np.zeros((n, n))
    for i, v in enumerate(V):
        for j, w in enumerate(V):
            if (v, w) in G:
                adjacency_matrix[i, j] = G[(v, w)]
            else:
                adjacency_matrix[i, j] = np.inf
    return adjacency_matrix
```

```
adjacency_matrix(G)
```

```
array([[inf,  2., inf,  3., inf, inf],
       [inf, inf,  7.,  4., inf, inf],
       [inf,  4., inf,  2.,  2.,  1.],
       [inf,  7.,  1., inf,  8., inf],
       [inf, inf,  7.,  6., inf,  2.],
       [inf, inf,  8., inf, inf, inf]])
```

The slow version

This is the $\mathcal{O}(n^3)$ version.

```
def dijkstra_n3(G, s):
    """
    Dijkstra's algorithm finding the shortest paths from s on graph G.

    Parameters
    ----------
    G : dictionary
        Describes the graph as edge-weight pair.
    s : integer
        The starting node

    Returns
    -------
    Y : list
        The cost to reach each node starting from s
    P : list
        The predecessor on the shortest path
    """
    V = get_nodes(G)
    L = adjacency_list(G)
    Ml = adjacency_matrix(G)

    n = len(V)
    P = np.zeros((n,))
    Y = np.zeros((n,))
    for i in range(n):
        Y[i] = np.inf

    # Start from s.
    S = [s]
    Y[s] = 0
    while len(S) < n:
        smallest_seen = np.inf
        v, w = -1, -1
        # Find shortest total path out of S
        for i in S:
            for j in L[i]:
                if j in S:
                    continue
                if smallest_seen > Y[i] + Ml[i][j]:
                    smallest_seen = Y[i] + Ml[i][j]
                    v, w = i, j
        # Update path out of S, add node.
        Y[w] = Y[v] + Ml[v][w]
        P[w] = v
        S.append(w)

    return Y, P
```

```
dijkstra_n3(G, 0)
```

```
(array([0., 2., 4., 3., 6., 5.]), array([0., 0., 3., 0., 2., 2.]))
```

Cross-check using `networkx`

```
G_nx = networkx.DiGraph()
G_nx.add_weighted_edges_from([(s, e, w) for (s, e), w in G.items()])
```

```
for target in range(1, 6):
    print(target, networkx.dijkstra_path_length(G_nx, 0, target),
networkx.dijkstra_path(G_nx, 0, target)[-2])
```

```
1 2 0
2 4 3
3 3 0
4 6 2
5 5 2
```

The fast version

This is the $\mathcal{O}(n^2)$ version.

```
def dijkstra_n2(G, s):
    """
    Dijkstra's algorithm finding the shortest paths from s on graph G.

    Parameters
    ----------
    G : dictionary
        Describes the graph as edge-weight pair.
    s : integer
        The starting node

    Returns
    -------
    Y : list
        The cost to reach each node starting from s
    P : list
        The predecessor on the shortest path
    """
    V = get_nodes(G)
    L = adjacency_list(G)
    Ml = adjacency_matrix(G)

    n = len(V)
    P = np.zeros((n,))
    Y = np.zeros((n,))
    for i in range(n):
        Y[i] = Ml[s][i]

    # Start from s.
    S = [s]
    Y[s] = 0
    while len(S) < n:
        smallest_seen = np.inf
        w = -1
        # Update all paths out of S
        for j in V:
            if j in S:
                continue
            if smallest_seen > Y[j]:
                smallest_seen = Y[j]
                w = j
        S.append(w)
        for h in L[w]:
            if Y[h] > Y[w] + Ml[w][h]:
                Y[h] = Y[w] + Ml[w][h]
                P[h] = w

    return Y, P
```

```
dijkstra_n2(G, 0)
```

```
(array([0., 2., 4., 3., 6., 5.]), array([0., 0., 3., 0., 2., 2.]))
```