Shortest paths: Dijkstra

Topics for this week:
- weighted graphs
- weighted shortest path algorithm: Dijkstra
- priority queue data structure
Problem: Distance between locations

We have a map with specified length for every road. What is the shortest distance from city A to city D?

For example:
Problem: Distance between locations

Let's look at previous approaches - do any of them work?

- DFS (depth-first search)
- BFS (breadth-first search)
- backtracking
  - try all paths
- a greedy approach:
  - always follow the shortest outgoing road
Graph definitions

We need to expand our graph definition: now we have distances for every direct connection between locations.

Let's review the original terminology:
- nodes
- edges

Here we also have:

edge weights

Note: directed/undirected
Dijkstra’s algorithm

- a different greedy approach

- idea: keep temporary distances from the initial vertex to every other vertex
Dijkstra's algorithm

Pseudo code:

```python
def Dijkstra(graph G, node start, node finish):
    create an empty priority queue Q
    insert start into Q, with cost 0
    while Q is not empty:
        let current be the return of extractMin(Q)
        for every neighbor u of current:
            if u has not been finalized:
                if u is not in Q:
                    insert u into Q, with cost current.dist + weight(current, u)
                else:
                    if u's cost in Q > current.dist + weight(current, u):
                        update u's cost in Q to

    return
```

Running time:

- The queue contains at most $n$ elements, the naive extractMin takes $O(n)$ operations.
- Every node has at most $n$ neighbors, and the posted implementation uses at most $O(n)$ steps to update the cost; hence $O(n^3)$ can be implemented faster: $O(n^2)$
Dijkstra's algorithm

Pseudo code:

```python
def Dijkstra(graph G, node start, node finish):
    create an empty priority queue
    insert start into the queue, with cost 0
    start. predecessor = None
    while queue is non-empty:
        let current be the return of extractMin(queue)
        let current.distance = cost of current
        if current = finish:
            return finish.distance
        for every neighbor u of current:
            if u has not been finalized (u.distance is undefined):
                if u is not in the queue:
                    insert u into the queue, with cost current.distance + weight of edge to u
                else:
                    if u's cost > current.distance + weight of edge to u:
                        update u's cost to current.distance + weight of edge to u
                        if successfully updated:
                            u.predecessor = current
                if not u has finished:
                    u.predecessor = current
```

Path reconstruction: traverse predecessor's back to start (see the posted code)
Data structures

Priority queue - supports operations:

- insert
- extractMin
- updateCost

A data structure that keeps a collection of elements, each has a cost (sometimes referred to as priority). → we remove the element with the lowest cost first (the extractMin function)