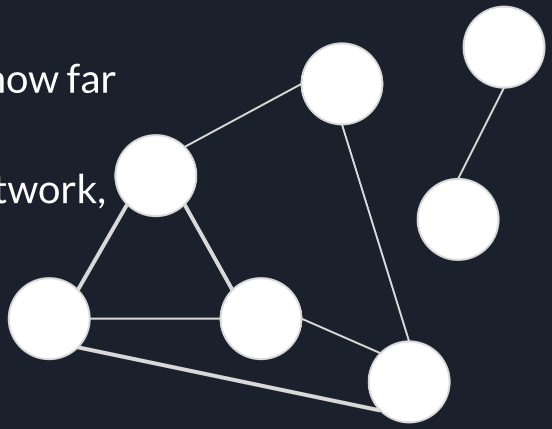
ProgTeam Week 7

Shortest Paths

What is a graph?

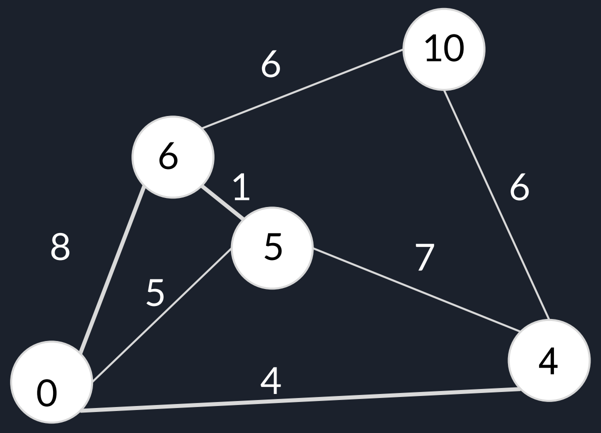
- Graph is a set of vertices and edges
- Edges are a pair of vertices (u,v)
  - Edges can be ordered or unordered; called *directed* and *undirected*
  - Edges can have an additional parameter, a *weight*
    - Some graphs are *unweighted*

- On a weighted graph, reasonable to ask how far two vertices are
  - Shortest distance in a computer network, route on road-system, etc.



Single Source Shortest Path: Dijkstra's Algorithm

- Edges all have positive weight (time taken to travel, etc.)
- What is the shortest path to a given destination from a given source?
- Solution: build paths by updating neighboring vertices
  - Select unprocessed vertex with current shortest path
    - All edges are positive: we'll never make it better!



Single Source Shortest Path: Dijkstra's Algorithm

Pseudocode: $O(E + N^2)$, N = # of vertices, E = # of edges

```
dist[i] = INF for all i in [0, n)

dist[s] = 0
while there are unvisited nodes:
    next = -1
    for j in [0, n):
        if dist[j] < dist[next] and j is unvisited:
            next = j
    next is visited // No path can improve dist[next]!
    for j in neighbors[next]:
        if dist[j] > dist[next] + weight[next][j]:
            dist[j] = dist[next] + weight[next][j]
            prev[j] = next // This is used for path recovery
```

Can we find the closest vertex faster than O(N)?

Single Source Shortest Path: Dijkstra's Algorithm

Solution: Use a heap! (A set can also work; in practice a heap is faster)

Heaps: Allow for insert/extract-minimum-element in Log(N) time

Pseudocode: O((E + N) Log N)

```
dist[i] = INF for all i in [0, n)
dist[s] = 0
pq = heap([0,n)) // will sort by minimum distance
while there are unvisited nodes:
    next = pq.extract_min()
    next is visited // No path can improve dist[next]!
    for j in neighbors[next]:
        if dist[j] > dist[next] + weight[next][j]:
            dist[j] = dist[next] + weight[next][j]
            prev[j] = next // This is used for path recovery
            pq.update_dist(j) // In practice, we just add another copy of j to the heap
```

Could use TreeSet instead of heap, but in practice heaps are much faster

Single-Source Shortest Path
Dijkstra's Algorithm

Final note: If we store the vertex that came before a given vertex, we can easily recover the path to a given vertex.

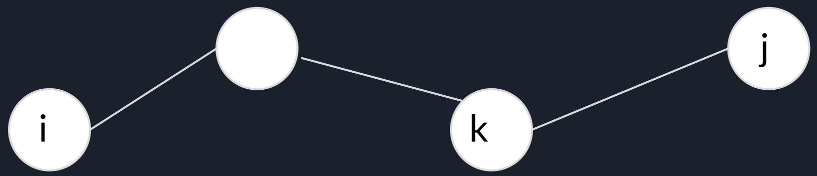
Pseudocode:

```
cur = ?
path = {cur}
while cur != s:
    cur = prev[cur]
    path.append(cur)
path.reverse()
```

All-Pairs Shortest Paths: Floyd-Warshall

- Goal: Find the shortest path from any vertex to any other vertex.
- (Every vertex is a source vertex)
- Usually use adjacency matrix instead of adjacency list; more efficient

- Key idea: combining paths
  - Suppose Dist[i][k] and Dist[k][j] store length of some paths from i to k and from k to j
    - Dist[i][k] + Dist[k][j] is length of some path from i to j

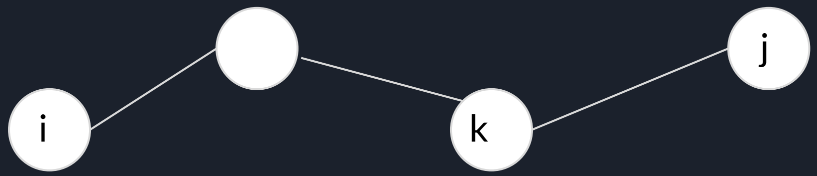


All-Pairs Shortest Paths: Floyd-Warshall

Pseudocode: $O(N^3)$

```
dist[i][j] = 0 if i == j // (for most problems)

dist[i][j] = INF if no edge between i and j
dist[i][j] = weight of edge between i and j else
for k in [0,n):
    for i in [0,n):
        for j in [0,n):
            new_dist = dist[i][k] + dist[k][j]
            if new_dist < dist[i][j]:
                dist[i][j] = new_dist
                next[i][j] = next[i][k] // Used for path recovery
```



All-Pairs Shortest Paths: Floyd-Warshall

We can recovery the path by storing the "next step"

Notice that when we merge two paths (i,k) and (k,j),

the second vertex in new path is second vertex in (i,k)

Initially next[i][k] should be k for all edges (i,k)

```
start = ?, end = ?

path = {start}
while start != end:
    start = next[start][end]
    path.add(start)
```