Shortest Paths
On weighted graphs
Weighted Shortest Paths

- The shortest path from a vertex $u$ to a vertex $v$ in a graph is a path $w_1 = u, w_2, ..., w_n = v$, where the sum:
  \[ \text{Weight}(w_1, w_2) + ... + \text{Weight}(w_{n-1}, w_n) \]
  attains its minimal value among all paths that start at $u$ and end at $v$.

- The length of a path of $n$ vertices is $n-1$ (the number of edges).

- If a graph is connected, and the weights are all non-negative, shortest paths exist for any pair of vertices:
  - Similarly for strongly connected digraphs with non-negative weights.
  - Shortest paths may not be unique.
Cycles and negative weights

- Negative weights may prevent the existence of shortest paths on graphs with cycles.
- In this example, there is no shortest path between $v_7$ and $v_8$.

The reason is that the simple cycle $c$ that starts and ends in $v_8$ has negative cost $-1$.

- The path $v_7, v_8, c, c, \ldots, c$, where $c$ repeats $t$ times has cost $2 - t$, so it can always be decreased, and no shortest path exists.

- In a connected graph, shortest paths exist if and only if no negative cost cycles exist.
How to compute shortest paths

- Consider initially the case where the edges are unweighted
  - Interested in shortest path in terms of minimum number of edges; conceptually equivalent to have all edges of same weight

- Answer: Breadth-first search

Shortest path from $v_3$ to $v_4$
How to compute shortest paths

- Consider initially the case where the edges are unweighted
  - Interested in shortest path in terms of minimum number of edges; conceptually equivalent to have all edges of same weight

- Answer: Breadth-first search

Shortest path from \( v_3 \) to \( v_4 \)
BFS for shortest path

```cpp
struct Vertex{
    private:
        list<Vertex> adj; // Adjacency list
        int dist;
    // ... data elements, etc.
    public: // Assume all vertices with current dist = -1;
        int findShortestPath(Vertex & u, Vertex & v){
            queue<Vertex> bfs;
            u.dist = 0;
            bfs.push(u);
            for(Vertex curr = bfs.front(); !(bfs.empty()) &&
                (curr != v); curr = bfs.front()) {
                for(list<Vertex>::iterator iter = curr.adj.begin();
                    iter!= curr.adj.end(); iter++) {
                    if((*iter).dist == -1 || (*iter).dist >
                        curr.dist+1) {
                        (*iter).dist = curr.dist + 1;
                        bfs.push(*iter);
                    }
                }
                bfs.pop();
            }
            return(v.dist);
        }
    }
```
void extractShortestPath(Vertex &u, 
    Vertex &v, stack<Vertex> & s) {
    s.push(v);
    for(Vertex curr = s.top(); curr!= u; 
        curr = s.top()) {
        for(list<Vertex>::iterator iter = 
            curr.adj.begin(); iter!=curr.adj.end(); 
            iter++) {
            if( (*iter).dist == curr.dist - 1) {
                s.push(*iter);
                break;
            }
        }
    }
}

- In the preceding, we learned how to compute minimum distances
- To compute shortest paths, we need to maintain an auxiliary structure
- Alternatively, if the graph is undirected, after computing the distances, one may extract the paths by tracing back from the destination
- If w is the last vertex found to be in the path, recover the predecessor of w as the neighbor z of w such that z.dist+weight(z,w)==w.dist
Weighted graphs: Dijkstra’s algorithm

- The weighted-edge version of the previous algorithm is called *Dijkstra’s algorithm for shortest paths*.
- The process is equivalent, except that the update of distance function uses the weight of the edge, instead of assuming it 1.
- For simplicity, assume no negative weights exist.
  - Then as before, use -1 to indicate an uncomputed distance.

Explicit changes:
- Change adjacency list to store pairs of (Vertex neigh, int edgeWeight).
- Update weight with the following test:
  ```c
  if ((*iter).vert.dist == -1 || (*iter).vert.dist > curr.dist+(*iter).edgeWeight) {
    (*iter).dist = curr.dist+1;
    bfs.push(*iter);
  }
  ```

The `bfs` must be now a priority queue, not a simple queue. Instead of iterating through `bfs`, the vertex of minimum distance is extracted at each time. Since we are assuming no negative-weight edges, the minimum distance-vertex cannot have its distance decreased by newfound paths from other vertices.