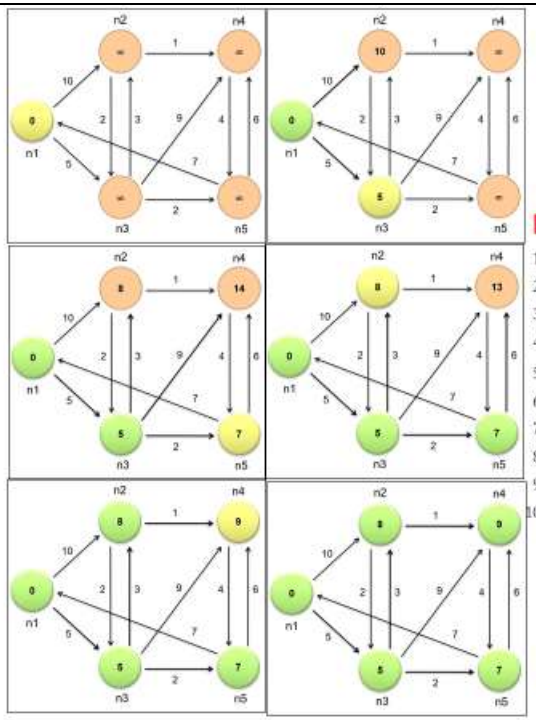
Dijkstra's Algorithm

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*. Set the initial node as the current node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance with the current assigned value, and assign the smaller one.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If all of destination nodes have been marked visited or if the smallest tentative distance among the nodes in the *unvisited set* is infinity, then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

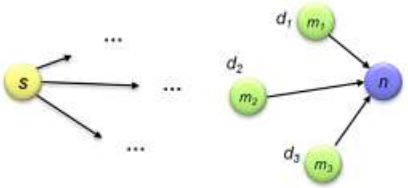


Dijkstra's Algorithm

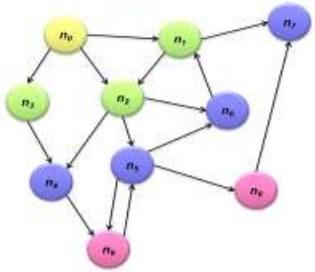
1: DIJKSTRA(G, w, s)
2: ~~$d[s] \leftarrow 0$~~
3: **for all** vertex $v \in V$ **do**
4: $d[v] \leftarrow \infty$
$d[s] \leftarrow 0$
5: $Q \leftarrow \{V\}$
6: **while** $Q \neq \emptyset$ **do**
7: $u \leftarrow$ EXTRACTMIN(Q)
8: **for all** vertex $v \in u$.ADJACENCYLIST **do**
9: **if** $d[v] > d[u] + w(u, v)$ **then**
10: $d[v] \leftarrow d[u] + w(u, v)$

Finding the Shortest Path

- Consider simple case of equal edge weights (i.e., weight=1)
- Solution to the problem can be defined inductively
- Here's the intuition:
  - Define: *b* is reachable from *a* if *b* is on adjacency list of *a*
  - DISTANCETO(*s*) = 0
  - For all nodes *p* reachable from *s*, DISTANCETO(*p*) = 1
  - For all nodes *n* reachable from some other set of nodes *M*, DISTANCETO(*n*) = 1 + min(DISTANCETO(*m*), $m \in M$)



Visualizing Parallel BFS



From Intuition to Algorithm

- Data representation:
  - Key: node n
  - Value: d (distance from start), adjacency list (list of nodes reachable from n)
  - Initialization: for all nodes except for start node, d = ∞
- Mapper:
  - $\forall$m $\in$ adjacency list: emit (m, d + 1)
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reducer:
  - Selects minimum distance path for each reachable node
  - Additional bookkeeping needed to keep track of actual path

Multiple Iterations Needed

- Each MapReduce iteration advances the "known frontier" by one hop
  - Subsequent iterations include more and more reachable nodes as frontier expands
  - Multiple iterations are needed to explore entire graph
- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits (n, adjacency list) as well

BFS Pseudo-Code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes

1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         dmin ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d                           ▷ Recover graph structure
8:             else if d < dmin then               ▷ Look for shorter distance
9:                 dmin ← d
10:        M.Distance ← dmin                       ▷ Update shortest distance
11:        Emit(nid m, node M)
```

if dmin < current distance, update; otherwise, keep the current distance

Weighted Edges

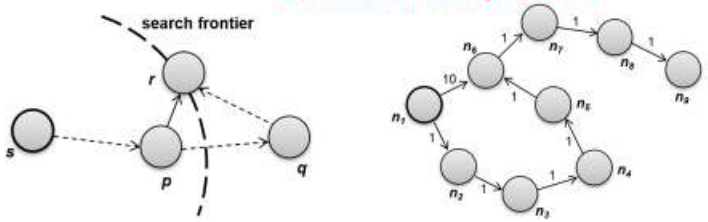
- Now add positive weights to the edges
- Simple change: adjacency list now includes a weight *w* for each edge
  - In mapper, emit $(m, d + w_p)$ instead of $(m, d + 1)$ for each node *m*

Stopping Criterion

- How many iterations are needed in parallel BFS (positive edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path
  - A node becomes "discovered" when the cost of the node becomes non-infinity.

NOT TRUE!

Additional Complexities



Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: "traversing" the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass along partial results via outlinks, keyed by destination node
  - Perform aggregation in reducer on inlinks to a node
  - Iterate until convergence: controlled by external "driver"
  - Don't forget to pass the graph structure between iterations

A practical implementation

- A node is represented by a string as follows
  - ID EDGES|WEIGHTS|DISTANCE_FROM_SOURCE|COLOR

The mappers

- All white nodes and black nodes only reproduce themselves
- For each gray node (e.g., an exploding node)
  - For each node n in the adjacency list, emit a gray node
    - n null|null|distance of exploding node + weight|gray
  - Turn its own color to black and emit itself
    - ID edges|weights|distance from source|black

The reducers

- Receive the data for all "copies" of each node
- Construct a new node for each node
  - The non-null list of edges and weights
  - The minimum distance from the source
  - The proper color

Choose the proper color

- If only receiving a copy of white node, color is white
- If only receiving a copy of black node, color is black
- If receiving copies consisting of white node and gray nodes, color is gray
- If receiving copies consisting of gray nodes and black node
  - If minimum distance comes from black node, color is black
  - Otherwise, color is gray