CS 6423

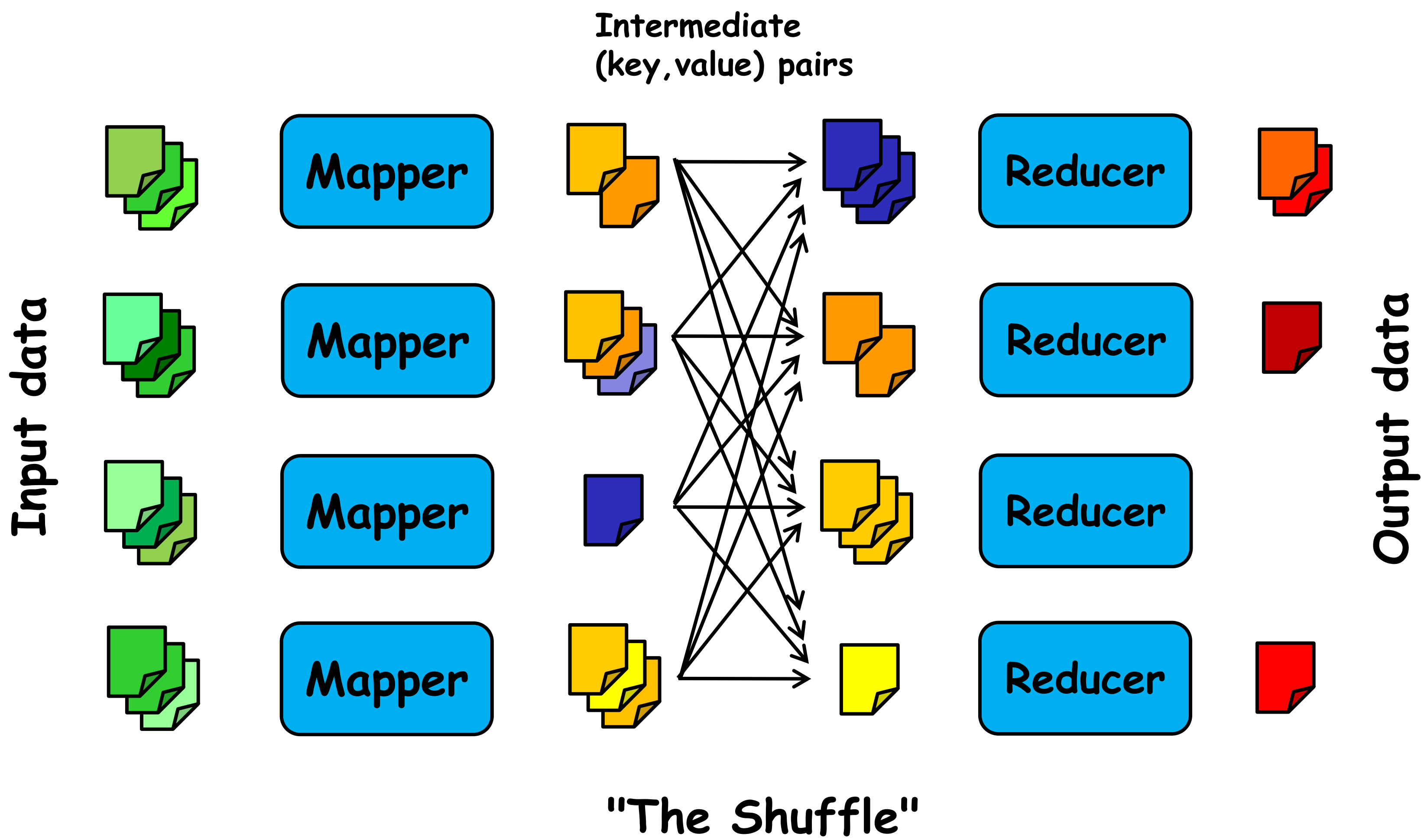
Scalable Computing for Big Data Analytics

Lecture 4: MapReduce: Graph Algorithhms

Prof. Gregory Provan
Department of Computer Science
University College Cork

Lecture adapted from:NETS 212: Scalable and Cloud Computing

Recap: MapReduce dataflow



What we have seen so far

- Initial algorithms
  - map/reduce model could be used to filter, collect, and aggregate data values
- Useful for data with limited structure
  - We could extract pieces of input data items and collect them to run various reduce operations
  - We could “join” two different data sets on a common key
- But that’s not enough…

Beyond average/sum/count

- Much of the world is a network of relationships and shared features
  - Members of a social network can be friends, and may have shared interests / memberships / etc.
  - Customers might view similar movies, and might even be clustered by interest groups
  - The Web consists of documents with links
  - Documents are also related by topics, words, authors, etc.

Goal: Develop a toolbox

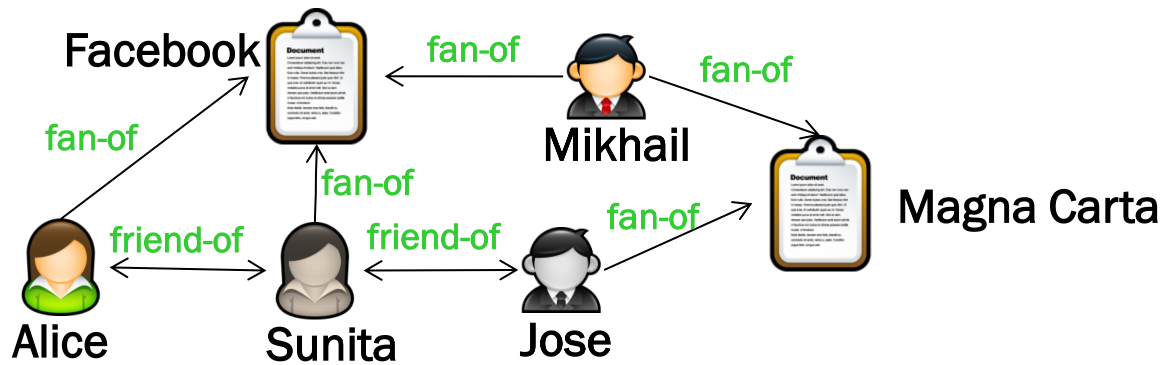
- We need a toolbox of algorithms useful for analyzing data that has both relationships and properties

- For the next ~2 lectures we'll start to build this toolbox
  - Compare the "traditional" and MapReduce solution

Plan for today

- Representing data in graphs NEXT
- Graph algorithms in MapReduce
  - Computation model
  - Iterative MapReduce
- A toolbox of algorithms
  - Single-source shortest path (SSSP)
  - k-means clustering
  - Classification with Naïve Bayes

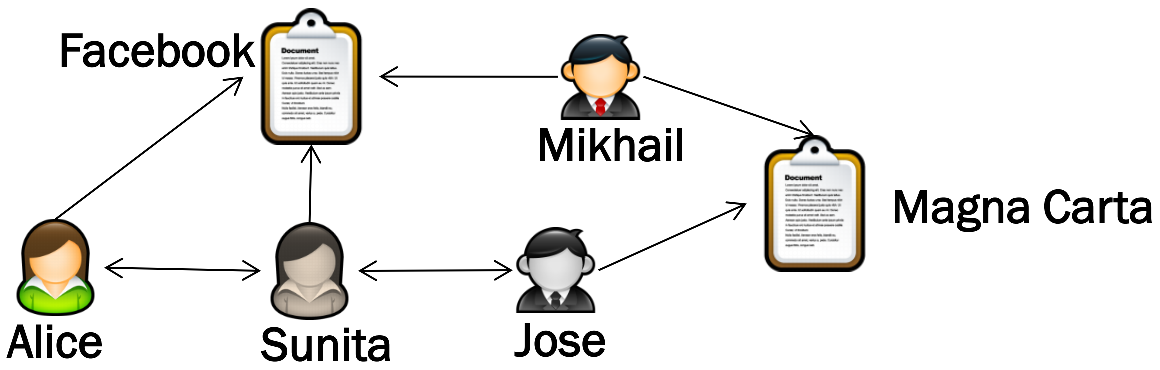
Thinking about related objects



- We can represent related objects as a labeled, directed graph
- Entities are typically represented as nodes; relationships are typically edges
  - Nodes all have IDs, and possibly other properties
  - Edges typically have values, possibly IDs and other properties

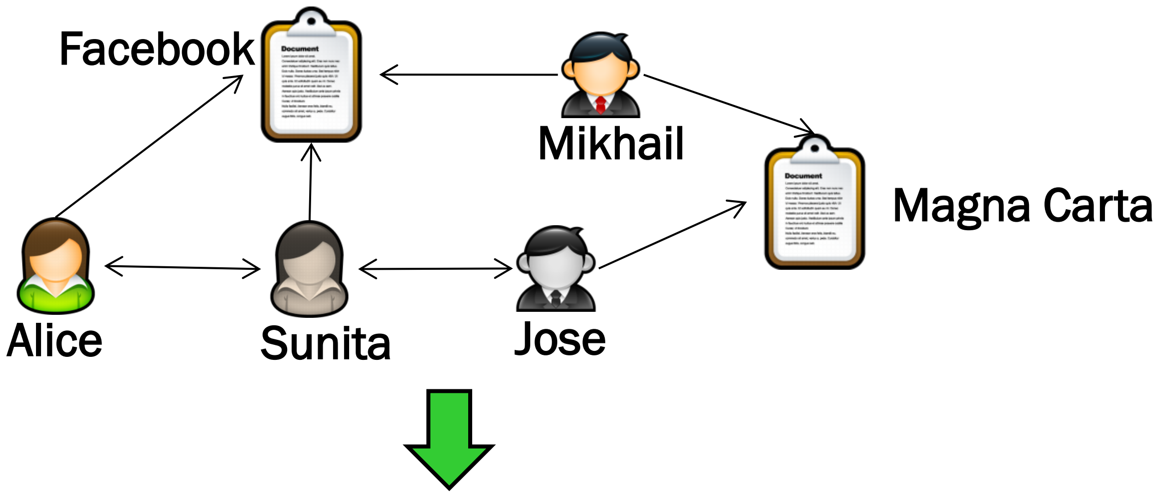
Images by Jojo Mendoza, Creative Commons licens

Encoding the data in a graph



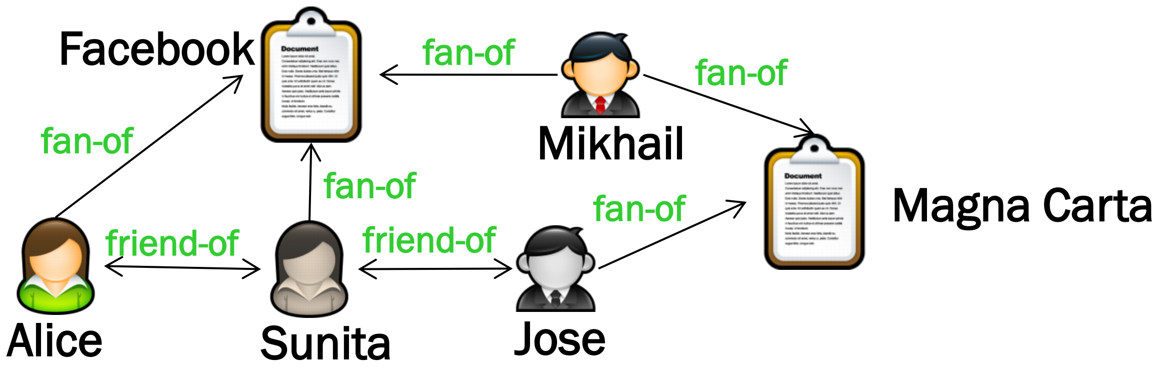
- Recall basic definition of a graph:
  - $G = (V, E)$ where V is vertices, E is edges of the form (v_1,v_2) where $v_1,v_2 \in V$
- Assume we only care about connected vertices
  - Then we can capture a graph simply as the edges
  - ... or as an adjacency list: v_i goes to $[v_j, v_{j+1}, \ldots]$

Graph encodings: Set of edges



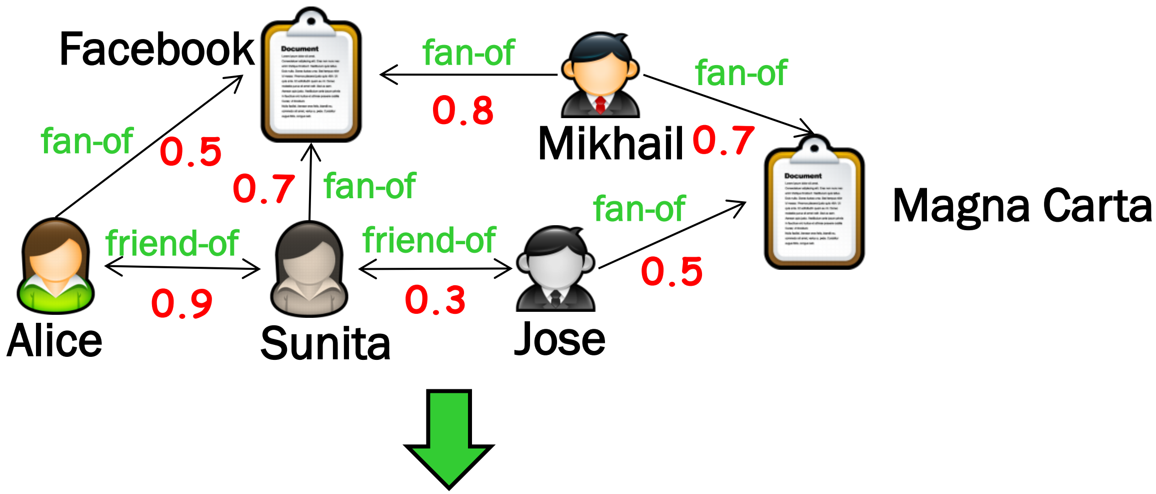
(Alice, Facebook)
(Alice, Sunita)
(Jose, Magna Carta)
(Jose, Sunita)
(Mikhail, Facebook)
(Mikhail, Magna Carta)
(Sunita, Facebook)
(Sunita, Alice)
(Sunita, Jose)

Graph encodings: Adding edge types



(Alice, fan-of, Facebook)
(Alice, friend-of, Sunita)
(Jose, fan-of, Magna Carta)
(Jose, friend-of, Sunita)
(Mikhail, fan-of, Facebook)
(Mikhail, fan-of, Magna Carta)
(Sunita, fan-of, Facebook)
(Sunita, friend-of, Alice)
(Sunita, friend-of, Jose)

Graph encodings: Adding weights



(Alice, fan-of, 0.5, Facebook)
(Alice, friend-of, 0.9, Sunita)
(Jose, fan-of, 0.5, Magna Carta)
(Jose, friend-of, 0.3, Sunita)
(Mikhail, fan-of, 0.8, Facebook)
(Mikhail, fan-of, 0.7, Magna Carta)
(Sunita, fan-of, 0.7, Facebook)
(Sunita, friend-of, 0.9, Alice)
(Sunita, friend-of, 0.3, Jose)

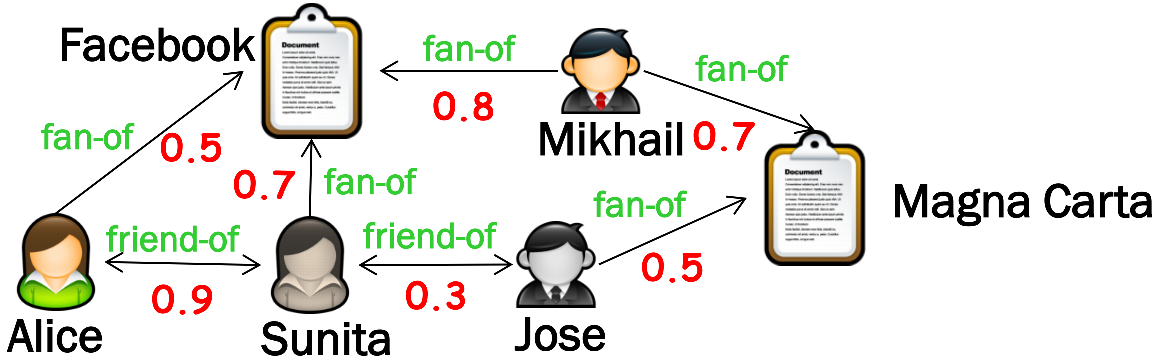
Recap: Related objects

- We can represent the relationships between related objects as a directed, labeled graph
  - Vertices represent the objects
  - Edges represent relationships
- We can annotate this graph in various ways
  - Add labels to edges to distinguish different types
  - Add weights to edges
  - ...
- We can encode the graph in various ways
  - Examples: Edge set, adjacency list

Plan for today

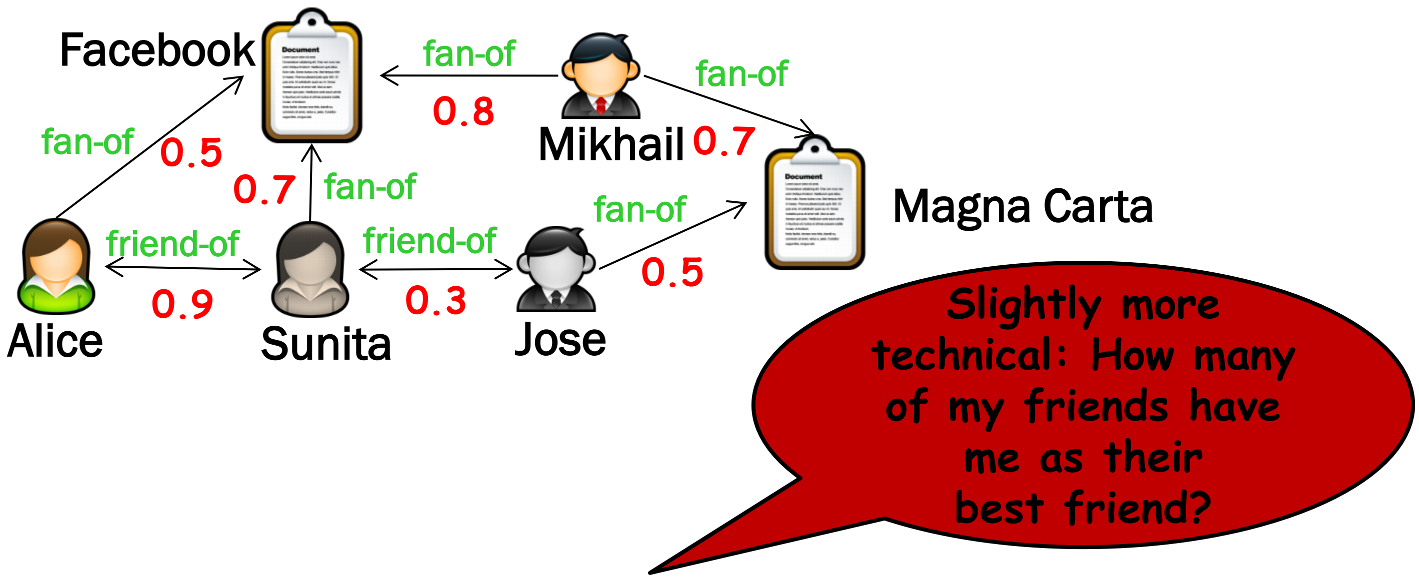
- Representing data in graphs
- Graph algorithms in MapReduce NEXT
  - Computation model
  - Iterative MapReduce
- A toolbox of algorithms
  - Single-source shortest path (SSSP)
  - k-means clustering
  - Classification with Naïve Bayes

A computation model for graphs



- Once the data is encoded in this way, we can perform various computations on it
  - Simple example: Which users are their friends' best friend?
  - More complicated examples (later): Page rank, adsorption, ...
- This is often done by
  - annotating the vertices with additional information, and
  - propagating the information along the edges
  - "Think like a vertex"!

A computation model for graphs



- Example: Am I my friends' best friend?

Can we do this in MapReduce?

```
map(key: node, value: [<otherNode, relType, strength>])
{


}
reduce(key: ________, values: list of ________)
{


}
```

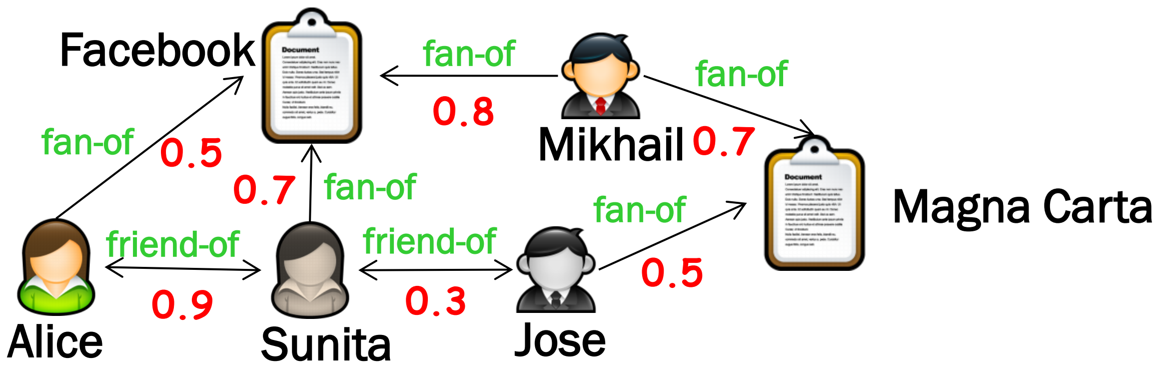
- Using adjacency list representation?

Can we do this in MapReduce?

```
map(key: node, value: <otherNode, relType, strength>)
{


}
reduce(key: ________, values: list of ________)
{


}
```

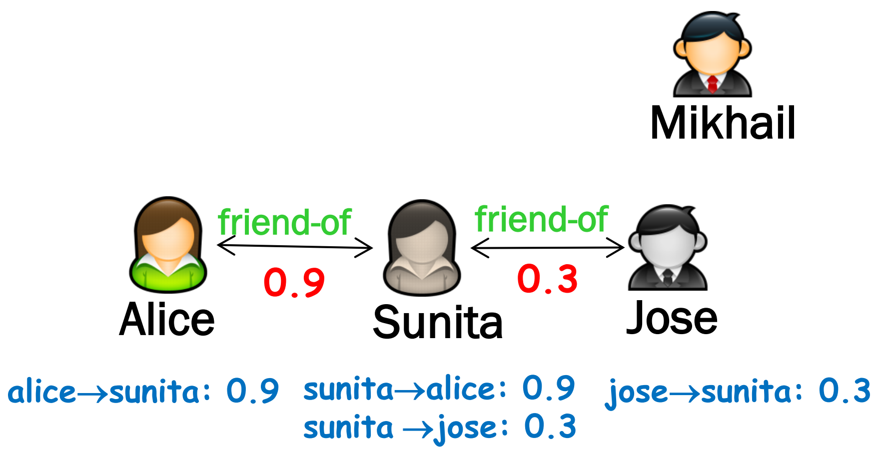
- Using single-edge data representation?

A computation model for graphs



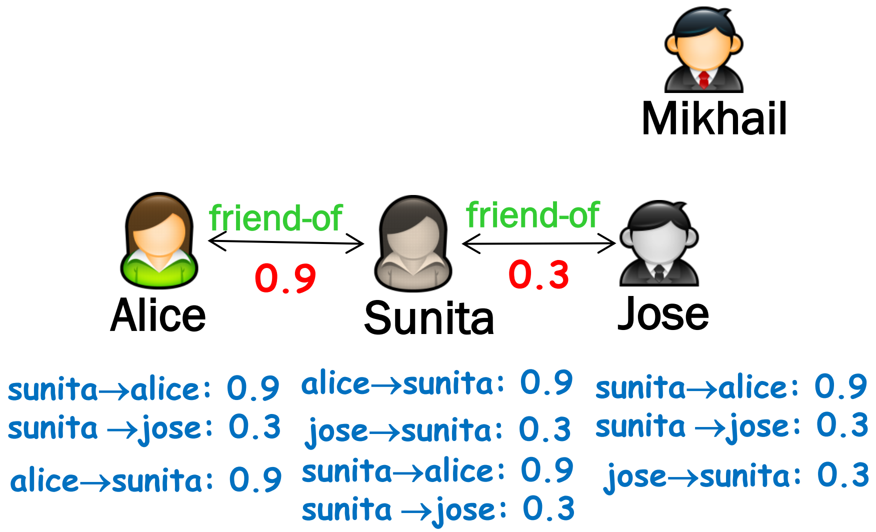
- Example: Am I my friends' best friend?
  - Step #1: Discard irrelevant vertices and edges

A computation model for graphs



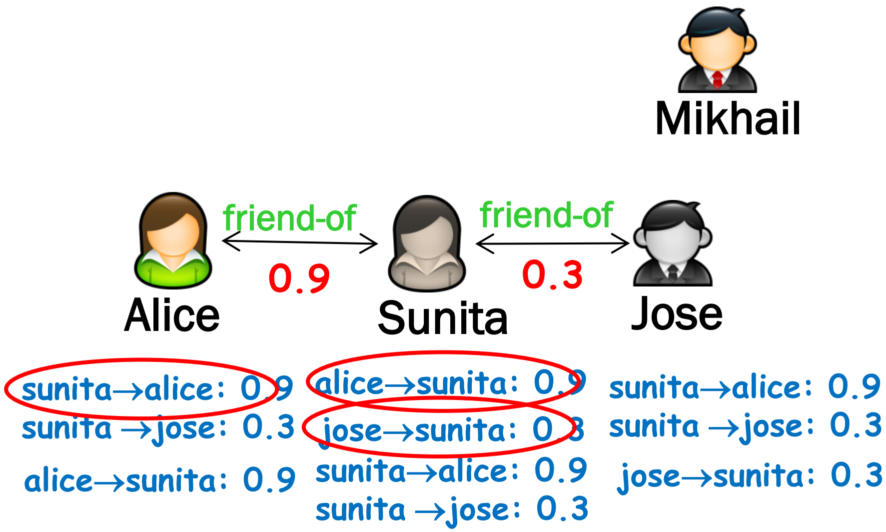
- Example: Am I my friends' best friend?
  - Step #1: Discard irrelevant vertices and edges
  - Step #2: Annotate each vertex with list of friends
  - Step #3: Push annotations along each edge

A computation model for graphs



- Example: Am I my friends' best friend?
  - Step #1: Discard irrelevant vertices and edges
  - Step #2: Annotate each vertex with list of friends
  - Step #3: Push annotations along each edge

A computation model for graphs



- Example: Am I my friends' best friend?
  - Step #1: Discard irrelevant vertices and edges
  - Step #2: Annotate each vertex with list of friends
  - Step #3: Push annotations along each edge
  - Step #4: Determine result at each vertex

A real-world use case

- A variant that is actually used in social networks today: "Who are the friends of multiple of my friends?"
  - Where have you seen this before?

- Friend recommendation!
  - Maybe these people should be my friends too!

Generalizing...

- Now suppose we want to go beyond direct friend relationships
  - Example: How many of my friends' friends (distance-2 neighbors) have me as their best friend's best friend?
  - What do we need to do?

- How about distance k>2?

- To compute the answer, we need to run multiple iterations of MapReduce!

Iterative MapReduce

- The basic model:

```
copy files from input dir → staging dir 1
(optional: do some preprocessing)

while (!terminating condition) {
  map from staging dir 1
  reduce into staging dir 2
  move files from staging dir 2 → staging dir1
}

(optional: postprocessing)
move files from staging dir 2 → output dir
```

- Note that reduce output must be compatible with the map input!
  - What can happen if we filter out some information in the mapper or in the reducer?

Graph algorithms and MapReduce

- A centralized algorithm typically traverses a tree or a graph one item at a time (there's only one "cursor")
  - You've learned breadth-first and depth-first traversals
- Most algorithms that are based on graphs make use of multiple map/reduce stages processing one "wave" at a time
  - Sometimes iterative MapReduce, other times chains of map/reduce

"Think like a vertex"

- Let's think about a different model for a bit:
  - Suppose we had a network that has exactly the same topology as the graph, with one node for each vertex
  - Suppose each vertex A has some local state s_A

    (A,s_A) tuple in the input file
  - The computation proceeds in rounds. In each round:

    MapReduce rounds
    - Step #1: Each vertex A reads its local state s_A

      map(A,s_A) invocation
    - Step #2: A can then send some messages mi to adjacent nodes B_i

      map() emits a (B_i,m_i) tuples
    - Step #3: Then each vertex A looks at all the messages it has received in step #2

      reduce($A,\{m_1,m_2,...,m_k\}$) invocation
    - Step #4: Finally, each vertex can update its local state to some other value s_A' if it wants to

      reduce() emits an (A,s_A')
  - This would be a natural fit for many graph algorithms!

Recap: MapReduce on graphs

- Suppose we want to:
  - compute a function for each vertex in a graph...
  - ... using data from vertices at most k hops away
- We can do this as follows:
  - "Push" information along the edges
    - "Think like a vertex"
  - Finally, perform the computation at each vertex
- May need more than one MapReduce phase
  - Iterative MapReduce: Outputs of stage i $\rightarrow$ inputs of stage i+1

Plan for today

- Representing data in graphs
- Graph algorithms in MapReduce
  - Computation model
  - Iterative MapReduce
- A toolbox of algorithms   NEXT
  - Single-source shortest path (SSSP)
  - k-means clustering
  - Classification with Naïve Bayes

Path-based algorithms

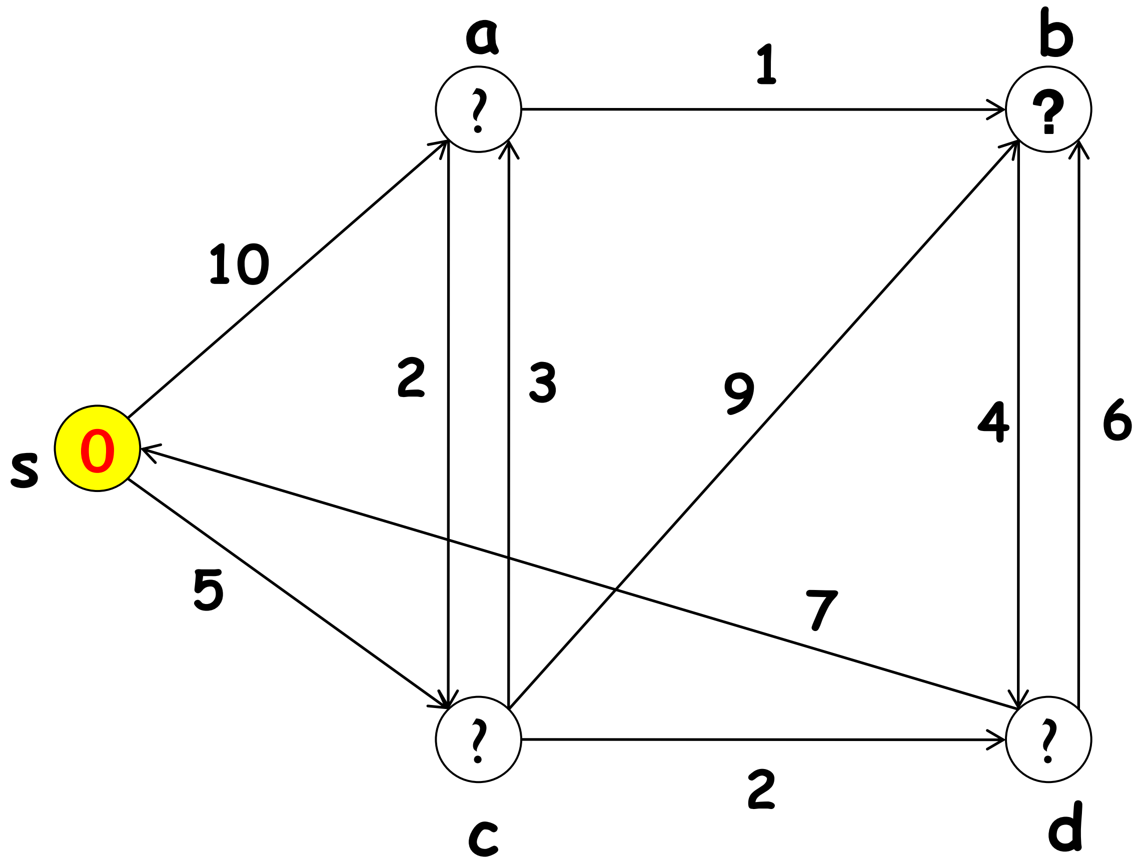
- Sometimes our goal is to compute information about the paths (sets of paths) between nodes
  - Edges may be annotated with cost, distance, or similarity

- Examples of such problems:
  - Shortest path from one node to another
  - Minimum spanning tree (minimal-cost tree connecting all vertices in a graph)
  - Steiner tree (minimal-cost tree connecting certain nodes)
  - Topological sort (node in a DAG comes before all nodes it points to)

Single-Source Shortest Path (SSSP)

Given a directed graph G = (V, E) in which each edge e has a cost c(e):

- **Compute the cost of reaching each node from the source node s in the most efficient way (potentially after multiple 'hops')**



SSSP: Intuition

- We can formulate the problem using induction
  - The shortest path follows the principle of optimality: the last step (u,v) makes use of the shortest path to u

- We can express this as follows:

```
bestDistanceAndPath(v) {
  if (v == source) then {
    return <distance 0, path [v]>
  } else {
    find argmin_u (bestDistanceAndPath[u] + dist[u,v])
    return <bestDistanceAndPath[u] + dist[u,v], path[u] + v>
  }
}
```

SSSP: traditional solution

- Traditional approach: Dijkstra's algorithm

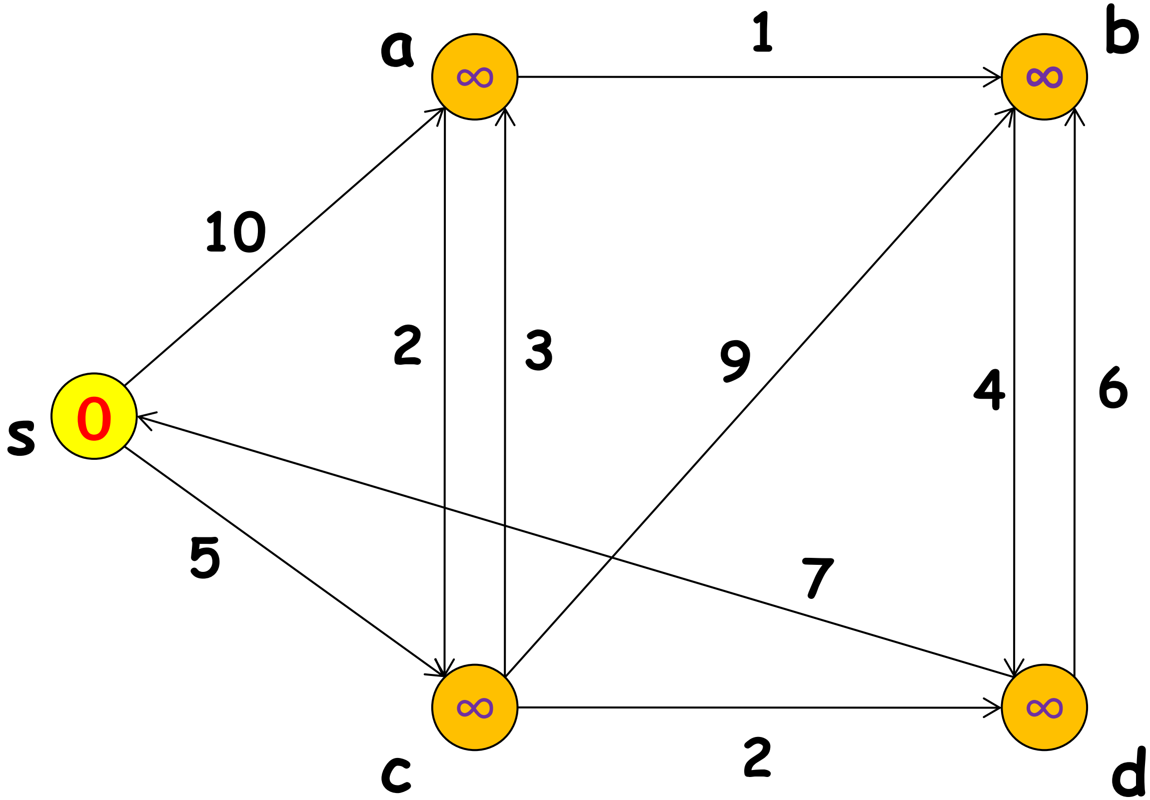
```
V: vertices, E: edges, S: start node

foreach v in V
  dist_S_to[v] := infinity
  predecessor[v] = nil
spSet = {}
Q := V
while (Q not empty) do
  u := Q.removeNodeClosestTo(S)
  spSet := spSet + {u}
  foreach v in V where (u,v) in E
    if (dist_S_To[v] > dist_S_To[u]+cost(u,v)) then
      dist_S_To[v] = dist_S_To[u] + cost(u,v)
    predecessor[v] = u
```

Initialize length and last step of path to default values

Update length and path based on edges radiating from u

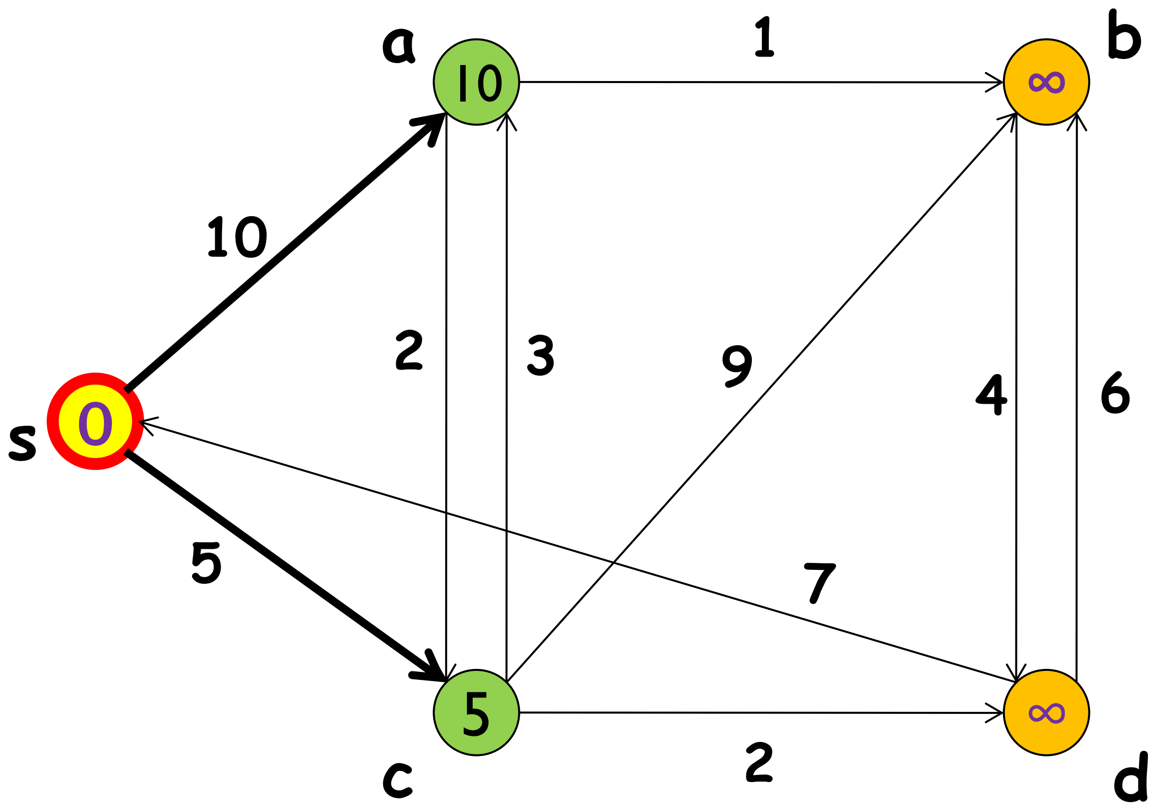
SSSP: Dijkstra in Action

Example from CLR 2nd ed. p. 528



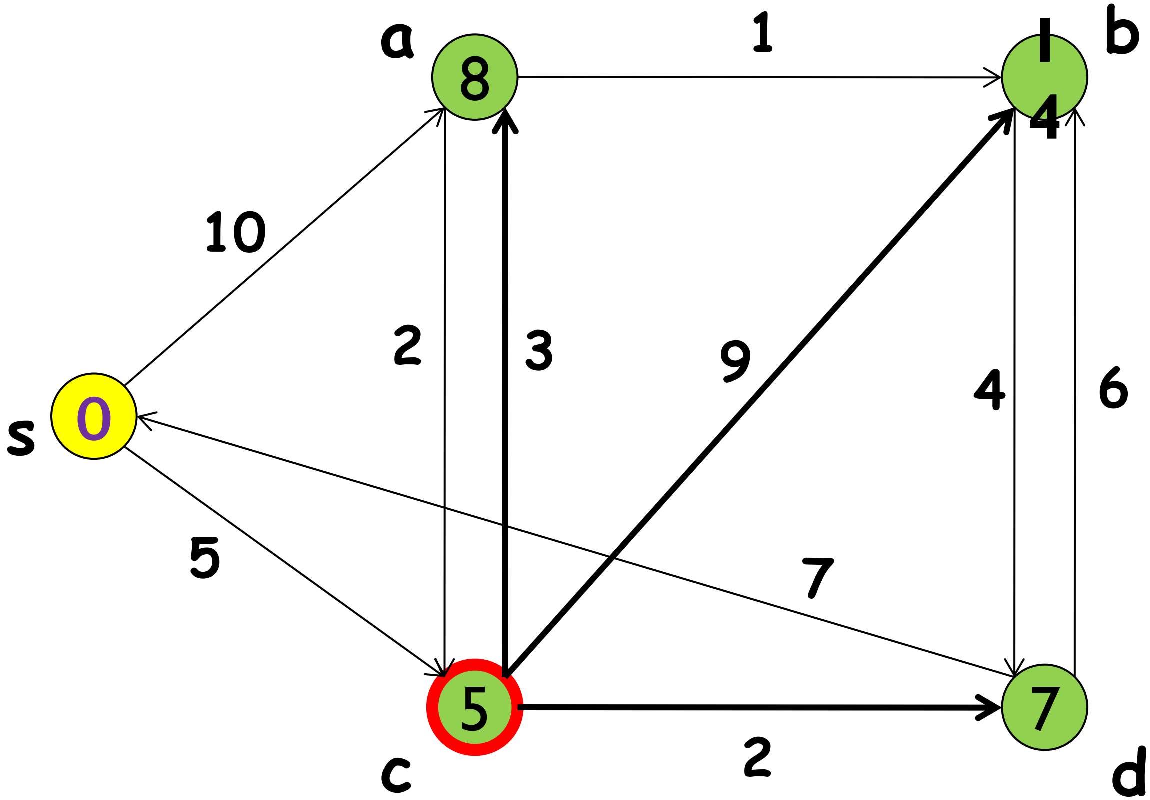
Q = {s,a,b,c,d} spSet = {}
dist_S_To: {(a,∞), (b,∞), (c,∞), (d,∞)}
predecessor: {(a,nil), (b,nil), (c,nil), (d,nil)}

SSSP: Dijkstra in Action



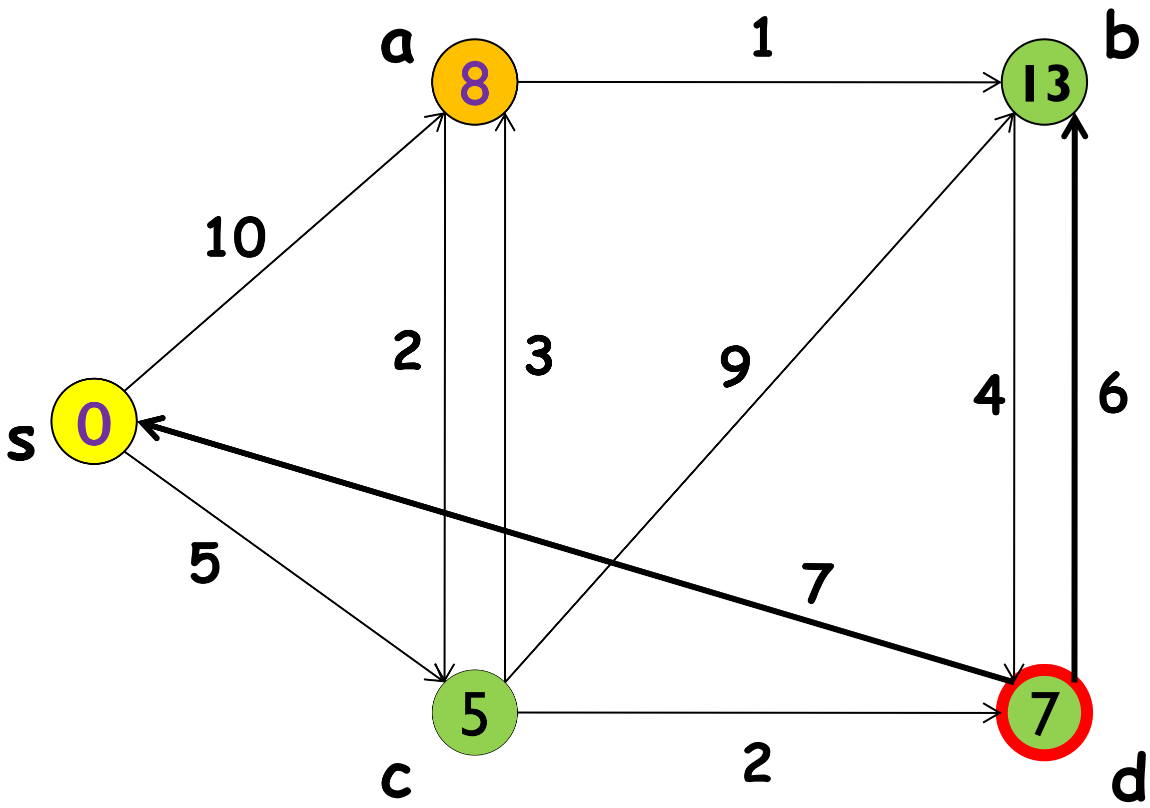
Q = {a,b,c,d} spSet = {s}
dist_S_To: {(a,10), (b,∞), (c,5), (d,∞)}
predecessor: {(a,s), (b,nil), (c,s), (d,nil)}

SSSP: Dijkstra in Action



Q = {a,b,d} spSet = {c,s}
dist_S_To: {(a,8), (b,14), (c,5), (d,7)}
predecessor: {(a,c), (b,c), (c,s), (d,c)}

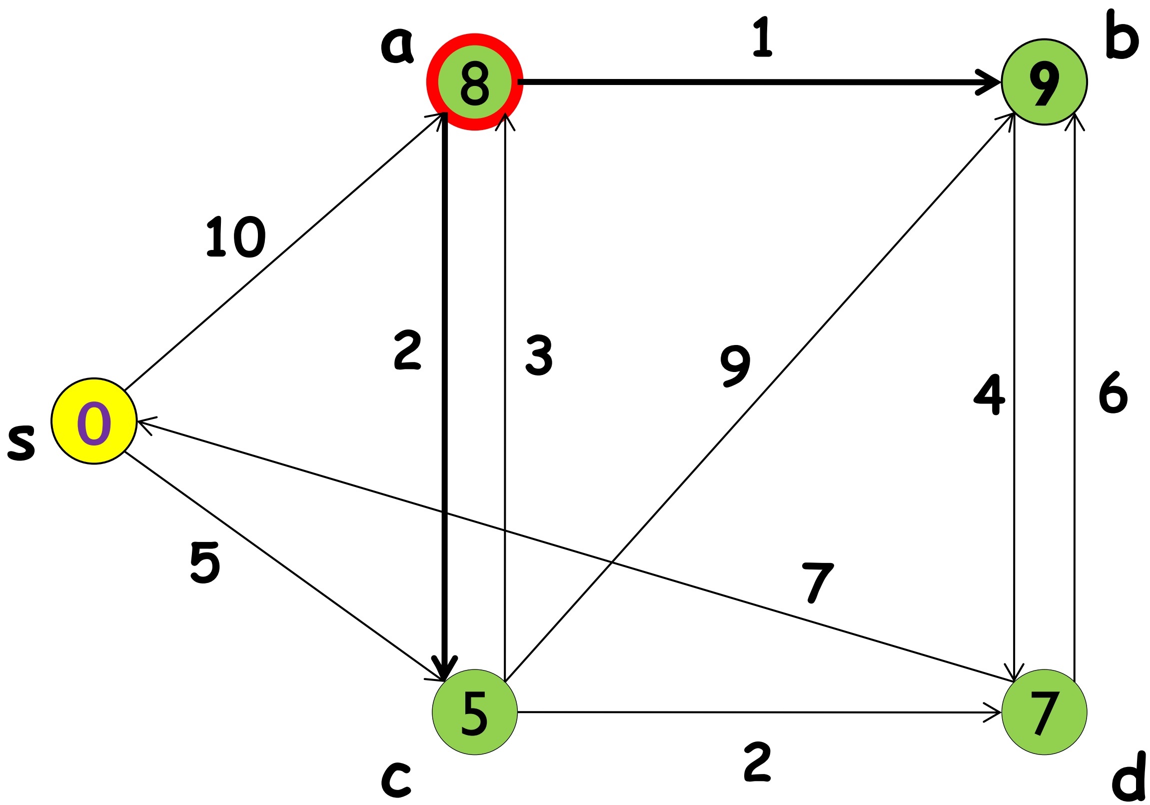
SSSP: Dijkstra in Action



Q = {a,b}　　　　　　spSet = {c,d,s}
dist_S_To: {(a,8), (b,13), (c,5), (d,7)}
predecessor: {(a,c), (b,d), (c,s), (d,c)}

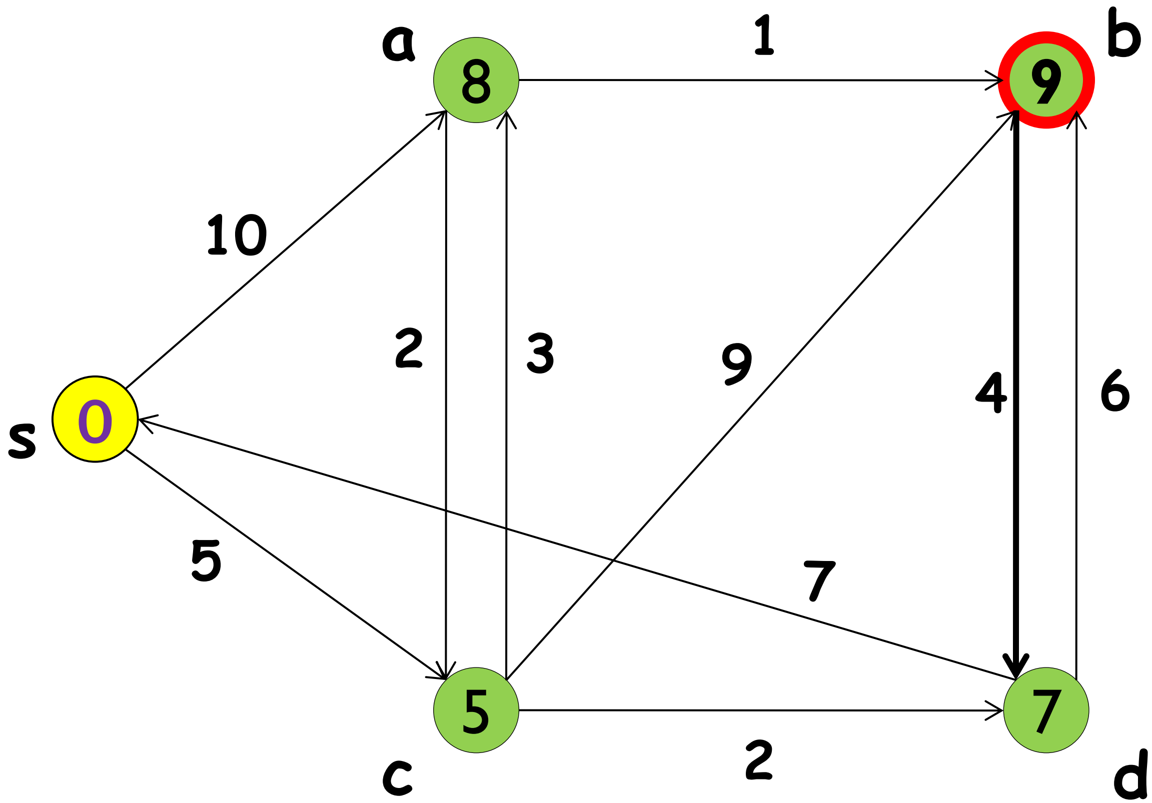
SSSP: Dijkstra in Action

Example from CLR 2nd ed. p. 528



Q = {b} spSet = {a,c,d,s}
dist_S_To: {(a,8), (b,9), (c,5), (d,7)}
predecessor: {(a,c), (b,a), (c,s), (d,c)}

SSSP: Dijkstra in Action

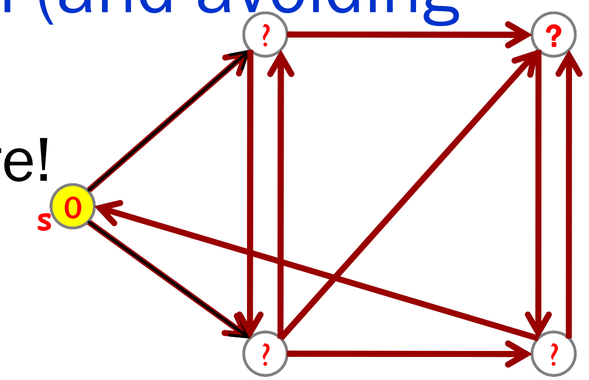


Q = {} spSet = {a,b,c,d,s}
dist_S_To: {(a,8), (b,9), (c,5), (d,7)}
predecessor: {(a,c), (b,a), (c,s), (d,c)}

Example from CLR 2nd ed. p. 528

SSSP: How to parallelize?

- Dijkstra traverses the graph along a single route at a time, prioritizing its traversal to the next step based on total path length (and avoiding cycles)
  - No real parallelism to be had here!



- Intuitively, we want something that “radiates” from the origin, one “edge hop distance” at a time
  - Each step outwards can be done in parallel, before another iteration occurs - or we are done
  - Recall our earlier discussion: Scalability depends on the algorithm, not (just) on the problem!

SSSP: Revisiting the inductive definition

```
bestDistanceAndPath(v) {
  if (v == source) then {
    return <distance 0, path [v]>
  } else {
    find argmin_u (bestDistanceAndPath[u] + dist[u,v])
    return <bestDistanceAndPath[u] + dist[u,v], path[u] + v>
  }
}
```

- Dijkstra's algorithm carefully considered each u in a way that allowed us to prune certain points
- Instead we can look at all potential u's for each v
  - Compute iteratively, by keeping a "frontier set" of u nodes i edge-hops from the source

SSSP: MapReduce formulation

The shortest path we have found so far.. from the source to nodeID has length ∞... this is the next... hop on that path... and here is the adjacency list for nodeID

- init:
  - For each node, node ID → <∞, -, {<succ-node-ID,edge-cost>}>
- map:
  - take node ID → <dist, next, {<succ-node-ID,edge-cost>}>
  - For each succ-node-ID:
    - emit succ-node ID → {<node ID, distance+edge-cost>}
  - emit node ID → distance,{<succ-node-ID,edge-cost>}

This is a new path from the source to succ-node-ID that we just discovered (not necessarily shortest)

- reduce:
  - distance := min cost from a predecessor; next := that predec.
  - emit node ID → <distance, next, {<succ-node-ID,edge-cost>}>

Why is this necessary?

- Repeat until no changes
- Postprocessing: Remove adjacency lists

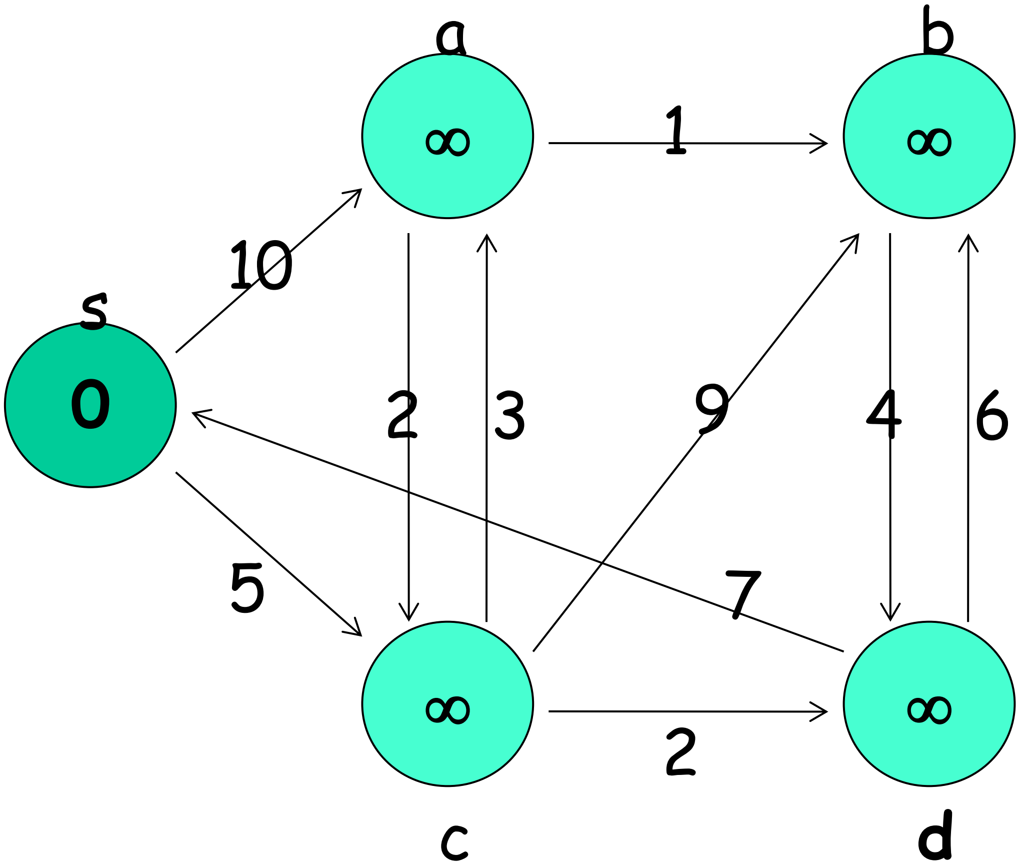
Example: SSSP – Parallel BFS in MapReduce

- Adjacency matrix

	s	a	b	c	d
s		10		5	
a			1	2	
b					4
c		3	9		2
d	7		6		

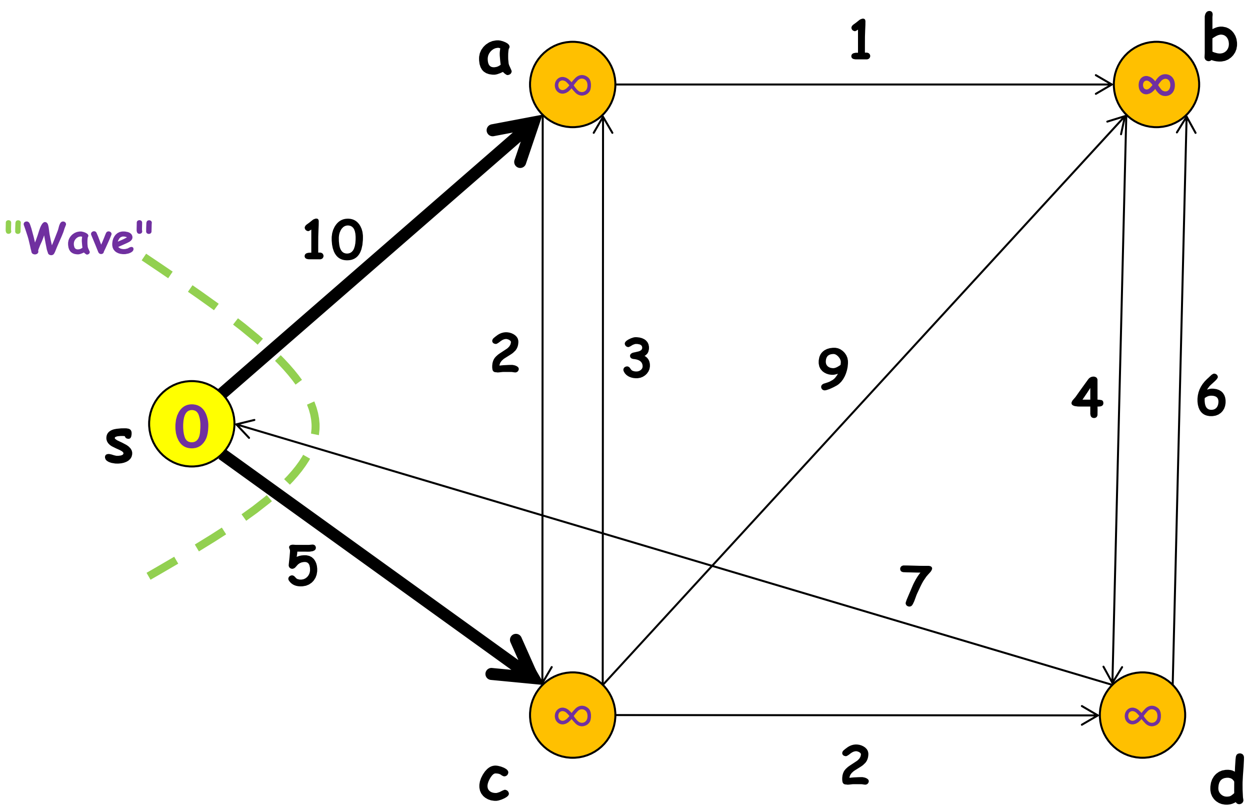
- Adjacency List

s: (a, 10), (c, 5)
a: (b, 1), (c, 2)
b: (d, 4)
c: (a, 3), (b, 9), (d, 2)
d: (s, 7), (b, 6)



Iteration 0: Base case

mapper:	(a,<s,10>) (c,<s,5>) edges
reducer:	(a,<10, ...>) (c,<5, ...>)



Iteration 0– Parallel BFS in MapReduce

- Map input: <node ID, <dist, adj list>>

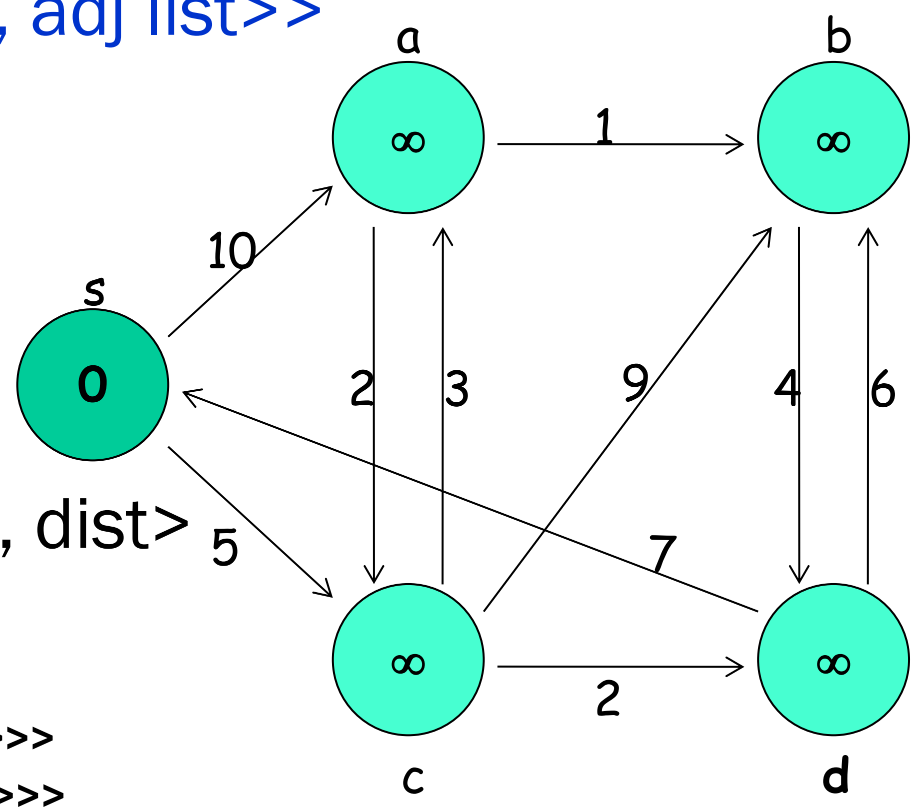
<s, <0, <(a, 10), (c, 5)>>>

<a, <inf, <(b, 1), (c, 2)>>>

<b, <inf, <(d, 4)>>>

<c, <inf, <(a, 3), (b, 9), (d, 2)>>>

<d, <inf, <(s, 7), (b, 6)>>>



- Map output: <dest node ID, dist>

<a, 10> <c, 5>
<b, inf> <c, inf>
<d, inf>
<a, inf> <b, inf> <d, inf>
<s, inf> <b, inf>

<s, <0, <(a, 10), (c, 5)>>>
<a, <inf, <(b, 1), (c, 2)>>>
<b, <inf, <(d, 4)>>>
<c, <inf, <(a, 3), (b, 9), (d, 2)>>>
<d, <inf, <(s, 7), (b, 6)>>>

Iteration 0 – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

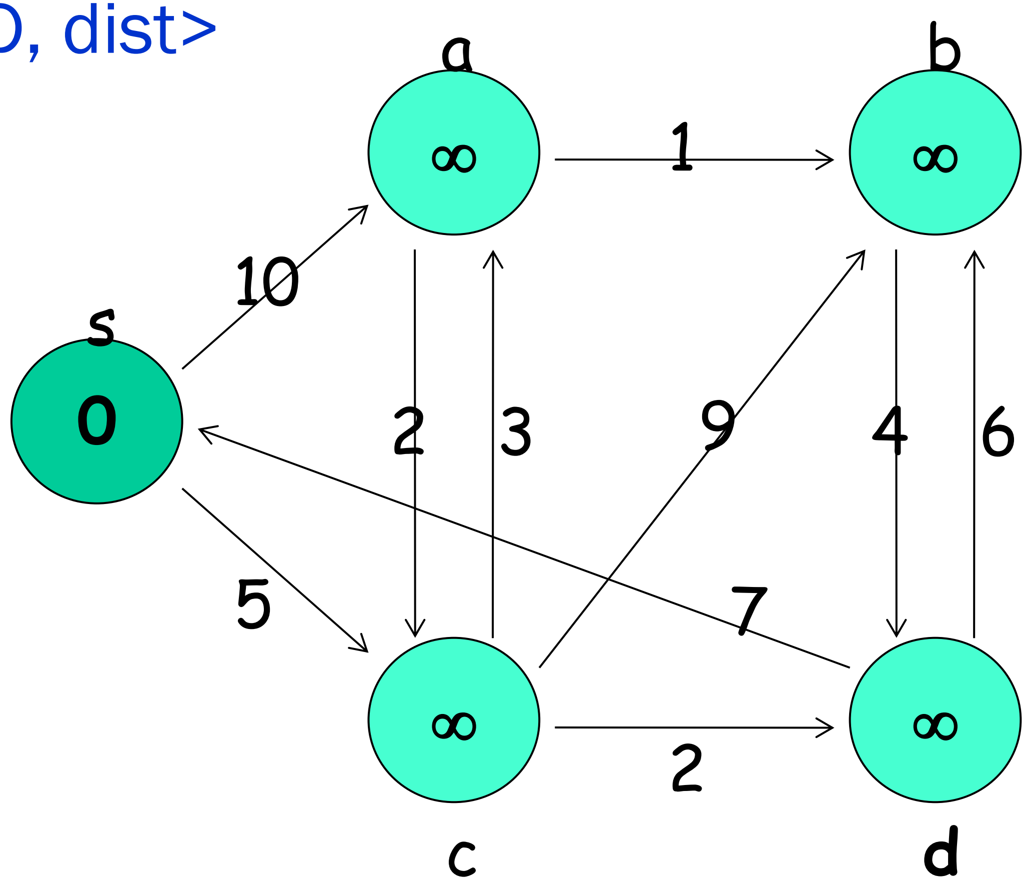
<s, <0, <(a, 10), (c, 5)>>>
<s, inf>

<a, <inf, <(b, 1), (c, 2)>>>
<a, 10> <a, inf>

<b, <inf, <(d, 4)>>>
<b, inf> <b, inf> <b, inf>

<c, <inf, <(a, 3), (b, 9), (d, 2)>>>
<c, 5> <c, inf>

<d, <inf, <(s, 7), (b, 6)>>>
<d, inf> <d, inf>



Iteration 0– Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<s, <0, <(a, 10), (c, 5)>>>

~~<s, inf>~~

<a, <~~inf~~, <(b, 1), (c, 2)>>>

<a, 10> ~~<a, inf>~~

<b, <inf, <(d, 4)>>>

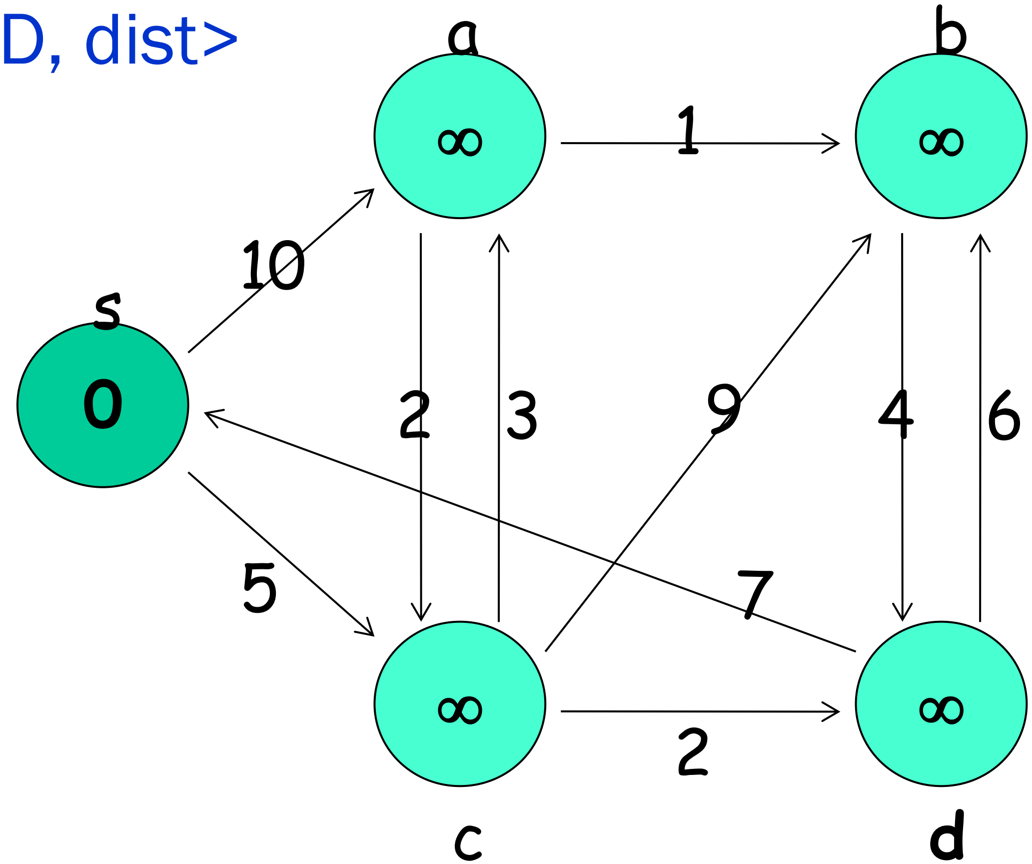
~~<b, inf> <b, inf> <b, inf>~~

<c, <~~inf~~, <(a, 3), (b, 9), (d, 2)>>>

<c, 5> ~~<c, inf>~~

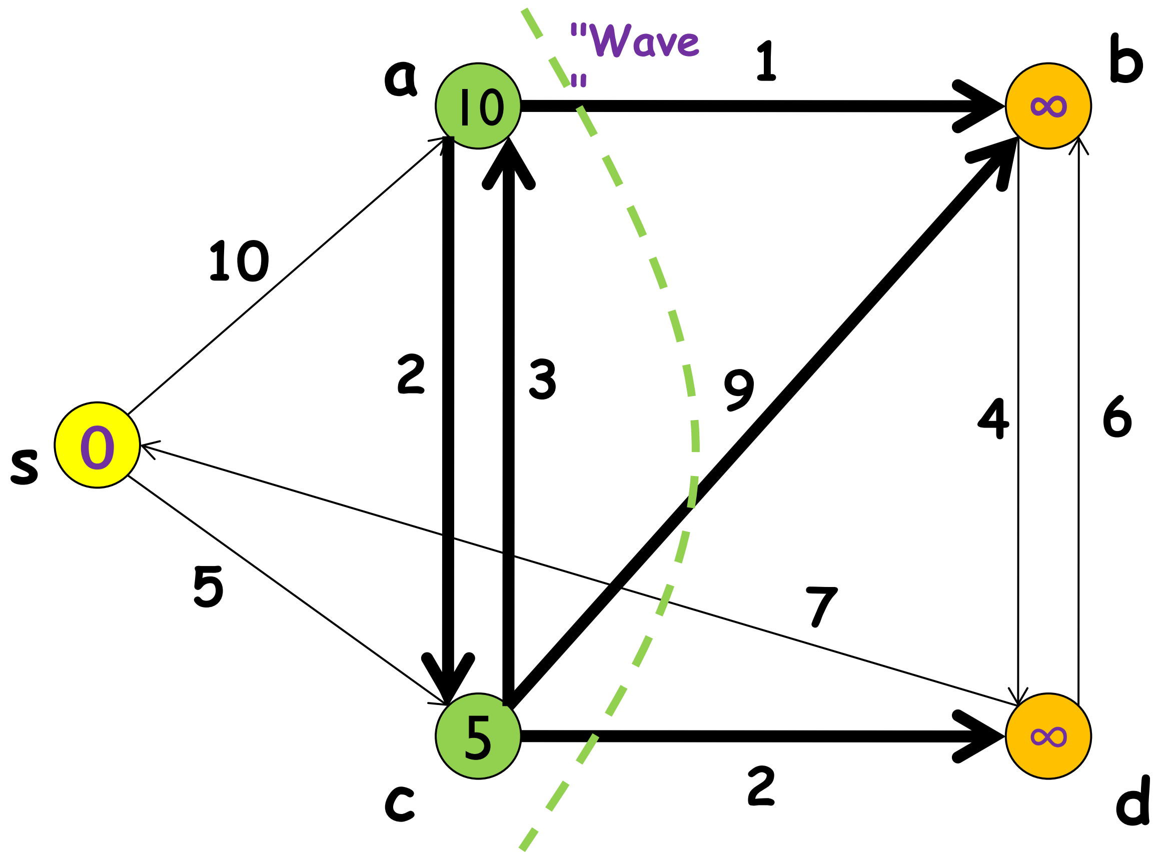
<d, <inf, <(s, 7), (b, 6)>>>

~~<d, inf> <d, inf>~~



Iteration 1

mapper:	(a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) **(b,<a,11>)** (b,<c,14>) (d,<c,7>) *edges*
reducer:	(a,<8, ...>) (c,<5, ...>) (b,<11, ...>) (d,<7, ...>)



Iteration 1– Parallel BFS in MapReduce

- Reduce output: <node ID, <dist, adj list>>

  = Map input for next iteration

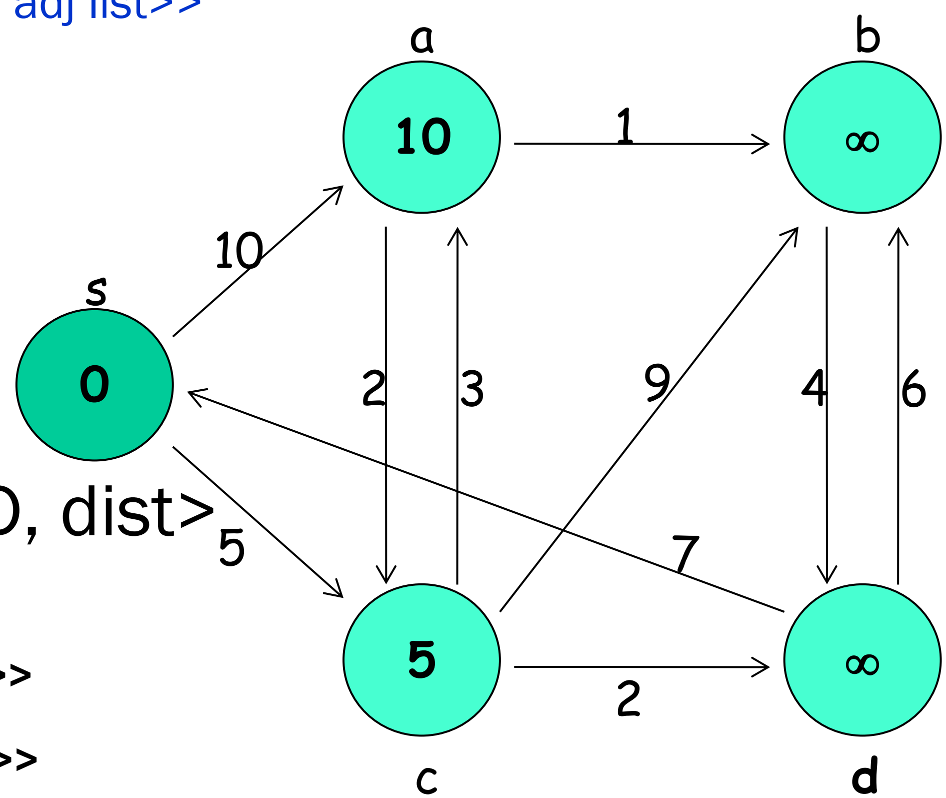
<s, <0, <(a, 10), (c, 5)>>>

<a, <10, <(b, 1), (c, 2)>>>

<b, <inf, <(d, 4)>>>

<c, <5, <(a, 3), (b, 9), (d, 2)>>>

<d, <inf, <(s, 7), (b, 6)>>>



- Map output: <dest node ID, dist>

<a, 10> <c, 5>

<b, 11> <c, 12>

<d, inf>

<a, 8> <b, 14> <d, 7>

<s, inf> <b, inf>

<s, <0, <(a, 10), (c, 5)>>>

<a, <10, <(b, 1), (c, 2)>>>

<b, <inf, <(d, 4)>>>

<c, <5, <(a, 3), (b, 9), (d, 2)>>>

<d, <inf, <(s, 7), (b, 6)>>>

Iteration 1 – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<s, <0, <(a, 10), (c, 5)>>>

<s, inf>

<a, <10, <(b, 1), (c, 2)>>>

<a, 10> <a, 8>

<b, <inf, <(d, 4)>>>

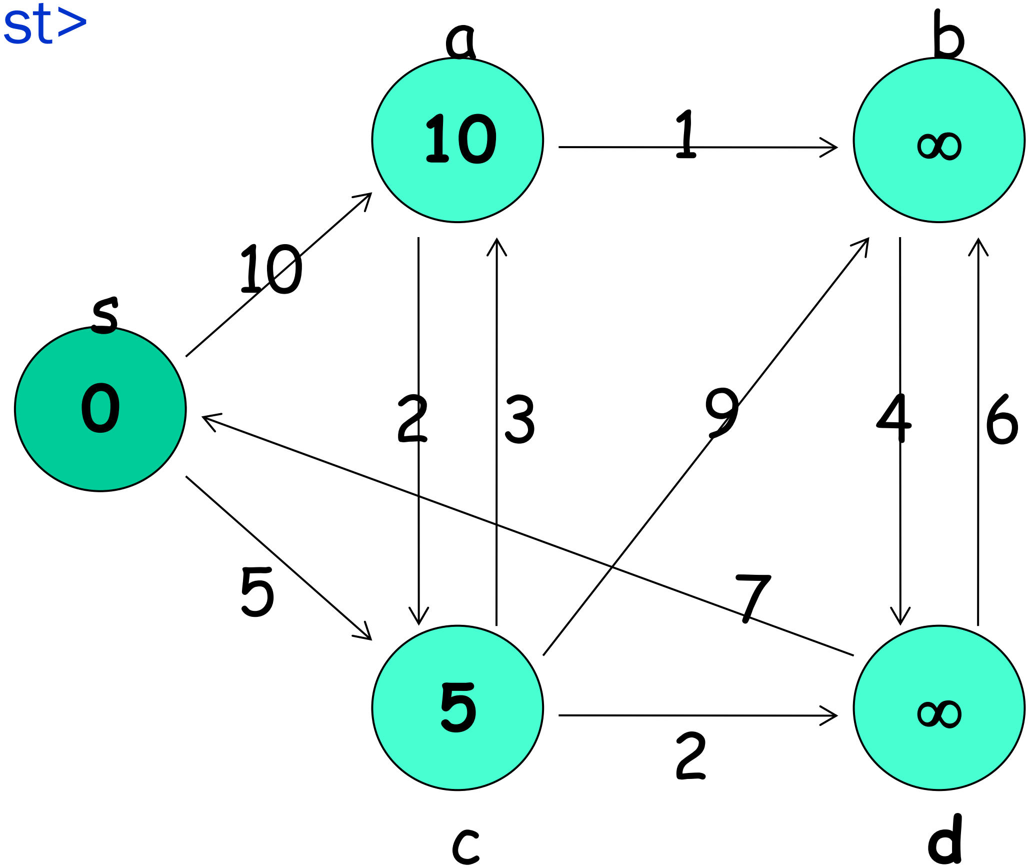
<b, 11> <b, 14> <b, inf>

<c, <5, <(a, 3), (b, 9), (d, 2)>>>

<c, 5> <c, 12>

<d, <inf, <(s, 7), (b, 6)>>>

<d, inf> <d, 7>



Iteration 1– Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<s, <0, <(a, 10), (c, 5)>>>

~~<s, inf>~~

<a, <~~10~~, <(b, 1), (c, 2)>>>

~~<a, 10>~~ <a, 8>

<b, <~~inf~~, <(d, 4)>>>

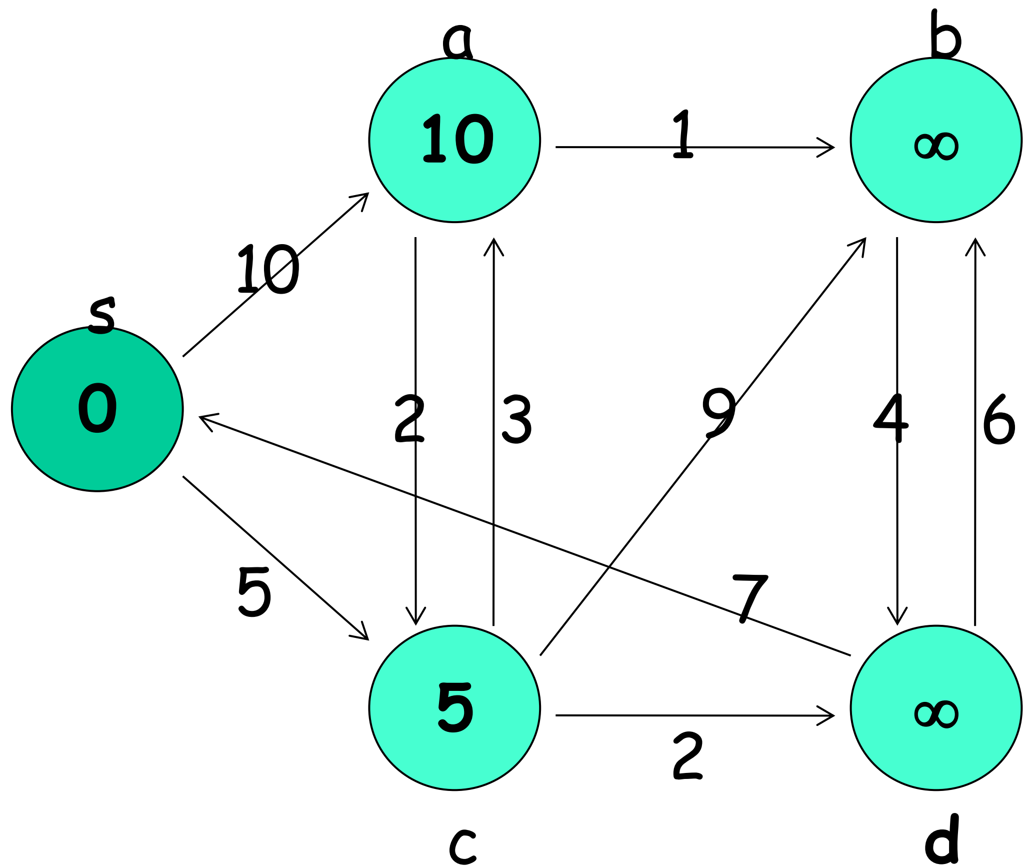
<b, 11> ~~<b, 14> <b, inf>~~

<c, <5, <(a, 3), (b, 9), (d, 2)>>>

~~<c, 5> <c, 12>~~

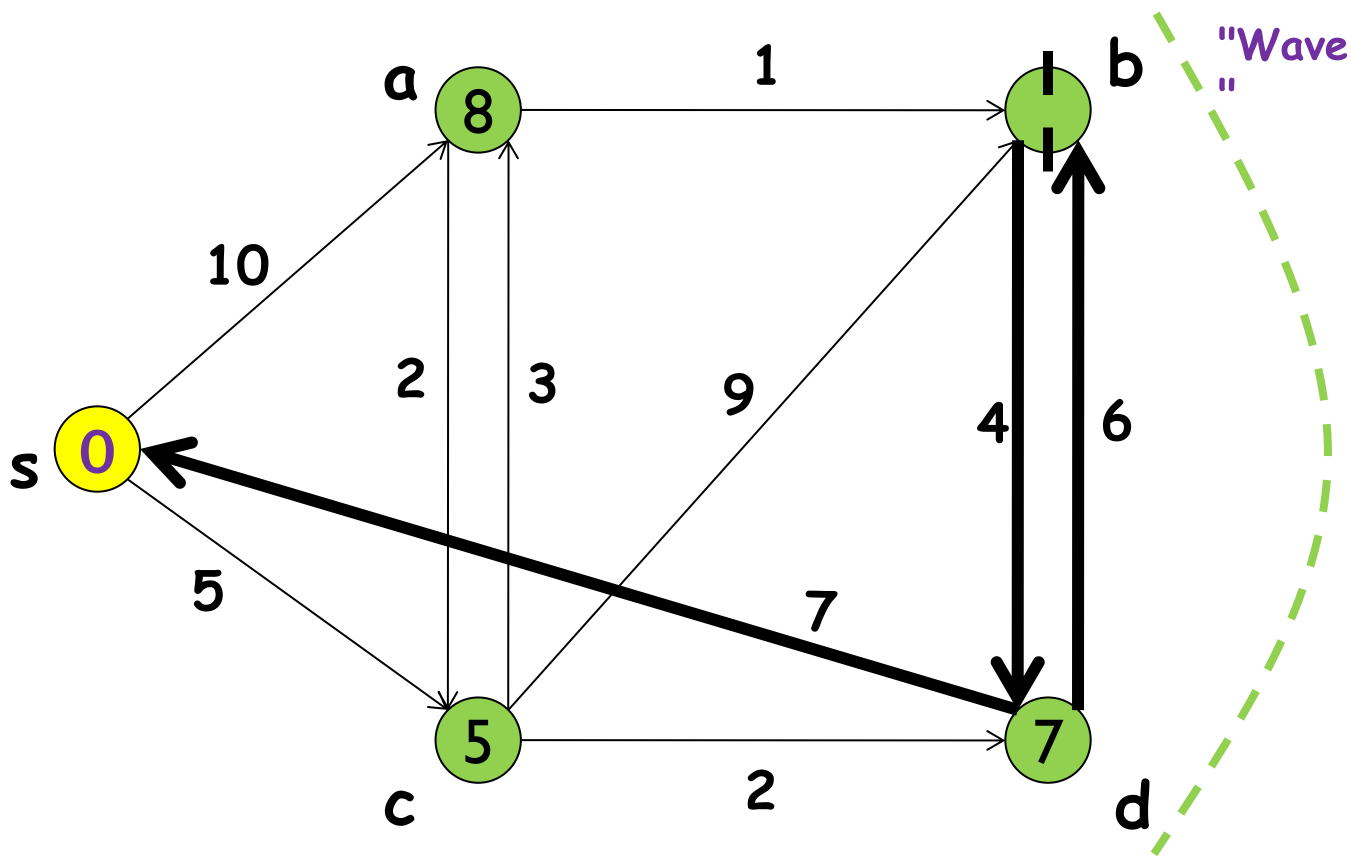
<d, <~~inf~~, <(s, 7), (b, 6)>>>

~~<d, inf>~~ <d, 7>



Iteration 2

mapper:	(a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) **(b,<a,11>)** (b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges
reducer:	(a,<8>) (c,<5>) (b,<11>) (d,<7>)



Iteration 2– Parallel BFS in MapReduce

- Reduce output: <node ID, <dist, adj list>>
  = Map input for next iteration

<s, <0, <(a, 10), (c, 5)>>>

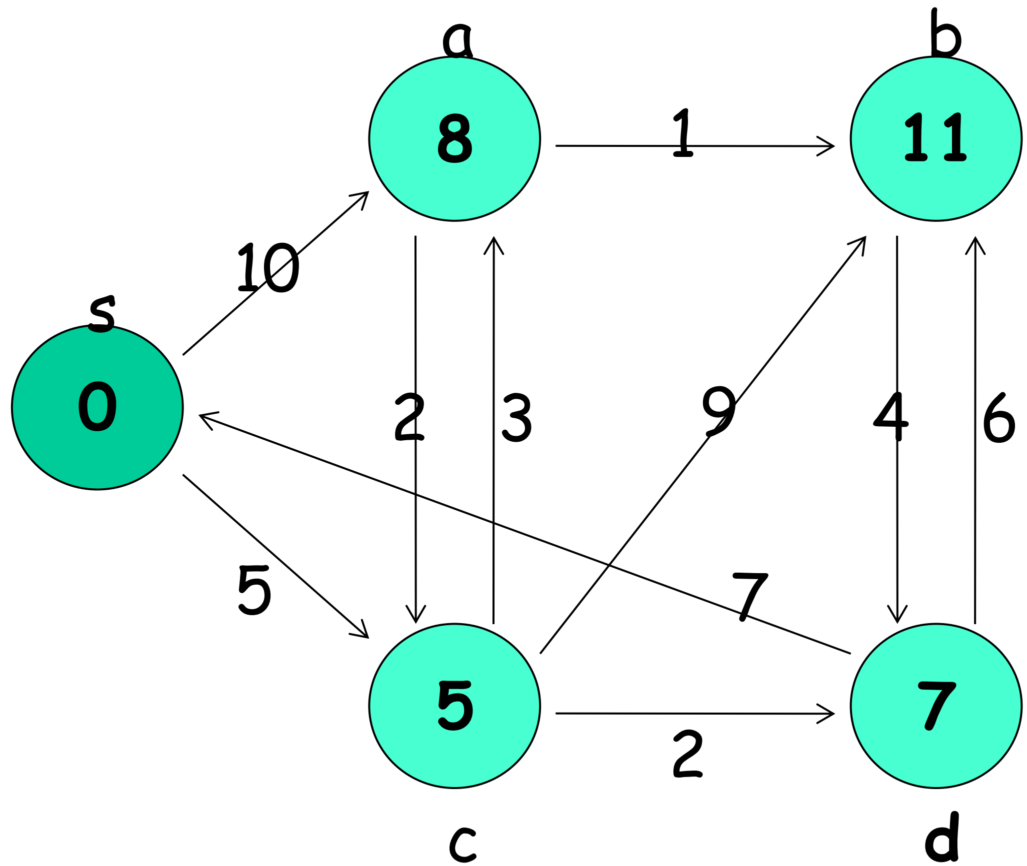
<a, <8, <(b, 1), (c, 2)>>>

<b, <11, <(d, 4)>>>

<c, <5, <(a, 3), (b, 9), (d, 2)>>>

<d, <7, <(s, 7), (b, 6)>>>

... the rest omitted ...

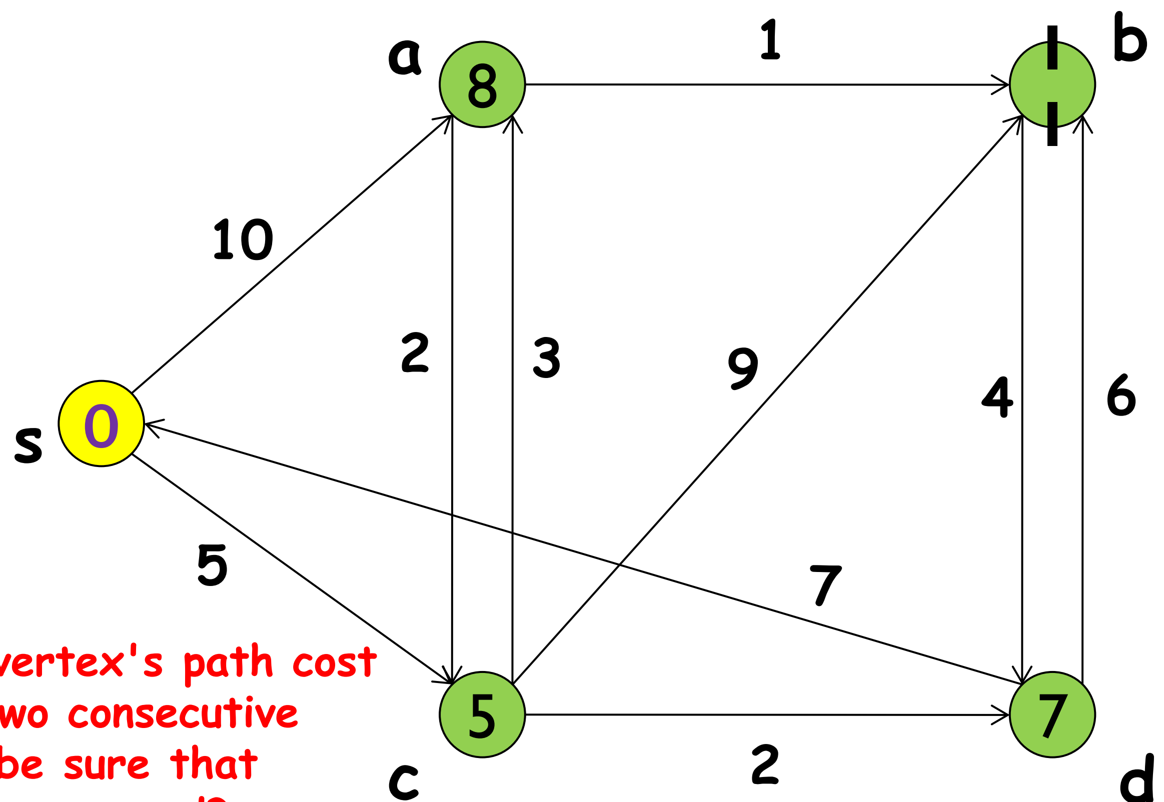


Iteration 3

No change!
Convergence!

mapper: (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) **(b,<a,11>)**
(b,<c,14>) (d,<c,7>) (b,<d,13>) (d,<b,15>) edges

reducer: (a,<8>) (c,<5>) (b,<11>) (d,<7>)



Question: If a vertex's path cost is the same in two consecutive rounds, can we be sure that this vertex has converged?

BFS Pseudo-Code

class MAPPER
    method MAP(nid n, node N)
        $d \leftarrow N.\textsc{Distance}$
        EMIT(nid n, N) ▷ Pass along graph structure
        for all nodeid $m \in N.\textsc{AdjacencyList}$ **do**
            EMIT(nid $m, d+1$) ▷ Emit distances to reachable nodes

class REDUCER
    method REDUCE(nid $m, [d_1, d_2, \ldots]$)
        $d_{min} \leftarrow \infty$
        $M \leftarrow \emptyset$
        for all $d \in$ counts $[d_1, d_2, \ldots]$ **do**
            if ISNODE(d) **then**
                $M \leftarrow d$ ▷ Recover graph structure
            else if $d < d_{min}$ **then** ▷ Look for shorter distance
                $d_{min} \leftarrow d$
        $M.\textsc{Distance} \leftarrow d_{min}$ ▷ Update shortest distance
        EMIT(nid m, node M)

Stopping Criterion

- How many iterations are needed in parallel BFS (equal edge weight case)?
- Now answer the question...
  - Six degrees of separation?
- Practicalities of implementation in MapReduce

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce explores all paths in parallel
  - Lots of "waste"
  - Useful work is only done at the "frontier"
- Why can't we do better using MapReduce?

Summary: SSSP

- Path-based algorithms typically involve iterative map/reduce
- They are typically formulated in a way that traverses in “waves” or “stages”, like breadth-first search
  - This allows for parallelism
  - They need a way to test for convergence
- Example: Single-source shortest path (SSSP)
  - Original Dijkstra formulation is hard to parallelize
  - But we can make it work with the "wave" approach