Common Algorithm Building Blocks

Mirek Riedewald

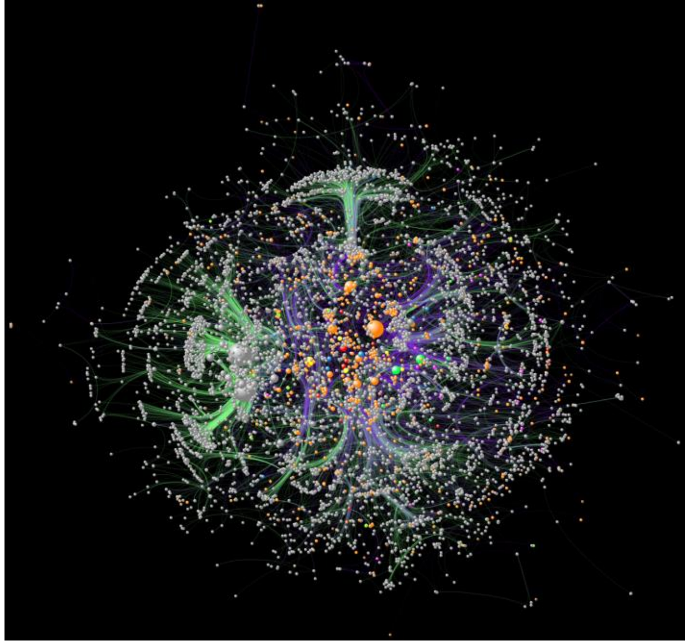
Key Learning Goals

- Write the pseudo-code for breadth-first search (BFS) in MapReduce and in Spark.
- Write the pseudo-code for single-source shortest path in MapReduce and in Spark.
- Write the pseudo-code for PageRank (without dangling pages) in MapReduce and in Spark.
- How can the single-source shortest path algorithm detect that no more iterations are needed?
- How can the PageRank algorithm detect that no more iterations are needed?

Key Learning Goals

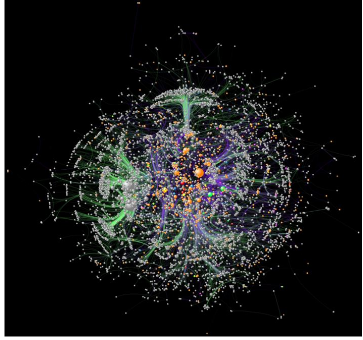
- Why is Spark better suited for BFS and BFS-based algorithms than MapReduce?
- How can we handle dangling pages in PageRank?

Introduction



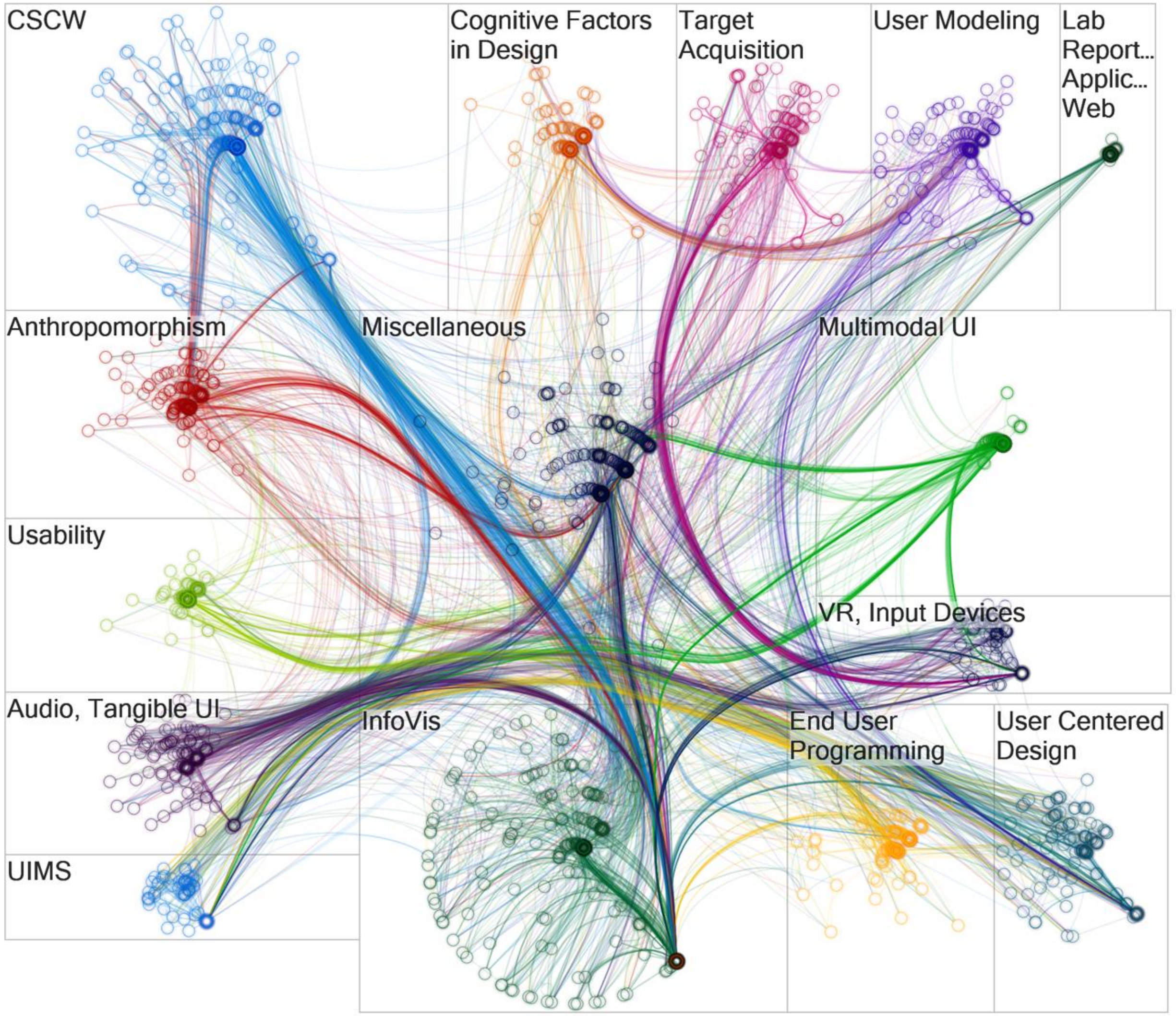
- Graphs are very general and a large variety of real-world problems can be modeled very naturally as graph analysis or mining problems. Data occurring naturally as graphs include:
  - The hyperlink structure of the Web
  - Social interactions, e.g., Facebook friendships, Twitter followers, email flows, and phone call patterns
  - Transportation networks, e.g., roads, bus routes, and flights
  - Relationships between genes, proteins, and diseases
- Since graphs are so general, many graph problems are inherently complex—a perfect target for distributed data-intensive computation.

Explanation for the Example

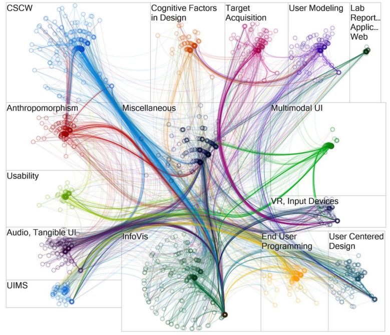


- This image shows a network of innovation relationships in the state of Pennsylvania in 1990 collected by Christopher Scott Dempwolf. The node-link visualization is laid out with the Harel-Koren FMS layout algorithm. Orange nodes are companies, e.g., Westinghouse electric; silver nodes inventors. Size indicates some measure of prominence. Edge colors represent the type of collaborations encoded and edges with similar routes are bundled together.
  - Chaturvedi, S, Dunne, C, Ashktorab, Z, Zacharia, R, and Shneiderman, B. “Group-in-a-Box meta-layouts for topological clusters and attribute-based groups: space efficient visualizations of network communities and their ties”. In: CGF: Computer Graphics Forum 33.8 (2014), pp. 52-68
  - Note: Cody Dunne is a faculty member in our college.

Relationships between CHI Topics



Explanation for the Example



- Papers of the ACM CHI conference, grouped disjointly by the main topic they cover and laid out individually for each topic using a polar layout. Node colors show the various groups, and the color of edges shows citations from papers in that same colored group to other papers. Edges with similar routes are bundled together. Radius and angle encode the number of citations and the betweenness centrality (how much of a gatekeeper the paper is between groups).
  – Same reference as previous image.

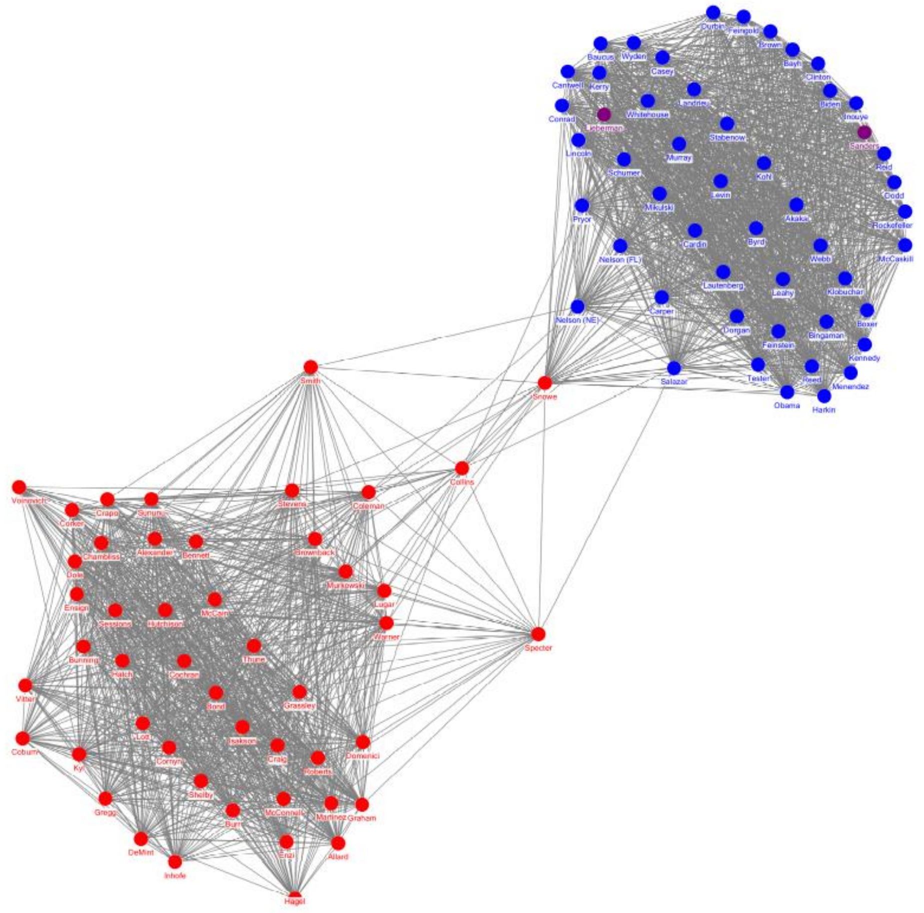
What is a Graph?

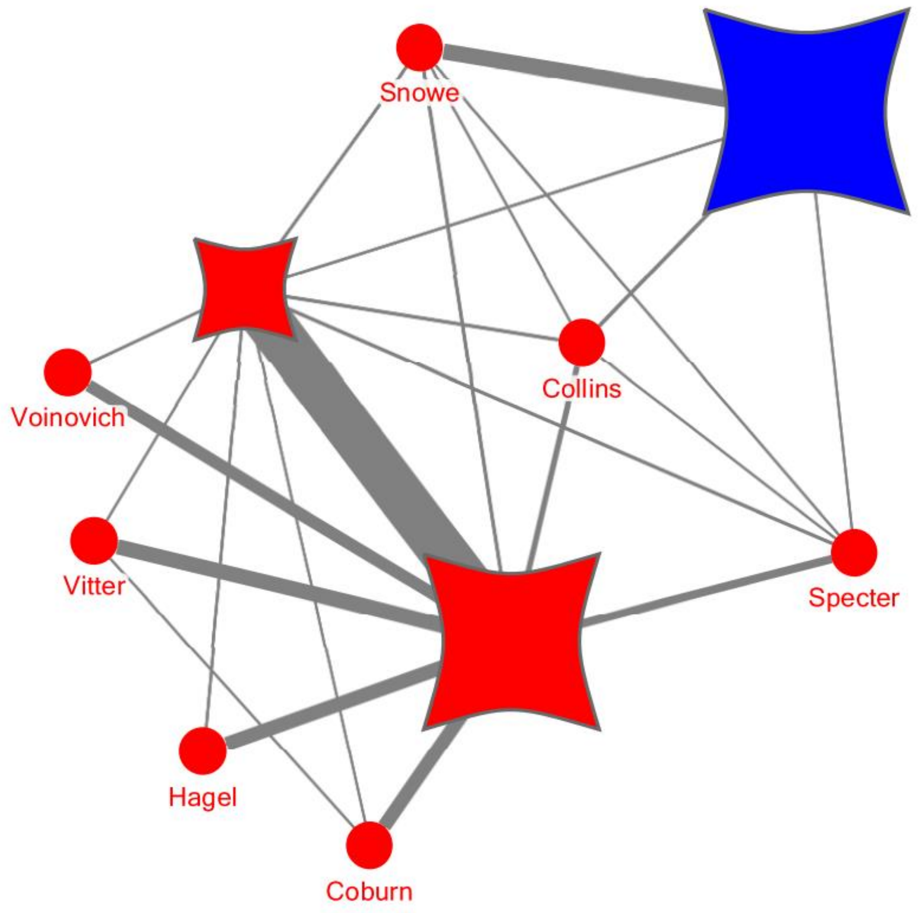
- Intuitively, a graph consists of vertices (a.k.a. nodes) and edges (a.k.a. links) between them. Vertex and edge labels can be used to encode heterogeneous information networks.
  - In an online social network, a vertex could be a person annotated with demographic information such as age. An edge could be annotated by the type of relationship, e.g., “friend” or “family.” This richer structure can improve the quality of the results obtained from graph analysis.
- We will focus on unlabeled graphs. Formally, a graph G is a pair (V, E), where V is a set of vertices and E is set of edges, such that $E \subseteq V \times V$.
- Edges can be directed or undirected. In a directed graph, edges (v1, v2) and (v2, v1) are different, while in an undirected graph they are the same. A standard trick is to encode undirected edge (v1, v2) using the corresponding two directed edges.
  - A road network should be modeled as a directed graph, because of the existence of one-way streets.
- Graphs might contain cycles. If a graph does not contain any cycle, it is said to be acyclic.

Graph Problems

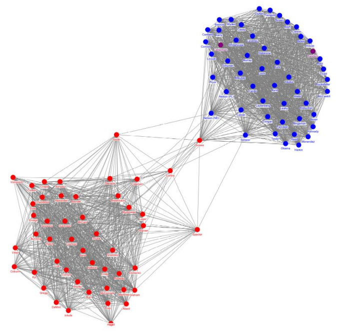
- There is a huge variety of graph analysis problems, including the following common examples:
- Graph search and path planning
  - Find driving directions from point A to point B.
  - Recommend possible friends in a social network.
  - Find the best route for IP packets or delivery trucks.
- Graph clustering
  - Identify user communities in social networks based on graph structure, e.g., connectedness.
  - Partition a large graph into homogeneous partitions to parallelize graph processing.
- Minimum spanning trees
  - Find the connected graph of minimum total edge weight.
- Bipartite graph matching
  - Match nodes on the “left” with nodes on “right” side. For example, match job seekers and employers, singles looking for dates, or papers with qualified reviewers.
- Maximum flow
  - Determine the greatest possible traffic between a source and a sink node, e.g., to optimize transportation networks.
- Finding “special” vertices
  - Find vertices that are of particular importance, e.g., disease hubs, leaders of a community, authoritative Web pages on a topic, or people with influence.

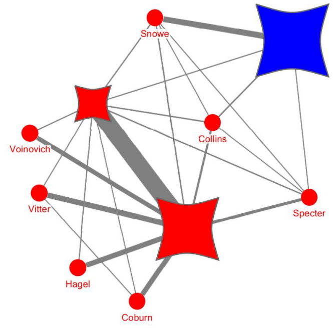
US Senate Co-Voting Patterns





Explanation for the Example





- US Senate co-voting patterns in 2007. Nodes represent senators and are colored by party: blue democrats, red republicans, purple independents. Edges join pairs of senators that vote the same way at least 70% of the time. The left visualization shows a node-link visualization of the data arranged using the Fruchterman Reingold network layout algorithm. The right visualization shows the same network simplified using motif simplification, which aggregates maximal cliques in the network into tapered square glyphs. In this the size of edges shows the overall number of edges between any pair of glyphs or glyphs and nodes.
  - Dunne, C and Shneiderman, B. “Motif simplification: improving network visualization readability with fan, connector, and clique glyphs”. In: CHI '13: Proc. SIGCHI Conference on Human Factors in Computing Systems. 2013, pp. 3247-3256

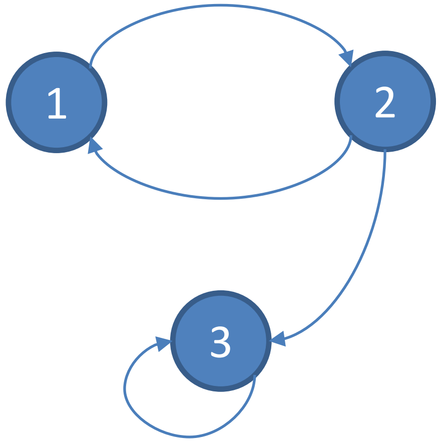
Graph Representations

- Graphs are usually represented in one of these three formats:
  - Adjacency matrix
  - Adjacency list
  - Set of individual edges
- We will take a quick look at each of them.

Adjacency Matrix Representation

- The adjacency matrix M represents a graph in a matrix of size |V| by |V|. Its size is quadratic in the number of vertices. Entry M(i,j) contains the weight of the edge from vertex i to vertex j; or 0 if there is no edge.
  - Notice that the adjacency matrix of an undirected graph is symmetric along the diagonal.

	1	**2**	**3**
1	0	1	0
2	1	0	1
3	0	0	1



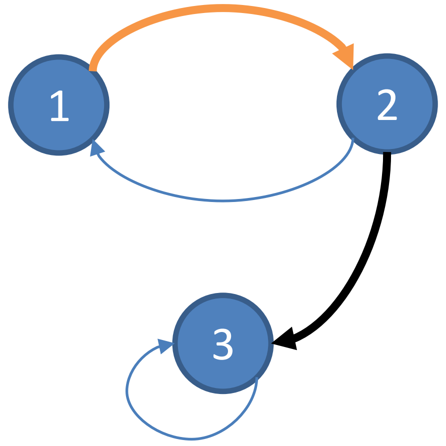
Adjacency Matrix Properties

- Advantages
  - A graph stored as an adjacency matrix is easy to process using linear algebra. For example, in matrix product M×M, entry (i,j) indicates if there exists a two-step path from vertex i to vertex j (if the value is non-zero) or not (if the value is zero).
  - An operation on outgoing edges of a vertex (also called "outlinks") corresponds to an iteration over a single row of the adjacency matrix. Similarly, an operation on incoming edges of a vertex (also called "inlinks") corresponds to an iteration over a single column.
- Disadvantage
  - For large graphs, the adjacency matrix tends to be very sparse as most vertex pairs are not connected by an edge. For those graphs this representation would be inefficient (in terms of storage cost) or even infeasible. Consider the Facebook friendship graph: for 1 billion users, even if every user on average had 10,000 friends, still only 99.999% of the matrix would have value zero.

Finding Two-Hop Paths through Linear Algebra

- Consider entry (1, 3) in matrix M×M, which is marked in green. Its value 1 is caused by M(1, 2)=1 (marked orange) and M(2, 3)=1 (marked black). Intuitively, the two-step path from node 1 to node 3 is made up of edges (1, 2) and (1, 3).

$$\underset{M}{\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline 0 & 0 & 1 \\ \hline\end{array}} \times \underset{M}{\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 0 & \mathbf{1} \\ \hline 0 & 0 & 1 \\ \hline\end{array}} = \underset{M\times M}{\begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 0 & 1 & 1 \\ \hline 0 & 0 & 1 \\ \hline\end{array}}$$



Adjacency List Representation

- The adjacency list representation only stores the existing *outlinks* for each vertex. Hence its size is linear in the number of edges.

	1	**2**	**3**
1	0	1	0
2	1	0	1
3	0	0	1

Adjacency matrix

1: 2
2: 1, 3
3: 3

Adjacency list for the same graph

Adjacency List Properties

- Advantages
  - Compared to the adjacency matrix, the adjacency list is much more space-efficient for sparse graphs.
  - It is still easy to compute over the outlinks of a vertex, because they are stored in the list for that vertex.
- Disadvantage
  - Computation over the inlinks of a vertex is more challenging compared to the adjacency matrix. Instead of scanning through a column, each of the different lists has to be searched, e.g., using binary search if the lists are sorted.

Set of Edges Representation

- The set of edges approach stores each edge of the graph explicitly.
  - The adjacency list format is the set-of-edges, grouped by the "start" vertex of each edge.

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	1

Adjacency matrix

(1,2), (2,1), (2,3), (3,3)

Set of edges for the same graph

Set of Edges Properties

- Advantages
  - This representation is still space-efficient for sparse graphs and it is easy to perform per-edge manipulations, e.g., inverting each edge.
  - The uniform record format—each record is a pair of vertices, in contrast to adjacency lists, whose sizes can vary—can simplify programming and data processing.
- Disadvantages
  - Out of the three, this is the most challenging format for computations over inlinks or outlinks of a vertex. The records corresponding to a given vertex' inlinks and outlinks might be scattered all over the file storing the set of edges.

Let us first take a look at the classic breadth-first algorithm for exploring the k-hop neighborhood of a given source vertex.

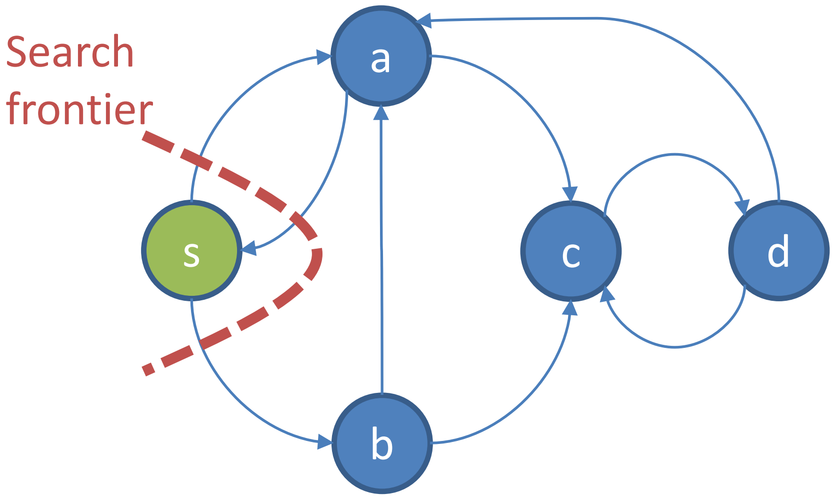
This algorithm forms the basis of two others we discuss later.

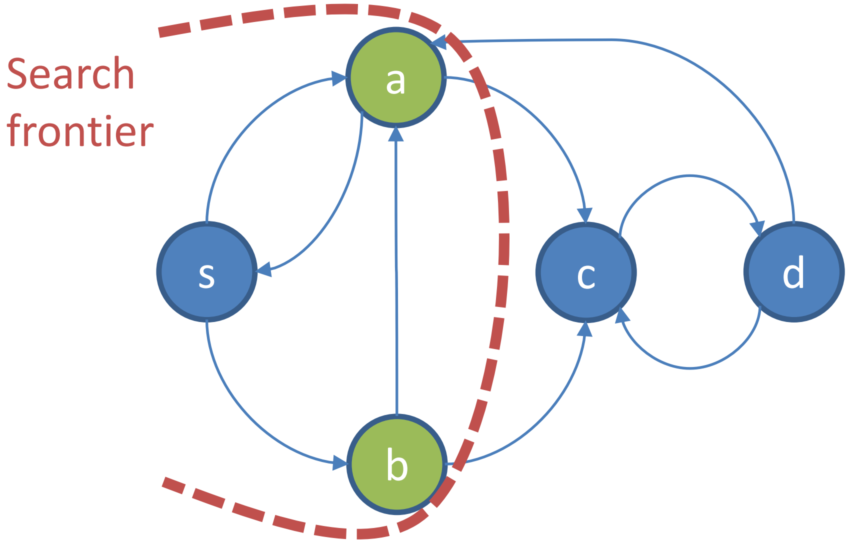
Breadth-First Search (BFS)

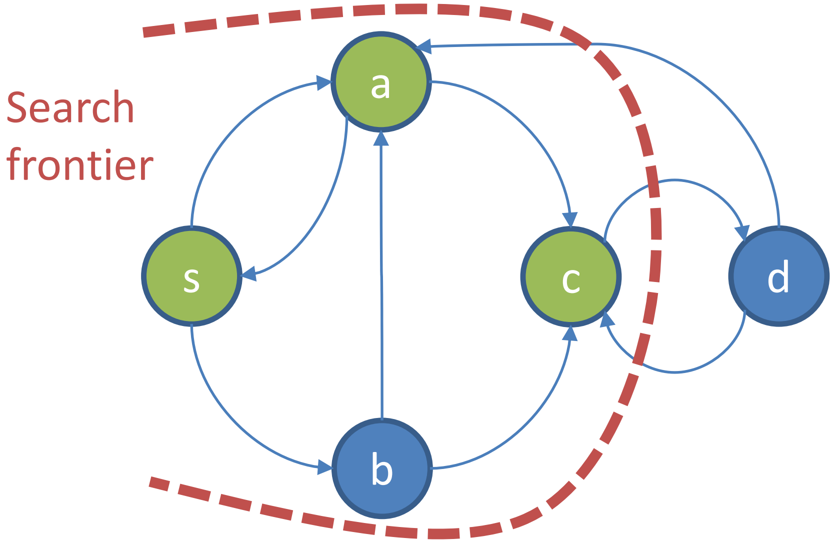
- Starting from a given source vertex s, breadth-first search first reaches all 1-hop neighbors of s, then all 2-hop neighbors, and so on.
- Notice that if the graph has cycles, then the search process can continue indefinitely, repeatedly traversing the cycles.
- In an acyclic graph, a vertex may also be encountered repeatedly, when it can be reached through different paths. However, breadth-first search will terminate after at most |V|-1 iterations—this is the length of the longest path possible in any acyclic graph.

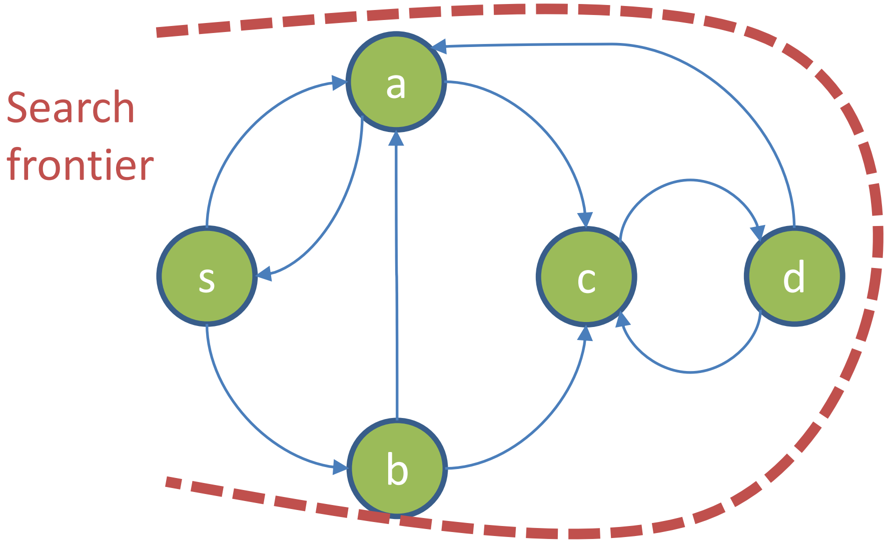
BFS Example

- The images below illustrate how breadth-first traversal reaches the different vertices in each iteration.
  - The green color highlights the vertices visited in the corresponding iteration. Notice how some vertices are visited repeatedly.
  - Due to the particular structure of the example graph, after iteration 3, each vertex will be reached in each of the following iterations.
- The search frontier illustrates how breadth-first traversal reaches new vertices.
  - Iteration 1: Vertices a and b are reached in exactly one hop from s.
  - Iteration 2: Vertex s is reached again (from a), and so is a (from b). Vertex c is reached for the first time.
  - Iteration 3: All vertices are reached, including d for the first time.


Search
frontier
a
s
c
d
b

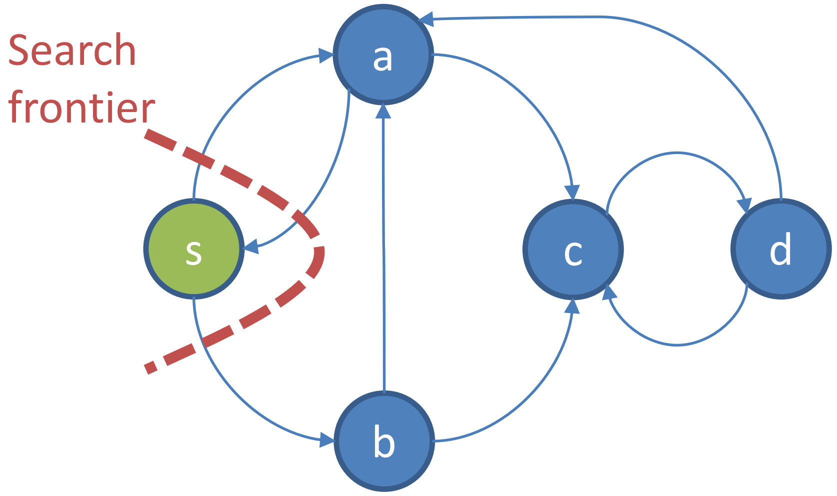

Search
frontier
a
s
c
d
b


Search
frontier
a
s
c
d
b


Search
frontier
a
s
c
d
b

Parallel BFS

- We will work with a graph represented in adjacency-list format.
- Parallel breadth-first search relies on the following observation: Consider a vertex v reached in iteration i. In iteration (i+1), the algorithm has to explore all adjacent vertices, i.e., the vertices in v's adjacency list.
    - Each vertex v reached in iteration i can be processed independently. In the beginning, only source vertex s is "reached," but in later iterations there could be a large set of reached vertices, enabling effective parallel computation.
- To turn this idea into an algorithm, let active(i) denote the set of vertices reached in iteration i. Then the next iteration simply replaces each active vertex by the vertices in its adjacency list. The corresponding parallel algorithm is as follows:
    - Each task receives a partition of active(i). For each vertex v in that partition, and for each vertex w in v's adjacency list, emit w. Then active(i+1) is obtained as the union of the emitted vertices.
- This algorithm has two problems.
    - The same w might be emitted multiple times. To eliminate duplicates, shuffling is needed.
    - How can the task access a vertex' adjacency list? For large graphs, it is infeasible to copy all adjacency lists (i.e., the entire graph) to all tasks. If each task receives only some of the adjacency lists, then it should only process vertices for which it has their adjacency lists!
- Let us return to our example to see what happens. Assume there are two tasks.



Initially, only source node s is active. The driver informs all tasks in iteration 1 about this.

Task 0:

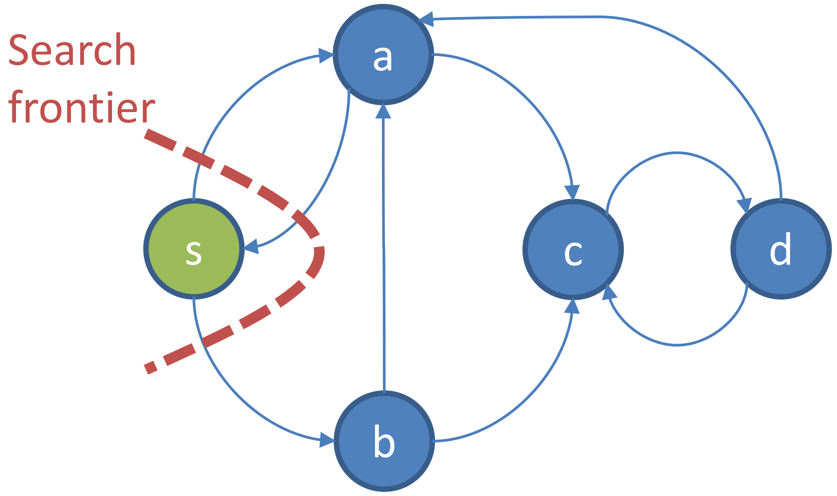
active: **s**
Adjacency lists:
s: a, b
b: a, c
d: a, c

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

Task 1:

active: **s**
Adjacency lists:
a: s, c
c: d

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```



In iteration 1, all nodes in the adjacency list of s are emitted. This is done by task 0. Task 1 has no active node and hence does not emit anything. The output is sent to the task that owns the corresponding adjacency list.

Task 0:

active: s
Adjacency lists:
s: a, b
b: a, c
d: a, c

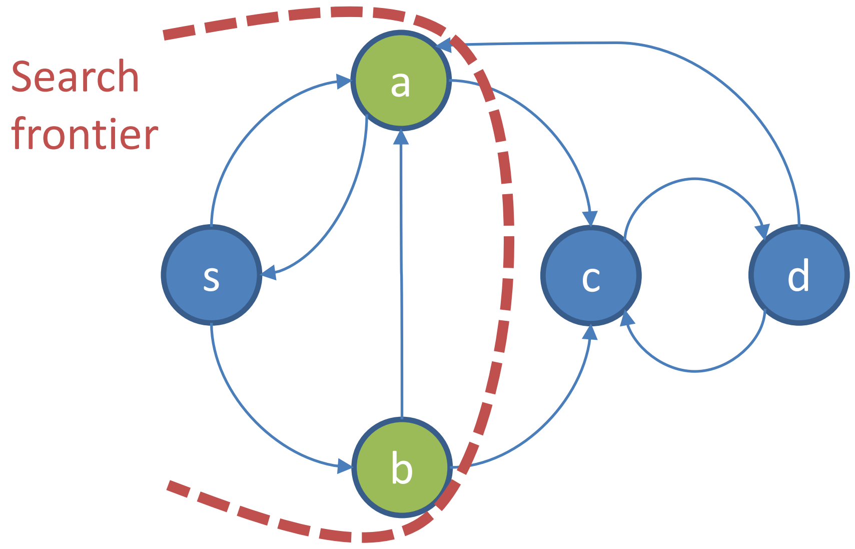
```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

Task 1:

active: s
Adjacency lists:
a: s, c
c: d

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

b **a**



Now vertices a and b are active. Notice that task 0 only knows about "its" vertex b, while task 1 only knows about a.

Task 0:

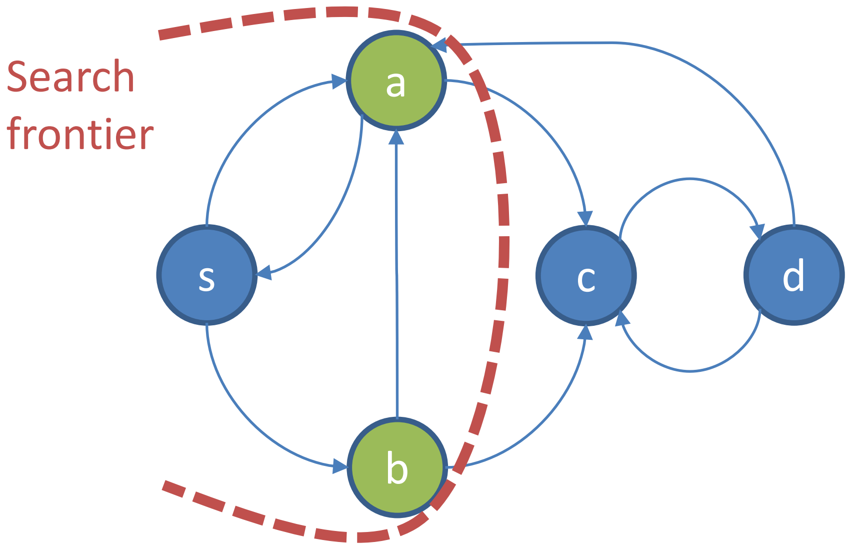
active: **b**
Adjacency lists:
s: a, b
b: a, c
d: a, c

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

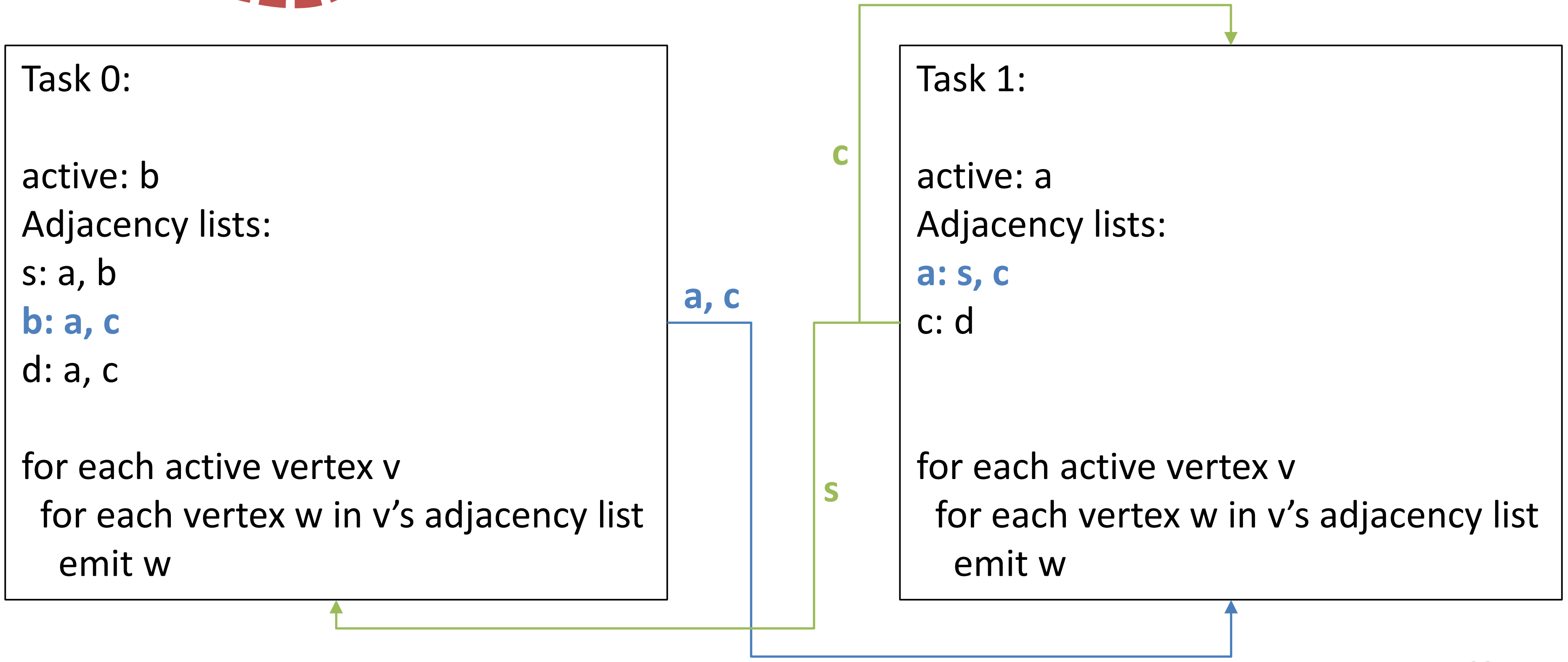
Task 1:

active: **a**
Adjacency lists:
a: s, c
c: d

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```



In iteration 2, all nodes in the adjacency lists of a and b are emitted. Since a and b are “owned” by different tasks, this work is done in parallel.
Note that c is received twice by task 1. It is straightforward to deal with such duplicates locally.



Task 0:

active: b
Adjacency lists:
s: a, b
b: a, c
d: a, c

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

Task 1:

active: a
Adjacency lists:
a: s, c
c: d

```
for each active vertex v
  for each vertex w in v's adjacency list
    emit w
```

BFS in MapReduce

- The communication of the emitted active vertices requires shuffling. In MapReduce, this means that each iteration is a full MapReduce job.
- The parallel algorithm requires a partition of the adjacency list to be kept in the memory of a task across iterations. This is *not possible* in MapReduce. Instead, each Map phase has to read the adjacency list info—again and again in every iteration.
- Hence both, newly active vertices and adjacency list data have to be read, shuffled, and emitted in every iteration.

MapReduce Implementation

```
// Map processes vertex n.
// Object N stores n's adjacency list
map(id n, vertex N) {
  // Pass along the graph structure
  emit(n, N)

  // If n is active, mark each vertex in
  // n's adjacency list as active
  if (N.isActive())
    for all id m in N.adjacencyList do
      emit(m, "active")
}
```

The driver program repeatedly calls this MapReduce job, each time passing the previous job's output directory as the input to the next.

```
// Reduce receives the vertex object for vertex m and
// between zero or more "active" flags.
reduce(id m, [o1, o2,…]) {
  active = false
  M = NULL

  for all o in [o1,o2,…] do
    if isVertexObject(o) then
      // The vertex object was found: recover graph structure
      M = o
    else
      // An active flag was found: m should be active
      active = true

  // Update active status of vertex m
  M.setActive(active)
  emit(m, M)
}
```

Improving the MapReduce Algorithm

- Can we avoid the expensive cycle of reading the graph in the Map phase, sending it from Mappers to Reducers, and then writing it in the Reduce phase, which repeats in every iteration? Distributed file system accesses are expensive, and the network is a precious shared resource.
- Unfortunately MapReduce does not support “pinning” data and the corresponding Map and Reduce tasks to a certain worker machine across different MapReduce jobs.
- One could attempt to use the distributed file cache for this purpose. However, it is very limited in the sense that it makes the same data available for each Mapper or Reducer, but does not support managing different partitions on different machines.
- In general, MapReduce lacks mechanisms to exploit the repetitive structure of iterative computations. Spark’s RDD abstraction addresses exactly this issue.

BFS in Spark

- We discuss the Spark Scala pseudo-code for BFS with RDDs. The version with DataSet is left as a voluntary challenge.
- The Spark program separates static data (the graph structure) from evolving data (the active vertex set).
  - This leverages cached RDD partitions and avoids the shuffling of the adjacency list data from the MapReduce implementation.
  - On the downside, the new activity status information has to be joined with the graph data, so that tuples of type (vertexID, adjacencyList) and (vertexID, activeStatus) are joined on vertexID. This join normally requires shuffling, but careful co-partitioning avoids shuffling.

Spark Implementation

```
// Assume the input file contains in each line a vertex ID and its adjacency list. Function getAdjListPairData
// returns a pair of vertex ID and its adjacency list, creating a pair RDD.
graph = sc.textFile(…).map( line => getAdjListPairData(line) )

// Some code here to make sure that graph has a Partitioner. This is needed for avoiding shuffling in the join below.

// Tell Spark to try and keep this pair RDD around in memory for efficient re-use
graph.persist()

// The active vertex set initially only contains source vertex s, which needs to be passed through the context.
// Create a pair RDD for s. Value 1 is a dummy value.
activeVertices = sc.parallelize( (“s”, 1) )

// Function extractVerticesAsPair returns each vertex id m in the adjacency list as (m, 1), where 1 is a dummy value.
for (iterationCount <- 1 to k) {
  activeVertices = graph.join( activeVertices )
          .flatMap( (id, adjList, dummy) => extractVerticesAsPair(adjList) )
          .reduceByKey( (x, y) => x )          // Remove duplicate vertex occurrences
}
```

BFS in a DBMS

- Assume the graph is stored in table Graph(id1, id2) and the active vertices in table Active(id), where id1, id2, and id are all vertex IDs. Then we can express the computation of an iteration as shown below.
  - This query expresses a left semi-join on Graph.

```
newActive =

          SELECT DISTINCT id2
          FROM Graph AS G
          WHERE G.id1 IN
                    (SELECT * FROM Active)
```

Next we take a look at two important graph algorithms: single-source shortest path and PageRank.

The parallel versions of both are built on BFS.

Single-Source Shortest Path

- Consider the well-known problem of finding the shortest path from a source vertex, s, to all other vertices in a given graph. The length of a path is equivalent to the total weight of all edges belonging to the path.
- Let all edges in the graph have *non-negative weights*.
- We will first discuss a classic sequential algorithm called Dijkstra’s algorithm, which finds the solution very efficiently. Then we will explore how to solve the problem in parallel.

Dijkstra’s Algorithm

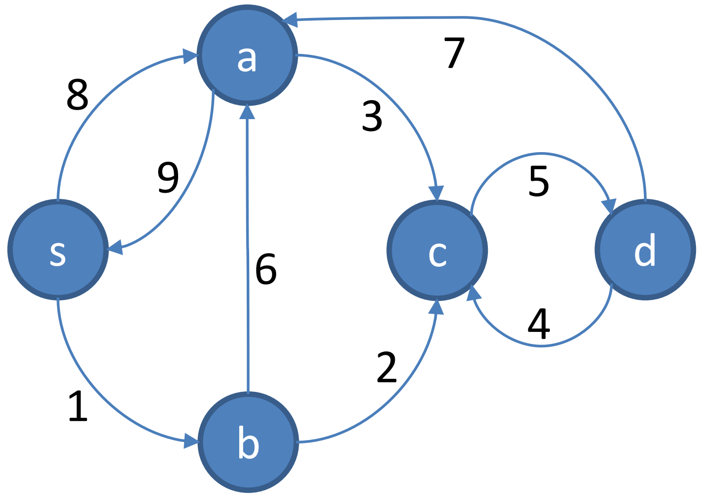
1. Let d[v] denote the distance of vertex v from source s. Initialize d[v] by setting d[s]=0 and d[v]=∞ for all v≠s.
2. Insert all graph vertices into a priority queue sorted by distance. (Initially s will be first, while all other vertices appear in some arbitrary order.)
3. Repeat until the queue is empty
    1. Remove the first vertex u from the queue. Output (u, d[u]). (The shortest path to u was found.)
    2. For each vertex v in u’s adjacency list do
        1. If v is in the queue and d[v] > d[u]+weight(u,v), then set d[v] to d[u]+weight(u,v). (A shorter path to v was found through u.)

Dijkstra’s Algorithm Example

Output:

Priority queue: (s,0), (d,∞), (a,∞), (b,∞), (c,∞)

Node being processed:



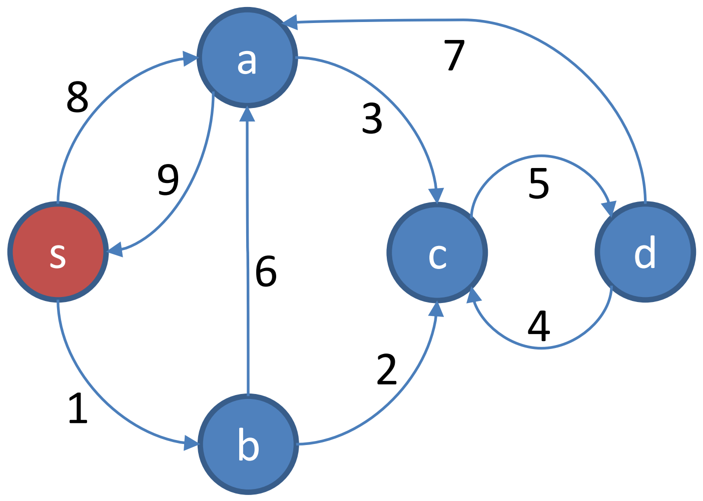
Initially all vertices and their current distance values are in the priority queue.

Dijkstra’s Algorithm Example

Output: (s,0)

Priority queue: ~~(s,0),~~ (d,∞), (a,∞), (b,∞), (c,∞)

Node being processed: s: (a,8), (b,1)



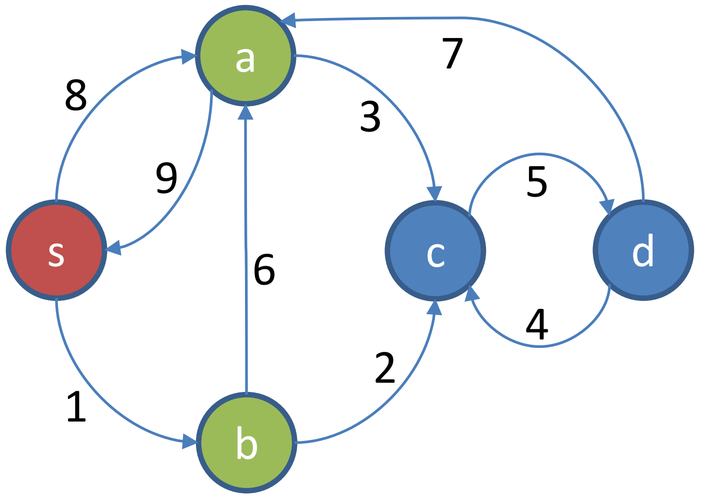
The first vertex, s, is removed and output.

Dijkstra’s Algorithm Example

Output: (s,0)

Priority queue: (b,1), (a,8), (d,∞), (c,∞)

Node being processed: s: (a,8), (b,1)



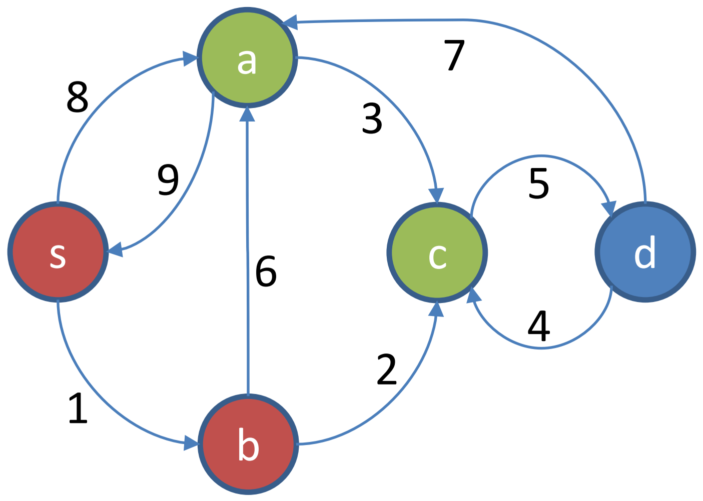
For all vertices in the adjacency list of the removed vertex s, the distances are updated. For example, since d[s]=0 and entry (a,8) appears in the adjacency list, there exists a path to node a with length 0+8 = 8.

Dijkstra’s Algorithm Example

Output: (s,0), (b,1)

Priority queue: (c,3), (a,7), (d,∞)

Node being processed: b: (a,6), (c,2)



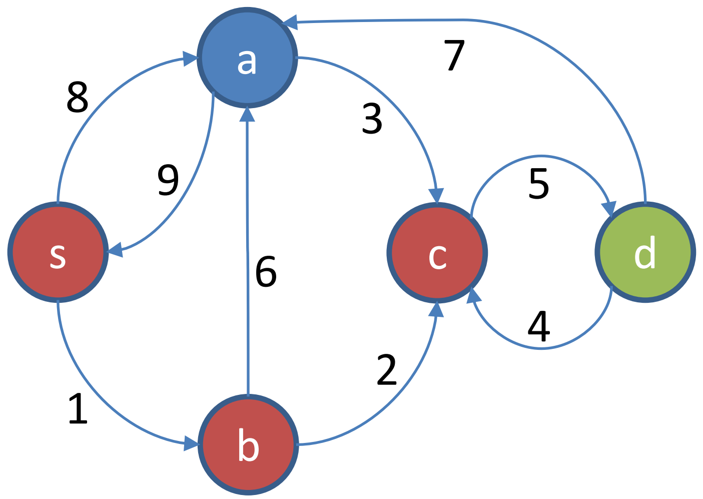
Now vertex b is removed and output, followed by updating of distances of nodes a and c.

Dijkstra’s Algorithm Example

Output: (s,0), (b,1), (c,3)

Priority queue: (a,7), (d,8)

Node being processed: c: (d,5)



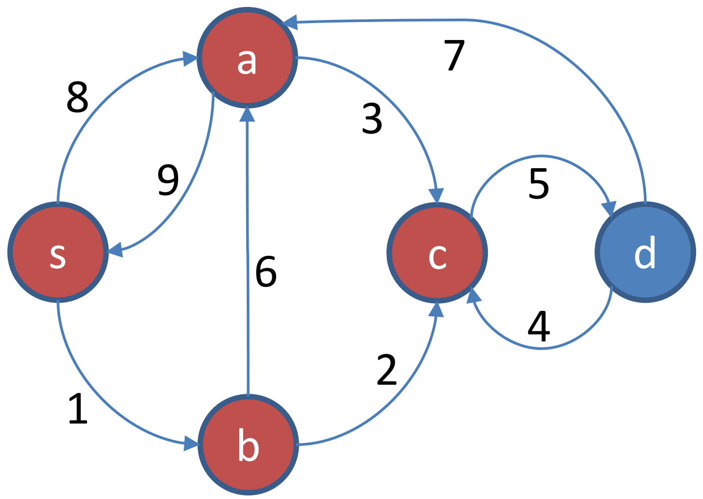
Now vertex c is removed and output, followed by updating of the distance of vertex d.

Dijkstra’s Algorithm Example

Output: (s,0), (b,1), (c,3), (a,7)

Priority queue: (d,8)

Node being processed: a: (c,3), (s,9)



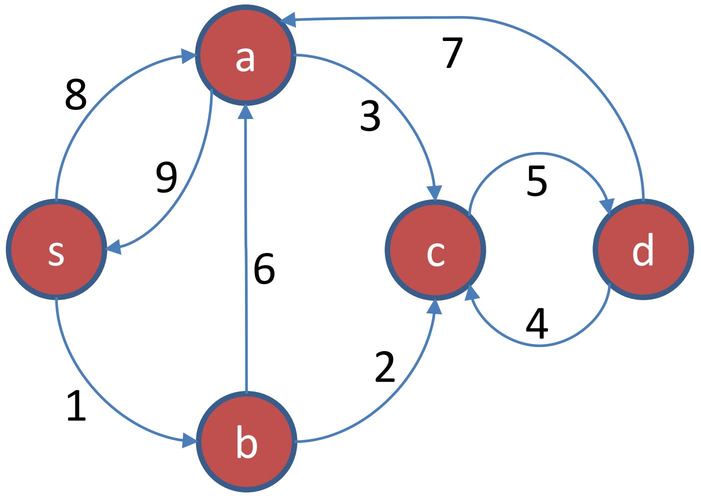
Now vertex a is removed and output. Since vertices c and s in a’s adjacency list are not in the queue, no distance updates are performed.

Dijkstra’s Algorithm Example

Output: (s,0), (b,1), (c,3), (a,7), (d,8)

Priority queue:

Node being processed: d: (c,4), (a,7)



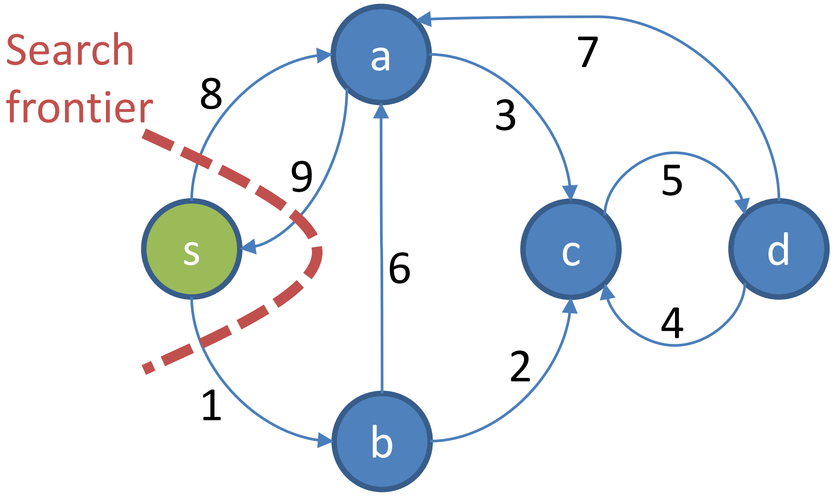
Finally vertex d is output and the algorithm terminates.

Parallel Single-Source Shortest Path

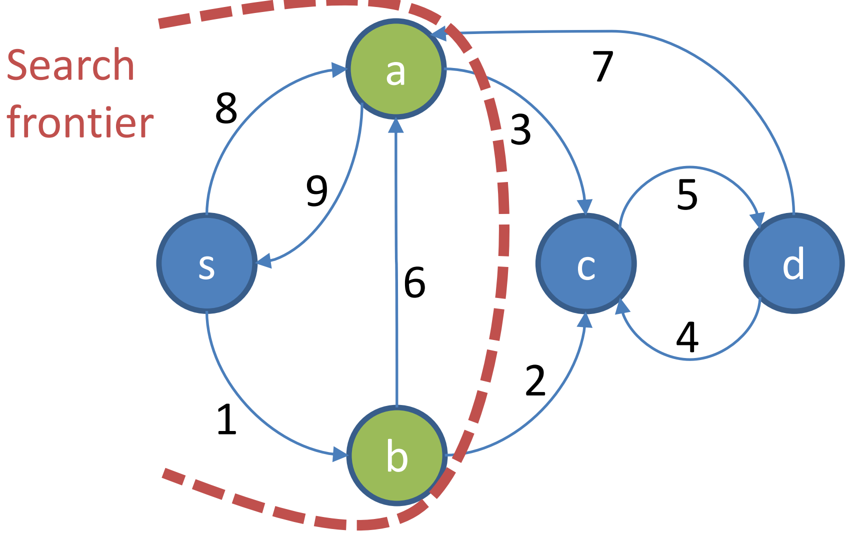
- Dijkstra’s algorithm is elegant and very efficient for sequential execution, but difficult to adapt for parallel execution: vertices are removed from the priority queue one-by-one. One cannot remove multiple vertices at once for parallel processing, without risking incorrect results.
  - Recall the example where after processing vertex s, entries (b,1) and (a,8) were the first entries in the queue. Removing both and processing them in parallel would not have worked, because the shortest path to a is going through b.
- While removing multiple vertices at once jeopardizes correctness in *any* parallel environment, the use of a single priority queue represents a particular challenge for MapReduce and Spark. It is not clear how to best implement such a shared data structure in a system without shared memory abstraction.

Shortest Path using BFS

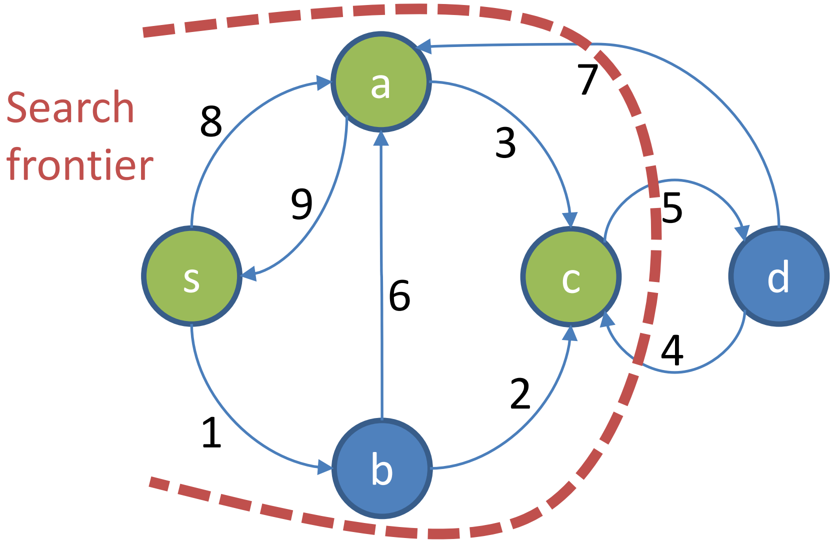
- Even without the priority queue, which is the key component of Dijkstra's algorithm, the parallel solution can still rely on the following property: If there exists a path of length d[u] to some vertex u, then for each vertex v in u's adjacency list there exists a path of length d[u]+weight(u,v).
- We design an algorithm that exploits this property by systematically exploring the graph using BFS:
  - In the first iteration, find all vertices reachable from s in exactly one hop and update their distances.
  - In the second iteration, find all vertices reachable from s in exactly two hops and update their distances. And so on.
  - Continue this process until the shortest path to each vertex was found.
- We illustrate the algorithm next for the same example graph, now with edge weights.



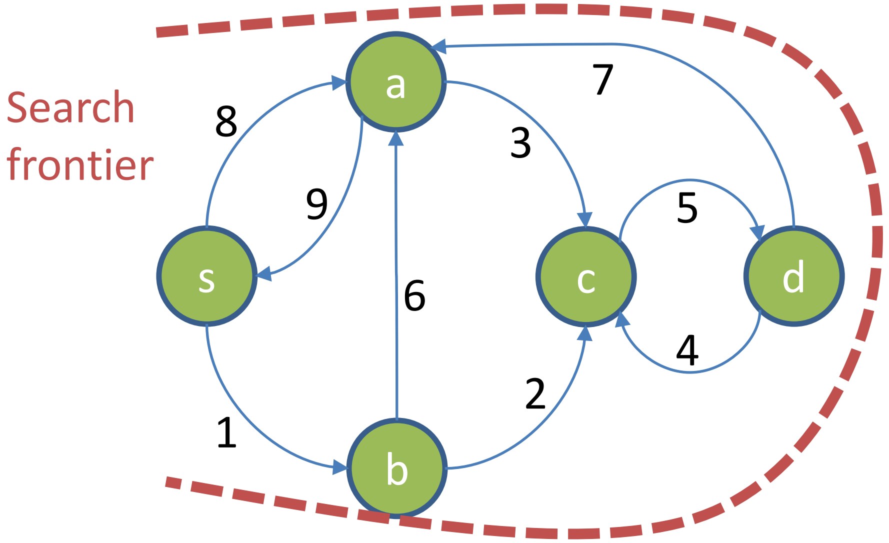
Initial state: (s,0), (a,∞), (b,∞), (c,∞), (d,∞)



Iteration 1: (s,0), (a,8), (b,1), (c,∞), (d,∞)



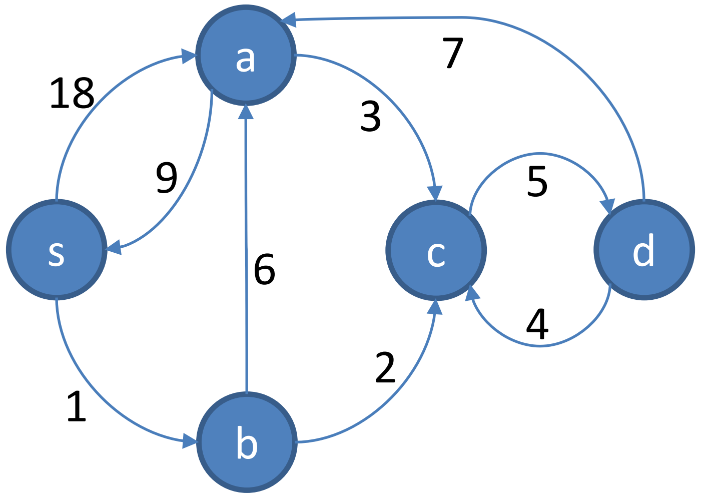
Iteration 2: (s,0), (a,7), (b,1), (c,3), (d,∞)



Iteration 3: (s,0), (a,7), (b,1), (c,3), (d,8)

When to Stop Iterating

- How does the algorithm know when all shortest paths were found?
- If all edges have the same weight, we can stop iterating as soon as no vertex has a distance of ∞ any more. This can be detected using a global counter or accumulator.
  - The number of iterations therefore depends on the graph diameter. In practice, many networks show the small-world phenomenon, i.e., have a small diameter.
- The situation becomes more complicated if edges have different weights. Then a "detour" path can be shorter (i.e., it has a lower total weight) than a more "direct" connection. In that case, we cannot stop as soon as all vertex distances are finite. Instead, iterations have to continue until no vertex' shortest distance changes any more. This can also be detected using a global counter or accumulator.
  - In the worst case, this can require |V|-1 iterations, where |V| is the number of vertices in the graph.
- If the graph contains cycles of negative weight, then the algorithm never terminates: each traversal of the cycle reduces the distances of all its vertices indefinitely.



This is an example for a graph where a "detour" path consisting of four edges is shorter than the direct path. Path (s, a) has length 18, while path (s, b, c, d, a) has length 15. Detecting this shortest path requires 4 iterations.

Single-Source Shortest Path in MapReduce

- The algorithm is virtually identical to BFS, but has to keep track of the shortest distance found for each vertex so far:
  - Map processes a single vertex u, emitting d[u]+weight(u,v) for each vertex v in u's adjacency list.
  - Reduce collects the newly computed distances for all inlinks of vertex v and determines if any of them is shorter than its currently known shortest distance.
- The driver program repeatedly calls the MapReduce program as many times as necessary, exploring ever longer paths.

MapReduce Code for a Single Iteration

```
// Map processes vertex n. Object N
// stores n's current min distance and
// its adjacency list
map(id n, vertex N) {
  // Pass along the graph structure
  emit(n, N)

  // Compute the distance for each outlink
  d = N.distance
  for all m in N.adjacencyList do
    emit( m, d + weight(n,m) )
}
```

```
// Reduce receives the vertex object for vertex m and
// the newly computed distances for all m's inlinks
reduce(id m, [d1, d2,…]) {
  dMin = ∞
  M = NULL

  for all d in [d1, d2,…] do
    if isVertex(d) then
      // The vertex object was found: recover graph structure
      M = d
    else
      // A distance value for an inlink was found: keep track
      // of the minimum.
      if d < dMin then dMin = d

  // Update distance of vertex m if necessary.
  if dMin < M.distance then M.distance = dMin
  emit( m, M )
}
```

MapReduce Algorithm Analysis

- Each iteration of the MapReduce algorithm performs a large amount of work:
  - The entire graph is read from the distributed file system.
  - The entire graph is transferred from Mappers to Reducers.
  - The entire graph, with updated distance values, is written to the distributed file system.
  - For every vertex u, no matter if it potentially lies on a shorter path to another vertex v or not, distance d[u]+weight(u,v) is computed and sent to the Reducers.
  - For every vertex v, no matter if its shortest path was already found in previous iterations or not, the Reduce function is executed to re-compute the shortest distance.
- This brute-force approach performs many irrelevant computations:
  - In early iterations, Map computes distances for vertices that have not yet been reached, therefore still have infinity distance.
  - In later iterations, the program keeps re-computing paths for vertices whose shortest path was already found.
- Contrast this with the elegance of Dijkstra’s algorithm, which avoids both of these wasteful types of computation, but needed the priority queue in order to achieve this.

Improving the MapReduce Algorithm: Avoiding Useless Work

- Can we avoid processing nodes that do not improve the path length values found so far? This turns out to be quite simple based on the following observations:
  - If a vertex u has distance d[u]=∞, then it cannot help reduce the distance for any of the vertices in its adjacency list.
  - Assume vertex u had the same distance d[u]=x in iterations i and (i+1). For any vertex v in its adjacency list, the Map function call for u would emit the same value x+weight(u,v) in both iterations. Since this value was already included in the Reduce computation in iteration i, it cannot result in any improvements in iteration (i+1).
- To exploit these properties, we can do the following:
  - Like BFS, the program distinguishes between "active" and "inactive" vertices. Active vertices are those that could potentially help reduce the distance for another vertex. Formally, we define a vertex to be active if and only if its distance value changed in the previous iteration. The only exception to this rule is source vertex s, which is set to "active" before the first iteration. Note that a vertex that was active in one iteration could become inactive in the next, and vice versa.
  - It is easy to prove that a vertex whose distance reached the final value, i.e., the shortest-path distance from s, will remain inactive afterwards. Hence the algorithm can stop iterating as soon as all vertices become inactive.
- The correspond program is shown below.

Improved MapReduce Code for a Single Iteration

```
// Map processes vertex n. Object N
// stores n's current min distance and
// its adjacency list
map(id n, vertex N) {
  // Pass along the graph structure
  emit(n, N)

  // Compute the distance for each outlink
  // of an active vertex
  if **N.isActive** {
    d = N.distance
    for all m in N.adjacencyList do
      emit( m, d + weight(n,m) )
  }
}
```

```
// Reduce receives the vertex object for vertex m and
// the newly computed distances for all m's inlinks
reduce(id m, [d1, d2,…]) {
  dMin = ∞
  M = NULL

  for all d in [d1, d2,…] do
    if isVertex(d) then
      // The vertex object was found: recover graph structure
      M = d
    else
      // A distance value for an inlink was found: keep track
      // of the minimum.
      if d < dMin then dMin = d

  // Update distance of vertex m if necessary.
  if dMin < M.distance then {
    M.distance = dMin
    // The distance change could affect the distance for
    // nodes in the adjacency list, hence set status to active
    M.setActive(true)
  }
  emit( m, M )
}
```

Single-Source Shortest Path in Spark

- We discuss the Spark Scala pseudo-code with RDDs. The version with DataSet is left as a voluntary challenge.
- The Spark program separates static data (the graph structure) from evolving data (the currently known shortest distance of each vertex).
  - This leverages cached RDD partitions and avoids the shuffling of the adjacency list data from the MapReduce implementation.
  - On the downside, the new distance status information has to be joined with the graph data, so that tuples of type (vertexID, adjacencyList) and (vertexID, distance) are joined on vertexID. This join normally requires shuffling, but careful co-partitioning avoids shuffling.

Spark Implementation

```
// Assume the input file contains in each line a vertex ID and its adjacency list. Function getAdjListPairData
// returns a pair of vertex ID and its adjacency list, creating a pair RDD. An entry in the adjacency list is a pair
// of destination vertex id and corresponding edge weight. The adjacency list of the source node has a special
// “source” flag used for the initial distance settings.
graph = sc.textFile(…).map( line => getAdjListPairData(line) )

// Some code here to make sure that graph has a Partitioner. This is needed for avoiding shuffling in the join below.

// Tell Spark to try and keep this pair RDD around in memory for efficient re-use
graph.persist()

// Create the initial distances. mapValues ensures that the same Partitioner is used as for the graph RDD.
distances = graph.mapValues( (id, adjList) => hasSourceFlag(adjList) match {
                                              case true => 0
                                              case _ => infinity }

// Function extractVertices returns each vertex id m in n’s adjacency list as (m, distance(n)+w),
// where w is the weight edge (n, m). It also returns n itself as (n, distance(n))
for (iterationCount <- 1 to k) {          // Use Accumulator instead to determine when last iteration is reached
  distances = graph.join( distances )
        .flatMap( (n, adjList, currentDistanceOfN) => extractVertices(adjList, currentDistanceOfN) )
        .reduceByKey( (x, y) => min(x, y) )          // Remember the shortest of the distances found
}
```

Single-Source Shortest Path in a DBMS

- Assume the graph is stored in table Graph(id1, id2, weight) and the currently known shortest distances in table Distances(id, distance), where id1, id2, and id are all vertex IDs. Then we can express the computation of an iteration as shown below.

```
tempDistances =
        SELECT G.id2 AS id, D.distance+G.weight AS distance
        FROM Graph AS G, Distances AS D
        WHERE G.id1 = D.id
        UNION
        SELECT * FROM Distances

newDistances =
        SELECT id, min(distance)
        FROM tempDistances
        GROUP BY id
```

PageRank

- PageRank was popularized by Google as a measure for evaluating the importance of a Web page. Intuitively it assigns greater importance to pages that are linked from many other important pages.
- More precisely, the PageRank reflects the probability that a random Web surfer will end up on this page. The random Web surfer can reach a page by jumping to it or by following the link from another page pointing to it.
- PageRank is a useful feature for Web search, helping identify the most relevant results for a given keyword query.
  - For example, consider query "Northeastern University." A person entering these keywords will most likely be looking for the Northeastern.edu homepage. If a search engine only considers traditional measures of importance such as TF/IDF, it might highly rank a spam page containing term "Northeastern" many times. Assuming that Northeastern.edu will be linked from many more important pages than the spam site, taking into account the PageRank value can help boost its rank in the result list.

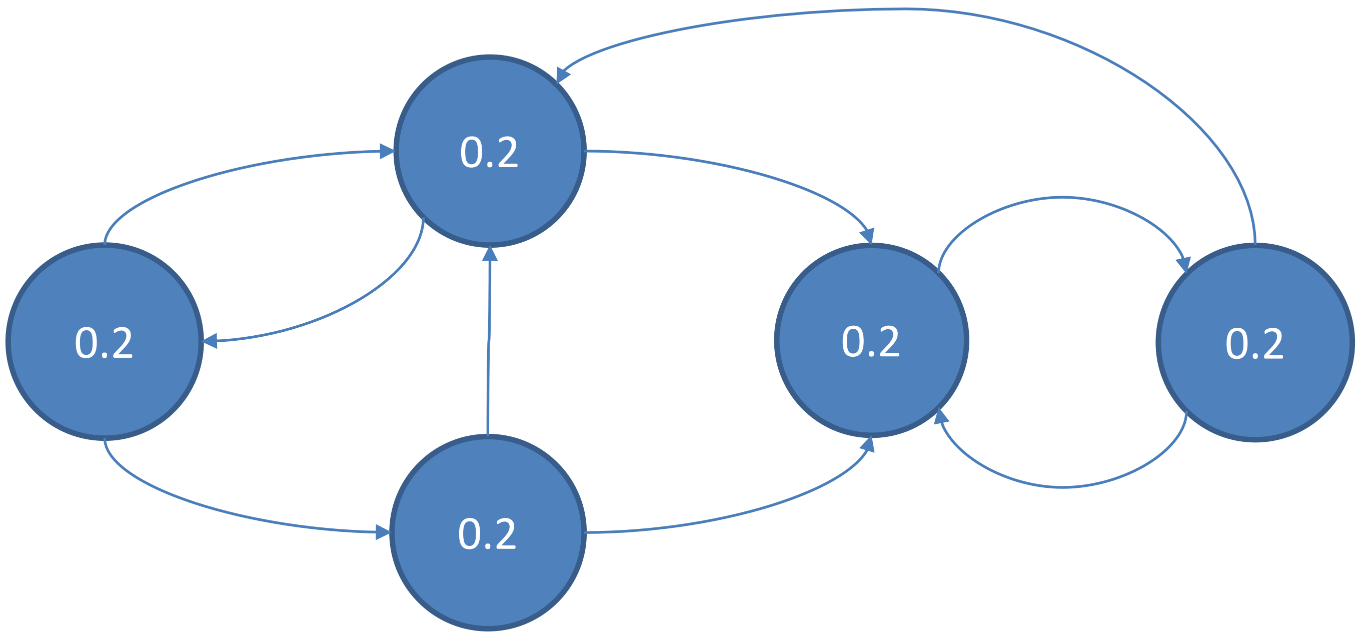
PageRank Definition

- Let's take a closer look at the random surfer model. Assume the random surfer is currently visiting some Web page. She first randomly decides to either type some new URL into the browser's address field (i.e., perform a random jump) or to follow one of the links on the current page to the page this link points to. The probability of doing the former is 1-α, the probability of doing the latter is α. If she chooses the random jump, then she will pick any of the Web pages in the graph uniformly at random. If she chooses to follow a link, then she will choose one of the links on the current page uniformly at random.
- Based on this model, the PageRank of some Web page n, P(n), is formally defined as
  - $P(n) = (1-\alpha)\frac{1}{|V|} + \alpha \sum_{m \in L(n)} \frac{P(m)}{C(m)}$
- The variables in the formula have the following meaning:
  - |V| is the number of pages (vertices) in the Web graph considered.
  - α is the probability of the surfer following a link; 1-α the probability of making a random jump.
  - L(n) is the set of all pages in the graph linking to n.
  - P(m) is the PageRank of another page m.
  - C(m) is the out-degree of page m, i.e., the number of links on that page.
- The formula has two major terms, corresponding to the two different ways of reaching page n:
  1. The random surfer visits n if she decides to randomly jump to it. The probability for this to happen equals the probability of the surfer deciding to make a random jump (1-α) times the probability that this random jump will end up on n, which is one out of |V| existing Web pages.
  2. The random surfer decided to follow a link (probability α) and was visiting another page m (which happens with probability P(m)) that contained a link to n. In that case the link to n is chosen with probability 1/C(m).

Iterative PageRank Computation

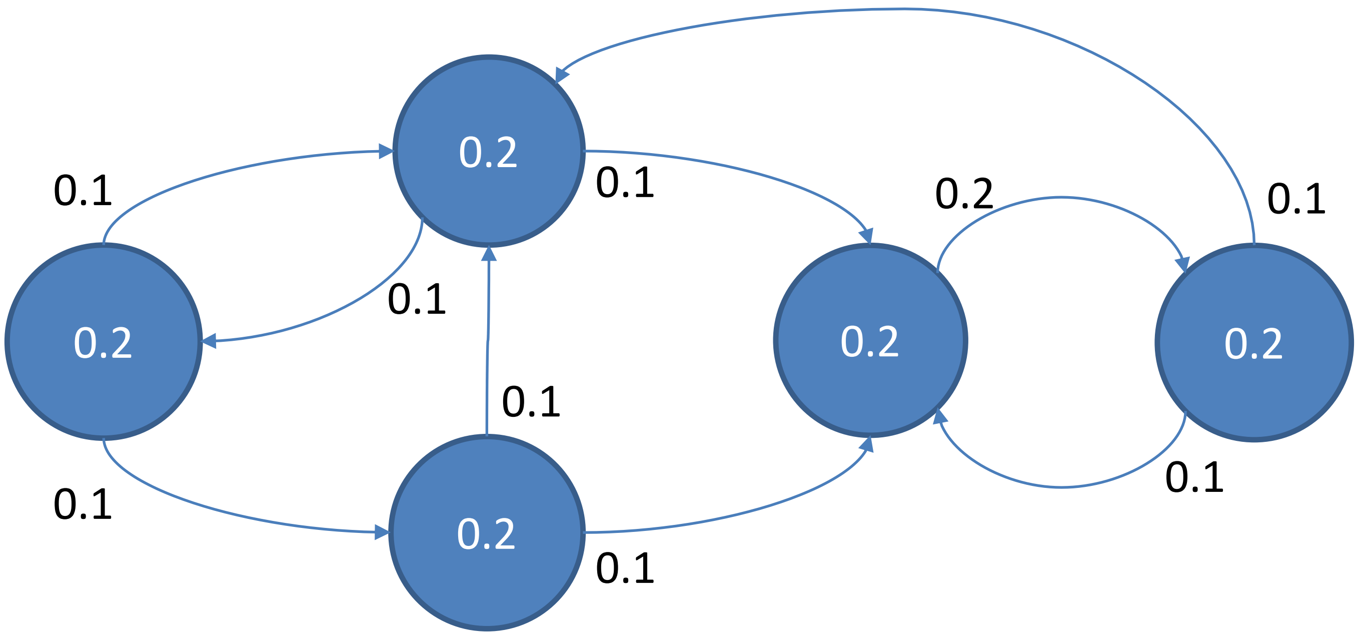
- Notice that the definition of PageRank creates a "chicken-and-egg problem":
  - To compute the PageRank of some page n, we need to know the PageRank of all other pages linking to it, which in turn might depend on n's PageRank.
- Fortunately, this recursive definition admits an iterative algorithm for computing all PageRank values in the graph. Starting with some initial values, each iteration computes the new PageRank for all pages. This process continues until a fixpoint is reached, meaning that the value for every single page does not change any more.
- Let us look at an example to see PageRank iterations in action.

Initial State



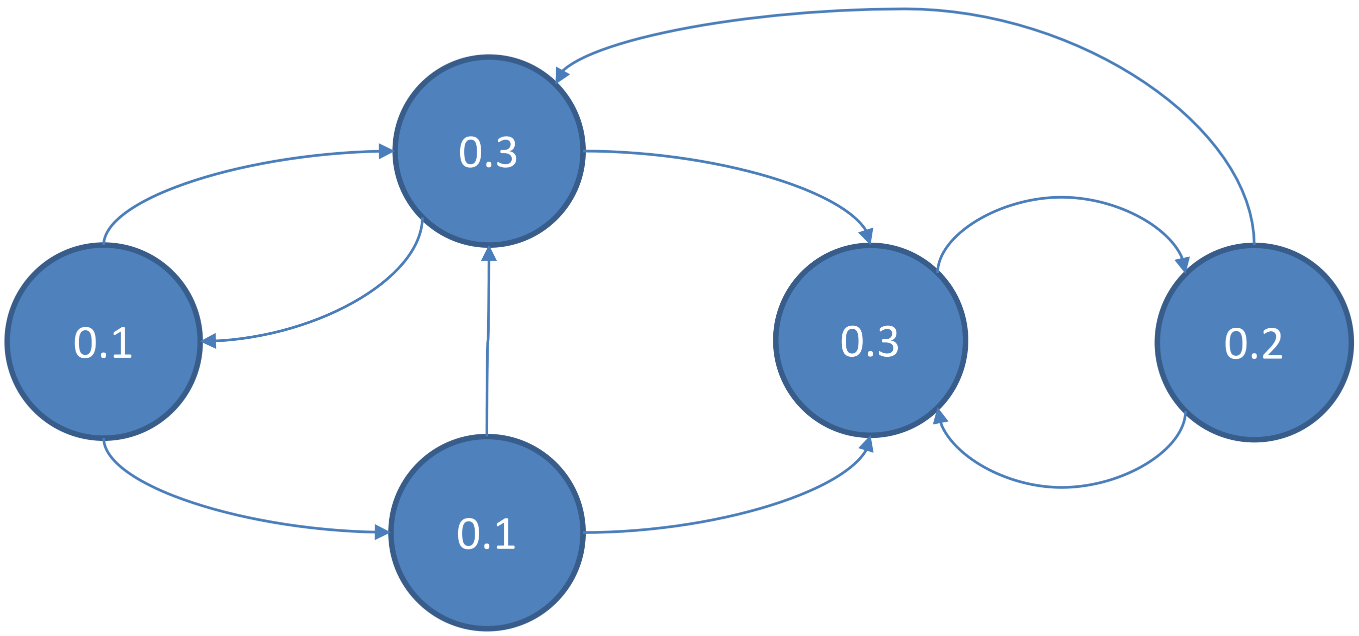
Assume that all PageRank values are initialized to 0.2. For simplicity we set α=1, i.e., the random surfer only follows links.

Iteration 1: PageRank Transfer



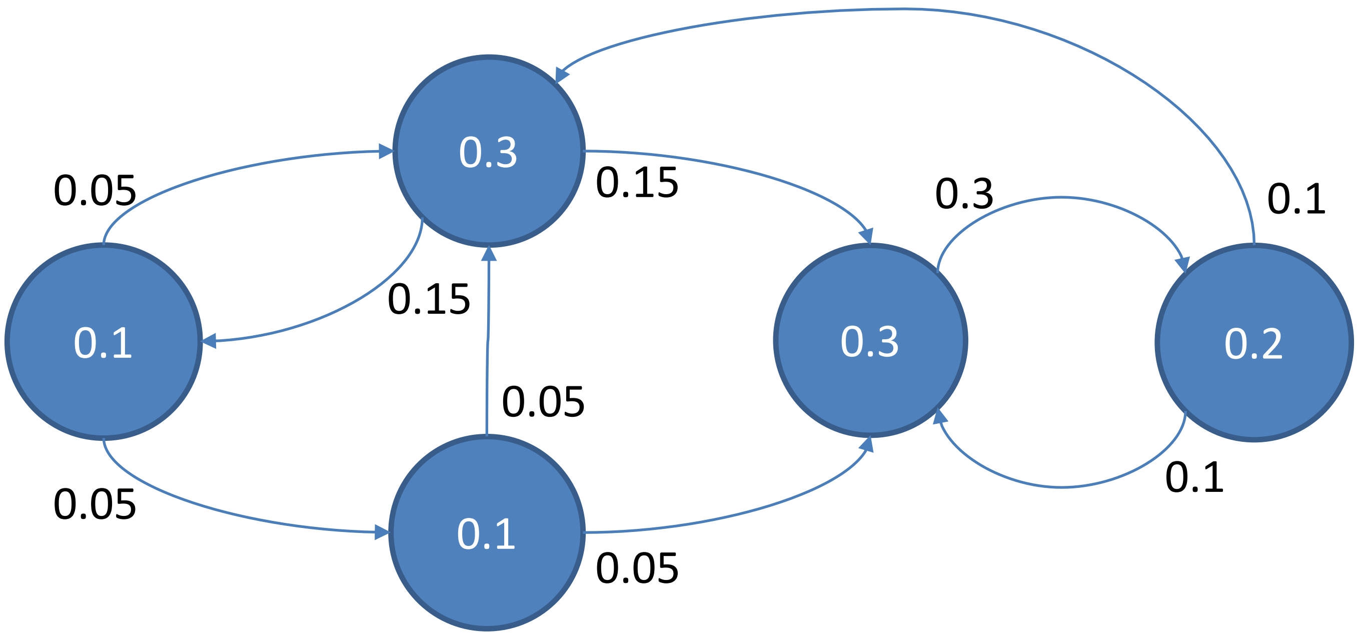
Since α=1, each page passes its full PageRank value, distributed equally over the outgoing links.

Iteration 1: Updated PageRank



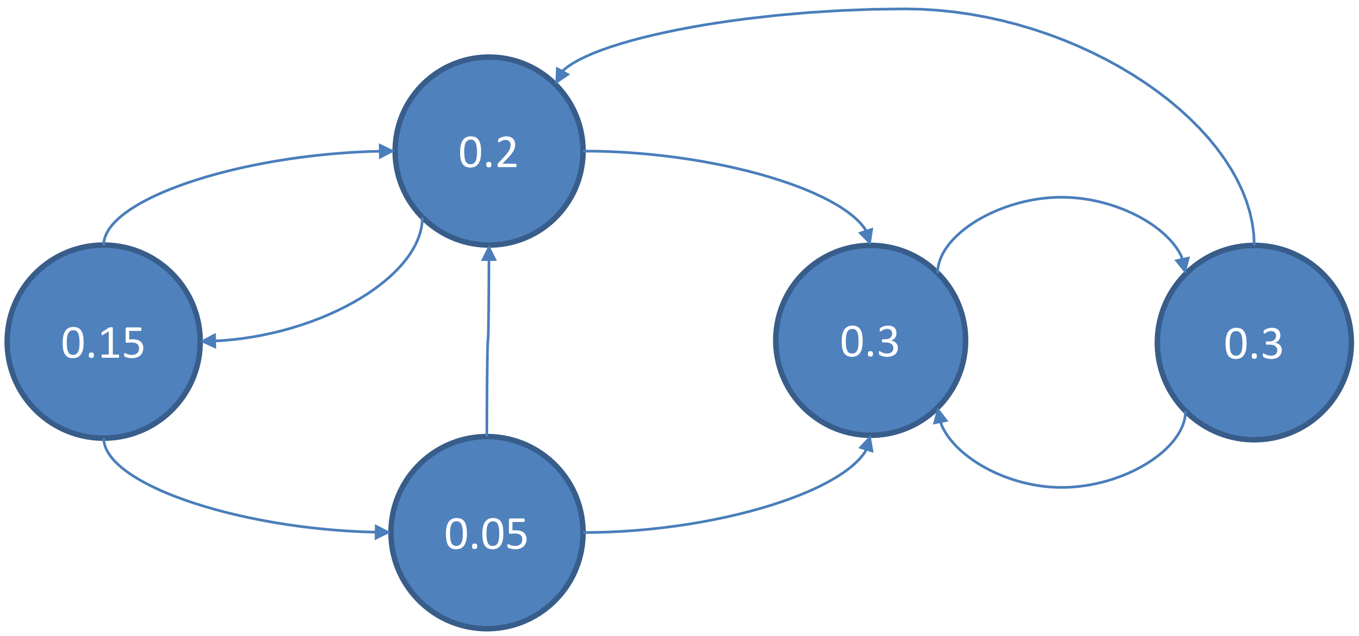
It is clearly visible how some pages receive more weight than others.

Iteration 2: PageRank Transfer



Since α=1, each page again passes its full PageRank value along the outgoing links.

Iteration 2: Updated PageRank



Already after two iterations, major properties of the algorithm show. First, from iteration to iteration, the PageRank of a page can oscillate between higher and lower values. E.g., the leftmost page changed from 0.2 to 0.1, then to 0.15. Over time, these changes become smaller as the values converge. Second, despite the oscillations, the general tendency is for some pages to accumulate larger values, while others drop.

PageRank using BFS

- Observing the steps during an iteration of PageRank, it becomes clear that there are two phases. In the first phase of the iteration, a page “sends out” fractions of its current PageRank along its *outgoing* edges. In the second phase, each page sums up the PageRank contributions along all its *incoming* edges.
- Given a page n’s current value P(n) and adjacency list, one can compute all its outgoing contributions.
- Adding the incoming contributions for some page m requires re-shuffling.
  - The term $(1-\alpha)/|V|$ remains constant throughout the computation. Both α and $|V|$ can be shared with all tasks.
- Interestingly, the computation pattern again matches BFS.

PageRank in MapReduce

- The algorithm is virtually identical to BFS, but an iteration has to keep track of the current PageRank value of each page:
  - Map processes a single vertex u, emitting the PageRank of u, divided by the outdegree, for each vertex v in u’s adjacency list.
  - Reduce collects the PageRank contributions from all inlinks of vertex v and then applies the formula.
- The driver program repeatedly calls the MapReduce program until (near) convergence, i.e., when all PageRank values stabilize.

MapReduce Code for a Single Iteration

```
// Map processes vertex n. Object N
// stores n's current PageRank and
// its adjacency list
map(id n, vertex N) {
  // Pass along the graph structure
  emit(n, N)

  // Compute contributions to send
  // along outgoing links
  p = N.pageRank / N.adjacencyList.size()
  for all m in N.adjacencyList do
    emit( m, p )
}
```

```
// Reduce receives the vertex object for vertex m and
// the PageRank contributions for all m's inlinks
reduce(id m, [p1, p2,…]) {
  s = 0
  M = NULL

  for all p in [p1, p2,…] do
    if isVertex(p) then
      // The vertex object was found: recover graph structure
      M = p
    else
      // A PageRank contribution from an inlink was found:
      // add it to the running sum.
      s += p

  M.pageRank = $(1-\alpha)/|V| + \alpha\cdot s$
  emit( m, M )
}
```

MapReduce Algorithm Analysis

- A careful look reveals that this program is structurally almost identical to the one for single-source shortest path. Hence it shares the same weaknesses caused by MapReduce’s inability to exploit repetitive structure in iterative programs. In each iteration,
  - …the entire graph is read from the distributed file system.
  - …the entire graph is transferred from Mappers to Reducers.
  - …the entire graph, with updated PageRank values, is written to the distributed file system.
- On the other hand, the PageRank program does not perform irrelevant computation. In contrast to single-source shortest path, all PageRank values have to be updated in every iteration.

PageRank in Spark

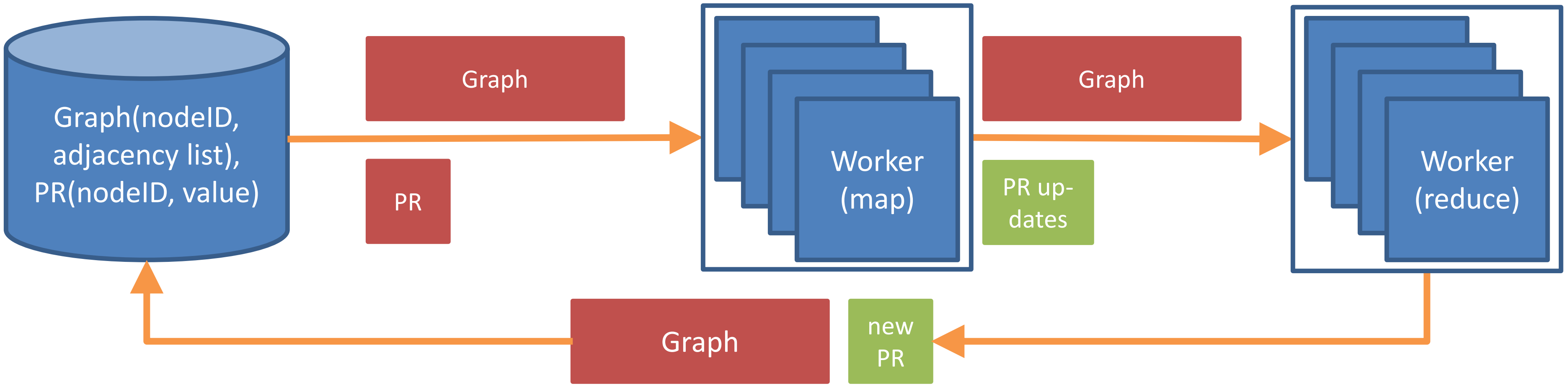
- We discuss the Spark Scala pseudo-code with RDDs. The version with DataSet is left as a voluntary challenge.
- The Spark program separates static data (the graph structure) from evolving data (the currently known PageRank of each vertex).
  - This leverages cached RDD partitions and avoids the shuffling of the adjacency list data from the MapReduce implementation.
  - On the downside, the new PageRank information has to be joined with the graph data, so that tuples of type (vertexID, adjacencyList) and (vertexID, PageRank) are joined on vertexID. This join normally requires shuffling, but careful co-partitioning avoids shuffling.

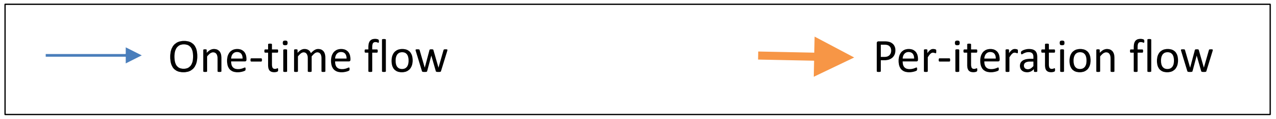
Spark Implementation

```
// Assume the input file contains in each line a vertex ID and its adjacency list. Function getAdjListPairData
// returns a pair of vertex ID and its adjacency list, creating a pair RDD. There is no special source node.
graph = sc.textFile(…).map( line => getAdjListPairData(line) )

// Some code here to make sure that graph has a Partitioner. This is needed for avoiding shuffling in the join below.

// Tell Spark to try and keep this pair RDD around in memory for efficient re-use
graph.persist()

// Create the initial PageRanks, using the page count |V|, which can be passed through the context.
// Function mapValues ensures that the same Partitioner is used as for the graph RDD.
PR = graph.mapValues( adjList => 1.0 / |V| )

// Function extractVertices returns each vertex id m in n's adjacency list as (m, n's PageRank / number of n's outlinks).
for (iterationCount <- 1 to k) {          // Use Accumulator instead to determine when last iteration is reached
  PR = graph.join( PR )
        .flatMap( (n, adjList, currentPRofN) => extractVertices(adjList, currentPRofN) )
        .reduceByKey( (x, y) => (x + y) ) // Add PageRank contributions
}
```

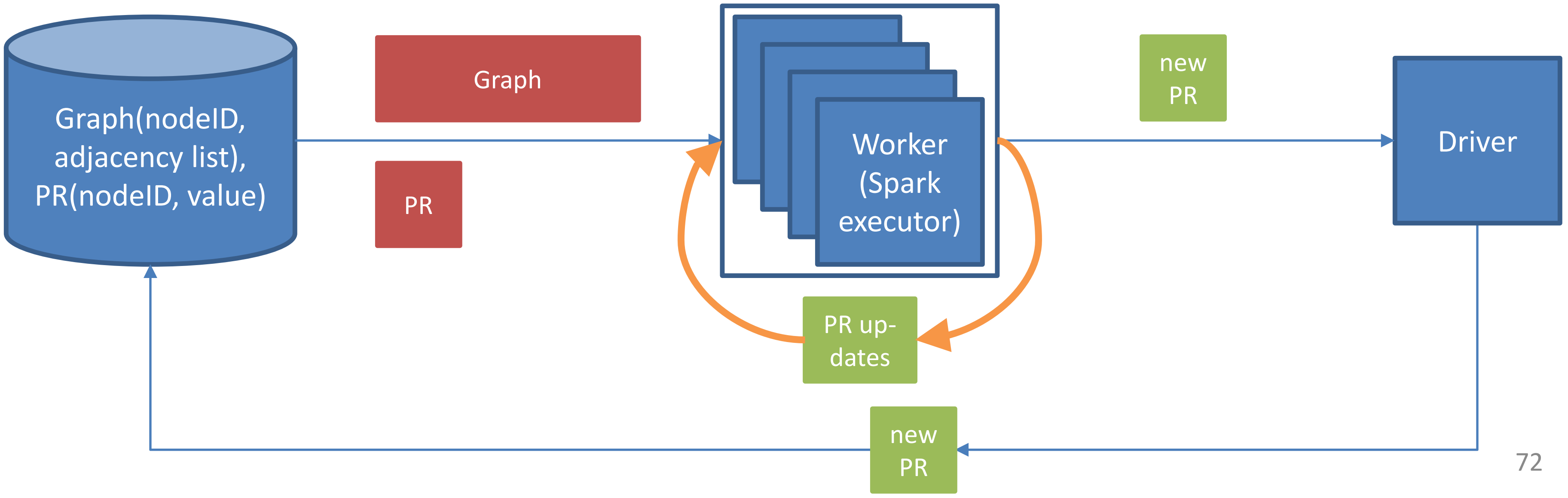
Let us look at the key difference between Spark and MapReduce.

Typical iterative job (PageRank) in Hadoop MapReduce:





Typical iterative job (PageRank) in Spark:



PageRank in a DBMS

- Assume the graph is stored in table Graph(id1, id2, outDegree) and the currently known PageRank values in table PageRank(id, PR), where id1, id2, and id are all vertex IDs, and outdegree is the number of vertices in the adjacency list of vertex id1. Then we can express the computation of an iteration as shown below.

```
newPR =

        SELECT G.id2, (1-alpha)*numPages + alpha*SUM(P.PR/G.outDegree)
        FROM Graph AS G, PageRank AS P
        WHERE G.id1 = P.id
        GROUP BY G.id2
```

Dangling Pages

- *Dangling pages*, i.e., pages that have no outgoing links, result in a loss of PageRank probability mass. Consider a dangling page x with PageRank P(x). During an iteration, P(x) is not passed along to another page and hence weight αP(x) just disappears. Stated differently, even if all PageRank values initially sum up to 1.0, due to dangling pages the total PageRank sum of the graph will decrease with every iteration.
- To avoid this problem, we need to correct for the missing probability mass. If the random surfer ends up on a page x without outgoing links, she cannot follow links and hence has to perform a random jump. We can model this mathematically by conceptually adding imaginary links from x to *every page in the graph*, including x itself. If we use δ to denote the total PageRank mass of dangling pages, then the corresponding formula for the PageRank of page n becomes
  - $P(n) = (1-\alpha)\frac{1}{|V|} + \alpha\left(\frac{\delta}{|V|} + \sum_{m \in L(n)} \frac{P(m)}{C(m)}\right)$
- For the iterative computation, this means that we have to compute the new value of δ in each iteration.

Dangling Pages Solution in MapReduce

- The formula computing the new PageRank in Reduce has to be modified to include the term with variable δ.
- Unfortunately, it is not supported to compute some value and also read it out in the *same* MapReduce job. We next discuss multiple possible solutions in more detail.
- Solution 1: add a separate phase to each iteration to compute δ.
  - During an iteration, first execute a MapReduce job that computes δ. This is a simple global aggregation job, summing up PageRank values for all dangling nodes.
  - Then pass the newly computed δ as a parameter to the modified MapReduce program that updates all PageRanks using the new formula with δ.
- Can we avoid the extra job in each iteration?

Alternative Solutions

- Solution 2: Merge computation of δ into previous Reduce phase.
  - Instead of computing δ in a separate job in the beginning of iteration (i+1), it could already be computed at the end of iteration i. To do so, we use a global counter that is updated by all Reduce calls for dangling nodes. The counter value can be read by the driver, and then passed into the next job's context.
- Solution 3: dummy page.
  - For a dangling page n, the Map function emits (dummy, n's PageRank). The Reduce call for the dummy node then adds up all contributions. The driver has to read this output and pass it into the next job's context.
- Solution 4: order inversion.
  - Notice that in Map, the page's PageRank has the old value computed in the previous iteration i. Reduce computes the new value, i.e., the one output in iteration (i+1). Hence we can apply the order inversion pattern to make sure each Reducer receives the old PageRank values of all dangling pages. Each Reducer can then compute δ right before executing any of the "normal" Reduce calls.

Dangling Pages Solution in Spark

- The Spark solutions mirror those in MapReduce.
  - We can compute the dangling PageRank mass in a separate step.
  - Instead of a global counter, we can use an Accumulator to compute the dangling mass in iteration i, then use it in iteration (i+1).
  - We can also add the dummy page to the graph RDD. Its value can be accessed using `lookup`.
- All these options result in an action: the separate computation step uses a "global" aggregate on the RDD (not by key!), Accumulator access is an action, and so is the lookup operation.
  - This creates a complex mix of transformations and actions in each loop iteration. Consider the dangling PageRank-related action in iteration i. It forces immediate evaluation. This evaluation depends on the PageRanks and the dangling mass in iteration (i-1). Those in turn depend on iteration (i-2), and so on.
  - Challenge question: How many times is each iteration executed? Does the use of `persist` change this behavior?

Dangling Pages Solution in a DBMS

- In SQL, we simply create an intermediate table with the dangling PageRank mass and hope that the optimizer will find an efficient computation strategy.
  - Note that for dangling pages n, there is no tuple with id1=n in Graph. Hence we find all dangling pages by checking for page ids that are not among the id1 values in Graph.

```
dangling =
        SELECT SUM(PR)
        FROM PageRank
        WHERE id NOT IN
                (SELECT id1 FROM Graph)

newPR =
        SELECT G.id2, (1-alpha)*numPages + alpha*(dangling / numPages + SUM(P.PR/G.outDegree))
        FROM Graph AS G, PageRank AS P
        WHERE G.id1 = P.id
        GROUP BY G.id2
```

Number of Iterations

- In theory, the computation iterates until the fixpoint is reached, i.e., none of the PageRank values changes any more from one iteration to the next. In practice it suffices to obtain approximate values, hence the computation can stop as soon as all PageRank values change only “very little.”
  - It is often reasonable to stop iterating as soon as none of the PageRank values changes by more than 1% or 0.1%. This threshold for *relative* change would be computed for each page n as |new(n) – old(n)| / old(n), where old(n) and new(n) are the PageRank of n in previous and current iteration, respectively.
  - Alternatively, one could use an *absolute* change threshold, computed as |new(n) – old(n)|. However, it is often difficult to choose an appropriate threshold, because PageRank values can vary by multiple orders of magnitude in large graphs.
- The easiest way to check for convergence is by using a global counter or Accumulator that determines for how many nodes the PageRank change was greater than the threshold. This counter is updated in the Reduce function (or the corresponding Spark function), where both old and new PageRank of a page are available. The driver program then checks the counter value in order to decide about running another iteration.
- Depending on graph size and structure, and the convergence threshold selected, the number of iterations can vary widely. In the paper that proposed PageRank, it was reported that convergence was achieved after 52 iterations for a graph with 322 million edges.

Summary

- Large graphs tend to be sparse and hence are often stored in adjacency list format or as a set of edges. This representation enables per-vertex computation in a single round, which can pass information along outgoing edges to all direct neighbors.
  - It is possible to extend this capability by pre-computing other data structures, e.g., the list of neighbors within a certain distance, to push information directly to nodes further away.
- Computation along incoming edges requires shuffling.
- The driver program controls execution of iterations, until a stopping condition is met.
- For some problems, the most efficient or most elegant sequential algorithms rely on a "centralized" data structure. These algorithms are often difficult to parallelize directly. Parallel solutions need to be developed "from scratch."
- Iterative algorithms are common in practice and can be implemented easily in MapReduce and Spark. However, MapReduce's lack of persistent in-memory data results in inefficiencies due to costly data transfer.

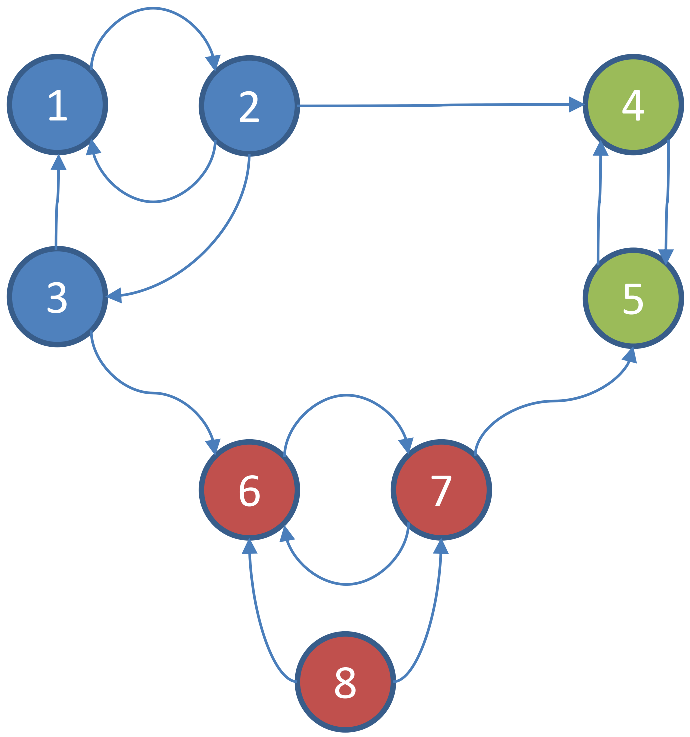
Summary (Cont.)

- The complexity of an iteration of each of the presented algorithms is linear in graph size. Hence these algorithms can scale to fairly large graphs.
  - For other problems, the complexity can be different.
- Despite the comparably low algorithm complexity, it is wasteful to read and write the entire graph in each iteration. Efficiency can be improved by partitioning the graph and keeping partitions on worker nodes, as shown on the next pages. While Spark RDDs support this, MapReduce does not.
  - A good partitioning keeps highly connected parts of the graph together, separating it along less connected nodes. This minimizes communication cost for algorithms that push data along graph edges. Finding a balanced min-cut style graph partitioning is a hard problem.
- Iterative computations on large data raise the issue of numerical stability. For example, for a graph with millions of nodes, the PageRank of an individual page might be so small that it underflows standard floating point representation. In that case, one has to carefully analyze the problem to determine if a number type of greater precision is needed or if a numerically more robust algorithm can be used. For instance, it might be possible to work with logarithm-transformed numbers. Since $\log(x \cdot y)=\log(x)+\log(y)$, one can replace multiplication by addition of log-transformed numbers.

Side-Note: Graph Partitions versus Remote Communication

- Consider the PageRank algorithm in Spark, where the graph RDD is partitioned over the different tasks. After loading the graph before the first iteration, no further graph data movement occurs.
- However, PageRank values still change and need to be passed along outgoing edges, then aggregated along incoming edges.
- Consider the example partitioning below, showing the adjacency lists managed by each task.

Partitions versus Communication



Graph partitions:

Partition A:
- node 1: 2
- node 2: 1, 3, 4
- node 3: 1, 6

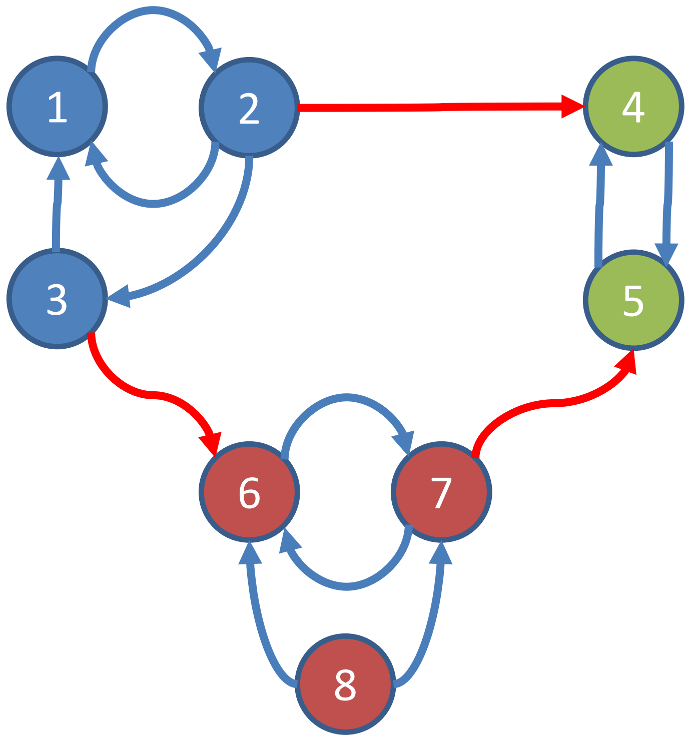
Partition B:
- node 4: 5
- node 5: 4

Partition C:
- node 6: 7
- node 7: 5, 6
- node 8: 6, 7

Partitions versus Communication

- During each iteration of the PageRank computation, each vertex sends its contributions along the outgoing edges. For destination vertices in the same partition, data transfer is local (indicated by a blue arrow).
- To avoid costly network transfer, the graph should be partitioned such that the number of edges connecting vertices in different partitions is minimized. In the example, only three edges (shown in red) require data transfer to a different task. After receiving the incoming contributions, for each vertex the new PageRank can be computed.

Efficient Iterative Processing



PageRank mass transfer:

→ local transfer

→ transfer to a different task

CYK 1

- True, false, or not enough information?
- Sparse graphs with millions of vertices should be stored as an adjacency *list*, not an adjacency *matrix.*

CYK 2

- True, false, or not enough information?
- The MapReduce program for single-source shortest path only works correctly if all edges have the same weight.

CYK 3

- True, false, or not enough information?
- The MapReduce program for single-source shortest path only works correctly if all edges have non-negative weight.

CYK 4

- True, false, or not enough information?
- Each iteration of the MapReduce program for single-source shortest path will read data from HDFS and then write data to HDFS.

CYK 5

- True, false, or not enough information?
- By using global counters or Accumulators, we can determine if the PageRank algorithm converged. There is no need for a separate job to detect convergence.

References

- S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems, Elsevier, pages 107-117, 1998.
  - https://scholar.google.com/scholar?cluster=16140207188598694220&hl=en&as_sdt=0,22&as_vis=1