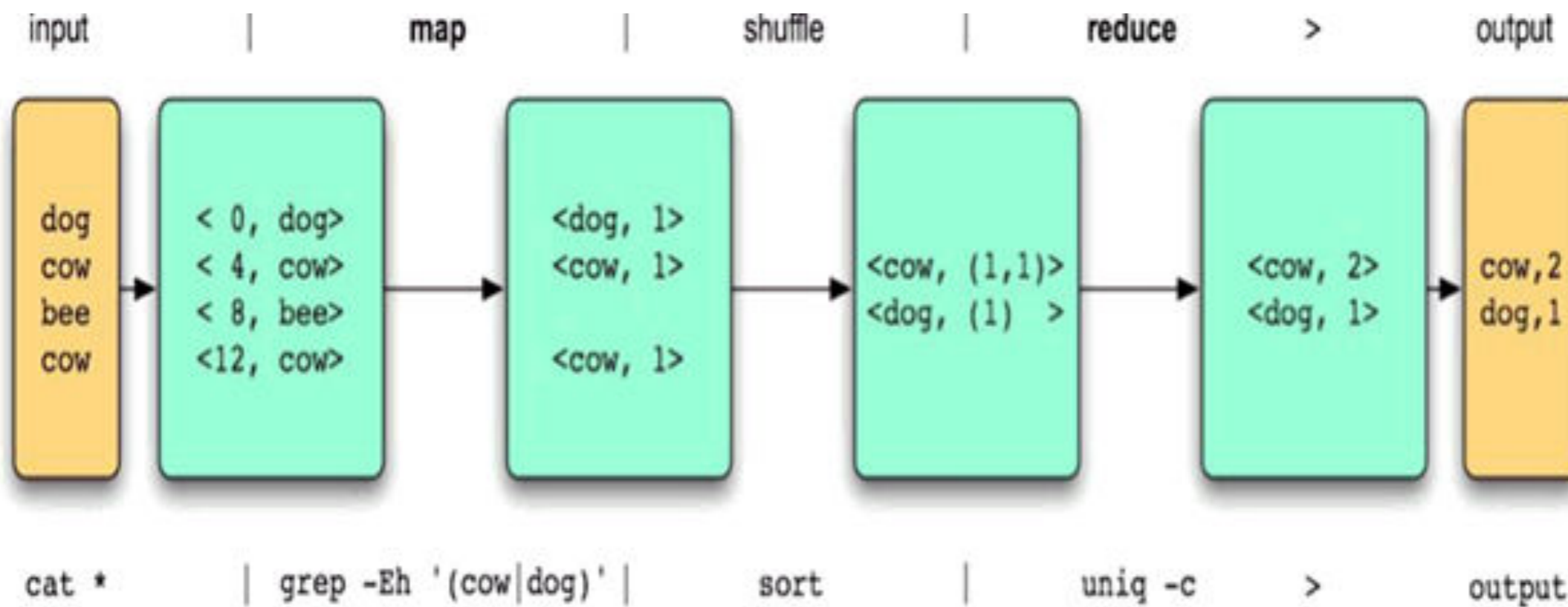


MapReduce Patterns



Intermediate Data

- **Written locally**
- **Transferred from mappers to reducers over network**
- **Issue**
 - Performance bottleneck
- **Solution**
 - Use combiners
 - Use **In-Mapper Combining**

Original Word Count

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term t ∈ doc d do
4:       EMIT(term t, count 1)

1: class REDUCER
2:   method REDUCE(term t, counts [c1, c2, ...])
3:     sum ← 0
4:     for all count c ∈ counts [c1, c2, ...] do
5:       sum ← sum + c
6:     EMIT(term t, count sum)
```

- How many intermediate keys per mapper?
- How can we improve this?
- Is it a “real” improvement?

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     H ← new ASSOCIATIVEARRAY
4:     for all term t ∈ doc d do
5:       H{t} ← H{t} + 1
6:     for all term t ∈ H do
7:       EMIT(term t, count H{t})
```

- Custom local aggregator
- Coding overhead
- Is it a “real” improvement?

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in \text{doc } d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:       EMIT(term  $t$ , count  $H\{t\}$ )
```

- Custom local aggregator
- Coding overhead
- Is it a “real” improvement?

In-Mapper Combining Analysis

- **Advantages:**

- Complete local aggregation control (how and when)
- Guaranteed to execute
- Direct efficiency control on intermediate data creation
- Avoid unnecessary objects creation and destruction (before combiners)

- **Disadvantages:**

- Breaks the functional programming background (state)
- Potential ordering-dependent bugs
- Memory scalability bottleneck (solved by memory foot-printing and flushing)

Matrix Generation

- **Common problem:**

- Given an input of size N , generate an output matrix of size $N \times N$

- **Example:** word co-occurrence matrix

- Given a document collection, emit the bigram frequencies

"Pairs"

```

1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:       for all term  $u \in \text{NEIGHBORS}(w)$  do
5:         EMIT(pair ( $w, u$ ), count 1)

```

▷ Emit count for each co-occurrence

```

1: class REDUCER
2:   method REDUCE(pair  $p$ , counts [ $c_1, c_2, \dots$ ])
3:      $s \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $s \leftarrow s + c$ 
6:     EMIT(pair  $p$ , count  $s$ )

```

▷ Sum co-occurrence counts

- We must use custom key type
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?
- Keys space?

"Stripes"

```

1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:        $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:       for all term  $u \in \text{NEIGHBORS}(w)$  do
6:          $H\{u\} \leftarrow H\{u\} + 1$ 
7:       EMIT(Term  $w$ , Stripe  $H$ )
  
```

▷ Tally words co-occurring with w

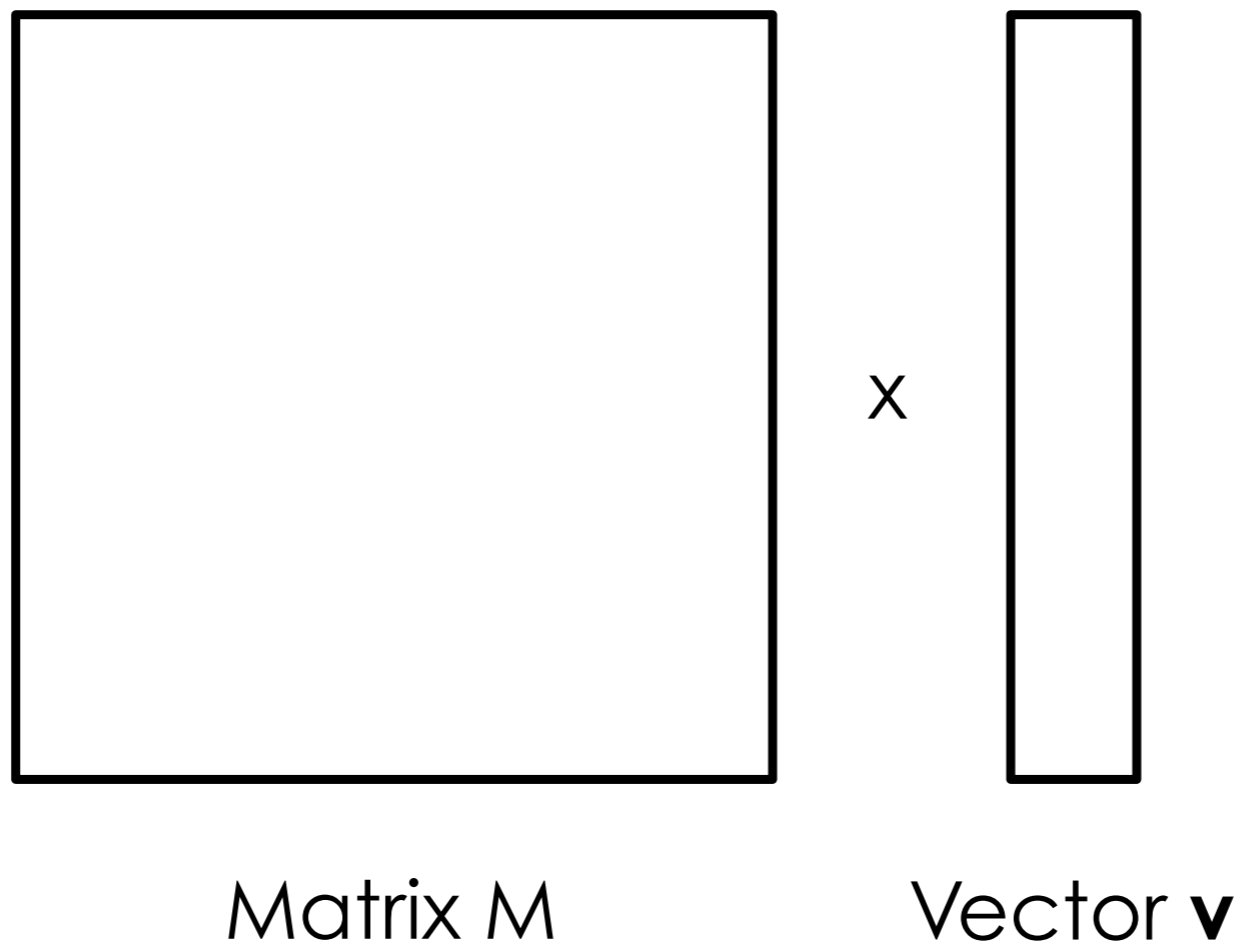
```

1: class REDUCER
2:   method REDUCE(term  $w$ , stripes [ $H_1, H_2, H_3, \dots$ ])
3:      $H_f \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:     for all stripe  $H \in \text{stripes } [H_1, H_2, H_3, \dots]$  do
5:       SUM( $H_f, H$ )
6:     EMIT(term  $w$ , stripe  $H_f$ )
  
```

▷ Element-wise sum

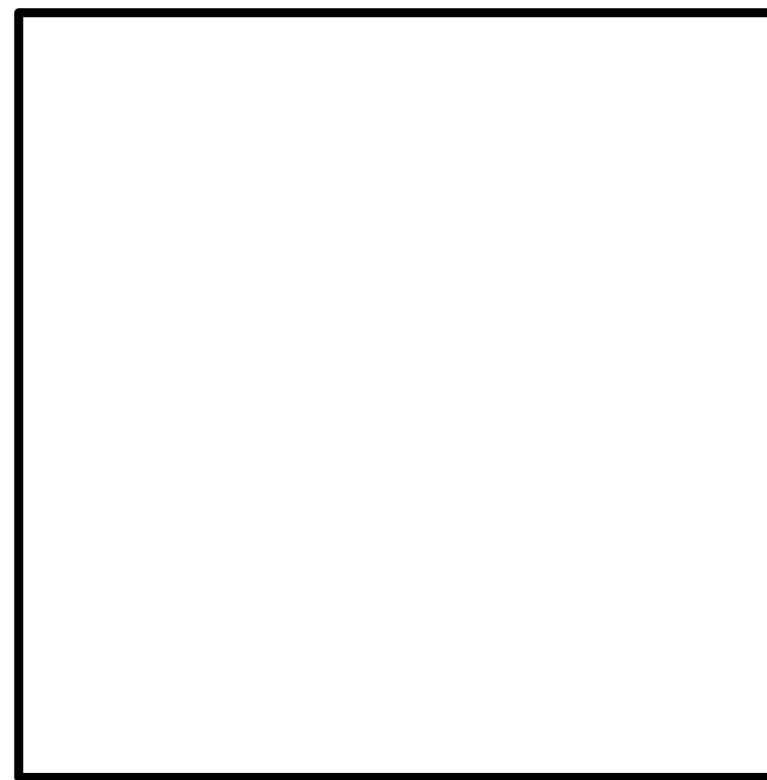
- We must use custom key and value types
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?
- Keys space?

Matrix Vector Multiplication



- The matrix does not fit in memory
 - 1 case: vector v fits in memory
 - 2 case: vector v does not fit in memory

Vector fits in memory



Matrix M

\times



Vector v

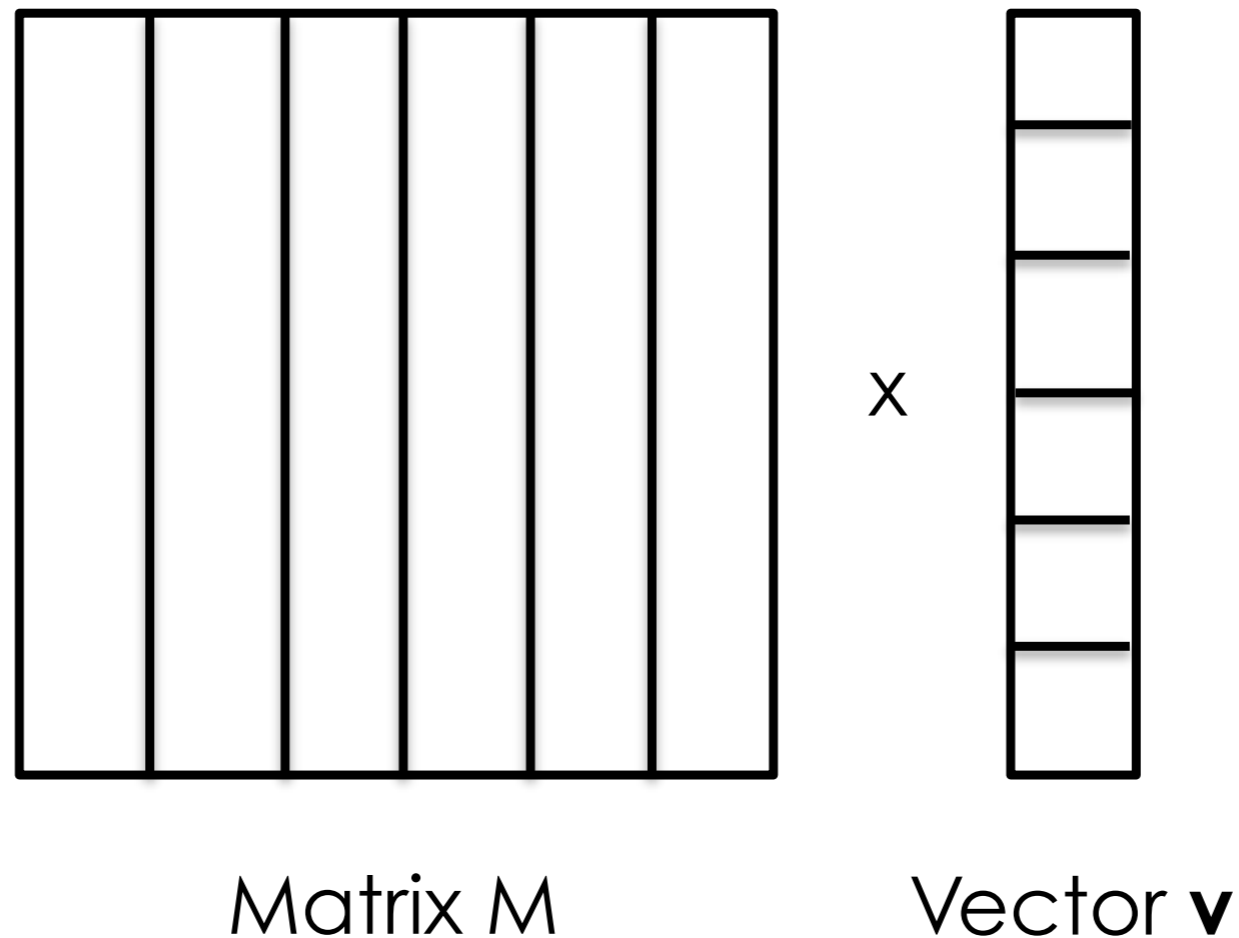
- Map

- input = (*, chunk of matrix M)
- vector v read from memory
- output = (i, $m_{ij}v_j$)

- Reduce

- sum up all the values for the given key i

Vector does not in memory



- Divide the vector in equal-sized subvectors that can fit in memory
- According to that, divide the matrix in stripes
- Stripe i and subvector i are independent from other stripes/subvectors
- Use the previous algorithm for each stripe/subvector pair

Relational Databases

A_1	A_2	A_3	A_4	A_5

Schema

Tuple

Relation R

- SELECTION: Select from R tuples satisfying condition C
- PROJECTION: For each tuple in R , select only certain attributes
- UNION, INTERSECTION, DIFFERENCE: Set operations on two relations with same schema
- NATURAL JOIN
- GROUPING and AGGREGATION

- MAP: Each tuple t , if condition C is satisfied, is outputted as a (t, t) pair
- REDUCE: Identity

- MAP: For each tuple t , create a new tuple t' containing only projected attributes. Output is (t', t') pair
- REDUCE: Coalesce input $(t', [t' t' t' t'])$ in output (t', t')

Union, Intersection, Difference

- MAP: Each tuple t is outputted as a (t, t) pair
- REDUCE: For each key t , there will be 1 or 2 values t . Coalesce them in a single output (t,t)

- MAP: Each tuple t is outputted as a (t, t) pair
- REDUCE: For each key t , there will be 1 or 2 values t . If 2 values, coalesce them in a single output (t,t) , else ignore

- MAP: For each tuple t in R , produce $(t, "R")$. For each tuple t in S , produce $(t, "S")$.
- REDUCE: For each key t , there will be 1 or 2 values t . If 1 value, and being "R", output (t,t) , else ignore

Natural Join

- We have two relations $R(A,B)$ and $S(B,C)$. Find tuples that agree on B components
- MAP: For each tuple (a,b) from R , produce $(b,("R",a))$. For each tuple (b,c) from S , produce $(b,("S",c))$.
 - REDUCE: For each key b , there will a list of values of the form $(("R",a)$ or $(("S",c)$. Construct all pairs and output them with b .

Grouping and Aggregation

- We have the relation $R(A,B,C)$ and we **group-by** A and **aggregate** on B .
- **MAP:** For each tuple (a,b,c) from R , output (a,b) .
Each key a represents a group.
 - **REDUCE:** Apply the aggregation operator to the list of b values associate with group a , producing x .
Output (a,x) .

Graph Algorithms

- $G = (V, E)$, where
 - V represents the set of vertices (nodes)
 - E represents the set of edges (links)
 - Both vertices and edges may contain additional information
- Graph algorithms typically involve:
 - Performing computations at each node: based on node features, edge features, and local link structure
 - Propagating computations: “traversing” the graph
- Key questions:
 - How do you represent graph data in MapReduce?
 - How do you traverse a graph in MapReduce?

- **Adjacency Matrix**

- Represent a graph as an $n \times n$ square matrix M
- $n = |V|$
- $M_{ij} = 1$ means a link from node i to j

- **Advantages:**

- Amenable to mathematical manipulation
- Iteration over rows and columns corresponds to computations on outlinks and inlinks

- **Disadvantages:**

- Lots of zeros for sparse matrices
- Lots of wasted space

- **Adjacency List**

- Take adjacency matrices...
- and throw away all the zeros

- **Advantages:**

- Much more compact representation
- Easy to compute over outlinks

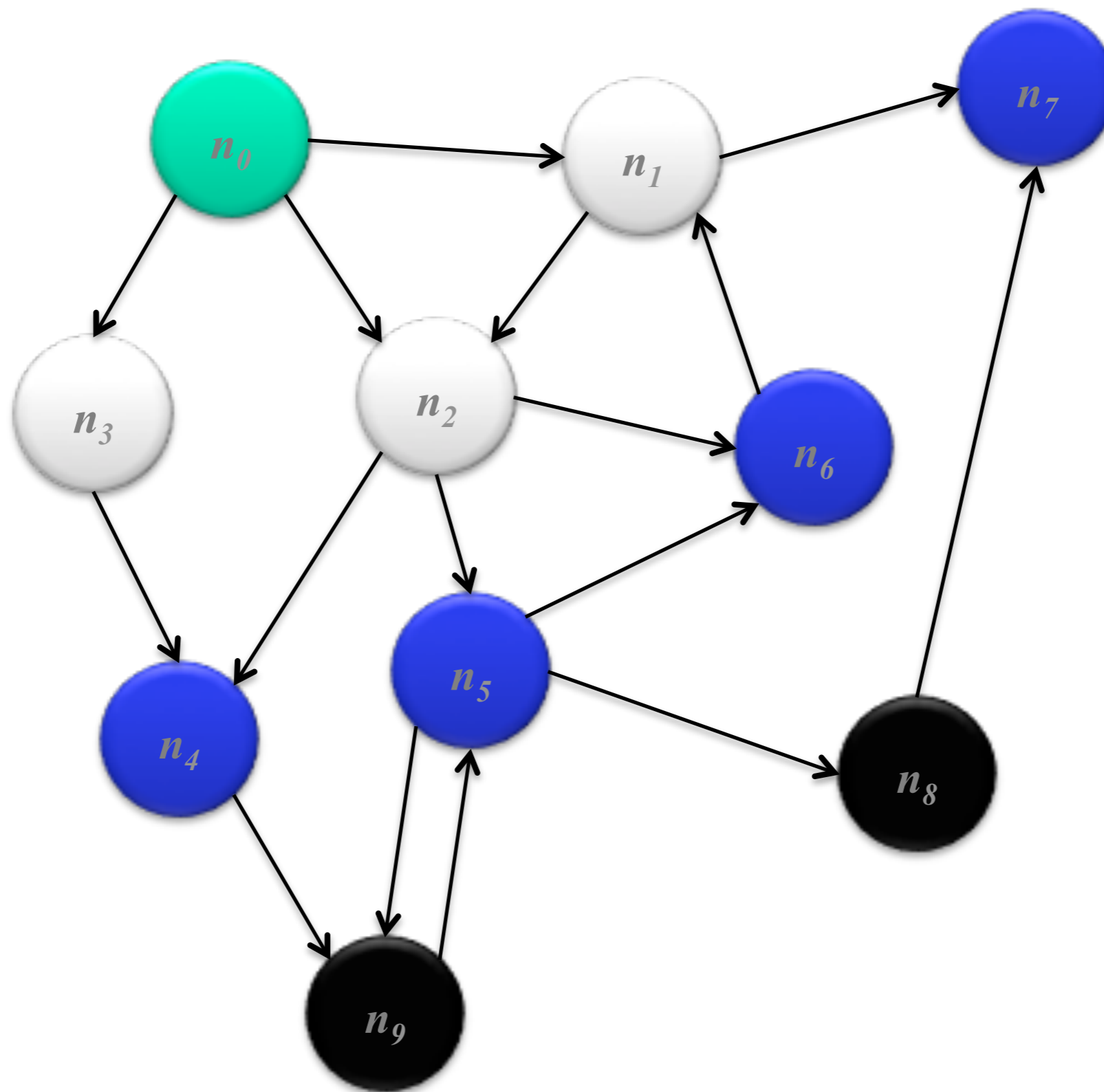
- **Disadvantages:**

- Much more difficult to compute over inlinks

Shortest Path

- Consider simple case of equal edge weights
- 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 ,
 $DISTANCETO(n) = 1 + \min(DISTANCETO(m), m \in M)$

Shortest Path



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, $d = \text{infinity}$

- Mapper:

- m Selects minimum distance path for each reachable node
- Additional bookkeeping needed to keep track of actual path
- 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

Details

- 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

```

1: class MAPPER
2:   method MAP(nid  $n$ , node  $N$ )
3:      $d \leftarrow N.DISTANCE$ 
4:     EMIT(nid  $n$ ,  $N$ )
5:     for all nodeid  $m \in N.ADJACENCYLIST$  do
6:       EMIT(nid  $m$ ,  $d + 1$ )

```

▷ Pass along graph structure
 ▷ Emit distances to reachable nodes

```

1: class REDUCER
2:   method REDUCE(nid  $m$ , [ $d_1, d_2, \dots$ ])
3:      $d_{min} \leftarrow \infty$ 
4:      $M \leftarrow \emptyset$ 
5:     for all  $d \in$  counts [ $d_1, d_2, \dots$ ] do
6:       if ISNODE( $d$ ) then
7:          $M \leftarrow d$ 
8:       else if  $d < d_{min}$  then
9:          $d_{min} \leftarrow d$ 
10:     $M.DISTANCE \leftarrow d_{min}$ 
11:    EMIT(nid  $m$ , node  $M$ )

```

▷ Recover graph structure
 ▷ Look for shorter distance
 ▷ Update shortest distance

Recipe

- 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