ENSF 612
Lecture MapReduce Algorithms

Dr. Gias Uddin,
Electrical and Software Engineering,
University of Calgary.
https://giasuddin.ca/

Topics

- Relational operations
- Matrix operations
- Graph algorithms

Relational operations

- Can do SQL-like operations using MapReduce
- Consider *select* and *join* operations as examples
- Recap some terminology first
- **Relation (*R*)** – a table with names of its columns
- **Attributes** – names of columns
- **Tuples** – rows of a relation
- **Schema** – collection of attributes
  - denoted by $R(A_1,A_2,\ldots A_N)$ where the As are attributes

Relational operators – cont'd

- How can we implement *select* using MapReduce?

  *select * from EMPLOYEES where salary>50,000*

- Apply condition *C* on each tuple in a relation
- Can implement using only a map step
- Input for **Map** is <*t,t*> where t is a tuple in the relation
- **Map**(<*t,t*>) outputs <*t,t*> if *C* is satisfied; nothing if not.
- **Reduce** is an identity function – simply emits input from **Map**
- Can either use key or value from **Reduce** as result

Relational operators– cont'd

- How to implement a *join* using MapReduce?
- **Compute the natural join $R(A,B) \bowtie S(B,C)$**
  - *R* and *S* are each stored in files
  - Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Relational operators– cont'd

- **Use a hash function *h* from B-values to *1…k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes**
  - send each k-v pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process**
  - matches all the pairs *(b,(a,R))* with all *(b,(c,S))*
  - outputs *(a,b,c)*.

Graph algorithms

- Graph algorithms have many applications
  - Social network analysis
  - Shortest path in road networks
- Consider the breadth first search (BFS) algorithm
- Traverse a graph in a breadth first manner
- Many applications – e.g., calculating shortest path
- How can we implement BFS using MapReduce?

Graph algorithms – cont'd

- BFS terminology
- **Graph** consists of **nodes**
- Two nodes may be connected by an **edge**
- Designate one node as *source*
- Nodes have **colours**
  - **White (*W*)**– node has not been visited
  - **Grey (*G*)** – node visited but its kids not visited, i.e., *W*
  - **Black (*B*)** – node visited all its kids are *G*
- **Goal – start from source -stop when no G nodes**

BFS

Wikipedia:
https://en.wikipedia.org/wiki/Breadth-first_search

Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'[1]), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
It uses the opposite strategy of depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.[2]

Pseudocode [edit]

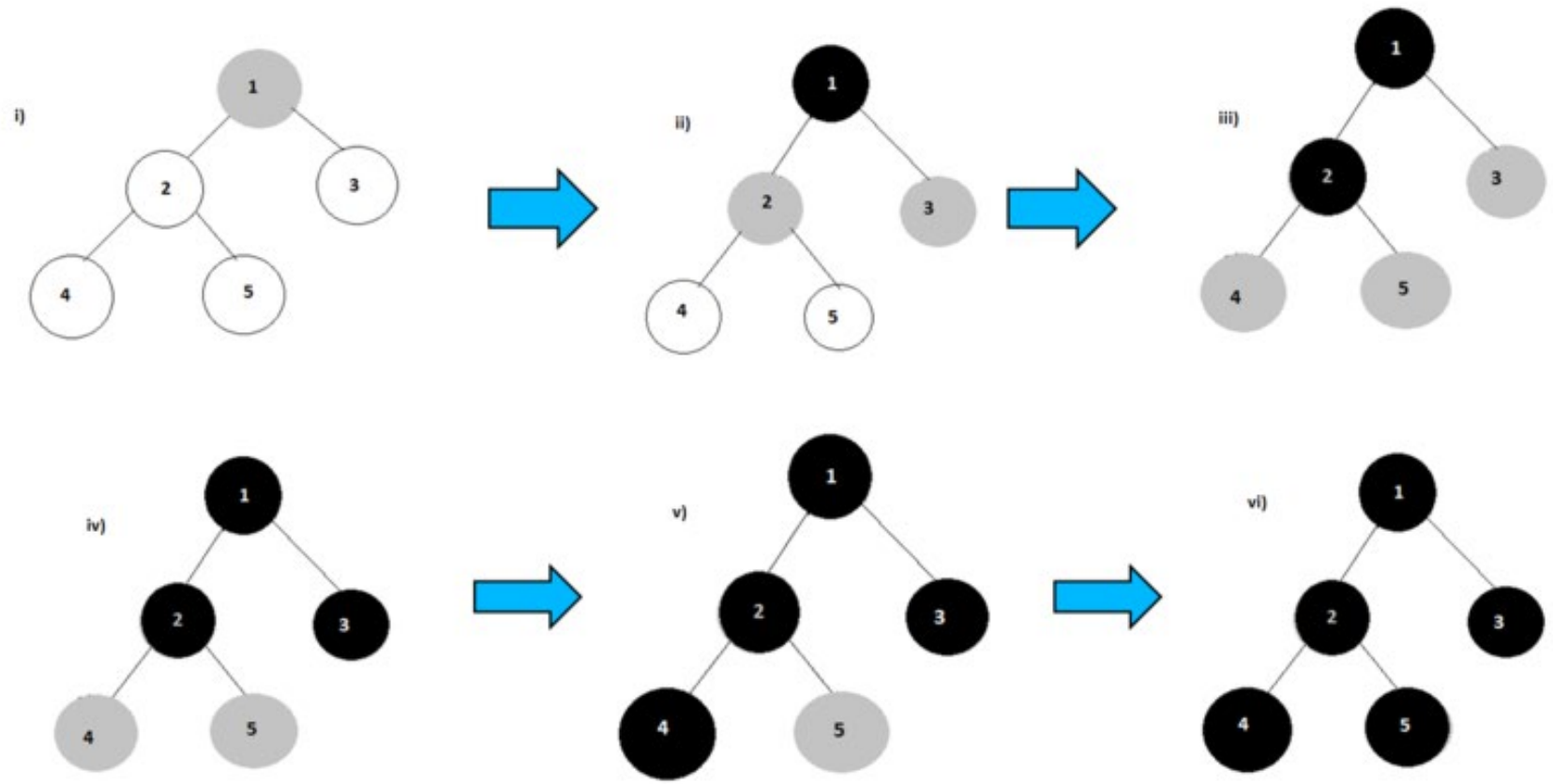
Input: A graph *G* and a *starting vertex root* of *G*

Output: Goal state. The *parent* links trace the shortest path back to *root*[7]

```
1  procedure BFS(G, root) is
2      let Q be a queue
3      label root as discovered
4      Q.enqueue(root)
5      while Q is not empty do
6          v := Q.dequeue()
7          if v is the goal then
8              return v
9          for all edges from v to w in
G.adjacentEdges(v) do
10             if w is not labeled as discovered
then
11                 label w as discovered
12                 Q.enqueue(w)
```

Queue – First in First Out (FIFO)
Stack – Last in First Out (LIFO)

BFS



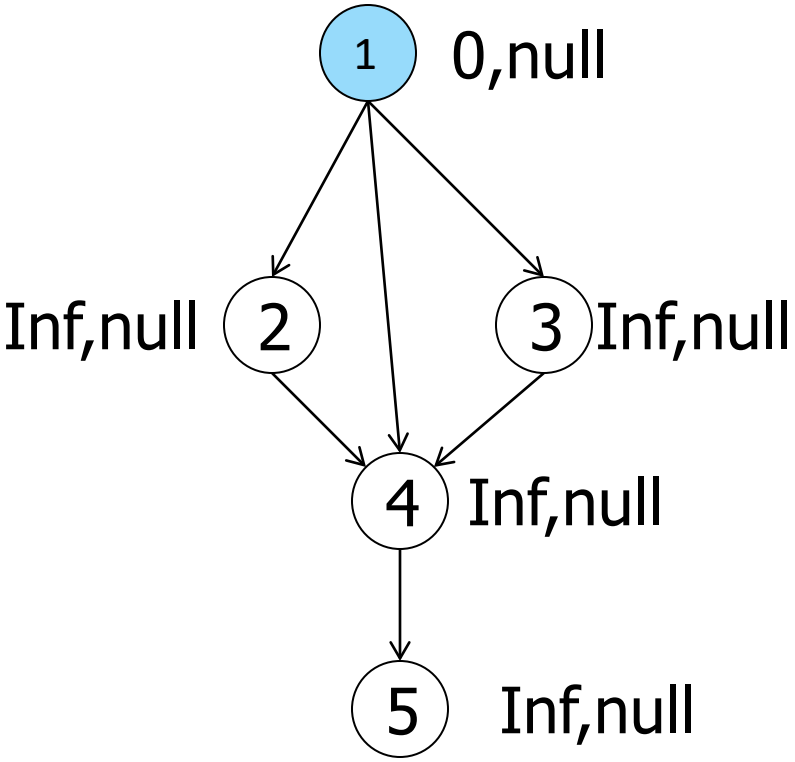
BFS

- Using BFS to get shortest path from *source* to nodes
- Need a couple of extra variables for a node
- **Score** – in how many hops can I get to *source* node
  - Initially 0 for *source* and infinity for others
- **Parent** – from which parent did I visit this node
  - Initially null for all nodes
- How does the algorithm work?

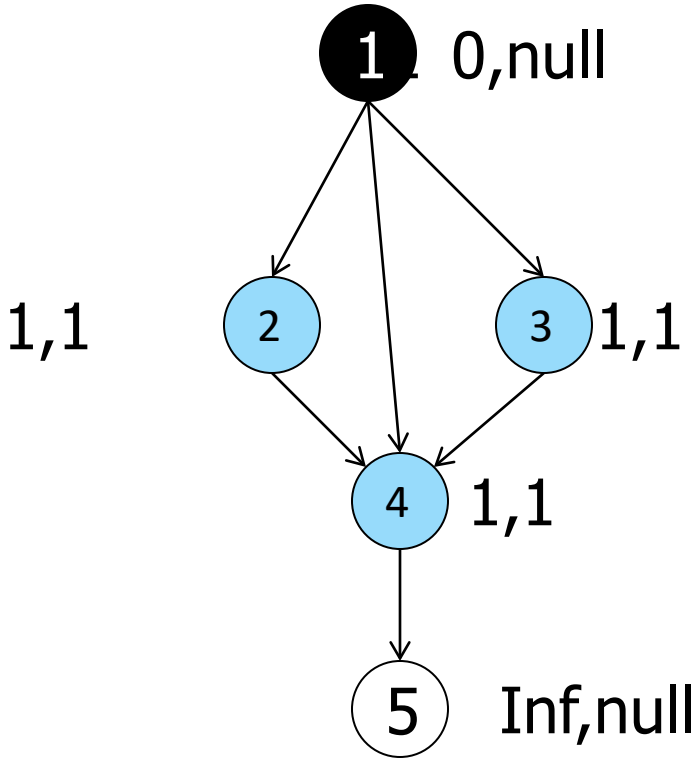
BFS

- Start from *source (*process all *"G"* nodes*)*
  - Mark from "*G*" to "*B*"
  - Visit unvisited (i.e., "*W*") kids - mark such kids from "*W*" to "*G*"
  - Set *parent* of kids as the node from which visited, i.e., *source*
  - Set *score* of kids as score of *parent* + 1, i.e., 0+1
- Repeat process for all the new "*G*" nodes
- Keep going till all nodes reachable from *source* are "*B*"
- We visit nodes with *score K+1* from those with *score K*
- Score of destination gives shortest hop to source
- Can also trace shortest path by using *parent* variable

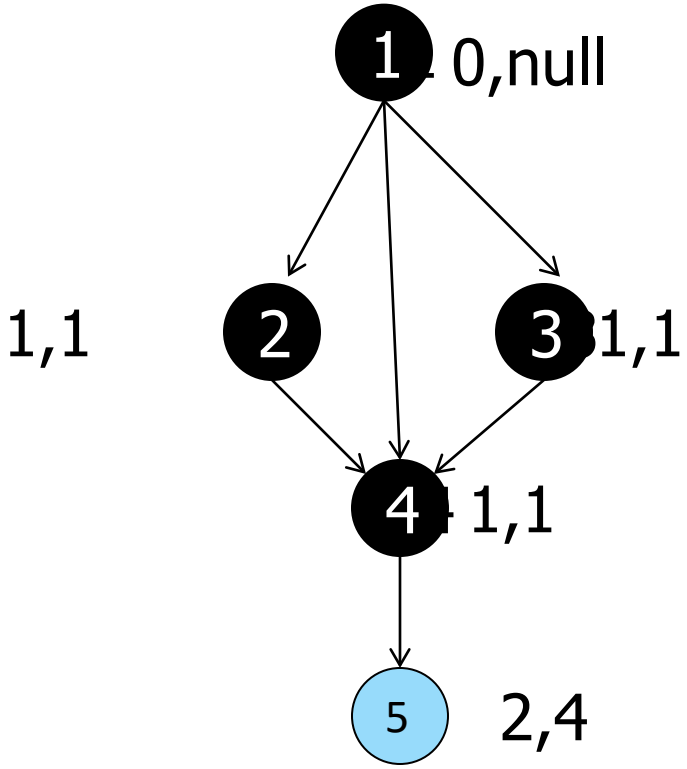
BFS



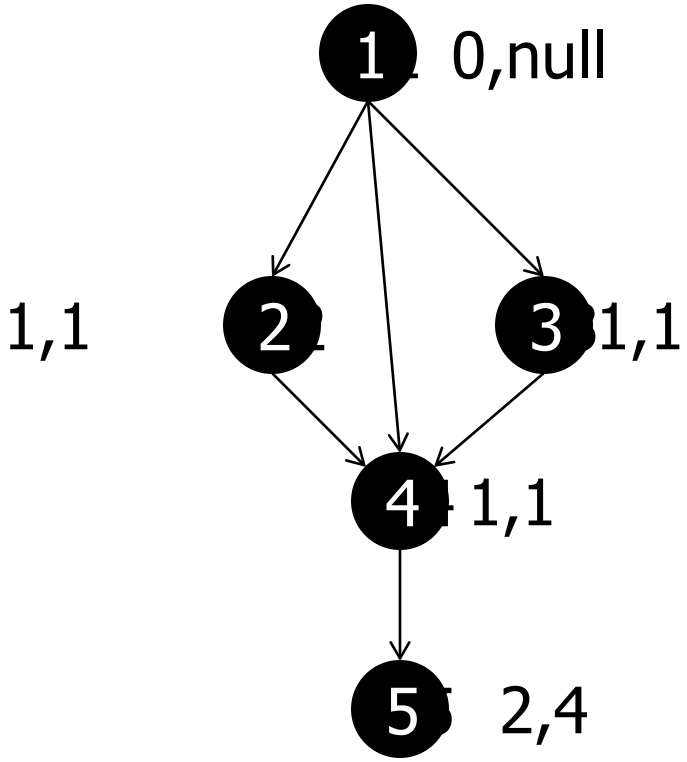
BFS



BFS



BFS



Finding the Shortest Path using BFS in MapReduce

Intuition

1. DistanceTo (startNode) = 0
2. For all nodes *n* directly reachable from startNode, DistanceTo (n) = 1
3. For all nodes n reachable from some other set of nodes S,
   DistanceTo (n) = 1 + min(DistanceTo(m), $m \in S$

Algorithm

1. A map task receives a node n as a key, and (D, points-to) as its value
   - D is the distance to the node from the start
   - "points-to" is a list of nodes reachable from n
   - $\forall p \in points - to, emit(p, D + 1)$
2. Reduce task gathers possible distances to a given p and selects the minimum one

BFS using MapReduce

- How to implement parallel BFS with MapReduce?
- Assume we have large input file describing graph
- Format of input file is as follows
- *node<tab>adjacency_list|score|colour|parentNode*
- *node* is the key, i.e., node id - everything else is a value
- *adjacency_list* – comma separated list of kid nodes
- *parentNode* – *"source"* for *source* node*, null* for others
- Algorithm is iterative
  - Do Map-Reduce pair – produce a result
  - Result is input to another Map-Reduce pair
  - Stop when all nodes processed

BFS using MapReduce

- Map step
  - Node is "*W*" – just emit the line
  - Node is "*B*" – just emit the line
  - Node is "*G*"
    - Make node "*B*" – emit a line corresponding to this
    - Make kids of node "*G*"
    - For each kid node, set parent as the node
    - For each kid node, set *score* as parent node *score* + 1
    - Emit records corresponding to each kid node
      - How do we handle kids' adjacency lists?

BFS using MapReduce

- Reduce step
  - Aggregate records with same key, i.e., node id
  - Will get multiple records for same node
  - Need to aggregate them and emit one valid record
  - A valid record should contain
    - a valid adjacency list for the node
    - the most up to date colour, i.e., state, of the node
    - the correct score and parent

BFS using MapReduce

- Reduce step
  - Questions to ponder
    - Why will there be multiple records for same node?
    - Why will the adjacency list of some records be empty?
    - Why will we get a node marked as both "*W*" and "*G*"
    - Why will we get a node marked as both "*G*" and "*B*"?
    - Why will we get 2 different scores and parents for same node?
  - Reducer is somewhat more complex than the mapper
- There are libraries which simply do graph processing
  - Pregel, graphX

Topics

- Relational operations
- Matrix operations
- Graph algorithms

Acknowledgements

Portions of these slides were adapted with permission from Leskovec et al. (http://www.mmds.org)

Portions of these slides were adapted with permission as per the following details:

License notice: http://creativecommons.org/licenses/by-sa/3.0/

Copyright notice: CC-BY-SA 3.0

Link to material: https://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters