Belief propagation

Data Intensive Programming

Andrea Scotti, Sandrine Idlas

KTH

October 28, 2019

Belief propagation

Message passing algorithm

- Used on probabilistic graphical models.
- Two types of nodes:
  - Variable nodes represent a stochastic variable present in the model.
  - Factor nodes contain a factored representation of the joint probability of its adjacent nodes.
- Algorithm aim: evaluate the marginal probability at every node.

Project aim

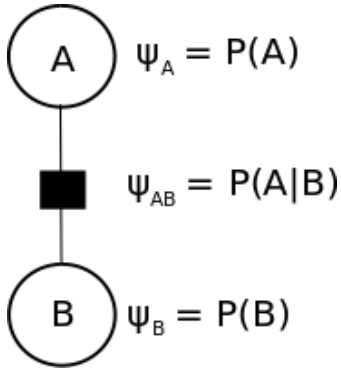
- Implement this algorithm in a distributed way.
- Use the model on trees.
- Stochastic variable is binary, with values 0 and 1.

Belief propagation

Joint distribution
$P(A, B) = P(A|B)P(B)$:

Alternative representation

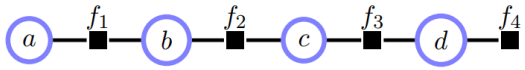
- circle nodes = variables
- dark square = factor node



- Marginalization:
  $P(B) = \sum_A P(B|A)$
- Message from A to the factor node: $\mu_{A \to AB} = P(A)$.
- Message from the factor node to B: $\mu_{AB \to B} = \sum_A \psi_{AB} \cdot \mu_{A \to AB}$.

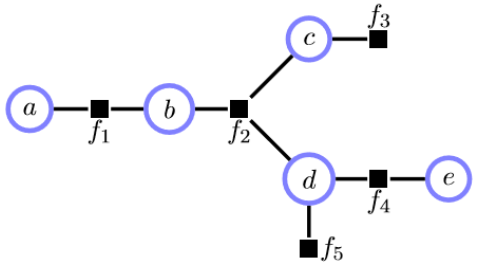
Belief propagation

Chain



- The joint distribution is $p(a,b,c,d) = f_1(a,b)f_2(b,c)f_3(c,d)f_4(d)$.
- $\mu_{d\to c}(c) = \sum_d f_3(c,d)f_4(d)$.
- $\mu_{c\to b}(b) = \sum_c f_2(b,c)\mu_{d\to c}(c)$.
- $p(a) = \sum_b f_1(a,b)\mu_{c\to b}(b)$

Branching



- Model:
  $p(a|b)p(b|c,d)p(c)p(d)p(e|d) = f_1(a,b)f_2(b,c,d)f_3(c)f_4(d,e)d_5(d)$.
- Marginalization: $\mu_{f_2\to b}(b) = \sum_{c,d} f_2(b,c,d)f_3(c)f_5(d)\sum_e f_4(d,e)$.

Belief propagation

Algorithm for updating messages as a function of incoming messages:

1. **Initialisation**:
   - The messages from leaf node factors are initialized to the factor.
   - The messages from leaf variable factors are initialized to unity.
2. **Factor to variable**:
   - Formed by summing the product of incoming node-to-factor messages.
   - $\mu_{x \to b}(x) = \prod_g \mu_{g \to x}(x)$, node x is excluded from the product.
3. **Variable to factor**:
   - Product of incoming factor-to-node messages, does not imply a summation.
   - $\mu_{f \to x}(x) = \sum_{X_f} \psi(X_f) \prod_y \mu_{y \to f}(y)$, summation over all states of all variables, except the current one.

Graphs in GraphX

- Graphs represented as VertexRDD and EdgeRDD.
- Executed using Pregel.
- Supersteps: sequences of iterations for message passing between distributed vertices.
- Allows to store trees bigger than the memory of a single machine.
- Can compute big trees since computation is shared between machines.

Input

- Fetched from a text file.
- Variable node ID: from 0 to the number of variables minus one.
- Factor node ID: shifted by the number of variable nodes.
- Each variable node x can have two different realizations (0 or 1), with probabilities $p(x = 0)$ and $p(x = 1)$.

Example of factor node with two neighbours:

```
0 1
0.3 0.7 0.7 0.3
```

- Adjacent variable nodes ID: 0 and 1.
- Distribution: $\begin{bmatrix} 0.3 & 0.7 \\ 0.7 & 0.3 \end{bmatrix} = \begin{bmatrix} p(0,0) & p(0,1) \\ p(1,0) & p(1,1) \end{bmatrix}$

Supersteps in Pregel

Running Pregel

```
val finalGraph = graphWithLeaves.pregel
    (initialMsg, maxIterations, EdgeDirection.Out)
    (vprog, sendMsg, mergeMsg)
```

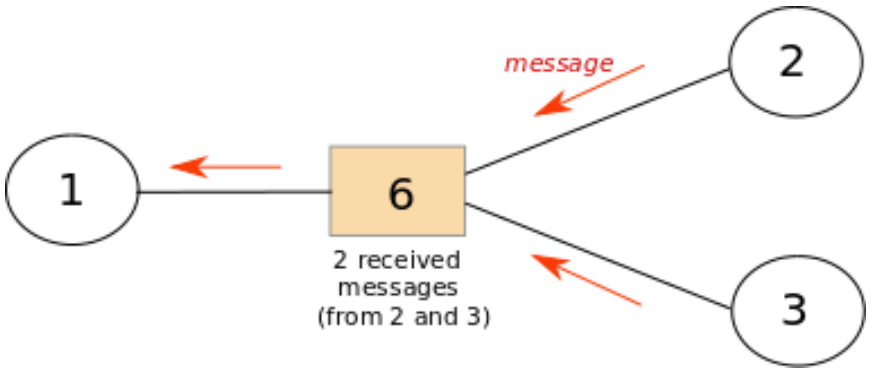
Functions mergeMsg, vprog and sendMsg

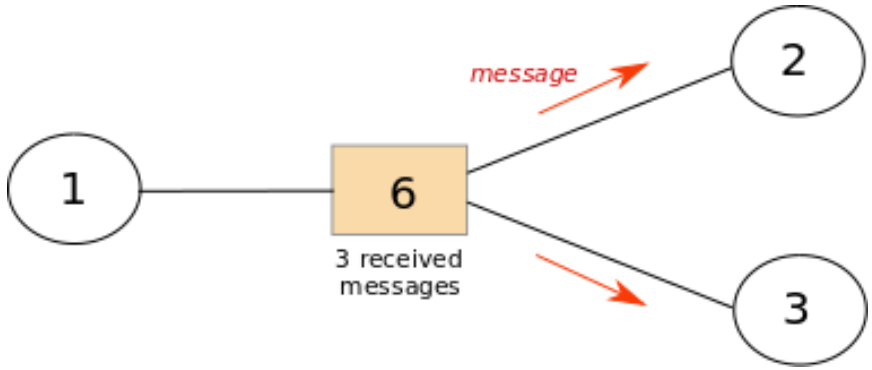
1. Merge (function mergeMsg): concatenates two lists (incoming messages to a factor node).
2. Apply (function vprog):
   - Adds new messages to the list of collected messages of the node.
   - Stores the source of a message in certain cases.
3. Scatter (function sendMsg):
   - Called for each edge of every scattered node.
   - Leaf nodes: only send messages at the beginning of the algorithm.
   - Internal nodes: send messages once they have received enough messages.
   - Computes messages to be sent using the message list created in vprog.

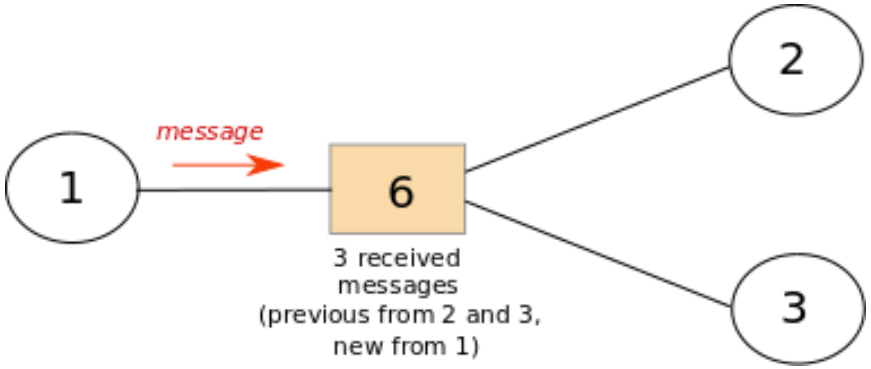
Initialization

- **Variable leaf nodes**: initialized to $(1, 1)$, or $(0, 0)$ for log.
- **Factor node**: initialized to the factor itself.

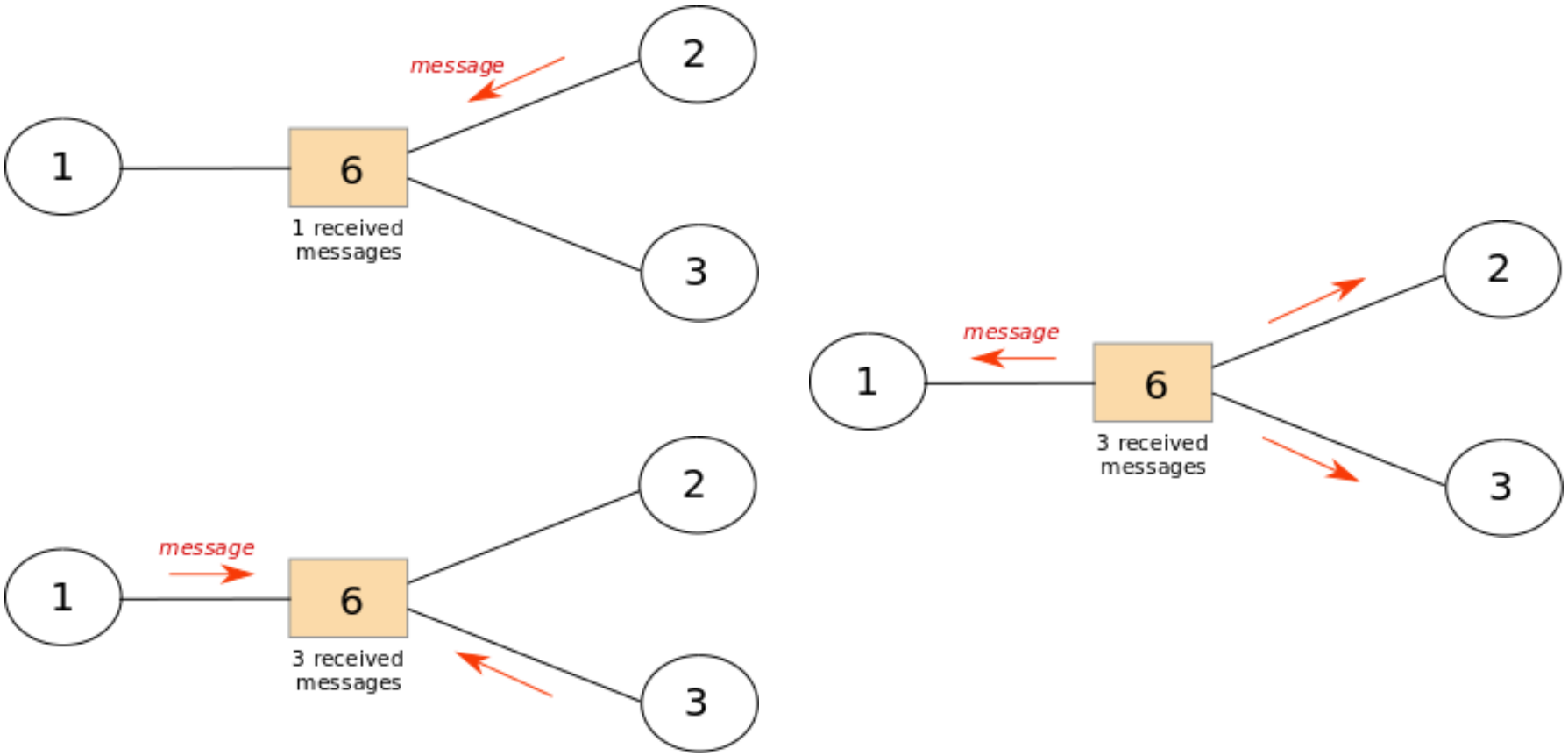
Choosing node destination







Choosing node destination



Output

After the algorithm has ended:

- Filter results by node type, keep only variable nodes.
- Compute probabilities for realizations 0 or 1.
- For each node, print the result:

```
(Node ID, (p(0), p(1), Normalization Constant))
```

Normalization constant

- Used to double-check that the algorithm functioned properly.
- Should be the same for every node $\implies$ information was passed properly through the network.

Results when using log

- e^x, where $x = log(p(x))$, can be too small $\implies$ retrieving the normalization constant is not possible.
- Print a list of non-normalized log probabilities:

```
(Node ID, (log(p(0)), log(p(1))))
```

Method

Implementation

- Python script to understand the algorithm.
- Inputs generated by Python script.
- Implement the algorithm on GraphX.
- Run it in a distributed way using the cluster Hopswork.

Debugging issues

- Cannot easily print with GraphX.
- Workaround: Accumulator function to accumulate the states during computation.

Next step

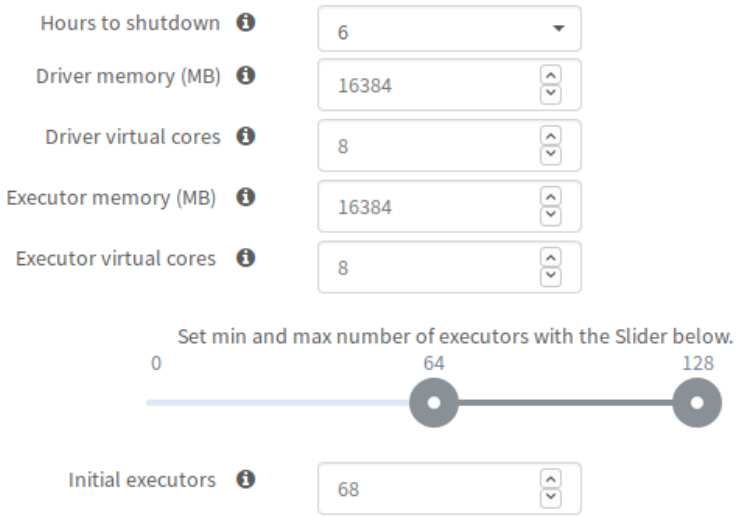
- Split the input file in several input files in Hadoop.
- Potential problems:
  - Need to create a local array that could contain all the nodes during the reading of the input.
  - Risk of runtime error during exectution.
- Instead: use $spark.kryoserializer.buffer.max = 1024M \Rightarrow$ could run half a million of nodes.

Dataset

- synthetic dataset
- python script called *factorTreeGenerator.py*
- generates a random tree rooted given:
  - total number of nodes n
  - minimum and maximum number of children for each node
- rooted in the node with ID 0

Results

- Number of nodes $100 \Rightarrow$ distributed algorithm with GraphX slower than centralized Python script.
- Computation time: several minutes, even in a big cluster. Example:
  - tree with 500000 nodes
  - number of children for each node uniformly between 1 and 10
  - cluster configurations:



  - **Computation time**: 20 minutes.

Results

- Trees of similar size: high influence of topology on the running time performance of the algorithm:
  - On a simple chain with 1 child per parent node, the algorithm runs very slowly in a distributed way.
  - Distributed computation more performant the more children there are per node, but the size of the distribution array exponentially increases with the number of neighbors k as 2^k.