Distributed implementation of Belief Propagation algorithm with GraphX on trees

Andrea Scotti, Sandrine Idlas

October 2019

1 Belief propagation algorithm

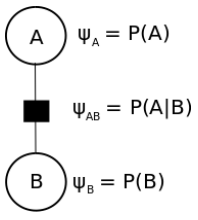
Belief propagation is a message passing algorithm used on probabilistic graphical models, in order to evaluate the marginal probability at every node. The graph is made of two types of nodes: variable nodes and factor nodes. Each stochastic variable present in the model is represented by a variable node. While the factor nodes contain a factored representation of the joint probability of its adjacent nodes, each variable node corresponds to a marginalized probability over all other variables (not including the current one).

The belief propagation algorithm is an efficient way to compute the marginal probability of each stochastic variable of a model, relying on the fact that some variables appear only locally and do not need to be re-computed at every calculation. Instead, the information they carry is combined locally with others, and then passed on through the graph to the relevant nodes. The aim of this project is to implement this algorithm on trees in a distributed way when the possible realizations of a variable are the values 0 and 1.

For a joint distribution of n variables, the marginal probability of one of the variables implies to sum over the possible values of all the other variables. The time complexity to do this for a single variable is $O(2^{n-1})$. With belief propagation, if the factor graph is a tree, a better complexity can be achieved. On trees it is possible to use dynamic programming to save subtree computations on memory and reuse them instead of computing them again and again.

To make this more concrete, let us start with the description of a simple Bayesian network containing only two nodes, represented by a directed graph. The following figure shows such such a network with 2 nodes, where there is a causal relationship between A and B:

The above graph can be described as a joint distribution. Writing this distribution with Bayes law, we obtain a factorized representation: $P(A, B) = P(A|B)P(B)$. A more convenient representation of this graph is the following, where circle nodes represent variables, and the dark square represents a factor node:

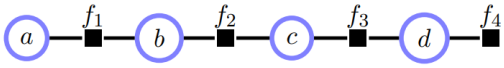


Generally, in order to obtain $P(B)$, we need to marginalize over A and compute: $P(B) = \sum_A P(B|A)P(A)$. This computation can be described in the above graph in the form of message passing from the variable node A to the factor node, then from the factor node to the variable node B. The message from A to the factor node is: $\mu_{A\to AB} = P(A)$. The message from the factor node to B is then:

$\mu_{AB\to B} = \sum_A \psi_{AB} \cdot \mu_{A\to AB}$, which is equal to the above expression for the marginal distribution.

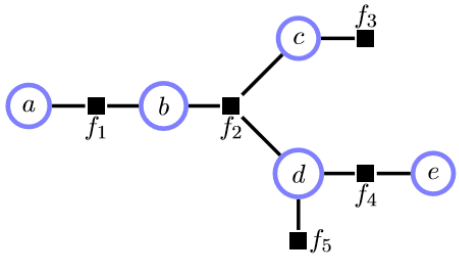
The equivalent computation can be achieved from node B to node A, so that all marginal distributions of the variables present in the model can be computed through such a message passing algorithm, once the whole graph is solved.

To see why this description could allow for more efficient computations, we can consider an example containing more than two nodes:



- The joint distribution is $p(a, b, c, d) = f_1(a, b)f_2(b, c)f_3(c, d)f_4(d)$.
- Marginalizing over $d = p(a, b, c)$ gives: $p(a, b, c) = \sum_d p(a, b, c, d) = \sum_d f_1(a, b)f_2(b, c)f_3(c, d)f_4(d)$.
- Since d only occurs locally, the summation can be "pushed" to the right in order to take into account only the two last factors:
  $p(a, b, c) = f_1(a, b)f_2(b, c) \sum_d f_3(c, d)f_4(d)$.
- Once computed, this sum can then be used again if we need to marginalize further over c to compute $p(a, b)$. This way, the information contained in this sum is passed on to c to be taken into account in the marginalization over c.
- Therefore, this sum can be considered as a **message** carrying information from d to c, which can be denoted as follows:
  $\mu_{d\to c}(c) = \sum_d f_3(c, d)f_4(d)$.
- The marginalization over c can be expressed as:
  $p(a, b) = \sum_c p(a, b, c) = f_1(a, b) \sum_c f_2(b, c)\mu_{d\to c}(c)$.
- We can continue further with the same idea in mind, denoting the message from c to b as:
  $\mu_{c\to b}(b) = \sum_c f_2(b, c)\mu_{d\to c}(c)$.
  Marginalizing over b, we obtain:
  $p(a) = \sum_b f_1(a, b)\mu_{c\to b}(b)$

For a chain of n variables, the marginal distribution can be computed in linear time with n.
We can generalize the above to trees rather than chains, and introduce branching in the model:



In this case, we can describe the belief network as follows:
$p(a|b)p(b|c, d)p(c)p(d)p(e|d) = f_1(a, b)f_2(b, c, d)f_3(c)f_4(d, e)d_5(d)$.
We can identify the different messages in order to compute the following marginal:
$p(a, b) = f_1(a, b) \sum_{c,d} f_2(b, c, d)f_3(c)f_5(d) \sum_e f_4(d, e)$.

- **Factor to variable**:

  The sum over c and d can be identified as the message from the factor f_2 to the variable b:

  $\mu_{f_2 \to b}(b) = \sum_{c,d} f_2(b,c,d) f_3(c) f_5(d) \sum_e f_4(d,e)$.

  This message can be constructed from the messages arriving from two branches through c and d, where:

  $\mu_{c \to f_2}(c) = f_3(c)$, and $\mu_{d \to f_2}(d) = \sum_e f_4(d,e)$.

  Therefore, a message from a factor to a node is formed by summing the product of incoming node-to-factor messages.

- **Variable to factor**

  Similarly, the message from a variable to a factor is is given by the product of incoming factor-to-node messages. However, it does not imply a summation.

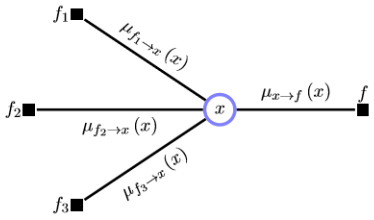
This implies that the messages can be re-used in order to compute other marginal probabilities for other variables. In fact, we can calculate the marginal for any variable that is part of the tree.

This algorithm is also called sum-product algorithm and it is based on updating messages as a function of incoming messages. The algorithm can be described as follows:

1. Initialisation:
   - The messages from leaf node factors are initialized to the factor.
   - The messages from leaf variable factors are initialized to unity.
2. The message from the variable to factor nodes is:

   $\mu_{x \to b}(x) = \prod_g \mu_{g \to x}(x)$,

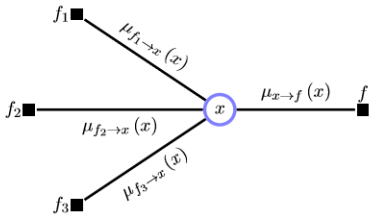
   where the node x is excluded from the product.



3. The message from the factor to variable nodes is:

   $\mu_{f \to x}(x) = \sum_{X_f} \psi(X_f) \prod_y \mu_{y \to f}(y)$,

   where the summation is over all the states of all the possible variables, except the current one.



2 Distributed programming

GraphX is the Spark library to perform graph-parallel processing. In Spark, a graph is logically represented as a pair of vertex and edge property collection: VertexRDD and EdgeRDD. The computations in the graph are executed using Pregel, which allows distributed vertices to communicate with each other by passing messages

to each other in sequences of iterations called Supersteps. At each superstep, a vertx can either send a message to another vertex, receive a message, or modify its own state.

When a vertex becomes inactive as its role in the algorithm ended, it votes to halt and will not perform any further work. The algorithm ends when all vertices have halted, so that there are no messages in transit and al vertices are inactive.

The choice of GraphX is motivated by the fact that trees can be too big to be stored in the memory of a single machine and they make the computation too heavy for a single machine.

3 Implementation

The algorithm has been implemented on Jupyter Notebook with GraphX while the input has been generated with an handcrafted script in Python.

3.1 Input

The input is fetched from a text file that contains a description of the variable and factor nodes, and the initial distributions. The nodes in the graph are identified by an ID number. For variable node, this number goes from 0 to the number of variables minus one. For factor node, the ID numbers are shifted by the number of variable nodes. For example, if there are 5 variable nodes, the first ID number for a factor node will be 5. Each variable node x can have two different realizations (0 or 1), with probabilities $p(x = 0)$ and $p(x = 1)$.

The input contains two lines for each factor node. The first line is a list of its adjacent variable nodes, and the second line contains its initial distribution in a serialized form. As an example, the following lines describe a factor node with two neighbours:

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

The first line indicates that this factor node has two adjacent variable nodes, with IDs 0 and 1. The second line represents the distribution, and should be interpreted as follows:

$$\begin{bmatrix} 0.3 & 0.7 \\ 0.7 & 0.3 \end{bmatrix} = \begin{bmatrix} p(0,0) & p(0,1) \\ p(1,0) & p(1,1) \end{bmatrix},$$

where $p(0, 0)$ is the joint probability that the value is 0 for node ID = 0, and that the value is 0 for node ID = 1.

3.2 Superstep

For iterative computations GraphX works upon Pregel. In this implementation, the exchanged messages are of the type $List[Message]$ where $Message$ is a class defined by us. The motivation behind this choice is that, for each node, we want to keep memory of all the received messages in order to compute to messages to send to the neighbors.
Pregel takes two argument lists:

- The first list contains configuration parameters:
  - The initial message (initialMsg) to start the computation
  - the maximum number of iterations (maxIterations), that will never exceed the double of the number of nodes because the graph is a tree
  - the edge direction in which to send messages (EdgeDirection.Out).
- The second list contains the user defined functions:
  - Apply (vprog)
  - Scatter: (sendMsg)
  - Gather: (mergeMsg)

According to the above description, in our implementation of the belief propagation algorithm, we run Pregel as follows:

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

The functions mergeMsg, vprog and sendMsg do the following:

1. Merge (function mergeMsg): the incoming messages to a factor node will be merged together using this function: it takes two list of messages and it returns a list that is the concatenation of the two input lists.

2. Apply (function vprog): In case the number of messages reaches the number of neighbours for a factor node upon receiving a single message, the source of this messages will be stored. This information will be used in such a way to not send a message again to this node and avoid duplicate messages (see explanations in the next paragraph). If the received list of messages contains the initial message, the state of the node doesn't change. In the other cases, the new messages are added to the current list of collected messages of the node.

3. Scatter (function sendMsg): This function is called for each edge of every scattered node. It needs to distinguish between leaf nodes and internal nodes. Leaf nodes send messages on the first superstep, but do not need to send any messages after this. Internal nodes send messages once they have accumulated enough information from their neighbouring nodes. The messages to be sent are computed depending on the message list created in vprog, the number of messages relative to the number of neighbours, and the type of node (whether it is a variable node or a factor node).

   For variable nodes the initial message sent by a leaf nodes is (1, 1), or equivalently (0, 0) in case the log-probability is used instead of the probability itself. Meanwhile a leaf factor node sends a message initialized to the factor itself.

In order to decide to which destination a node should send a messages, it needs to have received all the messages necessary for the computation. We can distinguish between three cases:

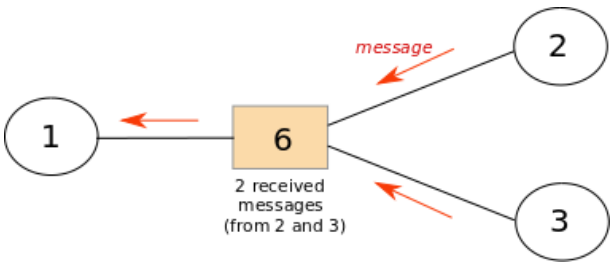
- The node has not received enough messages to do the computation and send messages further to other node. In this case, at the next superstep where the factor node should send something, it will not send any message.

- The factor node has received one less message than the number of its neighbours. In this case, it can send a message to the node that it has not received any message from yet.

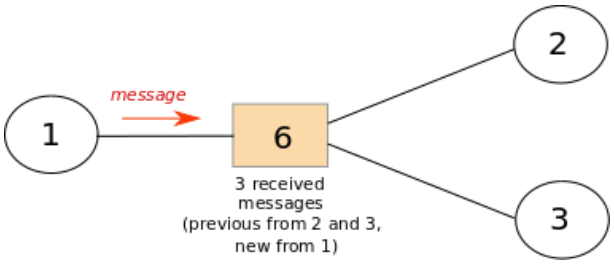
- The factor node has received as many messages as the number of its neighbours. It can now forward a message updated with the content of the newly received message to the nodes that have not received anything yet.

These last two points are represented in the figures below, where the square node represents a factor node (shaded in orange) and the circle-shaped nodes represent variable nodes.

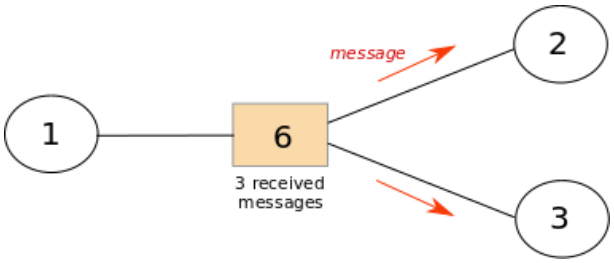
In the following representation, the factor node 6 has three neighbouring variable nodes: 1, 2 and 3. It first receives 2 messages from nodes 2 and 3, and has therefore received enough messages to pass on a message to node 1. In order to recognize this case, we can notice that the number of received messages by node 6 is the number of its neighbours -1. Therefore, when a factor node has received $(Number\_Of\_Neighbours - 1)$ messages, it can infer that it is time to send a message to the node that has not sent anything yet:



At a later stage, Node 1 will end up sending its message to Node 6:

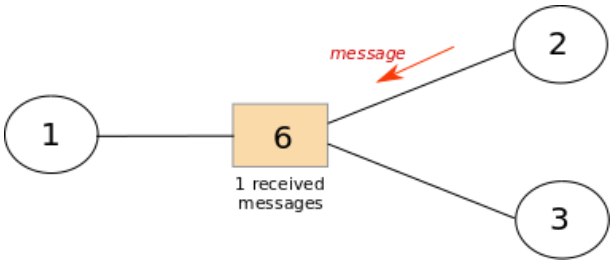


Node 6 will then have the same number of messages as it has neighbours, and will recognize this case as the time at which it can backpropagate this information to the nodes it has not sent anything yet (nodes 2 and 3). It will not send any message to node 1, since node 1 has already received its message previously. To distinguish node 1 from the others, the function vprog stores it in a variable called firstDestination.

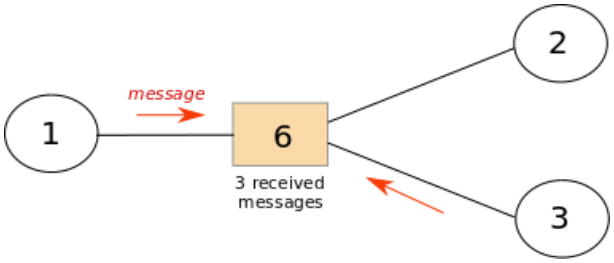


Not all cases would require node 6 to send all the its messages in two different steps. For example, in the following case, all three messages will be sent at once.

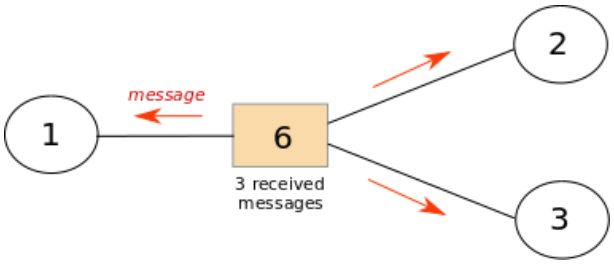
In the first step, only node 2 is sending its message to node 6:



Thereafter, nodes 1 and 3 are sending a message to node 6 simultaneously:



Since node 6 has now received all three messages, it can now send all the three messages to all its adjacent nodes at once:



At the end of the algorithm, all the factor nodes will have received a message from each of its adjacent node, and will have propagated a message to each of them as well. The computations ends when all variable nodes have received all information regarding the variable that they represents, and have updated their beliefs accordingly.

Since the computations involve products of probabilities, i.e small numbers $\leqslant 1$, big graphs face the problem of underflow as the resulting probabilities become too small. A common methods to deal with this problem in probability graphs is to compute the log-probabilities instead of looking at the probabilities themselves [1]. This improves numerical stability, since this allows to use summation instead of product so that the result does not decrease as fast with the number of computations.

For this reason, the log-messages was used instead of messages on large graphs. In this case, it was necessary to add a small constant ϵ inside the log, in order to maintain stability by avoiding the case $\log(0)$, which is undefined and goes towards $-\infty$. For very big graphs, it is not possible to retrieve the actual probability from the log due to underflow (small normalization constants causing division by 0), and the result is then expressed as a non-normalized log-probability.

3.3 Output

When the message passing algorithm is completed and each node has the complete list of collected messages, the results are filtered by node type first, and only the variable nodes are kept. For each remaining node, the probabilities for each realization 0 or 1 is computed. Finally, for each variable, the result is printed under the following form:

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

The normalization constant is used to double-check that the algorithm functioned properly: it is expected that the normalization constant will be the same for every node, since it is independent from the variables. The fact that the normalization constant is similar for every node means that the information about the variables were propagated appropriately to every node through the messages.

If the log-messages are really small, computing the non-normalized probability $p(x)$ from $log(p(x))$ is intractable because e^x, where $x = log(p(x))$, is too small and retrieving the normalization constant is not possible. In this case we just print a list of non-normalized log probabilities in the following form:

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

4 Method

In order to familiarize with the problem, our first step was to implement the algorithm on in a centralized way with python. At this stage, we checked the results locally. After this, the algorithm was implemented on GraphX to run in a distributed way. In order to run the code on several machines, we used the cluster in Hopswork.

One problem to implement the code was that GraphX does not easily let us print the different function outputs in order to debug the code. As a workaround, we implemented an accumulator function, that accumulates the states during the computation and finally prints them.

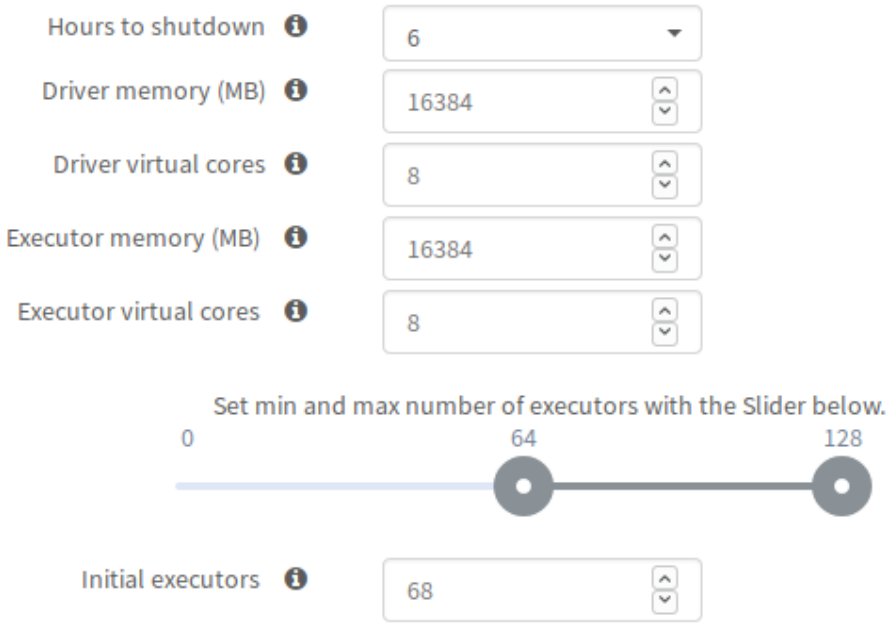
To manage bigger graph, the next step would be to split the input file in several input files in Hadoop. However, we didn't go this further because we expect that, on the attempt to create a local array that could contain all the nodes during the reading of the input, the execution could go on a runtime error. So we increased as much as possible the input buffer size in Hopswork Spark Configuration ($spark.kryoserializer.buffer.max = 1024M$) and we were still able to run the code on trees with more than half a million of nodes.

5 Dataset

In this project the algorithm is tested on a synthetic dataset. We implemented a script in python called $factorTreeGenerator.py$ that, given the total number of nodes n and minimum and maximum number of children for each node in the tree ($minNumberOfChildren$, $maxNumberOfChildren$), it generates a random tree rooted in the node with ID 0. It is possible to change these features of the tree by accordingly setting them directly in the code of the script.

6 Results

We run the algorithm over multiple test cases and overall GraphX resulted slower compared to the centralized algorithm written in python for trees with less than one hundreds nodes. Moreover, by increasing the number of nodes, the computation takes several minutes even in a big cluster. For instance, given a tree with 500000 nodes and a number of children for each node uniformly between 1 and 10 and these cluster configurations:



the algorithm ends its computation in twenty minutes.
For trees of similar size, the graph topology has an influence on the running time performance of the algorithm. The more children there are per node, the more we can take advantage of the distributed computation. In the limit of implementing a tree as a simple chain with only one child per parent node, it is not tractable to run the code even in a distributed way. This is due to the fact that each child needs to wait until its parent has a message to send and this is the worst case in the number of iterations. On the other hand, the more children we have per node, the more we can take advantage of the distributed computation, since a parent can send messages to several children in parallel. However, if the average number of neighbors per node increases, the serialized array, that represents the probability distribution for a factor node, exponentially increases. If k is the number of neighbors, the size of this array is 2^k.

References

[1] David Barber. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, New York, NY, USA, 2012.