Research Task D - The Comparison: Implementation of a Parallel Solution for Message Passing on Arithmetic Circuits

André E. dos Santos DOSSANTOS@CS.UREGINA.CA
Department of Computer Science
University of Regina
Regina, Canada

Abstract

Arithmetic Circuits (AC) is a graphical method for reasoning with Bayesian networks (BNs) based on partial differentiation. ACs is a viable framework for applications of BNs to embedded systems, which are characterized for their primitive computational resources. There are two phases to compile the AC: upward and inward. Once the BN is processed, one can compute in constant-time answers to a large class of probabilistic queries. In this paper we present the algorithm to compile the AC. We show the parallel solution and describe how it compares to a serial solution. In practice, our empirical evaluation shows that the parallel solution tends to be faster than the serial solution in ACs.

1. Introduction

Darwiche (2013) proposed a new approach for inference in Bayesian networks (BNs) (Pearl, 1988) based on partial differentiation called *Arithmetic Circuits* (Darwiche, 2013). The differential approach presents two key contributions. First, it enphasizes the role of partial differentiation on probabilistic reasoning, giving a new utility to central tasks of BNs. Second, it helps the migration of BN applications to embedded systems, which are known for constraint in computational resources.

An AC is a directed acyclic graph with four types of variables: evidence indicators, network parameters (probability values), sum nodes and product nodes. The leaves are evidence indicators ands network parameters and the rest of the graph consists of sum and product nodes. The AC is build according to the elimination ordering of the inference algorithm variable elimination. There are two phases to compile the AC given and evidence: upward and inward. The first phase consists of following the operators numbers upward. By the end of the upward phase we can compute the probability of the evidence. By the end of the downward phase we can compute in constant-time answers to a large class of probabilistic queries.

In this paper we present the algorithm to compile an AC. It computes both upward and inward phase. We show a parallel solution implementation and describe how it compares to a serial solution.

This paper is organized as follows. In Section 2, message passing in Acs are reviewed. Section 3 presents the parallel solution implementation for message passing on arithmetic circuits. Conclusions are given in Section 4.

2. Background

For a complete background please check the Task C - *The Problem* PDF paper.

2.1 Evaluating and Differentiating a Polynomial Representation

The evaluation of the AC and computing the partial derivatives is a two-phase message passing scheme in which each message is simply a number. The first phase, massages are sent from nodes to their parents in the AC following the operations of each node. The phase starts from the leaves up to the root. Note that the value of a of a node can not be computed before all its children values has been computed. First phase is described in lines 1-11 in Algorithm 1. Figure 1 shows this process, where it leads to assigning the value 0.3 to the root, indicating that the probability of evidence E is $P(E) = 0.3$.

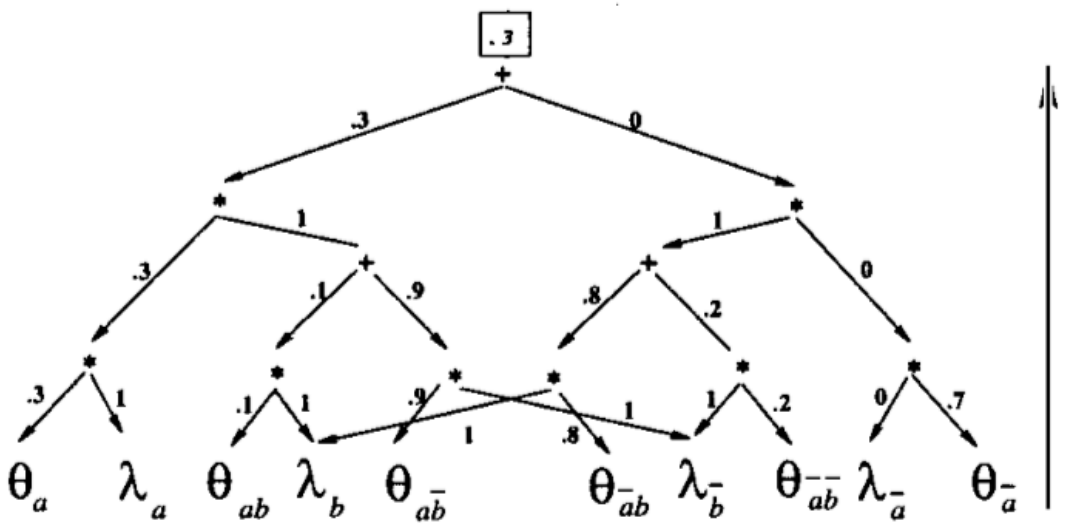

Figure 1: Upward phase on an AC.

Second phase, messages are sent from nodes to children in the same rooted DAG, leading to computation of all partial derivatives. This process is known as *back propagation* (Rumelhart et al., 1988), a common method of training artificial neural networks, and its proven to be able to compute the partial derivatives of variables on the leaves. The phase starts from the root down to the leaves. The derivative value of the root, by definition, is always 1. For the remaining nodes v, if the parent $Pa(v)$ is a summation node, the derivative value of v is equal to its parent. If the parent $Pa(v)$ is a multiplication node, the derivative value of v is equal to the summation of its parent times the others siblings of v. This processes is illustrated in Figure 3. Note that the derivative of a node of a node can not be computed before all its parents derivatives has been computed. Second phase is described in lines 13-27 in Algorithm 35.

After the two-phase steps we have the capacity to answer, in constant time, a very large class of probabilistic queries, relating to classical inference, parameter estimation,

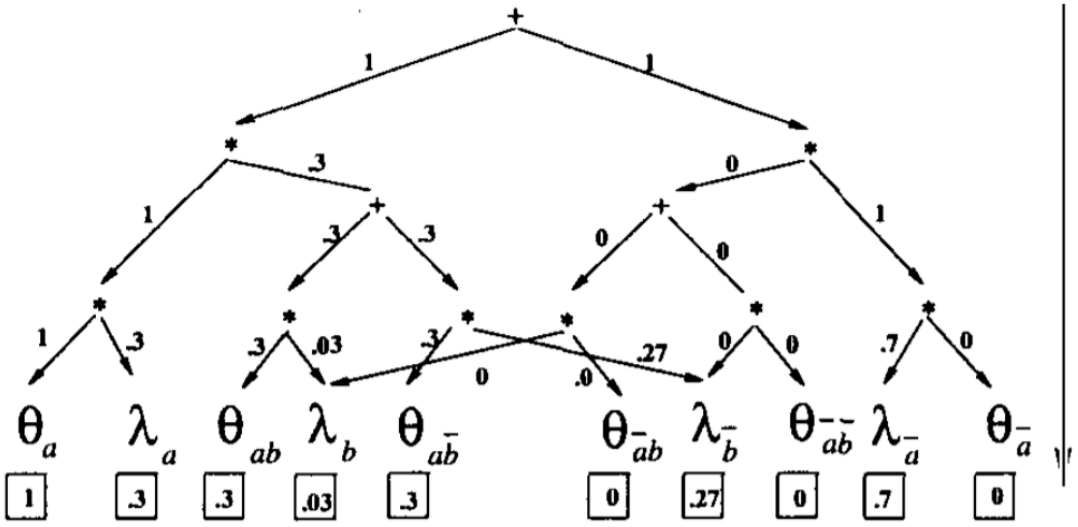

Figure 2: Second phase of the evaluation of AC in Figure 5, under evidence $E = A$.

model validation and sensitivity analysis. Some of those queries are (i) in the root node the probability of the evidence, and (ii) the posterior marginal of any network variable.

2.1.1 Serial Solution

The serial solution computes the values and derivatives of each node sequentially in both cases. The only restrictions are those imposed by the topological ordering. For instance, the second phase where the derivatives are propagated top-bottom, can be implemented with *breadth-first search* algorithm.

2.1.2 Parallel Solution

The phases of evaluating and differentiating an AC can improved by the parallelization of the nodes value calculation. Phase one, massages are sent from nodes to their parents in the AC following the operations of each node, as described in lines 1-11 in Algorithm 1. Since the phase starts from the leaves up to the root, the layers of operations (sum and product) can be computed in parallel. For instance, in the second last layer of Figure 1, the value of nodes $(\theta_a \star \lambda_a)$, $(\theta_{ab} \star \lambda_b)$, $(\theta_{a\bar{b}} \star \lambda_{\bar{b}})$, $(\lambda_b \star \theta_{\bar{a}b})$, $(\lambda_{\bar{b}} \star \theta_{\bar{a}\bar{b}})$, and $(\lambda_{\bar{a}} \star \theta_{\bar{a}})$ can be computed in parallel.

In the second phase, messages are sent from nodes to children in the same rooted DAG, leading to computation of all partial derivatives, as described in lines 12-27 in Algorithm 1. Since the phase starts from the root down to the leaves, the derivatives of the layers of children can be computed in parallel. For instance, in the last layers of Figure 3, the derivatives of nodes θ_a, λ_a, θ_{ab}, λ_b, $\theta_{a\bar{b}}$, $\theta_{\bar{a}b}$, $\lambda_{\bar{b}}$, $\theta_{\bar{a}\bar{b}}$, $\lambda_{\bar{a}}$, and $\theta_{\bar{a}}$ can be computed in parallel.

3. Experimental Results

ACs are known for been a special case of neural networks (Poon and Domingos, 2011; Delalleau and Bengio, 2011; Peharz et al., 2013). In fact, the main differences are only (i) the activations functions, which are either sum and products on AC, and (ii) the data is a joint probability distribution. That been said, we implement a parallel solution for Algorithm 35 using a Python library called *keras* (Chollet, 2015). Keras is a modular neural networks library, written in Python, for fast experimentation.

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Layer, Activation
import time
import tensorflow as tf

f = open("results.csv", "w")


INPUT_SIZE = 10
OUTPUT_SIZE = INPUT_SIZE
nb_class = 3

batch_size = 128
nb_epoch = 40

np.random.seed(123)

X_train = np.random.rand(INPUT_SIZE, nb_class)
Y_train = np.random.rand(OUTPUT_SIZE, nb_class)

X_test = np.random.rand(INPUT_SIZE)
Y_test = np.random.rand(OUTPUT_SIZE)

for i in range(1,51):

    start_time = time.time()

    model = Sequential()
    model.add(Dense(INPUT_SIZE, input_shape=(nb_class,)))
    model.add(Activation('linear'))
    model.add(Dense(OUTPUT_SIZE))
    model.add(Activation('linear'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    final_time = time.time()
    diff_time = final_time - start_time

    f.write(str(i)+","+str(diff_time)+","+"\n")

f.close()
```

We report an empirical comparison between parallel and serial AC compilation solutions. The experiments were conducted on a 2.9 GHz Intel Core i7 with 8 GB RAM. The experiments were performed 50 times for each solution. The total time in seconds required by the parallel and serial solutions are reported in Table 1.

From Table 1, the implementation of a serial solution is slower than that of d-separation on all runs. The main reason is that Algorithm 35 can have two parallelizations, the 0first and second phase. However, note that there are variations on the times reported. Those variations are due the fact an AC can be constructed in a different manner according to different elimination orderings[1].

Table 1: Comparison of parallel and serial solutions for ACs with 50 runs each

Solution	Time average	Standard deviation
Serial	0.1610	0.0536
Parallel	0.0434	0.0082

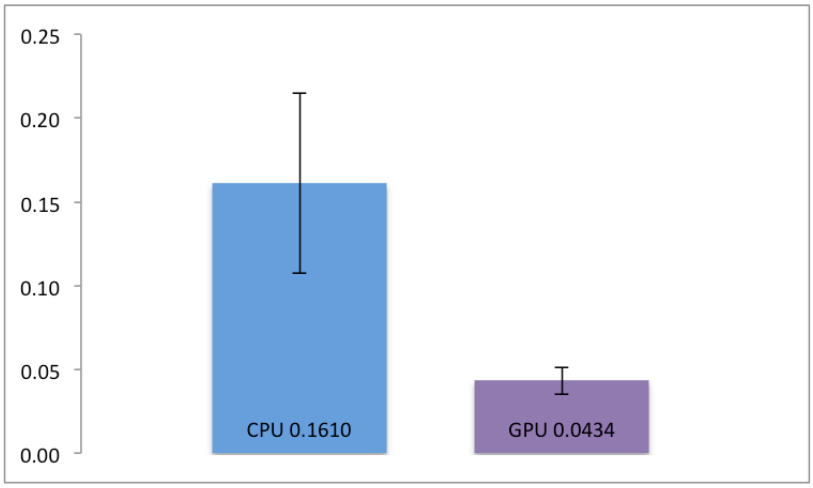

Figure 3: Comparison between CPU and GPU implementation of AC compiling.

4. Conclusion

Reasoning with ACs presents several advantages. It emphasizes the role of partial differentiation on probabilistic reasoning, giving a new utility to central tasks of BNs. Also, it helps the migration of BN applications to embedded systems, which are known for constraint in computational resources.

There are two phases to compile the AC: upward and inward. In this paper we presented the algorithm to compile the AC, drawn from Darwiche (2009). We have implemented a parallel solution and compared to a serial solution. Our experimental results indicate that the parallel solution is especially effective in ACs. However, different elimination ordering of variables upon the construction of the AC can influence on the compilation time.

1. For more information please check the Task A - *The Paper* PDF paper

References

Franois Chollet. keras. `https://github.com/fchollet/keras`, 2015.

Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

Adnan Darwiche. A differential approach to inference in bayesian networks. *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI2000)*, 2013.

Olivier Delalleau and Yoshua Bengio. Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems*, pages 666–674, 2011.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Robert Peharz, Bernhard C Geiger, and Franz Pernkopf. Greedy part-wise learning of sum-product networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 612–627. Springer, 2013.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 689–690. IEEE, 2011.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Algorithm 35 `CircP2`($\mathcal{AC}$, vr(), dr()). Assumes the values of leaf circuit nodes v have been initialized in vr(v) and the circuit alternates between addition and multiplication nodes, with leaves having multiplication parents.

input:

$\mathcal{AC}$: arithmetic circuit
vr(): array of value registers (one register for each circuit node)
dr(): array of derivative registers (one register for each circuit node)

output: computes the value of circuit output v in vr(v) and computes derivatives of leaf nodes v in dr(v)

main:

1: **for** each non-leaf node v with children c (visit children before parents) **do**
2: **if** v is an addition node **then**
3: vr(v)$\leftarrow \sum_{c:\ \text{bit}(c)=0}$ vr(c) {if bit(c) $= 1$, value of c is 0}
4: **else**
5: **if** v has a single child c' with vr(c') $= 0$ **then**
6: bit(v)$\leftarrow$1; vr(v)$\leftarrow \prod_{c \neq c'}$ vr(c)
7: **else**
8: bit(v)$\leftarrow$0; vr(v)$\leftarrow \prod_{c}$ vr(c)
9: **end if**
10: **end if**
11: **end for**
12: dr(v)$\leftarrow$0 for all non-root nodes v; dr(v)$\leftarrow$1 for root node v
13: **for** each non-root node v (visit parents before children) **do**
14: **for** each parent p of node v **do**
15: **if** p is an addition node **then**
16: dr(v)$\leftarrow$dr(v) + dr(p)
17: **else**
18: **if** vr(p) $\neq 0$ **then** {p has at most one child with zero value}
19: **if** bit(p) $= 0$ **then** {p has no zero children}
20: dr(v)$\leftarrow$dr(v) + dr(p)vr(p)/vr(v)
21: **else if** vr(v) $= 0$ **then** {v is the single zero child}
22: dr(v)$\leftarrow$dr(v) + dr(p)vr(p)
23: **end if**
24: **end if**
25: **end if**
26: **end for**
27: **end for**