Variable Elimination in Bayesian Networks

Tom Aarsen - s1027401

18 November 2020

1 Introduction

Variable Elimination (VE) is an inference algorithm that can be applied on Bayesian Networks (BN), which efficiently sums out variables in a sensible order. It can be used to calculate probabilities of values for variables within such a Bayesian Network.

The algorithm for BN takes advantage of the BN property which states that

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid \text{Parents}(X_i))$$

where $\text{Parents}(X_i)$ is defined as all parents of node X_i within the Bayesian Network.

This paper describes this algorithm, and implements it such that it can run on any given Bayesian Network. The BN's passed to this algorithm are in the Bayesian Interchange Format[1], samples of which can be found on the Bayesian Network Repository[2] by `bnlearn`, a R package for Bayesian network learning and inference.

Alongside the explanation and implementation of this algorithm, four separate elimination ordering functions will be tested for performance. These functions are:

1. `least_incoming_arcs`: Prioritises variables whose nodes in the BN have the **least parents**.
2. `most_incoming_arcs`: Prioritises variables whose nodes in the BN have the **most parents**.
3. `least_outgoing_arcs`: Prioritises variables whose nodes in the BN have the **least children**.
4. `most_outgoing_arcs`: Prioritises variables whose nodes in the BN have the **most children**.

These functions are evaluated on computation time. How exactly these questions will be answered is described in the Methods section. Before getting to that, the Variable Elimination algorithm will be explained.

2 Variable Elimination

The VE algorithm takes several parameters. Beyond the obvious BN, three other parameters must be supplied:

[1] `https://www.cs.washington.edu/dm/vfml/appendixes/bif.htm`
[2] `https://www.bnlearn.com/bnrepository/`

- Query variable
- Observed variables
- Elimination order

The first of which determines the variable for which the probability will be determined. When attempting to find the probability $P(X)$ for some variable X, this variable would be the query variable.

The second, observed variables, is a potentially empty mapping of variables to values. They represent what is already known. In the probability $P(X|A = \text{True}, B = \text{False})$, the values for variables A and B are already known to be True and False respectively, and thus the supplied observed variables are:

```
{
    "A": "True",
    "B": "False",
}
```

The third of the parameters is the determining factor of the performance of VE: the elimination order. It can either be a list of variables, or a function outputting a list of variables. The VE algorithm will "eliminate" variables in this order. Note that the elimination order list does not contain the query, nor the observed variables, as these are not eliminated.

The BN property can now be used to determine a function to calculate the probability of the query variable X_i:

$$P(X_i) = \sum_{\{X_1,\ldots,X_n\}-\{X_i\}} P(X_1,\ldots,X_n) = \sum_{\{X_1,\ldots,X_n\}-\{X_i\}} \prod_{j=1}^{n} P(X_j \mid \text{Parents}(X_j))$$

For example, with simple BN $A \to B \to C$ and query variable C, this would produce:

$$P(C) = \sum_{A,B} P(A,B,C) = \sum_{A,B} P(A)P(B|A)P(C|B)$$

The next step is to replace every $P(X_i, X_j, \ldots, X_n)$ with a factor over X_i *and* $X_j, \ldots, X_n$, like $f_0(X_i, X_j, \ldots, X_n)$. Each of these factors is a truth table for these variables, and each probability is filled in using the probability information from the BN. Note that the value next to f is only used for identification, and has no meaning.

Following the previous example, this produces:

$$P(C) = \sum_{A,B} f_0(A)f_1(A,B)f_2(B,C)$$

where factors might look like this:

$f_1(A,B) =$

A	B	val
$True$	$True$	0.3
$True$	$False$	0.7
$False$	$True$	0.8
$False$	$False$	0.2

This is where the observed values are first used. Each of the observed variables will be removed from the equation and the factors. For the factors, the options that cannot be will be scrapped,

simplifying the factors.

For example, if A is known to be $True$ in the given example, then the previous factor becomes:

$f_1(A) =$

B	val
$True$	0.3
$False$	0.7

Furthermore, the A will be removed from the sum in the formula, and all factors will be updated to no longer mention A. f_0 is removed completely.

$$P(C|A = True) = \sum_B f_1(B) f_2(B, C)$$

This concludes the setup portion of the VE algorithm.

For each variable in the elimination list, several steps are executed:

1. Find all factors containing the current variable to eliminate, and multiply them together. This might produce a factor with more variables than any of the factors used in the multiplication, e.g. for $f_9(A, B, C, D) = f_8(A, B) f_7(A, C) f_6(B, C, D)$.
   In the example, where B is next to be eliminated, this produces $f_1(B)$ and $f_2(B, C)$. These will be multiplied together to produce a new factor, e.g. $f_4(B, C) = f_1(B) f_2(B, C)$.

2. From this product factor, the variable to eliminate is summed out. This means that all rows in the truth table that are now equivalent due to the removal of the variable to eliminate will be summed together, and reduced to just one row.
   For the example this turns $f_4(B, C)$ into $f_4(C)$.

3. The factors that were used in the creation of this product factor will be removed from the equation, and the product factor is added to the equation. The variable that is eliminated is also removed from the sum in the equation.
   After one iteration of the example, the equation has transformed into $P(C|A = True) = f_4(C)$

These steps are repeated until all variables have been eliminated.

For any BN, this will produce a formula in this form:

$$P(Q|O) = f_1(Q) \cdot f_2(Q) \cdots f_n(Q)$$

where Q is the query variable, O is the mapping of observed variables, and the righthand side is one or more factors that only hold values for Q.

These factors will all be multiplied together to create one final factor for the query variable Q. As the very last step, this final factor is normalized by dividing the factor by the sum of all values. This ensures that it is a probability distribution.

This normalized factor holds the probabilities for each option for the query variable, given the observed variables, and this concludes the Variable Elimination algorithm.

3 Implementation

With the concept of the algorithm out of the way, the implementation will be discussed. This section is split up into two parts: The factor calculations and the variable elimination. With the factor calculations prepared, the VE implementation becomes fairly straightforward.

3.1 Factor Calculations

Within the main file, `variable_elim.py`, a new class `Factor` is introduced. This class is initialized with a reference back to the Variable Elimination class introduced later, a set of strings for nodes that represent the variables stored in this factor, as well as a `pandas DataFrame` object called `df`. This `DataFrame` is similar to a truth-table but with a `prob` column as the final column. This `df` may look like:

```
    Alarm Burglary Earthquake   prob
0    True     True       True  0.950
1   False     True       True  0.050
2    True    False       True  0.290
3   False    False       True  0.710
4    True     True      False  0.940
5   False     True      False  0.060
6    True    False      False  0.001
7   False    False      False  0.999
```

Several operations exist on factors, that are implemented as (class)methods for the `Factor` class:

- Marginalization
- Reduction
- Product
- Normalization

Each of these will be explained next. Note that each of these methods have examples in comments in the file that are not mentioned in this paper.

3.1.1 Marginalization

Factor marginalization is the act of summing over one variable to produce a new factor:

$$\sum_{B} f_3(A, B, C) = f_4(A, C)$$

It is used in the second iteration step of the VE algorithm, to sum away the variable to be eliminated.
The method introduced, called `marginalize`, takes one parameter: `node`, the nodes meant to be summed away. This parameter may either be a string representing a node, or a set of strings representing nodes. The `DataFrame` that holds the data for this factor is grouped by all nodes represented by this factor that are not `node`, and then the result is summed. The method is implemented accordingly:

```
    def marginalize(self, node: Union[str, Set[str]]) -> None:
        if isinstance(node, str):
            node = {node}
        self.nodes -= node
        if self.nodes:
            self.df = self.df.groupby(list(self.nodes)).sum().reset_index()
```

3.1.2 Reduction

Factor reduction is the act of removing rows where variables now have known values:

$$f_3(A, B, C = True) = f_5(A, B)$$

It is used in the preparation steps of the VE algorithm, to remove the cases that cannot occur due to the observed variables.
The method for it, called `reduce`, takes a parameter `node_dict`. This is a `dict` of strings representing nodes as keys, with values corresponding to those nodes. An example would be:

```
{
    'Burglary': 'True',
    'Earthquake': 'False'
}
```

A set of relevant nodes is constructed using the intersection of nodes represented by the current `Factor` object, and `node_dict`. For these relevant nodes it is checked for which rows the `DataFrame` has the right values, the ones listed in `node_dict`. The indices where this is the case, for each observed variable, are AND-ed together with `&`, and passed as indices to the `DataFrame` object. Then, the columns for the relevant nodes are removed, and these nodes are removed from the `nodes` class variable. This method is implemented like:

```
    def reduce(self, node_dict: dict) -> None:
        relevant_nodes = set(node_dict.keys()).intersection(self.nodes)
        if relevant_nodes:
            self.df = self.df[reduce(
                lambda x, y: x & y,
                (self.df[key] == val for key, val in node_dict.items()
                                     if key in relevant_nodes))
            ]
            self.df = self.df.drop(columns=relevant_nodes)
            self.nodes -= relevant_nodes
```

3.1.3 Product

A factor product is when two factors are multiplied together such that they merge on overlapping segments, often creating a larger factor:

$$f_1(A, B) \times f_2(B, C) = f_3(A, B, C)$$

This is used in the first iteration step of the VE algorithm, to reduce multiple factors into one.
The classmethod for this is called `product`, and takes two `Factor` objects. It returns a completely new `Factor` object.
These factors are merged on the overlapping columns, and the probability for each row between the two factors is multiplied together. This classmethod is implemented as such:

```
    @classmethod
    def product(cls, factor_x: "Factor", factor_y: "Factor") -> "Factor":
        new_df = factor_x.df.merge(factor_y.df, on=list(
            factor_x.nodes.intersection(factor_y.nodes)))
        new_df["prob"] = new_df["prob_x"] * new_df["prob_y"]
        new_df = new_df.drop(columns=["prob_x", "prob_y"])
        return cls(factor_x.ve, factor_x.nodes.union(factor_y.nodes), new_df)
```

3.1.4 Normalization

Factor normalization is converting a factor into a probability distribution - ensuring the values sum to 1.

This is used as the very last step in the VE algorithm.
The `DataFrame` is simply divided by the sum of all probability values, which causes the new sum of all probability values to be 1. The implementation is simple:

```
    def normalize(self):
        self.df["prob"] /= self.df["prob"].sum()
```

3.2 Variable Elimination

Now that the building blocks of the VE algorithm, the factors, have been discussed, the real VE implementation will be shown. This implementation exists in the `run` method of the `VariableElimination` class. This class is initialised with a BN network, which is read from a file by another module. This class may also be initialized with a verbosity level, where 0 indicates no debug information, 1 means some, and 2 means all debug information.

The `run` method takes three parameters, corresponding exactly to the three required parameters for the VE algorithm: the string `query`, the `dict observed`, and either the method or list `elim_order`. The steps executed in this method will be described next. They are the core of the VE algorithm.

1. If the supplied elimination ordering is a function that generates the real ordering, then this function is called. From this step the `elim_order` variable is a list of strings representing variables or nodes, potentially including the query and observed variables.

   ```
       def run(self, query: str, observed: dict, elim_order: Union[List[str], Callable]):
           # Convert function that determines ordering into a list of nodes
           if isinstance(elim_order, FunctionType):
               elim_order = elim_order(self.network)
   ```

2. The nodes in the network are converted to `Factor` objects. Each object is given:

   - A reference to the current `VariableElimination` object.
   - A list of nodes relevant for this `Factor`, i.e. the node itself and all of its parents.
   - An initial `DataFrame` object from the network's probability mapping.

   This produces a list of `Factor` objects.

   ```
           # Create a list of Factor objects *before* reducing observed variables
           factors = [
               Factor(self,
                      {node, *set(self.network.parents[node])},
                      self.network.probabilities[node])
               for node in self.network.nodes
           ]
   ```

3. The factors are reduced by the observed variables. This is where the `Factor.reduce` method is used. After reducing, (now) empty factors are removed.

   ```
           # Reduce factors by observed variables.
           for factor in factors:
               factor.reduce(observed)

           # Remove (now) empty factors.
           factors = [factor for factor in factors if factor.nodes]
   ```

4. The `elim_order` list is stripped from the query variable, and all observed variables.

   ```
           # Remove query and observed variables from elimination ordering
           elim_order = [node for node in elim_order if node != query and node not in
       observed]
   ```

5. Next are the three iteration steps of the VE algorithm:

   (a) For each node to be eliminated, the relevant factors are put in a list. The `reduce` function from `functools` is used alongside `Factor.product` to reduce this list of factors into one product factor.

```
        for i, node in enumerate(elim_order):
            # Get factors that use `node`
            filtered_factors = [factor for factor in factors
                                if node in factor.nodes]
            # Multiply factors containing `node`
            product_factor = reduce(Factor.product, filtered_factors)
```

   (b) This product factor is marginalized with the node to be eliminated using `Factor.marginalize`.

```
            # Sum out `node`
            product_factor.marginalize(node)
```

   (c) The list of factors is stripped from all factors that were used to create the product factor, and the product factor is appended as long as it is not empty. (This can occur if the product factor only uses one variable, and that is the variable to be marginalized).

```
            # Remove the multiplied factors, and add the new one,
            # unless the new one is empty
            factors = [factor for factor in factors
                       if factor not in filtered_factors]
            if product_factor.nodes:
                factors.append(product_factor)
```

   These are looped until all variables have been eliminated.

6. All remaining factors are multiplied into one factor, again using `reduce` mixed with `Factor.product`.

```
        # Multiply remaining factors, i.e. factors with `query`
        final_factor = reduce(Factor.product, factors)
```

7. This final factor is normalized.

```
        # Normalize this factor
        final_factor.normalize()
```

8. And finally, the `DataFrame` from this factor is returned in a singleton `dict` with the query variable as the key.

```
        return {query: final_factor.df}
```

This implementation allows for variables with any number of values.
In addition to this implementation of the VE algorithm, the `VariableElimination` class has several methods that act as heuristics for elimination order. These are `least_incoming_arcs`, `most_incoming_arcs`, `least_outgoing_arcs` and `most_outgoing_arcs`. These are simply based on the `parents` dict that the BN network has. How the performance of each will be measured will be explained in the following Method section.

4 Method

This section is about the four different elimination ordering heuristics supplied with the VE implementation. If the VE implementation is correct, then the results for each of these heuristics should be identical, even if the path to get there is different.

Furthermore, each of these elimination orderings are compared in terms of run-time performance. Alternatively, it would be possible to compute the amount of FLOPS required, but run-time performance is a more realistic performance gauge - as some operations may be implemented more or less efficiently in the underlying data structures by `pandas`.

There are 3 different bayesian networks that will be tested:

Name	#Nodes	#Arcs	#Params	Avg degree	Max in-degree
EARTHQUAKE	5	4	10	1.6	2
SACHS	11	17	178	3.09	3
ALARM	37	46	509	2.49	4

The queries `'Alarm'`, `'Akt'` and `'MINVOLSET'` will be used for these BN's respectively, with evidences `'{'Burglary': 'True'}'`, `'{'Mek': 'LOW', 'Plcg': 'AVG', 'Jnk': 'HIGH'}'` and `'{'VENTMACH': "LOW"}'`.

These 3 BN's will be run with their respective queries and evidences for each elimination ordering heuristic. Each combination of BN and heuristic is executed 5 times to gather an average run-time. Afterward, each combination of BN and heuristic is also ran for each of the three verbosity levels, and the output is placed in text files for inspection. These files can be found in the `outputs` folder.

Note that `'read_bayesnet.py'` was also modified to move the `values`, `probabilities` and `parents` variables to within the `__init__` function, as otherwise multiple bayesian networks will have overlapping values.

5 Results & Discussion

The execution time of the VE algorithm in seconds:

For **Earthquake**:

Heuristic	Iter #1	Iter #2	Iter #3	Iter #4	Iter #5	Avg
`least_incoming_arcs`	0.0928	0.0905	0.1044	0.0913	0.0919	0.0942
`most_incoming_arcs`	0.0929	0.0899	0.0945	0.0931	0.0929	0.0927
`least_outgoing_arcs`	0.09475	0.09272	0.08890	0.08942	0.09428	0.0920
`most_outgoing_arcs`	0.0888	0.0920	0.0885	0.0859	0.0865	0.0884

For **Sachs**:

Heuristic	Iter #1	Iter #2	Iter #3	Iter #4	Iter #5	Avg
`least_incoming_arcs`	0.2405	0.2474	0.2871	0.3504	0.2970	0.2845
`most_incoming_arcs`	0.2971	0.2468	0.2367	0.2323	0.2436	0.2513
`least_outgoing_arcs`	0.2553	0.2263	0.2317	0.2354	0.2325	0.2363
`most_outgoing_arcs`	0.3012	0.3092	0.2777	0.2581	0.2590	0.2811

For **Alarm**:

Heuristic	Iter #1	Iter #2	Iter #3	Iter #4	Iter #5	Avg
`least_incoming_arcs`	1.2183	1.3034	1.2172	1.2113	1.1719	1.2245
`most_incoming_arcs`	1.3943	1.4068	1.3978	1.3457	1.3366	1.3763
`least_outgoing_arcs`	1.0549	1.1010	1.1004	1.1304	1.1453	1.1064
`most_outgoing_arcs`	16.7423	16.5832	16.7124	16.6272	16.5904	16.6512

So, in the end:

Heuristic	**Earthquake**	**Sachs**	**Alarm**
`least_incoming_arcs`	0.0942	0.2845	1.2245
`most_incoming_arcs`	0.0927	0.2513	1.3763
`least_outgoing_arcs`	0.0920	**0.2363**	**1.1064**
`most_outgoing_arcs`	**0.0884**	0.2811	16.6512

`least_outgoing_arcs` prioritises nodes with no children, and as such focuses on leaf nodes. This is likely the cause of the generally good performance of this heuristic. It is not surprising, as one of the suggested methods for picking elimination ordering is by picking leaf nodes first. Interestingly, `most_outgoing_arcs` performes the best for **Earthquake**, while performing horribly for **Alarm**. This is likely a consequence of the difference in network structure.

The results for `least_incoming_arcs` and `most_incoming_arcs` do not allow a black-and-white conclusion to be drawn. `least_incoming_arcs` beats `most_incoming_arcs` fairly handily for **Alarm**, but loses to it for **Sachs**, while the performance is similar for **Earthquake**.

Beyond the performance, each of these heuristics produce identical results in all tested cases, as can be seen in the `verbose_0.txt` files for each combination of BN and heuristic, within the `outputs` folder.

The output for all of the heuristics, on **Earthquake**, for query `'Alarm'` and observed variables and values {`'Burglary': 'True'`} is:

```
{
'Alarm':
       Alarm    prob
    0  False  0.0598
    1   True  0.9402
}
```

These results are correct when compared to AISpace[3].

6 Conclusion

In conclusion, the Variable Elimination algorithm on Bayesian Networks can easily and elegantly be implemented in Python, using clever use of calculations on factors. In doing so, every step of the algorithm in concept can be implemented in just a few lines. The result is a simple yet effective implementation allowing for different elimination ordering heuristics and verbosity levels. In fact, the code is so simple that the lines used for logging take up more space than the actual logic.

The `least_outgoing_arcs` heuristic prioritising leaf nodes performs best on average, though `least_incoming_arcs` and `most_incoming_arcs` also consistently perform up to standard, according to the tests taken. Only `most_outgoing_arcs` was documented to perform considerably worse than the standard, for one of the tested Bayesian Networks.

[3] `http://aispace.org/bayes/`