# Causal Explanation with Background Knowledge

Bhaskara Reddy Moole

bhaskarareddy@wondertechnology.com
School of Management
Walden University,
155 Fifth Ave South
Minneapolis, MN 55401.

Marco Valtorta

mgv@cse.sc.edu
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208.

**Abstract.** This paper presents a new sequential algorithm to answer the question about the existence of a causal explanation for a set of independence statements (a dependency model), which is consistent with a given set of background knowledge. Emphasis is placed on generality, efficiency and ease of parallelization of the algorithm. From this sequential algorithm, an efficient, scalable, and easy to implement parallel algorithm with very little inter-processor communication is derived.

Keywords: uncertainty in AI, Bayesian networks, causality, Distributed AI, background knowledge

## 1. Introduction

Bayesian Networks are proving to be very useful in data mining, machine learning, knowledge acquisition, knowledge representation, and in causal inference [1-3]. Sequential algorithms for the recovery of Bayesian Belief Networks are of polynomial or exponential time complexity and many of them have very little impact on practical problems. With the advancement of VLSI technology and as parallel computers are becoming commonplace, it is important to explore parallel algorithms for Bayesian Network construction. While a fair amount of research on parallel inference in Bayesian networks exists, there is surprisingly little work on parallel learning of Bayesian networks [4-6]. These two papers and the book represent the conditional independence testing and scoring approaches to Bayesian network learning. In this paper a new sequential algorithm is presented to answer the question about the existence of a causal explanation for a set of independence statements (a dependency model), which is consistent with a set of background knowledge. Using this sequential algorithm as the basis, a very efficient, scalable, and easy to implement parallel algorithm with very little inter-processor communication is designed and analyzed. This parallel algorithm is implemented using simulation in c language[7].

## 1 Definitions

*U* is Universe of events. If $P(e_ie_j) = P(e_i)P(e_j)$, then events $e_j$ and $e_i$ are (mutually) statistically independent. Similarly, if $P(e_i,e_j \mid S) = P(e_i \mid S)P(e_j \mid S)$ when $P(S) \neq 0$, then $e_i$ and $e_j$ are statistically independent given *S*, where *S* is any subset of *U* that does not contain $e_i$ and $e_j$. This is also written as $I(e_i, S, e_j)$ [8, 9] and called an independence statement.

A dependency Model is a list M of conditional independence statements of the form I(A,S,B). M is graph isomorphic if all the independencies in M and no independencies outside M can be represented using an undirected graph G. Similarly, M is DAG isomorphic if it can be represented in this manner using a Directed Acyclic Graph (DAG).

A DAG *D* is said to be **I-map** of a dependency model *M* if every *d-separation* condition (as defined in [1, 2]) displayed in *D* corresponds to a valid conditional independence relationship in *M*, i.e. if for every three disjoint sets of vertices *X*, *Y*, and *Z* we have $<X|Z|Y>_D \Rightarrow I(X,Z,Y)_M$.

A DAG is a **minimal I-map** of M if none of its arrows can be deleted without destroying its I-mapness.

## 2 Bayesian Belief Network

Given a probability distribution P on a set of variables V, a DAG D=(V,E) where E is an ordered pair of variables (each of which corresponds to a vertex in graphical representation) of V is called a *Bayesian Belief Network* of P iff D is a minimal I-map of P.

However, results proved by Valtorta [10], Cooper [11], and others show that the synthesis, inference, and refinement of Belief networks is NP-Hard. These results force researchers to focus their efforts on special purpose algorithms, approximate algorithms, and parallel algorithms. Parallel algorithms can be very useful if they are generic and exact, as highly parallel computers are likely to become a commonplace with advancement of technology.

## 1 Algorithm to construct Bayesian Belief Networks

The algorithm presented below is similar to the algorithms presented by Meek [12], Pearl and Verma [9], and Spirtes and Glymour [13]. This algorithm has four phases. Phase 1 is similar to the phase 1 of Pearl and Verma's [9] algorithm (produces a partially directed acyclic graph (pdag) from dependency model). Phase 2 handles background knowledge and phase 3 extends the result of phase 2 (pdag) into a DAG using a simple algorithm reported by Dor and Tarsi [14]. Phase 4 is similar to that of Pearl and Verma's [9]. This algorithm is different from the previously reported algorithms at a gross level. It uses slightly different data structure to represent the graphs (undirected, partially directed, and fully directed graphs). This algorithm is different from Pearl and Verma [9]'s algorithm as theirs cannot handle the background knowledge, and it is different from Meek [12]'s algorithm as Meek handles the background knowledge in a different way (during the extension of partially directed graph of earlier phase, which can cause some extensions not to conform to the background knowledge). The concept of background knowledge is extended and more intuitive types of background knowledge are introduced. Our algorithm ensures that the background knowledge is handled correctly in all possible extensions. The modifications and new concepts introduced in this algorithm can handle more generic knowledge and will provide the domain expert with greater choice. The simplicity and parallelism of our algorithm significantly contribute to the value of the algorithm.

As we mentioned in the previous paragraph, Meek introduced the concept of background knowledge and presented an algorithm [12] to handle background knowledge. That algorithm fails to extend the partially directed acyclic graph (pdag) to conform to the background knowledge in all possible extensions.

This problem can be solved by adding S1 (step 1) of Phase II" of Meek's algorithm to Phase III as S1 (step 1). This modification to Meek's algorithm works correctly because Phase III is the only place where undirected edges are directed but not checked for consistency with Background Knowledge.

In the next section we present a new sequential algorithm that is more generic and prove that background knowledge is handled correctly in all possible extensions.

## 3 Sequential Bayesian Network Construction Algorithm

### 2 Data Structure

A graph is a pair G = (V, E) where V is the set of *n* nodes numbered from 0 to (n-1), and E is the set of ordered pairs of nodes representing edges, exactly *n*x*n*. An edge (a, b) can be directed-outwards (a→b), directed-inwards (a←b), undirected (a—b), non-existing (ab) (also called 'noedge'), or unknown (a?b). An edge is said to be directed if it is directed-outwards or directed-inwards. An edge is of known type if it is not of the unknown type. Two edges between a pair of nodes in the opposite directions (a→b and a←b, one directed-outwards and another directed-inwards) are not equivalent to an undirected edge, but they constitute a directed cycle between these two nodes. Node *a* is adjacent to node *b* if and only if there is an edge directed-outwards (a→b), directed-inwards (a←b), or undirected (a—b). All adjacent nodes of node *a* are also called neighbors of *a*. A node is an Island if it is not adjacent to any other node. An empty graph is a graph with only non-existing edges. An edge is adjacent to another if they share a node. A directed cycle is a set of ordered directed edges which leads us to the starting node by following an adjacent edge in the set in arrow's direction. A DAG is a graph with all its edges directed and with no directed cycles.

### 3 Input

(1) A set M of independence statements of the form I(x, A, y) defined over a set V of *n* variables (dependency model). (2) A consistent set of background knowledge K = {F, R} where F is a graph that represents forbidden edges in the result and R is a graph that represents a set of required edges. Both F and R have the same set of nodes as the considered set of variables in the dependency model. The following tables

show allowed types of corresponding edge in the result for each required edge and forbidden edge.

**Table 2.1**

| An Edge of R | Result is allowed to have |
|---|---|
| undirected (a—b) | directed (a←b or a→b) |
| directed-outwards (a→b) | directed-outwards (a→b) |
| directed-inwards (a←b) | directed-inwards (a←b) |
| noedge (ab) | noedge (ab) |
| unknown (a?b) | directed edge or noedge |

**Table 2.2**

| An Edge of F | Result is allowed to have |
|---|---|
| undirected (a—b) | noedge (ab) |
| directed-outwards (a→b) | directed-inwards (a←b) or noedge (ab) |
| directed-inwards (a←b) | directed-outwards (a→b) or noedge (ab) |
| Noedge (ab) | directed (a←b or a→b) |
| unknown (a?b) | directed (a←b or a→b) or noedge (ab) |

## 4 Output

Result is FAIL or a DAG. If the algorithm is successful and returns a fully oriented graph G (also called DAG D), then the input set of independence statements (or underlying probability distribution or dependency model) is DAG isomorphic , and that result D is consistent with background knowledge (i.e. all the required edges of R and forbidden edges of F have only the allowed types on corresponding edges in D). The DAG D is said to represent a causal explanation of the input set of independence statements (dependency model) [3, 9].

*1. Phase 1:*

Start with an empty graph G on the set of vertices V.

1. Search for an I statement I(x, A, y) for each pair $(x, y) \in E = VxV$, $A \subset \{V \setminus \{x, y\}\}$. If no such I statement is found, connect x and y of G with an undirected edge (x—y). If I statement is found, mark Separator(x, y) with A.
2. For every triplet (x, z, y) such that (s.t.) (x, y) are not adjacent and (x, z) and (y, z) are adjacent, direct edges (x→z) and (y→z) in G if $z \notin$ Separator(x, y).

*2. Phase 2:*

Check for the following Background Knowledge conformances and set possible edges.

1. For each required undirected edge (x—y) in R, if the corresponding edge (x, y) in G is a non-existing edge then FAIL.
2. For each required non-existing edge (xy) in R, if the corresponding edge (x, y) in G is not a non-existing edge then FAIL.
3. For each required directed-outwards edge (x→y) in R, if the corresponding edge (x, y) in G is a non-existing edge (xy) or directed-inwards edge (x←y) then FAIL, else set (x, y) in G with directed-outwards (x→y) edge.
4. For each required directed-inwards edge (x←y) in R, if the corresponding edge (x, y) in G is a non-existing edge (xy) or directed-outwards edge (x→y) then FAIL, else set (x, y) in G with directed-inwards (x←y) edge.
5. For each forbidden undirected edge (x—y) in F, if the corresponding edge (x, y) in G is not a non-existing edge then FAIL.
6. For each forbidden non-existing edge (xy) in F, if the corresponding edge (x, y) in G is a non-existing edge then FAIL.
7. For each directed edge (x→y) in F, if the corresponding edge in G is of type (x→y) then FAIL.
8. Check for directed cycles in G and FAIL if a directed cycle exists.

3. *Phase 3:*
Try to extend G into a DAG.
1. While there are nodes not marked as DELETED, do
(a) select a vertex x not marked with DELETED, which satisfies the following criteria:
 (a1) *x* is a sink (i.e. there is no outward directed edge from *x* in G)
 (a2) there is no edge in F that is directed-inwards w.r.t. *x* s.t. the corresponding edge in G is not marked as DELETED
 (a3) If all the edges incident on *x* in G are directed-inwards w.r.t. *x* or if *x* is an Island, then mark *x* and all the edges incident on *x* as DELETED
 (a4) If there are some edges incident on *x* in G that are undirected, check if for every undirected edge (*x, y*), *y* is a neighbor of all the neighbors of *x* in G. If so, direct all undirected edges (*x, y*) inwards (i.e. *x*←*y*) and mark *x* and all the edges incident on *x* as DELETED
(b) If vertex *x* is not found in step (a), then FAIL. Else Go To 1.

4. *Phase 4:*
Check the faithfulness of DAG D = G.
1. Test that every I statement of M is in D (using d-separation condition).
2. In a total ordering of the nodes of D which agrees with the directionality of edges of D, suppose that parents(a) are the direct parent nodes of a in D and predecessors(a) is the set of all the nodes preceding a without including parents(a) in this ordering. For every node a, test if I(a, parents(a), predecessors(a)) is in M.
3. If both tests are successful return D. Else return FAIL.

## 5 Complexity Analysis

Phase 1 can be completed in O(|M| + $|V|^2$) [9]. Phase 2 requires O($|V|^3$) [12]. Phase 3 can be completed in O($|V|^2$). Phase 4 can be completed in O(|M|*$|V|^2$ + |M|*|V|)[9].

## 6 Proof of Correctness

For Phase 1 steps 1 and 2 refer to Pearl and Verma [9].

Phase 2, Steps 1, 2, 3 and 4: For an undirected edge in R, only type not allowed is a non-existing edge. Therefore, algorithm fails if there is a non-existing edge in G. For a non-existing edge in R, only type allowed is a non-existing edge. If that is not the case algorithm fails. If an edge is directed in R and it is undirected in G, it is set to have the same direction in G as in R, failing otherwise. These steps ensure that algorithm works correctly for all the edges in R.

Phase 2, Steps 5, 6 and 7: For an undirected edge in F, only type allowed is a non-existing edge. Therefore, algorithm fails if it is not a non-existing edge in G. For a non-existing edge in F, a non-existing edge is forbidden in the result. Therefore algorithm fails if it is the case. Then all the directed edges of F are compared with corresponding edges in G to see if their direction is indicated incorrectly (by the dependency model – by following the steps in phase 1). If G has a forbidden directed edge then algorithm fails, thus ensuring correctness. In Phase 3, the remaining edges in G that needs to be directed are checked for forbidden direction in each possible extension and directed only if it is not forbidden. If every possible extension results in a forbidden edge, algorithm fails. These steps ensure that the algorithm works for all the edges in F.

For correctness of phase 3, Dor and Tarsi [14] state their argument in a way similar to the following: (1) A DAG should have a sink. (2) Removing a sink and all the incident edges on it should result in a DAG. (3) Step (a4) prevents new vee-structures being introduced (Three nodes, say a,b,c, form a vee structure if a→bb←cTherefore, Phase 3 extends partially directed acyclic graph (pdag) into a fully directed acyclic graph (DAG) without introducing new vee-structures, if the pdag is extendible.

Phase 4, steps 2 and steps 3 are to ensure that the result is faithful with dependency model. For proof of correctness of these two steps refer to Pearl and Verma[9].

## 4 Intuition behind Background Knowledge:

We have introduced four basic types for background knowledge that directly correspond to the types of knowledge of a domain expert. An unknown piece of knowledge means that dependency model indications are the only source in the causal explanation. A directed edge indicates that domain expert knows the cause and effect (or temporal relationship). An undirected edge means that the domain expert knows about existence of the local relationship (existence of dependency) but not the temporal or causal relationship. A non-existing edge indicates that the domain expert knows the non-existence of the relationship. These pieces of knowledge can be either in the form of required or forbidden relationships.

When we try to find a complete causal explanation from both 'Dependency Model' and 'Background Knowledge,' and they conflict with each other, we have three possibilities: (1) Constructing a complete causal explanation FAILS. (2) Dependency

Model overrides the Background Knowledge. (3) Background Knowledge overrides the Dependency Model.

Our algorithm shown above is for the case 1. This algorithm can be used for the other two cases with a few simple modifications. If we want to override background knowledge with dependency model, wherever the algorithm FAILS we ignore the background knowledge. For case 3 we will use the edge type indicated by the domain expert and continue with the rest of the algorithm in the normal way.

## 2. Parallel Construction of Bayesian Belief Networks

### 5 Parallel Computer Model

This section describes the Parallel Computer Model used to develop the Bayesian Belief Network Construction Algorithm, which is based on the sequential algorithm described earlier. The Parallel Computer Model for which this algorithm is developed is well known Parallel Random Access Memory (PRAM) model.

A PRAM consists of $p$ general-purpose processors, $P_0$, $P_1$, $P_1$, … $P_{p-1}$, all of which are connected to a large shared, random access memory SM. The Processors have a private, or local, memory for their own computation, but all communication among them takes place via shared memory.

While this looks unrealistic, it is simple and easy to simulate on real systems and algorithms developed for this model can easily be analyzed for target computer with straightforward conversions. A detailed discussion of Parallel Computer Models can be found in [11, 15-17] and is beyond the scope of this paper.

The following two definitions will also be used in the later parts of the presentation.

(1) **Speedup**: Let Sequential Time Complexity of Problem P be $T^*(n)$ for an input of size $n$. Let $T_p(n)$ be the time required to solve P using the parallel algorithm with $p$ processors. Then speedup achieved by parallel algorithm is defined as:

$$S_p(n) = T^*(n)/T_p(n)$$

(2) **Efficiency**: Efficiency of the parallel algorithm is defined as:

$$E_p(n) = T_1(n)/pT_p(n)$$

### 6 Algorithmic Notation

The algorithmic notation used below is same as the one described in [14] and is easy to understand. Where it is appropriate pure English description is used to simplify the algorithm.

## 7 Parallel Algorithm

Assumptions: In the following, n, $n^2$ and $n^3$ are divisible by *p*. Nodes are numbered from 0 to (n-1). Processors are numbered from 0 to (p-1).

**Input**: (1) The processor number $p_k$ is available to each processor. (2) The total number of processors *p* is also available to each processor. (3) An empty DAG D = (V, E) is available in shared memory SM. (4) The number of nodes in DAG *n* is available to each processor. (5) The set of background knowledge K = {F, R} where F is the Graph that contains forbidden edges and R is the Graph that contains required edges. (6) A function get_I_StatementSeparatingAandB( node a, node b ) which returns an Independence Statement that has the form I(a,S,b) or NULL if no such statement is found. This function may be implemented in different ways depending on the actual input that is available. As we know, a set of I statements in the Dependency Model M will grow exponentially as the number of variables grows. It may be impractical to assume that M is available through explicit enumeration of I-statements. There are possibilities of representing a basis L where logical closure of L is M (i.e. CL(L) = M). Implementing this function in a most efficient manner is out of the scope of this project. Another, possibility is to implement the above function to return the answer by searching through the probability distribution for I-statements with the requested qualification. This assumption simplifies the development of algorithm greatly. Each call to this function counts as *m* operations for the complexity analysis.
.

**Output**: A Directed Acyclic Graph (DAG) D = (V, E) where V is the set of *n* nodes, and E is the set of ordered pairs of nodes representing edges, exactly *n*x*n*. If the algorithm is successful and returns a fully oriented DAG D, then the considered set of independence statements or probability distribution is DAG isomorphic and has a causal explanation and D graphically represents that consistent set of independence statements and background knowledge. D will be available in SM at the end of successful execution.

We also assume there exist the following ***constant time O(1) functions*** to perform various operations.

| | |
|---|---|
| connectEdge( node a, node b ) | makes (a—b) in a dag |
| deleteEdge( node a, node b ) | makes (ab) in a dag |
| directEdge( node a, node b ) | makes (a$\rightarrow$b) in a dag |
| isEdgeDeleted( node a, node b ) | answers yes if (a, b) is marked deleted |
| EdgeDirection( node a, node b ) | answers: directed, undirected, noedge |
| directedOutwards(node a, node b) | answers yes if (a$\rightarrow$b) |
| directedInwards(node a, node b) | answers yes if (a$\leftarrow$b) |
| isUndirectedEdge(node a, node b) | answers yes if (a—b) |
| markEdge( a, b, S) | mark each edge with a statement I(a,S,b) |
| deleteNode( node a ) | deletes node *a* and all the edges incident to it |
| isNodeDeleted( node a ) | answers yes if node *a* was already deleted |
| isNonEmptyDag( DAG d ) | answers yes if some nodes are not deleted |
| isDescendant( node a, node b ) | is node b a descendant of node *a*? |
| isHeadToHead( node a, node b, node c ) | does this triplet form head-to-head node at *c*? |
| isLabeled( label l, node a, node b ) | is edge between *a* and *b* labeled with *l*? |

| | |
|---|---|
| isAdjacent( node a, node b ) | answers question: is *a* adjacent to *b*? |
| areAdjacentEdges(node a, node b, node c, node d) | are (a, b) and (c, d) adjacent? |
| LabelEdge( label l, node a, node b ) | label this edge between *a* and *b* with *l* |
| LabelNode( node a ) | label *a* as reachable (with constant label R) |
| getAll_I_Statements() | returns all the I statements in M. |
| NodeHasNoForbiddenEdges( node a ) | YES if *a* has no inward directed edge in F |
| isNotConsistentWithBackgroundKnowledge(node a, node b) | returns YES if edge is consistent with K |
| findDirectionFromBackgroundKnowledge(node a, node b) | find correct direction for the edge |

***and other O(n) sequential functions***:

| | |
|---|---|
| doesBelongToSet(node a, S) | answers yes if a belongs to separating set S |

(This will also be written as $a \in S$ or the negation of it as $a \notin S$)

| | |
|---|---|
| orientEdgesTowards( node a ) | makes (a←x) where x isAdjacent to a |

**Algorithm**

```
begin
1. for i := 0 to $n^2/p$ do
j := i + $p_k$ * ($n^2/p$)
a := int (j/n)
b := j%n
S := get_I_StatementSeparatingAandB( a, b )
if ( S $\neq$ NULL) then
begin
markEdge( a, b, S )
global write( D(a, b) = S)
end
else
begin
connectEdge( a, b )
global write( (a, b) )
end
2. global read (D)
3. for i := 0 to $n^3/p$ do
j := i + $p_k$ * ($n^3/p$)
a := int ($j/n^2$)
b := int (j/n)%n
c := int (j%n)
if ( (! isAdjacent( a, b ) ) and
   (  isAdjacent( a, c ) ) and
   (  isAdjacent( b, c ) ) and
```

```
    (   c ∉ S( a, b ) ) )
then
begin
directEdge( a, c )
directEdge( b, c )
global write ( ( a, c ), ( b, c ) )
end
4. for i := 0 to $n^2/p$ do
j :=  i + $p_k$ * $(n^2/p)$
a := int (j/n)
b := j%n
if ( isNotConsistentWithBackgroundKnowledge( a, b ) ) then
begin
return FAIL
end
findDirectionFromBackgroundKnowledge( a, b )
global write ( a, b )
5. Construct Transitive Closure of this partially directed graph D
6. If there is a directed cycle in D, then return FAIL.
7. u := n
8. for i := 0 to n/p do
sink[i] := 2
9. for i := 0 to n/p do
SM(sink[i]) := 0
10. while ( u != 0 ) do
for i := 0 to $n^2/p$ do
j :=  i + $p_k$ * $(n^2/p)$
a := int (j/n)
b := j%n

if ( isNodeDeleted( a ) ) then
begin
sink[a] := 0
continue
end
if ( isNodeDeleted( b ) ) then
begin
sink[b] := 0
continue
end
if ( directedOutwards( a, b ) ) then
begin
sink[a] := 0
AdjacencySet[b][a] := 1
end
else if ( directedInwards( a, b ) ) then
begin
```

```
sink[b] := 0
AdjacencySet[a][b] := 1
end
else if ( isUndirectedEdge( a, b ) ) then
begin
sink[a] := 1
sink[b] := 1
AdjacencySet[a][b] := 1
AdjacencySet[b][a] := 1
allAdjacencySet[a][b] := 1
allAdjacencySet[b][a] := 1
end

flag := 1
for i := 0 to $n^3/p$ do
j := i + $p_k$ * ($n^3/p$)
a := int ($j/n^2$)
b := int (j/n)%n
c := int (j%n)
if ( sink[a] = 0 ) then
begin
continue
end
else if ( sink[a] = 2 ) then
begin
flag = 1
break
end
else if ( AdjacencySet[a][b] = 0
        or AdjacencySet[a][c] = 0 ) then
begin
continue
end
else if ( ! isAdjacent( b, c ) )  then
begin
flag = 0
break
end

if ( flag = 1) then
begin
global write ( SM(sink[a]) = 1 )
end
for  i := 0 to n/p do
global read (a := SM(sink[i]))
if ( a = 1 && NodeHasNoForbiddenEdges( a ) ) then
begin
```

```
global write (SM(sink) := a)
end
global read ( j := SM(sink))
orientEdgesTowards ( j )
deleteNode ( j )
u := u-1
11. M := getAll_I_Statements()
12. m := number of I statements in M
13. Construct Closure of D for descendants
14. for i := 0 to m/p do
Use d-separation condition on D and answer whether M[i] is implied by D. If M[i]
is not implied by D, return FAIL.
15. Generate an ordering of nodes D using breadth first traversal or depth first traversal.
16. for i := 0 to n/p do
Use d-separation condition and test if each node is shielded from all its predecessors given its parents. If not return FAIL.
17. DAG D in SM is the result.
end
```

## Complexity Analysis

We analyze each step of the algorithm separately and then conclude on the overall complexity of the algorithm.

**Step 1** of the algorithm can be completed in $O(m + n^2/p)$ time, $O(n^2)$ operations and $O(n^2)$ communications, where *m* is the number of operations performed by get_I_StatementSeparatingAandB(a,b). The time complexity approaches $O(m)$ with $n^2$ processors.

Therefore, $S_p(n) = O(m + n^2) / O(m + n^2/p)$

If *m* is a constant, $S_p(n) = O(n^2) / O(n^2/p)$ and $S_p(n) \approx p$ as *p* approaches $n^2$.

Similarly, efficiency of the algorithm in step 1 is $E_p(n) = (m + n^2) / p*(m + n^2/p)$

If *m* is a constant, $E_p(n) = (n^2) / (p*n^2/p) \approx 1$

Since Speedup of *p* and efficiency of 1 are the optimality indicators, this part of the algorithm achieves ideal performance for $p=n^2$.

If *m* is a constant, time complexity approaches $O(1)$ as p approaches $n^2$, and can not be improved further.

**Step 2** of the algorithm needs $O(n^2)$ communications.

**Step 3** of the algorithm can be completed in $O(n^3/p)$ time, $O(n^3)$ operations and $O(n^3)$ communications. The time complexity approaches $O(1)$ as p approaches $n^3$ and can not be improved by adding more processors. Therefore, $S_p(n) \approx p$, and $E_p(n) \approx 1$ for this step. This part of the algorithm achieves ideal performance for $p=n^3$.

**Step 4** requires $O(1)$ (constant time) with $n^2$ processors.

**Step 5** requires $O(\log n)$ time using $O(n^3 \log n)$ operations on CRCW PRAM, or in $O(\log^2 n)$ time using $O(M(n)*\log n)$ operations on CREW PRAM, where $M(n)$ is the best known sequential bound for multiplying two *n*x*n* matrices, according to [17] [pages 249-250].

**Step 6** requires O(1) time with n processors to find directed cycle with the result of the step 4.

**Step 7** is constant time O(1) operation.

**Step 8** is initialization of local sink array and can be done in constant time with n processors.

**Step 9** is initialization of global sink array and can be done in constant time with n processors.

**Step 10** While loop (outer loop) runs in O(n). Complexity analysis for the first *for* loop is exactly same as for step 1. That is, as p approaches $n^2$ time complexity approaches O(1). Complexity analysis for second *for* loop is exactly same as step 2. Therefore, as p approaches $n^3$ time complexity approaches O(1). Hence, this step has a time complexity of O(n).

**Step 11** is a constant time O(1) operation.

**Step 12** is constant time O(1) operation.

**Step 13** requires O(log n) time using $O(n^3 \log n)$ operations on CRCW PRAM, or in $O(\log^2 n)$ time using O(M(n)*log n) operations on CREW PRAM, where M(n) is the best known sequential bound for multiplying two *n*x*n* matrices, according to [17] [pages 249-250].

**Step 14** can be completed in O(m/p) time with O(p) communications. However, noting that m grows exponentially as n grows in explicit enumeration method, it is not practical to analyze this step thoroughly without knowing representation method.

**Step 15** Can be completed in linear time.

**Step 16** requires O(n) with n processors.

**Step 17** returns result and requires constant time.

***Conclusions on Complexity Analysis***: Using this algorithm, with at most $n^3$ number of processors, a given set of independence statements can be verified if they have a causal explanation in time O(n) if m is a constant or $m<n$. Otherwise O(m) if $m > n$.

## 8 Proof of Correctness for Parallel Algorithm

A formal proof of correctness is presented in [18]. A simulated implementation is presented in [7].

## 9. Conclusion

We presented a new constraint-based algorithm for learning Bayesian networks from data, analyzed its complexity and proved its correctness. In particular, the algorithm handles correctly several types of background knowledge, as an expert could provide it, thus extending and correcting an algorithm by Meek [12]. From the sequential algorithm, we derived an efficient, scalable, and easy to implement parallel algorithm for the PRAM model of computation, and proved its correctness using number-theoretic arguments.

## References

1. Pearl, J., *Probabilistic Reasoning in Intelligent Systems*. 1988, San Mateo, CA: Morgan Kaufmann.
2. Jensen, F.V., *Bayesian Networks and Decision Graphs*. 2001, New York, NY: Springer.
3. Pearl, J., *Causality*. 2001, Cambridge, UK: Cambridge University Press.
4. Lam, W. and A.M. Segre, *A Parallel Learning Algorithm for Bayesian Inferece Networks.* IEEE Transactions on Knowledge and Data Engineering, 2002. **14**(1): p. 159-208.
5. Mechling, R. and M. Valtorta, *A Parallel Constructor of Markov Networks*, in *Selecting Models from Data: Artificial Intelligence and Statistics IV*, P. Cheeseman and R.W. Oldford, Editors. 1994, Springer: New York, NY. p. 255-261.
6. Neapolitan, R.E., *Learning Bayesian Networks*. 2004, Upper Saddle River, NJ: Pearson Prentice Hall.
7. Moole, B.R., *Parallel Construction of Bayesian Belief Networks*, in *Dept of Computer Science*. 1997, University of South Carolina: Columbia, SC, USA.
8. Rao, C.R., *Linear Statistical Inference And Its Applications*. 1973: John Wiley & Sons.
9. Verma, T. and J. Pearl. *An Algorithm for deciding if a set of observed independencies has a causal explanation*. in *Proceedings of the 8th Conference on Uncertainty in AI*. 1992. Stanford, CA: 323-330.
10. Valtorta, M. and D.W. Loveland, *On the complexity of Belief Network Synthesis and Refinement.* International Journal of Approximate Reasoning, 1992. **7**(3-4): p. 121-148.
11. Cooper, G.F., *The complexity of probabilistic inference using Bayesian Belief Networks.* Artificial Intelligence, 1990. **42**: p. 393-405.
12. Meek, C. *Causal Inference and causal explanation with background knowledge*. in *Proceedings of 11th Conference on Uncertainty in AI*. 1995. San Mateo, CA: Morgan-Kaufman.
13. Spirtes, P. and C. Glymour, *An algorithm for fast recovery of sparse causal graphs.* Social Science Computer Review, 1991. **9**(1).
14. Dor, D. and M. Tarsi, *A simple algorithm to construct a consistent extension of a partially oriented graph*. 1992, Cognitive Systems Laboratory, Dept of CS, University of California at Los Angeles (UCLA): Los Angeles, CA.
15. Almasi, G.S. and A. Gottlieb, *Highly Parallel Computing*. 1989: Benjamin/Cummings Publishing Company.
16. Baase, S., *Computer Algorithms: Introduction to Design and Analysis*. 2 ed. 1988: Addison-Wesley Publishing Company.
17. JaJa, J., *An Introduction to Parallel Algorithms*. 1992: Addison-Wesley Publishing Company.
18. Moole, B. and M. Valtorta, *Causal Explanation with Background Knowledge*. 2002, Dept of CSE, University of South Carolina: Columbia, SC 29208, USA.