International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems
© World Scientific Publishing Company

SEQUENTIAL AND PARALLEL ALGORITHMS FOR CAUSAL EXPLANATION WITH BACKGROUND KNOWLEDGE

BHASKARA REDDY MOOLE
School of Management
Walden University
155 Fifth AvenueSouth
Minneapolis, MN 55401, USA
bhaskarareddy@wodertechnology.com

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

Received xx yyyy 2004
Revised xx yyy 2004
Accepted xx yyyyyy 2004

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; artificial intelligence;, Bayesian networks; causality; dependency; background knowledge.

1. Introduction and Motivation

Bayesian Belief 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 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, 5]. The two papers just cited represent the scoring approach and the conditional independence testing approach to Bayesian network learning, respectively. 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. A simulated implementation of this parallel algorithm in the C language and related experimental results are presented in another paper[6].

1.1 *Definitions*

Let U be a Universe of events. An event e_i is statistically independent of another event e_j if $P(e_i \mid e_j) = P(e_i)$. Similarly, e_j is statistically independent of e_i if $P(e_j \mid e_i) = P(e_j)$. If both are true, then $P(e_ie_j) = P(e_i)P(e_j)$, which implies 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 $I(e_i, S, e_j)$ and called an independence statement.[7,8]

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 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.

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 and Cooper (among others) show that the synthesis, inference, and refinement of Belief networks is NP-Hard.[9,10] 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 commonplace with as technology advances.

2 Algorithm to construct Bayesian Belief Networks

The algorithm presented below is similar to the algorithms presented by Meek, Pearl and Verma, and Spirtes and Glymour.[11,8,12] This algorithm has four phases. Phase 1 is similar to phase 1 of Pearl and Verma's algorithm (and produces a partially directed acyclic graph (pdag) from a dependency model).[8] 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.[13] Phase 4 is similar to that of Pearl and Verma's.[8] 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's algorithm as

theirs cannot handle the background knowledge, and it is different from Meek's algorithm as Meek handles the background knowledge in a different way (during the extension of the partially directed graph computed in an earlier phase, which can cause some extensions not to confirm to the background knowledge)[8,11] 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 to handle background knowledge.[11] 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.

2.1 *Bayesian network construction algorithm*

2.1.1 Data structure

A Graph 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 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.

2.1.2 Input

(1) A set M of independence statements of the form I(x, A, y) defined over a set 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 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)

Table 2.2

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

2.1.3 Output

The 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 has a causal explanation, 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), which graphically represents the causal explanation for that set of independence statements or dependency model. We note that, in general, there are multiple directed graphs that correspond to an input set of independence statements, and not all of them are causally interpretable. The presence of background knowledge reduces the number of possible directed graphs and makes it more likely for the graph constructed by the algorithm in this paper to admit a causal interpretation. We therefore use the term “causal explanation” in this paper, while admitting that in some cases the graph returned by the algorithm is not causal.

2.1.4 Method

Phase 1:
Start with an empty graph G on the set of vertices V.
Search for an independence statement (I statement) I(x, A, y) for each pair (x, y) $\in$ E = VxV, A $\subset$ {V \ {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.
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).

Phase 2:
Check for the following Background Knowledge conformances and set possible edges.
For each required undirected edge (x—y) in R, if the corresponding edge (x, y) in G is a non-existing edge then FAIL.
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.
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.
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.
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.
For each forbidden non-existing edge (xy) in F, if the corresponding edge (x, y) in G is a non-existing edge then FAIL.
For each directed edge (x→y) in F, if the corresponding edge in G is of type (x→y) then FAIL.
Check for directed cycles in G and FAIL if a directed cycle exists.

Phase 3:
Try to extend G into a DAG.
While there are nodes not marked as DELETED, do
select a vertex x not marked with DELETED, which satisfies the following four 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. 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

If vertex *x* is not found in step (a), then FAIL. Else Go To 1.

Phase 4:
Check the faithfulness of DAG D = G.
Test that every I statement of M is in D (using d-separation condition).
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.
If both tests are successful return D. Else return FAIL.

2.1.5 Complexity analysis

Phase 1 can be completed in $O(|M| + |V|^2)$.[8] Phase 2 requires $O(|V|^3)$.[11] Phase 3 can be completed in $O(|V|^2)$. Phase 4 can be completed in $O(|M|*|V|^2 + |M|*|V|)$.[8]

2.1.6 Proof of correctness of sequential algorithm

For Phase 1 steps 1 and 2 refer to Pearl and Verma.[8]

Phase 2, Steps 1, 2, 3 and 4: For an undirected edge in R, the only type not allowed is a non-existing edge. Therefore, the algorithm fails if there is a non-existing edge in G. For a non-existing edge in R, the only type allowed is a non-existing edge. If that is not the case, the 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, the only type allowed is a non-existing edge. Therefore, the algorithm fails if the corresponding edge in G is not a non-existing edge. For a non-existing edge in F, a non-existing edge is forbidden in the result. Therefore the algorithm fails if it is the case. Then all the directed edges of F are compared with the 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 need 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, the algorithm fails. These steps ensure that the algorithm works for all the edges in F.

For correctness of phase 3, Dor and Tarsi 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.[13] Therefore, 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 to the dependency model. For proof of correctness of these two steps refer to Pearl and Verma.[8]

2.2 *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 the domain expert knows the cause and effect relationship. An undirected edge means that the domain expert knows about existence of the local relationship (existence of dependency) but not the 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 the dependency model and the background knowledge, and they conflict with each other, we have three possibilities: (1) constructing a complete causal explanation FAILS; (2) the dependency model overrides the background knowledge; (3) the background knowledge overrides the dependency model.

Our algorithm shown above is for case 1. This algorithm can be used for other two cases by few simple modifications. If we want to override background knowledge with dependency model, wherever the algorithm FAILS we ignore 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.

3 Parallel Construction of Bayesian Belief Networks

3.1 *Parallel computer model and algorithmic notation*

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 the well known Parallel Random Access Memory (PRAM) model.

A PRAM consists of p general-purpose processors, P_0, P_1, P_2, … 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 computers with straightforward conversions.

The Bulk Synchronous Parallel (BSP) model concentrates on the synchronization of the processors at regular intervals, making it suitable to implement/translate sequential programs. Its main aim is to provide a bridge between software and hardware, and to avoid the burden on the programmer of managing memory, assigning communication, and performing low-level synchronization.[14] Our algorithm does not require synchronization at regular intervals and its requirement for inter-processor communication is very modest. The properties of our algorithm are maintained in the BSP model. By designing the algorithm for the PRAM model and implementing it a BSP

computer does not destroy any of its properties. The LogP parallel computer model reflects communication costs better the PRAM and BSP models, and it may be a more realistic model of today's parallel computers.[15] However, our algorithm requires very little inter-processor communication, and hence the impact of selecting PRAM is negligible. The implementation of our algorithm on a LogP computer does not destroy any of its properties. It is also easy to convert the sequential algorithm to work efficiently on BSP and LogP models due to its inherent parallelism. A detailed discussion of parallel computer models and parallel algorithm design can be found elsewhere and is beyond the scope of this paper.[16,17,18]

The period of time from 1975 to 1990 witnessed a rapid advancement of parallel architectures. The relative lull of 1990s was followed by massive parallelization (cluster computing, Symmetric Multi Processing (SMP), Grid computing). Most practical algorithms, and especially the ones that are implemented in any programming language, require polynomial time or less. Even then the parallelization trend continued its momentum. This is simply because the existing programs stretch the one resource that humans do not have control on, viz. time. Any savings achieved in this aspect are worth considering. Our algorithm achieves ideal Speedup and ideal Efficiency (as defined below) when parallelized converting a polynomial task to a worst case *linear* task, making the parallel version very attractive for constructing the causal explanations. We also note that our algorithm completes most operations in constant time, as shown in the complexity analysis provided later in the paper.

The following two definitions will also be used in the later parts of the presentation.
Speedup: Let the 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 a parallel algorithm with p processors. Then speedup achieved by the parallel algorithm is defined as:
$S_p(n) = T^*(n)/T_p(n)$
Efficiency: Efficiency of the parallel algorithm is defined as:
$E_p(n) = T_1(n)/(pT_p(n))$

The algorithmic notation used below is same as the one described by JaJa and is easy to understand.[18] Where it is appropriate, pure English description is used to simplify the algorithm.

3.2 *Parallel algorithm*

In the following, we assume that 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).

3.2.1 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 the 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 the algorithm greatly. A call to this function will be counted as *m* operations for the complexity analysis.

3.2.2 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.

3.2.3 Auxiliary Functions

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→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→b)
directedInwards(node a, node b)	answers yes if (a←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) 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) answers yes if a has no inward directed edge in F
isNotConsistentWithBackgroundKnowledge(node a, node b)
answers yes if edge is consistent with K
findDirectionFromBackgroundKnowledge(node a, node b)
find correct direction for the edge

and other O(n) sequential function(s):
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$\leftarrow$x) where x isAdjacent to a

3.2.4 Method

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 $\notin$ 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
```

Appendix B contains the complexity analysis of the above parallel algorithm, which proves claims of ideal speedup and ideal efficiency. Appendix C contains the proof of correctness of the parallel algorithm.

4 Conclusions and Future Work

This paper presented a simple intuitive sequential algorithm to recover Bayesian networks, which overcomes some of the problems identified in earlier algorithms when handling the background knowledge. The concept of background knowledge is addressed in a more general way than in earlier algorithms.

There is surprisingly little published research on parallel construction (learning) of Bayesian networks. A very comprehensive and recent treatise on Bayesian network learning only lists two references.[19,4,5] (We are also aware of work by Xiang and Chu.[20] They concentrate on learning Markov networks but are unaware of the other referenced work on the topic and state that "to the best of [their] knowledge, [their paper] is the first investigation on parallel learning of belief networks.") Any additional work on this topic, such as the parallel algorithm presented in this paper, should be considered a substantial contribution to the state of the art.

Similarly, there has not been a lot of research on learning Bayesian networks in the presence of background knowledge since the work of Meek.[12] Experience with applications of Bayesian networks indicates that situations in which learning happens in the presence of background knowledge (represented as forbidden and required edges) is common and is not well supported by existing learning algorithms. This paper contributes new results in this area also, by improving Meek's sequential algorithm.

The unique feature of our work is that it presents a *parallel* algorithm for learning Bayesian networks *in the presence of background knowledge*. The parallel algorithm for Bayesian network learning that we present in this paper is correct, efficient and scalable. The efficiency and scalability of the algorithm are based on a (possibly surprising) decomposition of work over each edge of the graph, whose proof of correctness is not trivial and based on number-theoretic properties.

The first author of this paper has implemented this algorithm on a virtual parallel computer, with results that confirm the theoretical analysis.[6] Further discussion of the implementation is outside the scope of this paper. We expect to continue the theoretical side of our work by taking advantage of recent results in the enumeration of PDAGs

(more commonly called essential graphs in the recent literature). This has the potential of restricting the search of Bayesian network structures to the space of PDAGs directly, rather than the larger space of DAGs. The exploitation of these results in the context of parallel algorithms in the presence of background knowledge remains a largely unexplored area for future work.

Acknowledgments

This work was supported in part by the Advanced Research and Development Activity (ARDA), an entity of the U.S. Government. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government.

References

1. Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 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 Inference Networks," *IEEE Transactions on Knowledge and Data Engineering*, 2002, **14**(1): pp. 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. Moole, B.R., "Parallel Construction of Bayesian Belief Networks" (Research Thesis), Dept of Computer Science, 1997, University of South Carolina: Columbia, SC, USA.
7. Rao, C.R., *Linear Statistical Inference and Its Applications*. 1973: John Wiley & Sons.
8. 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, pp. 323-330.
9. Valtorta, M. and D.W. Loveland, "On the complexity of Belief Network Synthesis and Refinement," *International Journal of Approximate Reasoning*, 1992 **7**(3-4): pp. 121-148.
10. Cooper, G.F., "The complexity of probabilistic inference using Bayesian Belief Networks," *Artificial Intelligence*, 1990. **42**: pp. 393-405.
11. 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.
12. Spirtes, P. and C. Glymour, "An algorithm for fast recovery of sparse causal graphs," *Social Science Computer Review*, 1991. **9**(1).
13. Dor, D. and M. Tarsi, "A simple algorithm to construct a consistent extension of a partially oriented graph," Technical Report, 1992, Cognitive Systems Laboratory, Dept of CS, University of California at Los Angeles (UCLA): Los Angeles, CA.
14. Valiant, L., "A bridging model for parallel computation," *Communications of the ACM*, 1990. **33**(8): pp. 103-111.
15. Culler, D., et al. "LogP: Torwards a Realistic Model of Parallel Computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1993.
16. Almasi, G.S. and A. Gottlieb, *Highly Parallel Computing*. 1989: Benjamin/Cummings Publishing Company.

17. Baase, S., *Computer Algorithms: Introduction to Design and Analysis*. 2nd ed. 1988: Addison-Wesley Publishing Company.
18. JaJa, J., *An Introduction to Parallel Algorithms*. 1992: Addison-Wesley Publishing Company.
19. Neapolitan, R.E., *Learning Bayesian Networks*. 2004, Upper Saddle River, NJ: Pearson Prentice Hall.
20. Xiang, Y and T. Chu. "Parallel Learning of Belief Networks in Large and Difficult Domains," Data Mining and Knowledge Discovery, 1999, **3**, pp.315-339.
21. Knuth, D., *Fundamentals of Algorithms*. 2nd ed. (The Art of Computer Programming, Vol. 1), 1973: Addison-Wesley Publishing Company.

Appendix A: List of Symbols and Abbreviations

$\rightarrow$	directed-outwards edge
$\leftarrow$	directed-inwards edge
$\forall$	for all values of
$\subset$	is a subset of
$\not\subset$	is not a subset of
$\in$	belonging to or member of
$\notin$	does not belong to or not a member of
?	unknown type edge
Δ	difference quantity
$\leq$	less than or equal to
$\neq$	is not equal to
$\Leftarrow$	implied by
$\Leftrightarrow$	implied by and implies
$\Rightarrow$	implies
s.t.	such that
w.r.t.	with respect to
wkt	we know that

Appendix B: Complexity Analysis of Parallel Algorithm

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) as the number of processors approaches n^2.

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, the 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 a speedup of *p* and an efficiency of 1 indicate optimality, this part of the algorithm achieves ideal performance for $p=n^2$.
If *m* is constant, the time complexity approaches O(1) as p approaches n^2, and cannot 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 cannot 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 Jaja [pages 249-250].[18]
Step 6 requires O(1) time with n processors to find a directed cycle in the graph resulting from step 5.
Step 7 is a constant time O(1) operation.
Step 8 is the initialization of the local sink array and can be done in constant time with n processors.
Step 9 is the initialization of the global sink array and can be done in constant time with n processors.
Step 10 While loop (outer loop) runs in O(n). The complexity analysis for the first *for* loop is exactly the same as for step 1. That is, as p approaches n^2, the time complexity approaches O(1). The complexity analysis for the second *for* loop is exactly the same as step 2. Therefore, as p approaches n^3, the 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 Jaja [pages 249-250].[18]
Step 14 can be completed in O(m/p) time with O(p) communications. However, noting that m grows exponentially as n grows if using the explicit enumeration method, it is not practical to analyze this step thoroughly without knowing the representation method used for the set of independence statements (the independence model).
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: This algorithm, with at most n^3 number of processors, can verify whether a given set of independence statements has a causal explanation in time O(n), if m is a constant or m<n, or in time O(m) if m > n.

Appendix C: Proof of Correctness of Parallel Algorithm

The formal proof of correctness is established using several lemmas and theorems. Informally, all the edges can be computed (learned) independently, i.e., without communicating with other processors. As we may note, a DAG can be described fully by just using the edges, so does the process of computing a DAG. Therefore, every step of the algorithm uses all the available processors to do the computation simultaneously and thus achieving an ideal speedup and ideal efficiency. The following proof of correctness heavily uses number theory. Knuth provides a very useful discussion of related topics.[21] The proof uses several properties of the quantity represented on the right-hand side of the following two assignments:

Equation I: $j := i + p_k * (n^2/p)$, $\forall i \ 0 \le i \le ((n^2/p)-1), \forall p_k \ 0 \le p_k \le (p-1), 1 \le p \le n^2$

Equation II: $j := i + p_k * (n^3/p)$, $\forall i \ 0 \le i \le ((n^3/p)-1), \forall p_k \ 0 \le p_k \le (p-1), 1 \le p \le n^3$

Lemma C.1: Equation I will produce j values s.t. $0 \le j \le (n^2 - 1)$.

Proof: From theories of integer addition, we know that (*wkt*) any two varying integer terms result in their lowest sum when both terms are at their lowest values and in their highest sum when both are at their highest values. The lowest value for both i and p_k is 0; hence the lowest value of j is 0. The highest value for j is obtained by substituting the highest values of the individual terms:

$j = ((n^2/p)-1) + (p-1) * (n^2/p) = (n^2/p) - 1 + p*(n^2/p) - (n^2/p) = n^2 - 1$

Therefore j, $0 \le j \le (n^2-1)$. □

Lemma 3.2: Equation I produces unique integer values for j.

Proof: Let $C_k = p_k * (n^2/p)$. If we let p_k to be constant at any particular value then C_k is constant. Since the value of i varies from 0 to $((n^2/p)-1)$ and each value for i is a unique integer, we obtain a set of unique integers varying from $0+C_k$ to $n^2/p+C_k$. Similarly, for any value of i, C_k varies from 0 to $(p-1)*(n^2/p)$ with unique integer values.

Uniqueness of the sum of these two terms can be proved by contradiction. Suppose we have two sums $(i_1 + C_{k1})$ and $(i_2 + C_{k2})$ which are not unique. If these two integer terms are not unique, then

$i_1 + C_{k1} = i_2 + C_{k2}$

$i_1 - i_2 = C_{k2} - C_{k1}$

$\Delta_i = \Delta_{ck}$

where Δ_i is the difference between any two i,

and Δ_{ck} is the difference between any two C_k

Since i varies from 0, 1, 2, ... $((n^2/p)-1)$, the highest difference between any two i can be $|((n^2/p)-1) - 0| = (n^2/p) - 1$.
Since C_k varies from 0, n^2/p, $2*n^2/p$, $3*n^2/p$, ... $(p-1)*n^2/p$, the lowest possible difference is $|n^2/p - 0| = (n^2/p)$. This implies that $\Delta_i < \Delta_{ck}$ always. This is contradictory to the above assumption, which says these two terms can be equivalent.
Therefore, the sum of these two terms must always be unique.
Moreover, i can have (n^2/p) values and p_k can have p values, therefore the total number of sums is $(n^2/p)*p = n^2$. □

Theorem C.1: Equation I produces unique consecutive integer values for j from 0, 1, ... (n^2-1).
Proof: From Lemma C.1, j, $0 \le j \le (n^2-1)$ and from Lemma C.2 j has n^2 unique values. It follows that j has all consecutive integer values from 0 to (n^2-1), as there are exactly n^2 integers in that inclusive range. □

Lemma C.3: In step 1 of the algorithm, the value of *a* varies from 0 to (n-1).
Proof:
$0 \le j \le (n^2-1)$
$0/n \le j/n \le (n^2-1)/n$
$\text{int}(0/n) \le \text{int}(j/n) \le \text{int}((n^2-1)/n)$
$0 \le a \le (n - 1)$
Moreover, as j has all the integer values from 0 to (n^2-1) it follows that a has all the integer values from 0 to (n-1). □

Lemma C.4: In step 1, the value of *b* varies from 0 to (n-1).
Proof:
$0 \le j \le (n^2-1)$
$0 \le j\%n$ for all positive j (from above)
$0 \le b$

For any positive integer x, the highest possible result of the mod operation in x % n is (n-1)
$((n^2-1) \% n)$ has a highest value of (n-1)
$b \le (n-1)$
$0 \le b \le (n - 1)$
Moreover, as j has all the consecutive integer values from 0 to (n^2-1) it follows that b has all the consecutive integer values from 0 to (n-1). □

Theorem C.2: The values of the ordered pair (a, b) are always unique.
Proof: From Lemma C.3 and C.4 *wkt* $0 \le a \le (n - 1)$ and $0 \le b \le (n - 1)$ i.e. a has n values and b has n values. There are ${}^nC_1 * {}^nC_1 = n^2$ possible ways to chose unique ordered pairs of a and b. The number of values for j is exactly n^2. One pair of values is chosen for each

j value. Therefore, for each value of j the ordered pair chosen is different from all others. That is, by the time the loop is completed all the unique ordered pairs of (a, b) values are computed.
Since we are using a and b as the nodes in the graph all the edges in the graph are computed. □

Theorem C.3: Step 1 of the algorithm constructs an undirected graph.
Proof: From theorem C.2 *wkt* the algorithm iterates over the edges working on each edge exactly once. Since work on one edge does not affect the work on another, the order in which these edges are visited does not matter. Therefore, each edge can be handled by any processor in any order and construct the edges. In the *for* loop of step 1 the indexes of each pair of nodes are generated first, then get_I_StatementSeparatingAandB(a, b) is used to find the separating set for this pair. If this separating set is found then the connectEdge(a, b) function is called to construct the edge. At the same time we mark this pair with the separating set that will be useful in the next *for* loop of this step. By the time the loop is completed, the undirected graph is constructed. □

Lemma C.5: Equation II will produce j values s.t. $0 \leq j \leq (n^3-1)$.
Proof: From theories of integer addition, *wkt* any two varying integer terms result in their lowest sum when both terms are at their lowest values and in their highest sum when both are at their highest values. The lowest value for both i and p_k is 0, hence the lowest value of j is 0. The highest value for j is obtained by substituting the highest values of the individual terms:
$j = ((n^3/p)-1) + (p-1) * (n^3/p) = (n^3/p) – 1 + p*(n^3/p) – (n^3/p) = (n^3 – 1)$
Therefore j, $0 \leq j \leq (n^3-1)$ □

Lemma C.6: Equation II produces unique integer values for j.
Proof: Let $C_k = p_k * n^3/p$. If we let p_k to be constant at any particular value, then C_k is constant. Since the value of i varies from 0 to $((n^3/p)-1)$ and each value for i is a unique integer, we obtain a set of unique integers varying from $0+C_k$ to $n^3/p+C_k$. Similarly, for any value of i, C_k varies from 0 to $(p-1)*(n^3/p)$ with unique integer values.
Uniqueness of the sum of these two terms can be proved by contradiction. Suppose we have two sums $(i_1 + C_{k1})$ and $(i_2 + C_{k2})$ which are not unique. If these two integer terms are not unique, then
$i_1 + C_{k1} = i_2 + C_{k2}$
$i_1 – i_2 = C_{k2} – C_{k1}$
$\Delta_i = \Delta_{ck}$
where Δ_i is the difference between any two i, and Δ_{ck} is the difference between any two C_k.
Since i varies from 0, 1, 2, ... $((n^3/p)-1)$, the highest difference between any two i can be $|(n^3/p)-1 –0 | = ((n^3/p)-1)$.

Since C_k varies from 0, n^3/p, $2*n^3/p$, $3*n^3/p$, ... $(p-1)*n^3/p$, the lowest possible difference is $|(n^3/p) - 0| = (n^3/p)$. This implies that $\Delta_i < \Delta_{ck}$. This is contradictory to the above assumption, which says these two terms can be equivalent.
Therefore, the sum of these two terms must always be unique.
Moreover, i can have (n^3/p) values and j can have p values; therefore, the total number of sums is $(n^3/p)*p = n^3$. □

Theorem C.4: Equation II produces unique consecutive integer values for j from 0, 1, ... (n^3-1).
Proof: From Lemma C.5, $0 \le j \le (n^3-1)$, and from Lemma C.6 j has n^3 unique values. It follows that j has all consecutive integer values from 0 to (n^3-1), as there are exactly n^3 integers in that inclusive range. □

Lemma C.7: In step 3 of the algorithm, the value of *a* varies from 0 to (n-1).
Proof:
$0 \le j \le (n^3-1)$
$0/n^2 \le j/n^2 \le (n^3-1)/n^2$
$\text{int}(0/n^2) \le \text{int}(j/n^2) \le \text{int}((n^3-1)/n^2)$
$0 \le a \le (n - 1)$
Moreover, as j has all the integer values from 0 to (n^3-1), it follows that a has all the integer values from 0 to n-1. □

Lemma C.8: In step 3, the value of *b* varies from 0 to (n-1).
Proof:
$0 \le j \le (n^3-1)$
$0/n \le j/n \le ((n^3-1)/n)$
$\text{int}(0/n) \le \text{int}(j/n) \le \text{int}((n^3-1)/n)$
$0 \le \text{int}(j/n) \le (n^2-1)$

The lowest value for int(j/n) is 0, and (int(j/n)%n) = 0%n = 0. Since for any integer x, x%n has highest possible value (n-1), the highest possible value for (int(j/n)%n) is (n-1).
Therefore,
$0 \le \text{int}(j/n)\%n \le (n-1)$
$0 \le b \le (n-1)$
Moreover, as j has all the consecutive integer values from 0 to (n^3-1), it follows that b has all the consecutive integer values from 0 to (n-1). □

Lemma C.9: In step 3, the value of *c* varies from 0 to (n-1).
Proof:
$0 \le j \le (n^3-1)$
$0 \le j\%n$ for all positive j (from above)
$0 \le c$

For any positive integer x, the highest possible result of the mod operation x % n is (n-1)
$((n^3-1) \% n)$ has a highest value of (n-1)
$c \leq (n-1)$
$0 \leq c \leq (n - 1)$
Moreover, as j has all the consecutive integer values from 0 to (n^3-1) it follows that c has all the consecutive integer values from 0 to (n-1). □

Theorem C.5: Values of the ordered triplet (a, b, c) are always unique.
Proof: From Lemmas C.7, C.8 and C.4 *wkt* $0 \leq a \leq (n - 1)$, $0 \leq b \leq (n - 1)$ and $0 \leq c \leq (n - 1)$ i.e. a has n values, b has n values, and c has n values. There are ${}^nC_1 * {}^nC_1 * {}^nC_1 = n^3$ possible ways to choose unique ordered triplets a, b, and c. There are exactly n^3 values for j. One triplet is chosen for each j value. Therefore, for each value of j the ordered triplet chosen is unique from others. That is, by the time the loop is completed all the unique ordered triplets of (a, b, c) values are computed.
Since we are using a, b, and c as the nodes in the graph all the edges in combination with each node in the graph are computed. □

Theorem C.6: Step 3 of the algorithm orients all the edges directly implied by independency relationships.
Proof: According to Pearl and Verma, if a and b are not adjacent and there exists a node c s.t. c is adjacent to both a and b and c does not belong to the separating set S(a, b), then orient edges (a→c) and (c←b) unless there already exists an edge in the opposite direction.[8] This implies that an edge (a→c) cannot be directed without knowing whether (c→a) exists. As we have defined our data structure to preserve the directionality of an edge by allowing the existence of edges in both directions at the same time, it is possible to direct them independently without destroying any information. When the independent processors work on each edge and direct it in its computed direction, we may end up with edges between a pair of nodes directed both ways, as it was not prevented. After step 3 is completed, it is possible to use this result and find if it really happened and terminate with failure just like in the sequential algorithm. (Our parallel algorithm delays this detection process until all the processors have completed their work, even though it would be possible to terminate the algorithm with failure as soon as a pair of edges directed in the opposite direction is added in step 3.) □

Theorem C.7: Step 10 directs all the undirected edges.
Proof: The proof essentially has the same line of arguments as the preceding part, except that a sink s and all the neighbors of s are found by scanning all the edges in the first part and then using this result in the second part of step 10, which just checks whether all the neighbors of a node x connected to s by an undirected edge are also the neighbors of s (i.e. finding the clique with s). Except for finding the sink and neighborhood relationships

of the sink in parallel, everything else is same as its counterpart sequential algorithm PDX, hence the same correctness arguments apply to this part as well.[13] □

Finally, steps 14, 15, and 16 have a trivial proof of correctness. All independence statements in M can be verified in any order, therefore verifying all of them in parallel and failing if any processor reports failure will verify the DAG.

This completes our proof of correctness for the whole algorithm.