Carnegie Mellon University
Research Showcase

Department of Philosophy | Dietrich College of Humanities and Social Sciences

5-31-2010

Bootstrapping the PC and CPC Algorithms to Improve Search Accuracy

Joseph Ramsey
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/philosophy

Part of the Philosophy Commons

Recommended Citation
Ramsey, Joseph, "Bootstrapping the PC and CPC Algorithms to Improve Search Accuracy" (2010). *Department of Philosophy.* Paper 101.
http://repository.cmu.edu/philosophy/101

This Technical Report is brought to you for free and open access by the Dietrich College of Humanities and Social Sciences at Research Showcase. It has been accepted for inclusion in Department of Philosophy by an authorized administrator of Research Showcase. For more information, please contact research-showcase@andrew.cmu.edu.

Bootstrapping the PC and CPC Algorithms to Improve Search Accuracy

Joseph D. Ramsey

May 31, 2010

Technical Report No. CMU-PHIL-187

Philosophy

Methodology

Logic

Carnegie Mellon

Pittsburgh, Pennsylvania 15213

Bootstrapping the PC and CPC Algorithms to Improve Search Accuracy[1]

Joseph D. Ramsey

Technical Report, Philosophy Department, Carnegie Mellon University
5/31/10

Abstract

By bootstrapping the output of the PC algorithm (Spirtes et al., 2000; Meek 1995), using larger conditioning sets informed by the current graph state, it is possible to define a novel algorithm, JPC, that improves accuracy of search for i.i.d. data drawn from linear, Gaussian, sparse to moderately dense models. The motivation for constructing sepsets using information in the current graph state is to highlight the differences between d-separation information in the graph and conditional independence information extracted from the sample. The same idea can be pursued for any algorithm for which conditioning sets informed by the current graph state can be constructed and for which an orientation procedure capable of orienting undirected graphs can be extracted. Another plausible candidate for such retrofitting is the CPC algorithm (Ramsey et al, 2006), yielding an algorithm, JCPC, which, when the true graph is sparse is somewhat more accurate than JPC. The method is not feasible for discovery for models of categorical variables, i.e., traditional Bayes nets; with alternative tests for conditional independence it may extend to non-linear or non-Gaussian models, or both.

1. Background.

By attaching a certain kind of bootstrapping procedure to the output of the PC algorithm (Spirtes et al. 2000; Meek 1995), is it possible to define a novel algorithm, JPC, that significantly improves total error for continuous-variable causal searches, under the assumptions of causal sufficiency, acyclicity and linearity, for i.i.d data with Gaussian errors. A similar algorithm can be defined as a bootstrapping post-processor to the CPC algorithm (Ramsey et al. 2006). These algorithms have an advantage over competing algorithms in that they minimize average total error—that is, the sum of false positive and negative adjacency error rates, directional error rate, and average number of bidirected edges erroneously inserted into the output. For sparse graphs, at relatively small sample sizes JCP and JCPC display a markedly lower total error rates than PC, CPC, and the Greedy Equivalence Search (GES), a consistent scoring algorithm. JCPC has an advantage over JPC for models with the number of edges equal to the number of nodes; for denser graphs, the advantage tends to reverse. Notably, for discrete variable searches, due to the large size of the conditioning sets involved, neither JPC nor JCPC has any real advantage over competing methods.

[1] The author thanks Clark Glymour, for many helpful comments and edits and to the James S. McDonnell Foundation, for support.

JPC and JCPC rely on standard theory for constraint-based causal searches of the sort used for the PC algorithm and other algorithms of that ilk. The types of paths and graphs that the algorithms use are as follows. A variable X is *adjacent to* a variable Y in G just in case X→Y, X←Y, or X—Y, or X↔Y; the adjacent nodes of X in G are denoted adj(X, G). X*→Y, X*—Y, or X*-*Y indicates an edge from X to Y, where the starred endpoints are unknown. X is a spouse of Y just in case X→W←Y for some W. A *path* is an n-tuple of variables <X, …, Y> in G such that each adjacent pair of variables in the tuple are adjacent in G. The *length* of a path is the number of edges along the path. A path of length 2 of the form X→Y←Z a *collider*, and a path of length 2 of one of the forms X→Y→Z, X←Y←Z, or X←Y→Z is a *noncollider.* A path P = <X,…,Y> is *directed* from X to Y, denoted X…→Y, if for each i from 1 to the length of the path minus 1, P(i)→P(i+1). A variable W such that X←…W…→Y is a *common cause* of X and Y provided the two paths intersect only at W. A set V of variables is *causally sufficient* just in case all common causes of X and Y are in V, for all X and Y in V. A directed path of the form X…→X in G is a *cycle*.

Let G be a *directed acyclic graph* (DAG) over variables in V—that is, a set of nodes taken together with a set of edges defined over those nodes, in which all edges are directed and there are no cycles. G is interpreted as a *causal model* over V just in case each directed edge X→Y in G is interpreted to mean that X is a direct cause of Y, relative to the variables in G. A causal model in which the underlying graph G is directed, acyclic and causally sufficient is a *DAG model*. In addition to single DAG models, one may consider equivalence classes of DAG models, represented using a graph type called a *pattern*, consisting of directed edges (X→Y) and undirected edges (X—Y), with the semantics that all edges directed in the pattern are so directed in each of the DAG models in the pattern, and all edges undirected in the pattern are directed for some DAG model in the pattern in the one direction and for some other DAG model in the pattern in the other direction.

The notion of d-separation is the following (Pearl, 1888). A descendant of X is a node Y such that X = Y or there exists a path X…→Y in G. A path P is *active* (*d-connecting*) relative to a set of vertices C (A, B not in C) iff (i) every noncollider on P is not a member of C, and (ii) every collider on P is an ancestor of some member of C. A path is *blocked* just in case it is not an active path. X and Y are d-separated conditional on C just in case all paths from X to Y conditional on C are blocked.

Causal graphs relate to probability distributions through the following assumptions. Let an *independence oracle* be a set of variables V together with a function I(X, Y, S) from pairs of nodes X, Y in V and conditioning sets of variables S in V to true or false. If I(X, Y, S), then X is independent of Y conditional on S; otherwise, X is dependent on Y conditional on S. The *Causal Markov Condition* is the condition that given a set of variables whose structure can be represented as a DAG, each variable is probabilistically independent of its non-effects (non-descendants in G) given its parents. The *Causal Faithfulness Condition* is the condition that given a set of

variables whose causal structure is represented by a DAG, no conditional independency holds unless the Causal Markov Condition implies it for the DAG. Given these two assumptions, X is independent of Y conditional on C just in case for the true DAG model G, X is d-separated from Y conditional on C.

An *unshielded collider* in G is a subgraph of G of the following form X*-*Y*-*Z*-*X. Orientations inside of unshielded colliders cannot be discerned in the general case; hence not all colliders in G may be identifiable in the pattern of G. The *graphical Markov blanket* of X in a DAG D, MB(X, D) consists of the parents, children, and spouses of X (a *spouse* of X being a parent of a child of X). If D* is the true model for a probability distribution, then MB(X, D*) is the *Markov blanket* of X. (Cases will be considered where D is not the true model.) If X is a variable in D*, then X is independent of and variable not in D* U {I} conditional on D*. In patterns, Markov blankets cannot strictly be identified. The parents and children of X and be identified, since these are the nodes adjacent to X. However, since not all colliders are marked in a pattern, it is not always the case the parents of children can be identified. The algorithmic code discussed below will condition on Markov blanket variables in order to block paths between variables X and Y in a pattern Π, doing extra searches where necessary to discover Markov blanket variables, or parents, or descendants, where these might otherwise not be identifiable.

The PC algorithm, which JPC extends, can be stated as follows:

PC(V, I)

1. Form the complete undirected graph U on the set of variables V.
2. n = 0
3. Repeat until for each ordered pair of adjacent variables X and Y, adj(U, X) \ Y has less than n elements
    a. For each pair of variables X and Y that are adjacent in (the current) U such that adj(U, X)\{Y} or adj(U,Y)\{X} has n elements, check through the subsets of adj(U, X)\{Y} and the subsets of adj(U, Y)\{X} that have exactly n variables. If a subset S is found conditional on which X and Y are independent, remove the edge between X and Y in U, and record S as *Sepset(X,Y)*.
    b. n <- n+1
4. Let P be the graph resulting from step 3. For each unshielded triple <A, B, C> in P, orient it as a A→B←C iff B is not in Sepset(A, C)
5. Execute the orientation rules given in Meek 1995.

The PC algorithm produces a *pattern*, that is, a graph consisting of directed and undirected edges, representing an equivalence class of DAGs, such that every directed edge in the pattern is found in every DAG in the equivalence class, and each undirected edge is directed the one way in some DAG in the pattern and the other way in some other DAG in the pattern. The Conservative PC (CPC) algorithm (Ramsey et al. 2006), which JCPC extends, modifies step 4 of the PC algorithm as follows:

4. Let P be the graph resulting from step 3. For each unshielded triple <A, B, C> in P, check all subsets of adj(A, G) and of adj(C, G):
    a. If B is not in any such set conditional on which A and C are independent, orient A—B—C as A→B←C;
    b. If B is in all such sets conditional on which A and C are independent, leave A—B—C as it is, i.e., a non-collider;
    c. Otherwise, mark the triple as "unfaithful" by underlining the triple.

The CPC algorithm produces an *e-pattern*, that is, an extended pattern, in which some triples are marked as ambiguous. For each algorithm steps 1-3 can be run to produce a graph showing the adjacencies of each node. Orientation is done in steps 4 and 5 of each algorithm. In the case of PC, this orientation requires a *sepset map*—that is, a map from each node pair to the conditioning set used to show the nodes in that node pairs are conditionally independent—or null if not the node pair was not shown to be conditionally independent. In the case of CPC, orientation does not require a sepset map and can be done directly from a graph.

Note that whereas a pattern characterizes a set of DAGs that are Markov equivalent, a e-pattern need not do so.

2. The Algorithms.

Pseudocode for JCP is given in Figures 1 and 2. The JPC algorithm bootstraps the PC algorithm by adding a meta-step. Given the graph, for each node pair X and Y of the graph, a set is constructed, S = Sepset(X, Y, G), that blocks all paths (conjectured by PC output) from X to Y except possibly for a direct connection between X and Y. If X and Y are independent conditional on S, then if there is an edge between X and Y in G, it is taken out. Alternatively, if X and Y are dependent conditional on S, then if there is no edge between X and Y in G, one is added. This procedure is carried out for all node pairs in G, and this entire procedure is iterated until no further changes are made to G. The graph at the end of each epoch is stored, and sepsets for the next epoch are calculated using this stored graph. Reorientation of the graph is performed once per epoch, using any new sepsets discovered and taking into account any adjacency changes that have been made.

Both the metastep and considerations of orientation require that some attention be paid to selection of separating sets in steps 3 and 4 of the algorithm. The sets chosen by the Sepset procedure have been shown in practice to yield good accuracy results for the algorithm. The rationale for PathBlockingSet is to construct a modified Markov blanket about X. It may be that Y is a spouse of X, in which case variables that are adjacent to both X and Y that are colliders must not be conditioned on. In order to separate X and Y, among all variables adjacent to both X and Y, those which form colliders with X and Y should not be conditioned on, and those that form noncolliders with X and Y should be conditioned on. As noted above, in general it is

not possible to know which variables along which paths form colliders in a pattern. Therefore, a search is done among all subsets of the variable adjacent to both X and Y for the subset that, when not conditioned on, allow X and Y to be separated, if such a set exists. Also, because conditioning on any descendant of such a collider would cause an otherwise blocked path to be unblocked, a search is done for variables (among those that might be conditioned on) that are descendants of one of these potential colliders, so that these may not be conditioned on either. The following theorem shows that failing to condition on descendants of common colliders does not lead to paths being unblocked that would otherwise be blocked by Markov blanket conditionings.

Theorem 1. Let X and Y be variables in a DAG G. Let C be the set of common colliders of X and Y. Let E consist of C taken together with its descendants in MB(X, G). Let S = MB(X, G) \ {E ∪ {X} ∪ {Y}}. Then X is d-separated from Y conditional on S.

Proof: Consider a path from X to Y. No matter how this path begins, it is blocked. Let D be the descendants of C in MB(X, G). If the path is of the form X→A→B…Y, then A is conditioned on, blocking the path, unless A is in D. But in the latter case, if there were a path A…->Y, then there would be a cycle Y→C→A…→Y, against assumption. If the path has the form X→A←B...Y, then conditioning on A and B would block the path, unless A or both A and B were in D.[2] Assuming there is no cycle Y→C→B…→Y, In either case, A would block the path, unless some descendant D of A were conditioned on. However, in that case, D would be a descendant of C, against assumption. If the path were of the form X←A…Y, then there would be a cycle C→A→X→C, against assumption. Thus all paths are blocked except possibly for length-2 paths of one of these forms: X→A→B, X←A→B, X←A←B, and X→A←B. In the first three cases, conditioning on A blocks the path, and A is in S. In the last case, not conditioning on A blocks the path, and A is not in S, and neither are any of its descendants. Thus all paths from X to Y are blocked. ∴

For patterns, the sepset S from Theorem 1 needs to be modified somewhat, since in a pattern, not all colliders and not all descendants of a variable can be determined from the graph alone. Because not all colliders can be determined, it's not possible to determine always whether a variable should be counted as a parent of a child. The following rule is used. Instead of including all parents of children, all adjacents of non-parents are used, a superset of the set of parents of children. By the same reasoning used in Theorem 1, including the extra variables in the sepset is not problematic. Since not all descendants of a variable can be determined from the graph alone, a secondary search is initiated from among the potential descendants of these the elements of C for a set that, when not conditioned on, in addition to the potential colliders not being conditioned on, will allow X and Y to be separated. The potential descendants of a variable X in G are those variables Y in G for which there

[2] If B is in D in this case, then so is A.

Figure 1. G is a graph, X and Y distinct variables in G, and E a set of variables to be excluded.

```
Procedure PathBlockingSet(X, Y, G, E)
1.  T ← adj(X, G)
2.  T ← T ∪ adj(non-parents(X) \ E)
3.  T ← T ∪ parents(Y, G)
4.  T ← T \ {X, Y}
5.  T ← T \ E

Procedure Sepset(X, Y, G)
1. B ← PathBlockingSet(X, Y, G, empty)
2. A ← {C such that adj(X, C, G) and adj(C, Y, G)}
3. For each subset E of A
4.   E2 ← E U descendants(X, E)
5.   E3 ← semiDirectedDescendants(X, E, G) \ E2
6.   For each subset E4 of E3
7.     S* ← PathBlockingSet(X, Y, G, E2 U E4)
8.     If I(X, Y | S*)
9.       Return S*
10. Return null
```

exists a *semi-directed path* from X to Y—that is, a path consisting of directed and/or undirected edges, in which all of the directed edges point away from X and toward Y.

As far as the code is concerned, Figure 1 consists of two methods, PathBlockingSet and Sepset. PathBlockingSet takes as arguments a graph G, two distinct variables X and Y in G, and a set of variables E to be excluded. It constructs a set containing the parents and children of X, and the parents of potential children of X (including parents of adjacents that aren't themselves parents but are not marked as children), and the marked parents of Y, with some exclusions. Excluded are any variables in E, any parents of variables of E, and X and Y themselves. If E consists entirely of the common colliders of X and Y and their descendants, then this is a set that blocks every path from X to Y with the exception of an edge between X and Y themselves. The job of the procedure Sepset, then, is to produce the correct E, if there is one. It does this by looking at every subset of the common adjacents of X and Y for the set of common colliders, and then attempting to discover from the graph, using a bit of search, what the descendants of those common colliders are, by initiating a search of the sort described above. If the common colliders are found, then the descendants of those common colliders will be found by the descendant search, and the search will be successful. Also, if the search for a sepset was successful, that means the common colliders were found, not conditioned on, and their descendants were also all not conditioned on.

The main loop is given Figure 2, along with an orientation procedure (taken from PC). To produce the JCPC algorithm, CPC is used instead of PC in line 1 of

Figure 2. The Reorient procedure takes a graph G and a map S from pairs of variables to sepsets (or to null). The JPC procedure takes as input a set of variables V over which a conditional independence relation has been defined, I. The procedure copy(G) returns a copy of graph G.

```
Procedure PC-orient(G, S)
1.  Remove all orientations from G.
2.  Orient colliders using sepsets in S.
3.  Add implied orientations using Meek rules (Meek 1995) and
sepsets in S.


Procedure JPC(V, I)
1. <G, S> ← PC(V, I)
2. Repeat while G changes:
3.   G2 ← copy(G)
4.   Repeat for every pair of nodes X and Y in G:
5.     S1 = Sepset(X, Y, G2)
6.     S2 = Sepset(Y, X, G2)
7.     If S1 is not null and S2 is not null
8.        S(X, Y)← the larger of S1 and S2
9.     Else if S1 is not null
10.       S(X, Y) ← S1
11.    Else if S2 is not null
12.       S(X, Y) ← S2
13.    Else
14.       S(X, Y) ← null
15.    If ~adj(X, Y, G)
16.      If S(X, Y) is null
17.        Add X—Y to G
18.    Else
19.      If S(X, Y) is not null
20.        Remove X*-*Y from G
21.   G ← PC-orient(G, S)
22. Return G.
```

Procedure JPC, and use CPC orientation is used instead of PC orientation in line 19 of Procedure JPC. With these substitutions, sepset map S does not need to be tracked, so lines 5-9 of the algorithm can be eliminated.

Obviously, any procedure such as Sepset in Figure 1 for choosing separating sets for nodes X and Y will work in theory so long as the chosen set blocks all paths from X to Y except for the possible adjacency between X and Y itself. A well-known alternative strategy is to block all parents of X or of Y. Since the adjacents of X are marked in G, one could simply search over all combinations of adj(X) to see if, conditional on any such combination, X is independent of Y. Incorporating known information about parents and children in G, this yields a “small” sepset, computed

by procedure Sepset2 (see below) as given in Figure 3. To compare this alternative to PC and JCP, an additional algorithm will be added to performance results, code-

Figure 3. An alternative procedure for producing sepsets. This procedure produces smaller sepsets than the Sepset procedure in Figure 1.

```
Procedure Sepset2(X, Y, G)
1.  T <- adj(X, G) \ {children(X, G) ∪ parents(X, G))
2.  For each subset E of T
3.     If (I(X, Y | E ∪ parents(X, G))
4.     Return E ∪ parents(X, G)
5.  Return null
```

named "JPC-small," using Sepset2 in place of Sepset to calculate sepsets in JPC. Unsurprisingly especially for denser searches, performance drops off radically using the "small" sepset.

4. Comparison to Alternative Algorithms.

Most algorithms recommended for causal search assuming causal sufficiency and acyclicity in the literature are aimed at the discrete variable case. Three are notably aimed at the continuous variable case: PC (Spirtes et al., 2000; Meek 1995), CPC (Ramsey et al., 2006), and GES (Checkering, 2002). These form a sensible comparison set for JPC and JCPC. Two further algorithms are added. Even though MMHC is aimed at the discrete variable case, arguments by its authors for its superiority in the discrete case invite a comparison. It is implemented in the continuous case by using the same adjacency phase as is published for the MMHC algorithm but using GES restricted to the adjacencies returned by the adjacency phase as the greedy hill-climbing orientation step. In addition, JPC-small is added, for reasons given above.[3]

These algorithms are compared in Tables 1 through 9. For each table, 30 random data sets are simulated, and error rates rages for each algorithm are listed, with the exception that in some cases certain algorithms are left out of the comparison for reasons of time. For each table, 30 random data sets are simulated; these same data sets are then given to each algorithm compared in that table and results tabulated. Data sets are simulated as follows. For each data set, a DAG is chosen uniformly from the set of graphs with a particular number of nodes and edges. For sparse graphs, options for 10 nodes and 10 edges (Table 1), 50 nodes and 50 edges (Table 2), and 100 nodes and 100 edges (Table 3) are considered. For denser graphs,

[3] GES restricted to a certain set of edges may in principle not return all of the edges given to it. This does not happen in the current comparison. Nevertheless, edges that are not returned are added as undirected edges to the output pattern, so no additional false negative counts are incurred.

options for 10 nodes and 20 edges (Tables 4 for n = 1000 and 7 for n = 5000), 50 nodes and 50 edges (Tables 5 for n = 1000 and 8 for n = 5000), and 100 nodes and 200 edges (Tables 6 for n = 1000 and 9 for n = 5000) are considered. The simulations are all random linear structural equation models with linear coefficients chosen randomly from the set (-1.5, -0.05) ∪ (0.05, 1.5). Errors are counted as follows. False positive adjacencies (Adj FP) are counted as adjacencies in the discovered graph that do not appear in the DAG used to generate the data. False negative adjacencies (Adj FN) are adjacencies that are in the randomly generated DAG but not in the discovered graph. Directional errors (DE) are counted as the number of directed edges in the discovered graph that are not so directed in the pattern of the randomly chosen DAG. These are adjacencies that may either be directed oppositely or not directed in the pattern of the randomly chosen DAG. Bidirected errors (BID) are counted as the number of bidirected edges that appear in the discovered graph; these are all errors, since the pattern of the randomly chosen DAG does not contain bidirected edges; each is counted once.

In all cases, random graphs are chosen with a maximum degree of 6 . In search, for JPC and JCPC, a maximum degree of 8 is assumed, along with a maximum descendant path length of 15. The latter is intended to avoid prolonged searches for possible descendants of possible common colliders of variables.

For all constraint-based algorithms (JPC, JPC-small, PC, CPC, MMHC), the Fisher Z independence test was used with alpha = 0.001. In the case of GES, to speed up the search and to reduce the number of false positive adjacencies, a discount penalty equal to the number of edges in the true graph is applied (Ramsey et al. 2010). In each table, algorithms are removed from consideration if they do not return in under 10 minutes.

Tables 1-3 show that JCPC and JPC are both low in average total error for small sparse graphs, of 10 nodes and 10 edges, 50 nodes and 50 edges, and 100 nodes and 100 edges, respectively. CPC runs a close third for 10 nodes, 10 edge graphs, but as the number of nodes and edges increases, JPC dominates all other algorithms considered. JPC-small—that is, JPC using the smaller sepsets (Sepset2) is shown in all three tables to have inferior performance to JPC, the difference becoming more pronounced as the number of nodes and edges increases. MMHC shows excellent false positive error rate, but total error is kept high by a high false negative error rate. GES, by contrast, shows excellent false negative error rate, but total error is kept high by a high false positive error rate, that in fact needs to be kept in check by applying a high penalty discount for larger models. PC, CPC, JPC, JPC-small, and JCPC are all liable to producing bidirected edges in the fact of conflicting conditional independence information. All bidirected edges are orientation errors, since the pattern of the randomly chosen DAG for each run does not contain any bidirected edges. JCPC controls these the best out of these algorithms, with JPC second and CPC third. In all, for sparse models, out of these options, JCPC is the algorithm of choice, with JPC a close second.

Table 1: Simulation results for graphs with 10 nodes and 10 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. GES uses a penalty discount of 10, equal to the expected number of edges. 'Time' is average time over 30 runs in milliseconds.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time[4]
JPC	0.27	0.27	0.87	0.07	1.47	32
JPC-small	0.10	0.37	0.23	0.60	1.30	16
JCPC	0.17	0.30	0.50	0.00	0.97	37
PC	0.10	0.80	0.87	1.33	3.10	5
CPC	0.07	0.80	0.33	0.07	1.27	6
GES	0.73	0.60	1.50	0.00	2.83	51
MMHC	0.03	1.00	1.70	0.00	2.73	21

Table 2: Simulation results for graphs with 50 nodes and 50 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. GES uses a penalty discount of 50, equal to the expected number of edges. Time' is average time over 30 runs in milliseconds.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	0.23	1.00	0.97	0.03	2.23	636
JPC-small	2.60	2.13	1.20	1.97	7.90	541
JCPC	0.33	1.30	0.27	0.00	1.90	1022
PC	1.33	6.43	2.40	4.87	15.03	92
CPC	1.33	6.43	0.43	0.07	8.27	102
GES	2.30	8.77	2.67	0.00	13.73	2024
MMHC	0.67	7.50	2.17	0.00	10.33	277

Table 3: Simulation results for graphs with 100 nodes and 100 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. GES uses a penalty discount of 100, equal to the expected number of edges. Time' is average time over 30 runs in milliseconds.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	1.73	3.80	3.37	0.07	8.97	4532
JPC-small	5.53	7.37	2.77	4.23	19.90	4047
JCPC	1.07	3.50	0.60	0.00	5.17	6198
PC	1.60	13.53	5.57	9.23	29.93	927
CPC	1.57	13.67	0.50	0.17	15.90	972
GES	1.13	43.67	1.27	0.00	46.07	12252
MMHC	0.97	16.00	2.40	0.00	19.37	1290

For denser models, with twice the number of edges as nodes, as shown in Table 4 for 1000 cases, for smaller models all of the algorithms being compared come back (and converge, for JPC-small); JPC has the lowest total error, with JCPC second, JPC-

[4] Time is shown in milliseconds. All runs were done on a MacBook Pro laptop, 2.8 GHz Intel Core 2 Duo, 4 GB, using Java 6.

small third, and others with higher total error. In this case, JPC actually has the lowest adjacency false positive rate as well, and the lowest false negative rate. In Table 5, for models with 50 nodes and 100 edges, JPC again has the lowest total error, lowest adjacency false and negative error rates, out of the algorithms compared. In Table 6, even though the total error rate for JPC is not small (65.6333), the next lowest total error rate is 135.4000 for MMHC. Nevertheless, these JPC results are not really usable, at this sample size, for this large dense model.

For the 5000-case examples, in Table 7 for 1000 cases, for smaller models, all of the algorithms being compared come back (and converge, for JPC-small); JPC again has the lowest total error, this time quite low, with JCPC second, JPC-small third, and others with higher total error. In this case, JPC actually has the lowest adjacency false positive rate as well, and the lowest false negative rate. In Table 8, for models with 50 nodes and 100 edges, JPC again has the lowest total error, lowest adjacency false and negative error rates, out of the algorithms compared. In Table 8, the total error rate for JPC is much smaller than for Table 6 (24.5667), the next lowest total error rate is 138.233 for MMHC, considerably higher. The JPC results in these cases are generally usable. The comparison of Tables 4-6 to Tables 7-9 illustrates rapid convergence of the algorithm with a moderate increase sample size. In addition, it is notable that JPC performs better than JCPC for these denser models.

Table 4: Simulation results for graphs with 10 nodes and 20 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	1.57	3.80	4.13	1.20	10.70	347
JPC-small	4.00	5.13	3.43	4.17	16.73	288
JCPC	1.23	6.73	3.17	0.00	11.13	1030
PC	2.13	9.63	2.27	5.30	19.33	5
CPC	2.07	9.53	2.43	0.97	15.00	9
GES	4.63	7.27	4.93	0.00	16.83	88
MMHC	1.37	9.80	3.17	0.00	14.33	21

Table 5: Simulation results for graphs with 50 nodes and 100 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	4.47	10.93	9.07	1.67	26.13	17285
JPC-small	45.27	38.50	9.63	54.30	147.70	25005
PC	14.37	50.60	8.17	23.70	96.83	168
CPC	15.00	50.70	4.77	1.27	71.73	205
GES	11.67	60.00	10.27	0.00	81.93	4029
MMHC	5.37	54.37	8.67	0.00	68.40	894

Table 6: Simulation results for graphs with 100 nodes and 200 edges, n = 1000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. JPC-small, JCPC, and GES have been removed from consideration because they do not return in under 10 minutes.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL
JPC	13.5000	35.7000	14.9000	1.5333	65.6333
PC	34.7333	109.2667	12.3333	43.3333	199.6667
CPC	34.9000	109.6000	7.3667	1.6000	153.4667
MMHC	10.9000	116.5667	7.9333	0.0000	135.4000

Table 7: Simulation results for graphs with 10 nodes and 20 edges, n = 5000. All numbers are average error rates over 30 runs. Columns are as indicated in the text.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	0.90	2.07	1.90	1.43	6.30	239
JPC-small	3.10	2.90	3.17	4.10	13.27	79
JCPC	1.30	4.27	1.77	0.00	7.33	1285
PC	2.43	6.80	2.57	6.67	18.47	7
CPC	2.50	7.00	2.43	1.53	13.47	15
GES	7.13	3.40	6.43	0.00	16.97	172
MMHC	1.63	7.53	3.77	0.00	12.93	33

Table 8: Simulation results for graphs with 50 nodes and 100 edges, n = 5000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. JCPC has been removed from consideration because it does not return in under 10 minutes.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	1.67	2.23	2.53	1.63	8.07	15077
JPC-small	49.13	32.53	7.73	60.60	150.00	38369
PC	21.63	45.10	6.90	39.80	113.43	249
CPC	21.63	45.13	6.10	4.07	76.93	326
GES	29.33	32.07	14.27	0.00	75.67	21801
MMHC	8.00	50.63	8.83	0.00	67.47	3700

Table 9: Simulation results for graphs with 100 nodes and 200 edges, n = 5000. All numbers are average error rates over 30 runs. Columns are as indicated in the text. JCPC has been removed from consideration because it does not return in under 10 minutes.

Algorithm	Adj FP	Adj FN	DE	BID	TOTAL	Time
JPC	4.50	6.90	4.50	1.50	17.40	99710
JPC-small	106.87	69.60	13.73	131.30	321.50	177988
PC	40.10	90.20	11.37	70.53	212.20	1728
CPC	40.03	90.43	9.63	5.87	145.97	1956
GES	38.97	78.87	19.40	0.00	137.23	191479
MMHC	12.17	101.23	13.63	0.00	127.03	33167

References

Chickering, D. M. (2002). Optimal Structure Identification with Greedy Search. *Journal of Machine Learning Research*, 3:507-554.

Meek, C. (1995). Causal Inference and Causal Explanation with Background Knowledge. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 403-411. Morgan Kaufmann.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Representation and Reasoning Series (2nd printing ed.). San Francisco: Morgan Kaufmann.

Ramsey, J. P. Spirtes, and J. Zhang (2006). Adjacency-Faithfulness and Conservative Causal Inference. *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, 401-408. Oregon: AUAI Press.

Spirtes, P., C. Glymour, and R. Scheines (2000). Causation, Prediction, and Search. MIT Press.

Tsamardinos, I., L. Brown, and C. Aliferis (2006). The Max-min Hill-Climbing Bayesian Network Structure Learning Algorithm. *Machine Learning* 65(1):31-78.