An Overview of the `pcalg` Package for `R`

M. Kalisch* A. Hauser† M.H. Maathuis*
Martin Mächler*

September 25, 2023

Contents

*ETH Zürich
†Bern University of Applied Sciences

1 Introduction

The paper Kalisch et al. [2012] reports on pcalg Version 1.1-4. Back then, the package covered basic functions for structure learning (`pc`, `fci`, `rfci`), as well as a method to compute bounds on total causal effects (`ida`). Since 2012, the `pcalg` package has been extended in two main areas: structure learning methods and covariate adjustment methods.

In this document, we give an overview of the functionality of `pcalg` (2.7-9). In section 2 and 3 we focus on methods for structure learning and covariate adjustment, respectively. In section 4 we discuss provided methods for random graph generation (e.g. for simulation studies). In section 5 we end with notes on some implementation details. The sections can be read independently.

We assume that the reader is familiar with basic terminology of structure learning and causal inference. For theoretical background or technical details we refer the reader to papers throughout the text. As a first introduction, one might consult the overview papers Kalisch and Bühlmann [2014], Maathuis and Nandy [2016], Maathuis and Drton [2017] or Heinze-Deml et al. [2018].

2 Structure Learning

2.1 Introduction

The goal in structure learning is to estimate the DAG or MAG representing the causal structure of the data generating mechanism, while the exact parameters of the causal structure are of less importance. For this, observational data or a mix of observational and interventional data might be available. Sometimes it might only be possible to estimate the Markov equivalence class of the true DAG or MAG.

The available functions for structure learning in package `pcalg` can be categorized in the following way:

- Constraint-based assuming no hidden confounders, i.i.d.:
  `skeleton()`, `pc()`, `lingam()`
- Constraint-based, allowing hidden variables, i.i.d.:
  `fci()`, `rfci()`, `fciPlus()`

- Score-based assuming no hidden confounders, i.i.d.:
  `ges()`
- Hybrid of constraint-based and score-based, assuming no hidden confounders, i.i.d.: ARGES (implemented in `ges()`)
- Score-based possibly with data from different settings, no hidden variables:
  `gies() and simy()`

More details on the assumptions can be found in section 2.7.

2.2 Constraint-based methods

This section follows Kalisch and Bühlmann [2014], where more details can be found. Given a causal structure one can derive constraints which every distribution generated from this causal structure must obey. The most prominent example of such constraints are conditional independence statements. Constraint-based learning checks for such constraints given data and thus ideally can reverse-engineer the causal structure of the data generating mechanism.

2.2.1 `pc()` and `skeleton()`

One prominent example of constraint-based learning (assuming no latent variables are present) is the PC-algorithm Spirtes et al. [2000] which estimates the CPDAG of the true causal structure. It can be outlined in three steps.

In the first step of the PC-algorithm, the skeleton of the DAG is estimated. The skeleton of a DAG is the undirected graph that has the same edges as the DAG but no edge orientations. The algorithm starts with a complete undirected graph. Then, for each edge (say, between a and c) the constraint is tested, whether there is any conditioning set s, so that a and c are conditional independent given s. If such a set (called a *separation set* or $sepset(a, c)$) is found, the edge between a and c is deleted.

In the second step of the PC-algorithm (i.e. after finding the skeleton as explained above), unshielded triples are oriented. An unshielded triple are three nodes a, b and c with $a - b$, $b - c$ but a and c are not connected. If node b is not in $sepset(a, c)$, the unshielded triples $a - b - c$ is oriented into an unshielded collider $a \to b \leftarrow c$. Otherwise b is marked as a non-collider on $a - b - c$.

In the third step, the partially directed graph from step two is checked using three rules to see if further edges can be oriented while avoiding new unshielded colliders (all of them were already found in step two) or cycles (which is forbidden in a DAG).

The first part of the PC-algorithm is implemented in function `skeleton()`. The main task of function `skeleton()` in finding the skeleton is to compute and test several conditional independencies. To keep the function flexible, `skeleton()` takes as argument a function `indepTest()` that performs these conditional independence tests and returns a p-value. All information that is needed in the conditional independence test can be passed in the argument

`suffStat`. The only exceptions are the number of variables `p` and the significance level `alpha` for the conditional independence tests, which are passed separately. Instead of specifying the number of variables `p`, one can also pass a character vector of variable names in the argument `labels`.

We show the usage of this function in a short example using built-in data. The dataset $gmG8$ contains $n = 5000$ observations of $p = 8$ continuous variables with a multivariate Gaussian distribution.

```
> data("gmG", package = "pcalg") ## loads data sets gmG and gmG8
```

In this example, the predefined function `gaussCItest()` is used for testing conditional independence. The corresponding sufficient statistic consists of the correlation matrix of the data and the sample size. Based on this input, the function `skeleton()` estimates the skeleton of the causal structure. The true DAG and the estimated skeleton of the causal structure are shown in Fig. 1.

```
> suffStat <- list(C = cor(gmG8$x), n = nrow(gmG8$x))
> varNames <- gmG8$g@nodes
> skel.gmG8 <- skeleton(suffStat, indepTest = gaussCItest,
                        labels = varNames, alpha = 0.01)
```

Finding the skeleton is also the first step in the algorithms FCI, RFCI and FCI+.

The PC-algorithm is implemented in function `pc()`. The arguments follow closely the arguments of `skeleton()`, i.e., the most important arguments consist of an conditional independence test and a suitable sufficient statistic.

We continue the previous example and illustrate function `pc()` using the built-in dataset $gmG8$. The result is shown in Fig. 1 (bidirected edges need to be interpreted as undirected edges in the resulting CPDAG).

```
> pc.gmG8 <- pc(suffStat, indepTest = gaussCItest,
                labels = varNames, alpha = 0.01)
```

The PC algorithm is known to be order-dependent, in the sense that the computed skeleton depends on the order in which the variables are given. Therefore, Colombo and Maathuis [2014] proposed a simple modification, called PC-stable, that yields order-independent adjacencies in the skeleton. In this function we implement their modified algorithm by using the argument `method = "stable"`, while the old order-dependent implementation can be called by using the argument `method = "original"`. While the argument `method = "stable"` calls an implementation of PC-stable written completely in `R`, the argument `method = "stable.fast"` calls a faster `C++` implementation of the same algorithm. In most cases, `method = "stable.fast"` will be the method of choice. The method `"stable"` is mostly of use in cases where strict backward-compatibility is required. Backward-compatibility is also the reason why the default value for the argument `method` is still `"stable"` rather than `"stable.fast"`.

We recall that in the default implementation unshielded triples $a - b - c$ are oriented based on $sepset(a, c)$. In the conservative (`conservative = TRUE`;

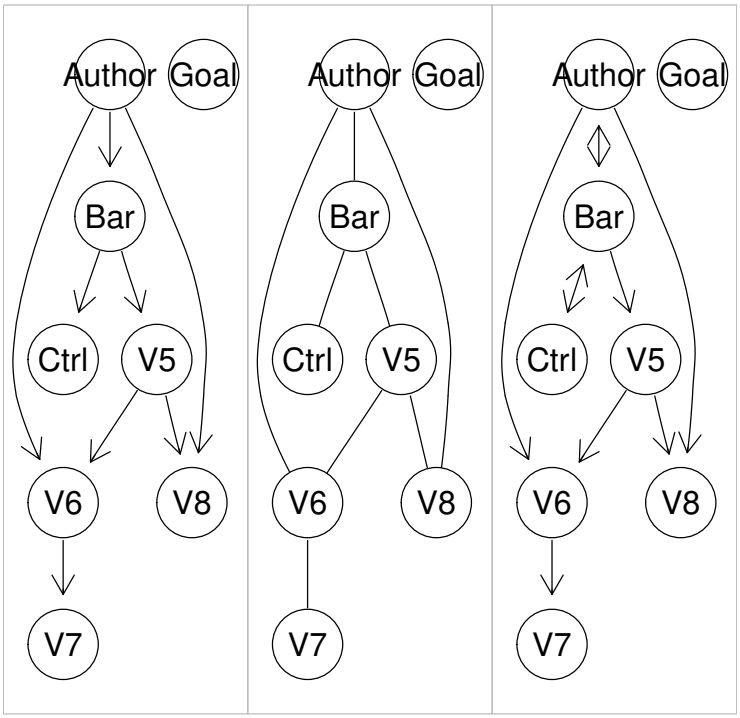

Figure 1: True causal DAG (left), estimated skeleton (middle) and estimated CPDAG (right). In the CPDAG, bidirected edges need to be interpreted as undirected edges.

see Ramsey et al. [2006]) or majority rule (`maj.rule = TRUE`; see Colombo and Maathuis [2014]) versions, the algorithm determines all subsets of $Adj(a) \setminus c$ (where $Adj(a)$ are all nodes adjecent to a) and $Adj(c) \setminus a$ that make a and c conditionally independent. They are called separating sets. In the conservative version $a - b - c$ is oriented as $a \to b \leftarrow c$ if b is in none of the separating sets. If b is in all separating sets, it is set as a non v-structure. If, however, b is in only some separating sets, the triple $a - b - c$ is marked as "ambiguous". Moreover, if no separating set is found among the neighbors, the triple is also marked as "ambiguous".

In the majority rule version the triple $a - b - c$ is marked as "ambiguous" if and only if b is in exactly 50 percent of such separating sets or no separating set was found. If b is in less than 50 percent of the separating sets it is set as a v-structure, and if in more than 50 percent it is set as a non v-structure.

Drawing a conclusion, the stable version of estimating the skeleton resolves the order-dependence issue wrt. the skeleton. Moreover, the useage of either the conservative or the majority rule versions resolve the order-dependence issues of the determination of the v-structures.

2.2.2 `fci()` and `rfci()`

Another prominent example of constraint-based learning, which allows for latent variables, is the FCI-algorithm Spirtes et al. [2000]. FCI estimates the PAG of the underlying causal structure and can be outlined in five steps.

In the first and second step, an initial skeleton and unshielded colliders are found as in the PC-algorithm. In the third step, a set called "Possible-D-SEP" is computed. The edges of the initial skeleton are then tested for conditional independence given subsets of Possible-D-SEP (note that Possible-D-SEP is not restricted to adjacency sets of X and Y). If conditional independencies are found, the conditioning sets are recorded as separating sets and the corresponding edges are removed (as in step 1 of PC-algorithm). Thus, edges of the initial skeleton might be removed and the list of separating sets might get extended. In step four, unshielded colliders in the updated skeleton are oriented based on the updated list of separating sets. In step five, further orientation rules are applied (see Zhang [2008]).

The FCI-algorithm is implemented in the function `fci()`. As in the function `pc()` we need two types of input: A function that evaluates conditional independence tests in a suitable way and a sufficient statistic of the data (i.e., a suitable summary) on which the conditional independence function works. Again, significance level `alpha` acts as a tuning parameter.

As an example, we show the usage of function `fci()` on a built-in dataset `gmL` containing four variables with a multivariate Gaussian distribution. The data was generated from a DAG model with one latent variable (variable 1) and four observed variables (variables 2, 3, 4 and 5) as shown in Fig. 2. We use the correlation matrix as sufficient statistic and function `gaussCItest()` as conditional independence test. Based on this input, the function `fci()` estimates the causal structure of the observed variables in the form of a PAG as shown in Fig. 2[1].

```
> data("gmL")
> suffStat <- list(C = cor(gmL$x), n = nrow(gmL$x))
> fci.gmL <- fci(suffStat, indepTest=gaussCItest,
                 alpha = 0.9999, labels = c("2","3","4","5"))
```

The function `rfci()` is a fast approximation of the function `fci()`. It avoids computing any Possible-D-SEP sets and does not conduct tests conditioning on subsets of Possible-D-SEP. This makes RFCI much faster than FCI. Mainly the orientation rules for unshielded triples were modified in order to produce an RFCI-PAG which, in the oracle version, is guaranteed to have the correct ancestral relationships.

Since the FCI and RFCI algorithms are build upon the PC algorithm, they are also order-dependent in their skeleton estimation. It is more involved, however, to resolve their order-dependence issues in the skeleton, see Colombo

[1] Due to a persistent bug in package `Rgraphviz` the edge marks are not always placed at the end of an edge, as here on the edge between node 2 and node 5.

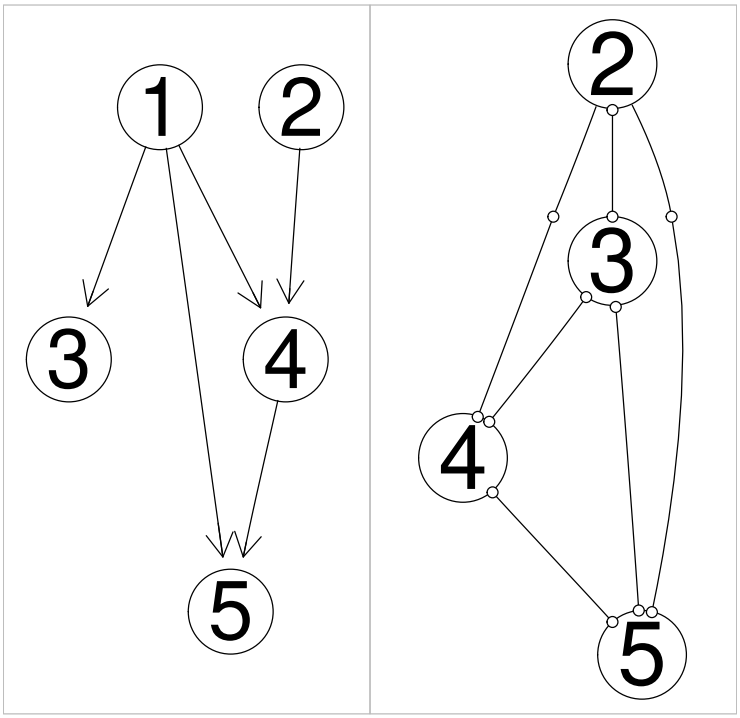

Figure 2: True causal DAG (left) and estimated PAG on the observed nodes with the labels 2, 3, 4 and 5 (right).

and Maathuis [2014]. However, the default function estimates an initial order-independent skeleton in these algorithms (for additional details on how to make the final skeleton of FCI fully order-independent, see Colombo and Maathuis [2014]).

2.2.3 `fciPlus()`

The FCI algorithm yields the true PAG (ignoring sampling error) but suffers (in the worst case) from exponential runtime even if the underlying graph is sparse. The RFCI algorithm is a fast approximation of the FCI algorithm. While the output of RFCI might be a different graph (even when ignoring sampling error), the derived ancestral informations are guaranteed to be correct but perhaps less informative than the true PAG.

Claassen et al. [2013] proposed the FCI+ algorithm which improves on FCI and RFCI in the following way: It yields the true PAG (ignoring sampling error) and is of polynomial complexity in the number of nodes, at least for sparse graphs with a bounded neighbourhood. The FCI+ algorithm in implemented in the function `fciPlus()`. The available arguments are a subset of the arguments

available in `fci()`.

We return to the example from section 2.2.2 and show the usage of function `fciPlus()` on the built-in dataset `gmL` containing four gaussian variables.

```
> suffStat <- list(C = cor(gmL$x), n = nrow(gmL$x))
> fciPlus.gmL <- fciPlus(suffStat, indepTest=gaussCItest,
                         alpha = 0.9999, labels = c("2","3","4","5"))
```

The estiamted PAG is identical to the PAG shown in Fig. 2.

2.3 Score-based methods

An alternative to constraint-based learning is a score-based approach. The idea behind score-based learning is the following: The agreement between data and a possible causal structure is assessed by a score. The causal structure is then estimated by the causal structure with the best score. With this approach the choice of the scoring function is crucial. Moreover, due to the large space of possible causal structures, heuristic search methods are often used.

2.3.1 `ges()` for the GES algorithm

A prominent example of score-based learning is Greedy-Equivalent-Search (GES) (Chickering [2002a, 2003]). This algorithm scores the causal structure using a score-equivalent and decomposable score, such as the BIC score (Schwarz [1978]). A score is score-equivalent, if it assigns the same value to all DAGs within the same Markov equivalence class. A score is decomposable, if it can be computed as a sum of terms (typically one term for each node) depending only on local features of the causal structure.

The idea of GES is to greedily search through Markov equivalence classes, i.e. the space of CPDAGs. It can be outlined in two (or three) steps. The GES algorithm starts with a CPDAG (often the empty CPDAG) and then adds, in a first step (called "forward phase"), edges in a greedy way (i.e. maximising the increase in score) until the considered score cannot be further increased. A single edge addition or, more precisely, forward step conceptually consists of getting a DAG representative of the former CPDAG, adding a single arrow to this DAG, and finally calculating the CPDAG of the new DAG. Then, in a second step (called "backward phase"), edges are greedily removed until, again, an optimum of the score is reached. As before, a single backward step in the space of CPDAGs is analogous to removing a single arrow from a graph in the space of DAGs. The benefit of the GES algorithm lies in the fact that it explores the search space in a computationally efficient manner, i.e. without actually generating the aforementioned representatives. GES was shown to be consistent in the setting where the number of variables remains fixed and the sample size goes to infinity (see Chickering [2002a]). This is quite remarkable, since it involves a greedy search.

The algorithm can be improved by including a turning phase ("third step") of edges (see Hauser and Bühlmann [2012]).

For the score-based GES algorithm, we have to define a score object before applying the inference algorithm. A score object is an instance of a class derived from the base class `Score`. This base class is implemented as a virtual reference class. At the moment, the `pcalg` package only contains classes derived from `Score` for Gaussian data: `GaussL0penObsScore` for purely i.i.d. data, and `GaussL0penIntScore` for a mixture of data sources (e.g. observational and interventional data); for the GES algorithm, we only need the first one here, but we will encounter the second one in the discussion of GIES, an extension of GES to interventional data (see section 2.3.2). However, the flexible implementation using class inheritance allows the user to implement own score classes for different scores.

The predefined score-class `GaussL0penObsScore` implements an ℓ_0-penalized maximum-likelihood estimator for observational data from a linear structural equation model with Gaussian noise. In such a model, associated with a DAG G, every structural equation is of the form $X = c + BX + \varepsilon$ where ε follows a mutivariate normal distribution and $B_{ij} \neq 0$ iff node X_j is a parent of node X_i in G. Given a dataset D, the score of a DAG G is then defined as

$$S(G, D) := \log(L(D)) - \lambda \cdot k \ , \tag{1}$$

where $L(D)$ stands for the maximum of the likelihood function of the model, and k represents the number of parameters in the model.

An instance of `GaussL0penObsScore` is generated as follows:

```
> score <- new("GaussL0penObsScore", data = matrix(1, 1, 1),
      lambda = 0.5*log(nrow(data)), intercept = FALSE,
      use.cpp = TRUE, ...)
```

The data matrix is provided by the argument `data`. The penalization constant λ (see equation (1)) is specified by `lambda`. The default value of `lambda` corresponds to the BIC score; the AIC score is realized by setting `lambda` to 1. The argument `intercept` indicates whether the model should allow for intercepts (`c` in the above equation) in the linear structural equations. The last argument `use.cpp` indicates whether the internal C++ library should be used for calculation, which is in most cases the best choice for reasons of speed.

Once a score object is defined, the GES algorithm is called as follows:

```
ges(score, labels = score$getNodes(),
    fixedGaps = NULL, adaptive = c("none",
        "vstructures", "triples"), phase = c("forward",
        "backward", "turning"), iterate = length(phase) >
        1, turning = NULL, maxDegree = integer(0),
    verbose = FALSE, ...)
```

The argument `score` is a score object defined before. The phases (forward, backward, or turning) that should actually be used are specified with the `phase` argument. The argument `iterate` indicates whether the specified phases should be executed only once (`iterate = FALSE`), or whether they should be executed

```
> score <- new("GaussL0penObsScore", gmG8$x)
> ges.fit <- ges(score)
> par(mfrow=1:2)
> plot(gmG8$g, main = "")          ; box(col="gray")
> plot(ges.fit$essgraph, main = ""); box(col="gray")
```

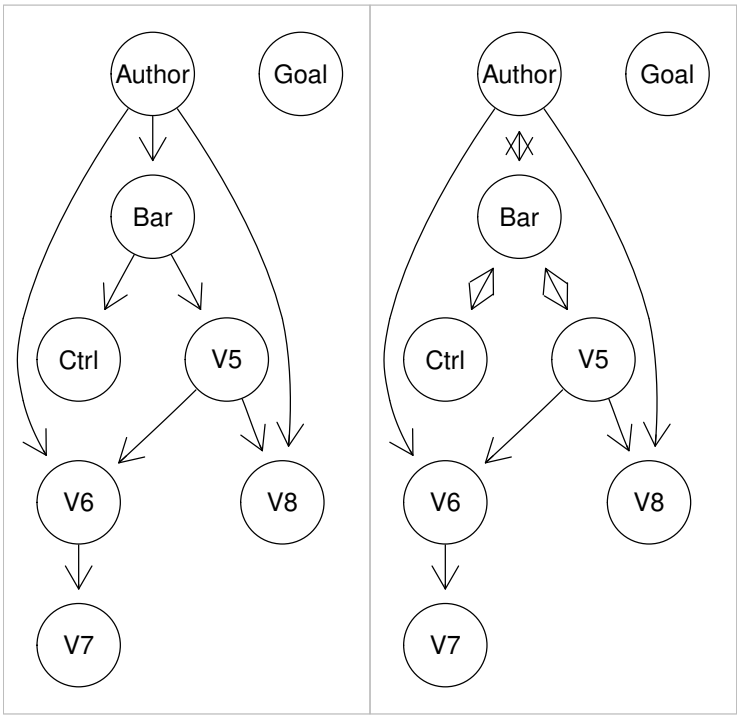

Figure 3: True underlying DAG (left) and CPDAG (right) estimated with the GES algorithm fitted on the simulated Gaussian dataset `gmG8`.

over and over again until no improvement of the score is possible anymore (`iterate = TRUE`). The default settings require all three phases to be used iteratively. The original implementation of Chickering [2002b] corresponds to setting `phase = c("forward", "backward"), iterate = FALSE`.

In Fig. 3, we re-analyze the dataset used in the example of Fig. 1 with the GES algorithm. The estimated graph is exactly the same in this case. Note that GES is order-independent (although the result depends on the starting graph of GES) by design, and that it does, in contrast to the PC algorithm, not depend on the `skeleton()` function.

2.3.2 `gies()`

The algorithms PC and GES both rely on the i.i.d. assumption and are not suited for causal inference from interventional data. The GIES algorithm, which stands for "greedy interventional equivalence search", is a generalization of GES to a mix of observational and interventional (where interventions are known) data or data from different settings (Hauser and Bühlmann [2012]). It does not only make sure that interventional data points are handled correctly (instead of being wrongly treated as observational data points), but also accounts for the improved identifiablity of causal models under interventional data by returning an *interventional CPDAG*.

The usage of `gies()` is similar to that of `ges()` described above. Actually, the function `ges()` is only an internal wrapper for `gies()`.

A dataset with jointly interventional and observational data points is *not* i.i.d. In order to use it for causal inference, we must specify the intervention target each data point belongs to. This is done by specifying the arguments `target` and `target.index` when generating an instance of `GaussL0penIntScore`:

```
> score <- new("GaussL0penIntScore", data = matrix(1, 1, 1),
      targets = list(integer(0)),
      target.index = rep(as.integer(1), nrow(data)),
      lambda = 0.5*log(nrow(data)), intercept = FALSE,
      use.cpp = TRUE, ...)
```

The argument `targets` is a list of all (mutually different) targets that have been intervened in the experiments generating the dataset (joint interventions are possible). The allocation of sample indices to intervention targets is specified by the argument `target.index`. This is an integer vector whose first entry specifies the index of the intervention target in the list `targets` of the first data point, whose second entry specifies the target index of the second data point, and so on.

An example can be found in the dataset `gmInt` which can be loaded by

```
> data(gmInt)
```

The dataset consists of 5000 data points sampled from the DAG in Fig. 3, among them 3000 observational ones, 1000 originating from an intervention at vertex 3 (with node label "Ctrl") and 1000 originating from an intervention at vertex 5 (with node label "V5"). These sampling properties are encoded by `gmInt$targets`, which is a list consisting of an empty vector, the (one-dimensional) vector `c(3)` and the vector `c(5)`, and by `gmInt$target.index`, which is a vector with 5000 entries in total, 3000 1's (referring to the first target, the empty one), 1000 2's (referring to the second target, `c(3)`), and finally 1000 3's (referring to the third target, `c(5)`).

Once a score object for interventional data is defined as described above, the GIES algorithm is called as follows:

```
gies(score, labels = score$getNodes(), targets = score$getTargets(),
    fixedGaps = NULL, adaptive = c("none", "vstructures",
```

```
        "triples"), phase = c("forward", "backward",
        "turning"), iterate = length(phase) > 1, turning = NULL,
    maxDegree = integer(0), verbose = FALSE, ...)
```

Most arguments coincide with those of `ges()`. The causal model underlying `gmInt` as well as the interventional CPDAG estimated by GIES can be found in Fig. 4.

2.3.3 `simy()`

As an alternative to GIES, we can also use the dynamic programming approach of Silander and Myllymäki [2006] to estimate the interventional CPDAG from this interventional dataset. This algorithm is implemented in the function `simy()` which has the same arguments as `gies()`. This approach yields the *exact* optimum of the BIC score at the price of a computational complexity which is exponential in the number of variables. On the small example based on 8 variables this algorithm is feasible; however, it is not feasible for more than approximately 25 variables, depending on the processor and memory of the machine. In this example, we get exactly the same result as with `gies()` (see Fig. 4).

2.4 Hybrid methods: ARGES

It is possible to restrict the search space of GES to subgraphs of a skeleton or conditional independence graph (CIG)[2] estimated in advance. Such a combination of a constraint-based and a score-based algorithm is called a hybrid method.

GES can be restricted to subgraphs of a given graph using the argument `fixedGaps`. This argument takes a symmetric boolean matrix; if the entry (i, j) is `TRUE`, `ges()` is not allowed to put an edge between nodes i and j. In other words, the argument `fixedGaps` takes the adjacency matrix of the graph complement [3] of a previously estimated CIG or skeleton, as illustrated in the follwing code example:

```
> score <- new("GaussL0penObsScore", gmG8$x)
> suffStat <- list(C = cor(gmG8$x), n = nrow(gmG8$x))
> skel.fit <- skeleton(suffStat = suffStat, indepTest = gaussCItest,
                       alpha = 0.01, p = ncol(gmG8$x))
> skel <- as(skel.fit@graph, "matrix")
> ges.fit <- ges(score, fixedGaps = !skel)
```

The resulting graph is not shown, it is the same as in Fig. 3.

The drawback of this straight-forward approach of a hybrid algorithm is the lack of consistency, even when using a consistent estimator for the undirected

[2] In a CIG an edge is missing if the two end-nodes are conditionally independent when conditioning on all remaining nodes.

[3] i.e. edges become gaps and gaps become edges

```
> score <- new("GaussL0penIntScore", gmInt$x, targets = gmInt$targets,
               target.index = gmInt$target.index)
> gies.fit <- gies(score)
> simy.fit <- simy(score)
> par(mfrow = c(1,3))
> plot(gmInt$g, main = "")            ; box(col="gray")
> plot(gies.fit$essgraph, main = "") ; box(col="gray")
> plot(simy.fit$essgraph, main = "") ; box(col="gray")
```

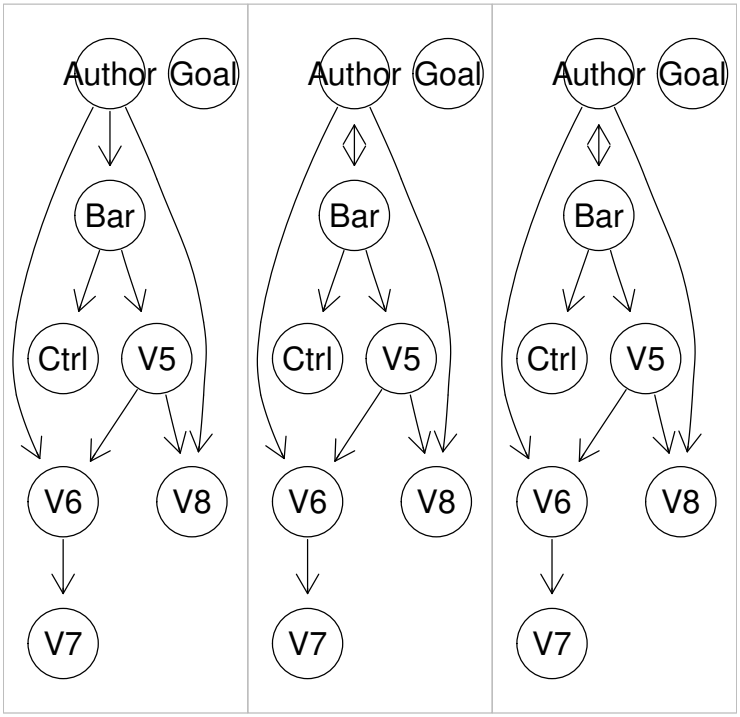

Figure 4: The underlying DAG (left) and the CPDAG estimated with the GIES algorithm (middle) and the dynamic programming approach of Silander and Myllymäki [2006] (right) applied on the simulated interventional Gaussian dataset `gmInt`. This dataset contains data from interventions at vertices 3 (with label "Ctrl") and 5 (with label "V5"); accordingly, the orientation of all arrows incident to these two vertices becomes identifiable (see also Fig. 3 for comparison with the observational case).

graph. Nandy et al. [2018] showed that a small modification of the forward phase of GES makes its combination with a consistent CIG or skeleton estimator consistent again; this modification is called ARGES, "adaptively restricted GES". ARGES can be called using the argument `adaptive` of the function `ges()`:

```
> score <- new("GaussL0penObsScore", gmG8$x)
> library(huge)
> huge.path <- huge(gmG8$x, verbose = FALSE)
> huge.fit <- huge.select(huge.path, verbose = FALSE)
> cig <- as.matrix(huge.fit$refit)
> ges.fit <- ges(score, fixedGaps = !cig, adaptive = "vstructures")
```

In this example (which again yields the same plot as in Fig. 3), we used methods from package `huge` to estimate the CIG in advance. Next, we called `ges()` with the argument `adaptive="vstructures"`, which calls a variant of ARGES called ARGES-CIG. When estimating the *skeleton* instead of the CPDAG in advance, one should use ARGES-skeleton instead, another variant of ARGES (called by the argument `adaptive="triples"`). In both variants of ARGES, only the forward phase is different from the base version of GES, the backward (and possibly turning) phase are identical. Note that the instruction on fixed gaps is not guaranteed to be respected in the final output as explained in Nandy et al. [2018].

2.5 Restricted structural equation models: LINGAM

Given observational data, the causal structure can in general only be determinded up to the Markov equivalence class of the causal structure. In special cases, however, full identification of the causal structure is possible.

A prominent example is the Linear Non-Gaussian Acyclic Model (LINGAM) for Causal Discovery, see Shimizu et al. [2006]. This method aims at discovering the complete causal structure of continuous-valued data, under the following assumptions: The data generating process is linear ($X = c + BX + \varepsilon$), there are no unobserved confounders, and error variables have non-Gaussian distributions of non-zero variances. The method is based on independent component analysis and is implemented in the function `lingam()`.

The input of `lingam()` is a data matrix with n rows (samples) and p columns (variables). The output is an `R` object of (S3) class `"LINGAM"`, basically a `list` with three components: `Bpruned` contains a $p \times p$-matrix B of linear coefficients, where $B_{i,j} \neq 0$ if $j \to i$. `stde` is a vector of length p with the standard deviations of the estimated residuals. `ci` is a vector of length p with the intercepts of each equation.

As an example, we show how to completely discover the true causal structure in the setting of only two correlated variables assuming no unobserved confounders. Note that when assuming a linear generating process with Gaussian errors, it would not be possible to completely discover the causal structure. However, since we now assume non-Gaussian errors, `lingam()` will succeed in completely determining the causal structure.

```
> set.seed(1234)
> n <- 500
> ## Truth: stde[1] = 0.89
> eps1 <- sign(rnorm(n)) * sqrt(abs(rnorm(n)))
> ## Truth: stde[2] = 0.29
> eps2 <- runif(n) - 0.5
> ## Truth: ci[2] = 3, Bpruned[1,1] = Bpruned[2,1] = 0
> x2 <- 3 + eps2
> ## Truth: ci[1] = 7, Bpruned[1,2] = 0.9, Bpruned[2,2] = 0
> x1 <- 0.9*x2 + 7 + eps1
> # Truth: x1 <- x2
```

Thus, the causal graph of variables x_1 and x_2 is $x_1 \leftarrow x_2$. In the linear coefficients matrix B, the only non-zero entry is $B_{1,2} = 0.9$. The true vector of intercepts has entries $c_1 = 7$ and $c_2 = 3$. Note that the equations are linear and the errors follow non-gaussian distributions, thus following the main assumptions of LINGAM. Now, we use the function `lingam()` to estimate the causal structure:

```
> X <- cbind(x1,x2)
> res <- lingam(X)
> res

$Bpruned
     [,1]      [,2]
[1,]    0 0.9105471
[2,]    0 0.0000000

$stde
[1] 0.8464599 0.2802133

$ci
[1] 6.922077 3.010565

attr(,"class")
[1] "LINGAM"
```

We can see that the structure of the causal model was estimated correctly: The only non-zero entry in the estimated linear coefficients matrix (called `Bpruned` in output) is $\hat{B}_{1,2}$, i.e., the estimated causal structure is $x_1 \leftarrow x_2$. Moreover, the estimated value of $\hat{B}_{1,2} = 0.91$ comes very close to the true value $B_{1,2} = 0.9$. The estimated vector of intercepts (called `ci` in the output: 6.92 and 3.01) is also close to the true vector of intercepts (7 and 3).

2.6 Adding background knowledge

In many applications background knowledge of the causal system is available. This information is typically of two kinds: Either it is known that a certain edge

must or must not be present. Or the orientation of a given edge is known (e.g. temporal information). This kind of background knowledge can be incorporated in several structure learning functions.

As explained in section 2.3.1, function `ges()` has the argument `fixedGaps` for restricting GES to a certain subgraph, which results in the ARGES algorithm.

In functions `skeleton()`, `pc()`, `fci()`, `rfci()` (but currently not `fciPlus()`) background knowledge on the presence or absence of edges can be entered through the arguments `fixedEdges` and `fixedGaps` and is guaranteed to be respected in the final output.

Moreover, for CPDAGs background knowledge on the orientation of edges can be added using function `addBgKnowledge()`. Note that adding orientation information to a CPDAG might not result in a CPDAG anymore but will always result in a PDAG. Applying the orientation rules from Meek [1995] might orient further edges resulting in a maximally oriented PDAG (see Perković et al. [2017] for more details). Function `addBgKnowledge()` is called as follows:

```
> showF(addBgKnowledge)

addBgKnowledge(gInput, x = c(), y = c(), verbose = FALSE,
    checkInput = TRUE)
```

The argument `gInput` is either a `graphNEL`-object or an adjacency matrix of type `amat.cpdag`. `x` and `y` are node labels so that edges should be oriented in the direction $x \to y$. If argument `checkInput` is `TRUE`, the input adjacency matrix is carefully checked to see if it is a valid graph. This is done using function `isValidGraph()`, which checks whether an adjacency matrix with coding `amat.cpdag` is of type CPDAG, DAG or maximally oriented PDAG. Based on this input, function `addBgKnowledge()` adds orientation $x \to y$ to the adjacency matrix and completes the orientation rules from Meek [1995]. If x and y are not specified (or empty vectors) this function simply completes the orientation rules from Meek [1995]. If x and y are vectors of length k, $k > 1$, this function tries to add $x_i \to y_i$ to the adjacency matrix and complete the orientation rules from Meek [1995] for every i in $1, ..., k$ (see Algorithm 1 in Perković et al. [2017]). The output of function `addBgKnowledge()` is a maximally oriented PDAG with coding `amat.cpdag`.

As an example, we force on the CPDAG in Fig. 5 (left) the orientation $a \to b$. By applying the orientation rules of Meek [1995] afterwards, edge $b \to c$ becomes oriented, too.

```
> amat <- matrix(c(0,1,0, 1,0,1, 0,1,0), 3,3) ## a -- b -- c
> colnames(amat) <- rownames(amat) <- letters[1:3]
> ## force a -> b
> bg <- addBgKnowledge(gInput = amat, x = "a", y = "b")
```

Note that it is currently *not* possible to define edge orientations *before* the CPDAG is estimated. Moreover, adding background knowledge in the form of edge orientations is currently not supported for PAGs or interventional CPDAGs.

```
> par(mfrow = c(1,2))
> plot(as(t(amat), "graphNEL")); box(col="gray")
> plot(as(t( bg ), "graphNEL")); box(col="gray")
```

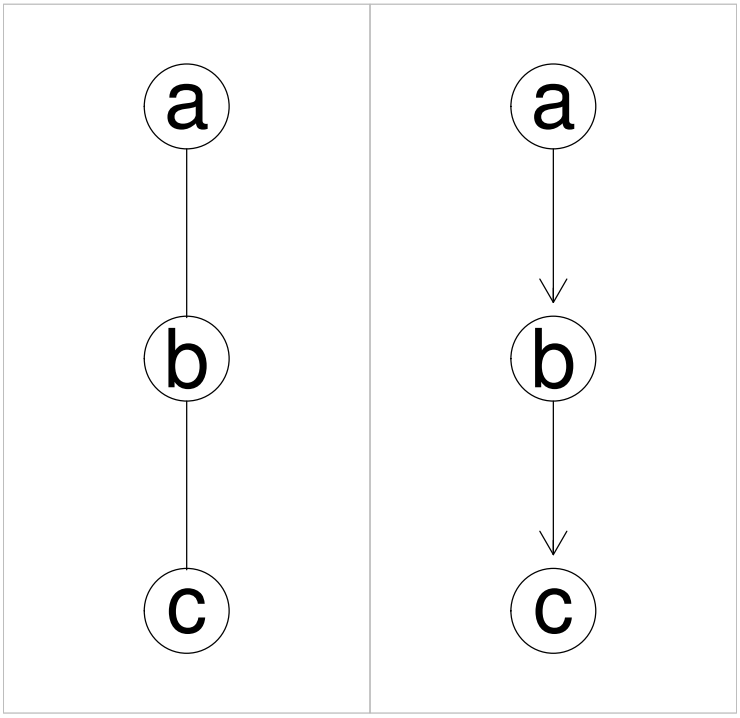

Figure 5: Left: Original CPDAG. Right: After adding the background knowledge $a \to b$ the edge $b \to c$ is automatically directed by applying the orientation rules from Meek [1995]. The result is a maximally oriented PDAG.

2.7 Summary of assumptions

In the following we summarize the assumptions of all mentioned structure learning methods.

PC algorithm: Faithfulness; no hidden or selection variables; consistent in high-dimensional settings given suitable assumptions; consistency in a standard asymptotic regime with a fixed number of variables follows as a special case; implemented in function `pc()`; see Kalisch and Bühlmann [2007] for full details.

FCI algorithm: Faithfulness; allows for hidden and selection variables; consistent in high-dimensional settings given suitable assumptions; consistency in a standard asymptotic regime with a fixed number of variables follows as a special case; implemented in function `fci()`; see Colombo et al. [2012] for full details.

RFCI algorithm: Faithfulness; allows for hidden and selection variables; polynomial runtime if the graph resulting in step 1 of RFCI is sparse; consistent in high-dimensional settings given suitable assumptions; consistency in a standard asymptotic regime with a fixed number of variables follows as a special case; implemented in function `rfci()`; see Colombo et al. [2012] for full details.

FCI+ algorithm: Faithfulness; allows for hidden and selection variables; polynomial runtime if the true underlying causal PAG is sparse; implemented in function `fciPlus()`. See Claassen et al. [2013] for full details.

LINGAM: No hidden or selection variables; data generating process is a linear structural equation model with non-Gaussian errors; see Shimizu et al. [2006] for full details.

GES algorithm: Faithfulness; no hidden or selection variables; consistency in high-dimensional setting given suitable assumptions (see Nandy et al. [2018]); consistency in a standard asymptotic regime with a fixed number of variables. Implemented in function `ges()`. See Chickering [2002b] for full details.

ARGES algorithm: Faithfulness; no hidden or selection variables; consistency in high-dimensional settings given suitable assumptions; implemented in function `ges()`, using argument `adaptive = TRUE`; see Nandy et al. [2018] for full details.

GIES algorithm: Faithfulness; no hidden or selection variables; mix of observational and interventional data; implemented in function `gies()`; see Hauser and Bühlmann [2012] for full details.

Dynamic programming approach of Silander and Myllymäki: Same assumptions as for GIES, but only up to approximately 25 variables (depending on CPU and memory resources). Implemented in function `simy()`. See Silander and Myllymäki [2006] for full details.

3 Covariate Adjustment

3.1 Introduction

Estimating causal effects from observational data is possible when using the right confounding variables as an adjustment set:

Adjustment set; Maathuis and Colombo [2015b] Let $\mathbf{X}, \mathbf{Y}$ and $\mathbf{Z}$ be pairwise disjoint node sets in a DAG, CPDAG, MAG or PAG G. Then $\mathbf{Z}$ is an adjustment set relative to $(\mathbf{X}, \mathbf{Y})$ in G if for any density[4] f consistent with G we have

$$f(\mathbf{y}|do(\mathbf{x})) = \begin{cases} f(\mathbf{y}|\mathbf{x}) & \text{if } \mathbf{Z} = \emptyset, \\ \int_{\mathbf{z}} f(\mathbf{y}|\mathbf{x}, \mathbf{z}) f(\mathbf{z}) d\mathbf{z} & \text{otherwise.} \end{cases} \tag{2}$$

Thus, adjustment sets allow to write post-intervention densities involving the do-operator (left-hand side of (2)) as specific functions of the usual conditional densities (right-hand side of (2)). The latter can be estimated from observational data.

However, in practice it is hard to determine what a valid adjustment set is. A common misconception is that adjusting for more variables is always better. This is not the case, as is detailed in the "M-bias graph" example (Shrier [2008], Rubin [2008]).

Given the practical importance of covariate adjustment, criteria have been developed for finding a valid adjustment set given the true causal structure underlying the data. For example, Pearl's Back-door Criterion (BC) (Pearl [1993b]) is a well known criterion for DAGs. Shpitser et al. [2010a] and Shpitser et al. [2010b] refined the back-door criterion to a sound and complete graphical criterion for adjustment in DAGs. Others considered more general graph classes, which can represent structural uncertainty. van der Zander et al. [2014] gave sound and complete graphical criteria for MAGs that allow for unobserved variables (latent confounding). Maathuis and Colombo [2015b] generalize this criterion for DAGs, CPDAGs, MAGs and PAGs (Generalized Backdoor Criterion, GBC). These two criteria are sound (i.e., if the criterion claims that a set is a valid adjustment set, then this claim is correct) but incomplete (i.e., there might be valid adjustment sets which are not detected by the criterion). Perković et al. [2015] present a criterion for covariate adjustment that is sound and complete for DAGs, CPDAGs, MAGs and PAGs without selection variables (Generalize Adjustment Criterion, GAC). The theoretical contribution of that paper closes the chapter on covariate adjustment for the corresponding graph classes. More details on GAC can be found in Perković et al. [2018].

Recently, Perković et al. [2017] extended the use of GAC on graphs incorporating background knowledge: PDAGs.

In the following example we show that, given suitable assumptions, the total causal effect of a variable X on another variable Y can be estimated using linear regression with the correct adjustment set $\mathbf{Z}$. Assume the data is generated by

[4] We use the notation for continuous random variables throughout. The discrete analogues should be obvious.

a multivariate Gaussian density that is consistent with a causal DAG G. Let $\mathbf{Z} \neq \emptyset$ be a valid adjustment set (e.g. according to GAC) relative to two variables X and Y in G such that $\mathbf{Z} \cap \{X \cup Y\} = \{\}$. Then

$$E(Y|do(x)) \quad = \quad \alpha + \gamma x + \beta^T E(\mathbf{z}). \tag{3}$$

We define the total causal effect of X on Y as $\frac{\partial}{\partial x} E(Y|do(x))$. Thus, the total causal effect of X on Y is γ, which is the regression coefficient of X in the regression of Y on X and $\mathbf{Z}$.

The available functions for covariate adjustment in package `pcalg` can be categorized in the following way:

- Compute causal effect by (conceptually) listing all DAGs in a given equivalence class: `ida()`, `jointIda()`
- Check if a given set is a valid adjustment set: `backdoor()`, `gac()` (also incorporating background knowledge)
- Given a graph, find a (or several) valid adjustment set(s): `adjustment()`, `backdoor()`

More details on assumptions can be found in section 3.3.

3.2 Methods for Covariate Adjustment

3.2.1 `ida()`

The first functions for estimating the causal effect of a single intervention variable X on a target variable Y using covariate adjustment included in `pcalg` were: `ida()` and a restricted but faster version `idaFast()` (for several target variables at the same time with restricted options). Conceptually, the method works as follows. First, an estimated CPDAG is provided as input (e.g. using the function `pc()`). Then we extract a collection of "valid" parent sets of the intervention variable from the estimated CPDAG. For each set of valid parent sets we compute a linear regression of Y on X using the parent set as covariate adjustment set. Thus, for each valid parent set, we derive one estimated effect resulting in a multi-set of causal effects.

This function can be called in the following way:

```
ida(x.pos, y.pos, mcov, graphEst, method = c("local",
    "optimal", "global"), y.notparent = FALSE,
    verbose = FALSE, all.dags = NA, type = c("cpdag",
        "pdag"))
```

`x.pos` and `y.pos` are the (integer) positions of variables X and Y, respectively. `mcov` is the covariance matrix that was used to estimte the CPDAG passed in argument `graphEst`. With argument `type` one can define if the estimated graph is either a CPDAG or a PDAG (e.g. after including background knowledge). If `method` is set to `global` the algorithm considers all DAGs in the

represented by the CPDAG or PDAG, hence is slow. If `method` is set to `local` the algorithm only considers the neighborhood of X in the CPDAG or PDAG, hence is faster. Moreover, the multiplicities of the estimated causal effects might be wrong. As an example, suppose that a CPDAG represents eight DAGs. The global method might produce the multiset 1.3, -0.5, 0.7, 1.3, 1.3, -0.5, 0.7, 0.7. The unique values in this set are -0.5, 0.7 and 1.3, and the multiplicities are 2, 3 and 3. The local method, on the other hand, might produce 1.3, -0.5, -0.5, 0.7. The unique values are again -0.5, 0.7 and 1.3, but the multiplicities are now 2, 1 and 1. Since the unique values of the multisets of the "global" and "local" method are identical, summary measures of the multiset that only depend on the unique values (e.g. minimum absolute value) can be estimate by the faster local method.

As an example, we simulate a random DAG, sample data, estimate the CPDAG (see Fig. 6) and apply `ida` to find the total causal effect from node number 2 to node number 5 using both the local and the global method. We can see that both methods produce the same unique values but different multiplicities.

```
> ## Simulate the true DAG
> set.seed(123)
> p <- 7
> myDAG <- pcalg::randomDAG(p, prob = 0.2) ## true DAG
> myCPDAG <- dag2cpdag(myDAG) ## true CPDAG
> ## simulate Gaussian data from the true DAG
> n <- 10000
> dat <- rmvDAG(n, myDAG)
> ## estimate CPDAG and PDAG -- see  help(pc)
> suffStat <- list(C = cor(dat), n = n)
> pc.fit <- pc(suffStat, indepTest = gaussCItest, p=p, alpha = 0.01)
> ## Supppose that we know the true CPDAG and covariance matrix
> (l.ida.cpdag <- ida(2,5, cov(dat),
                      myCPDAG, method = "local", type = "cpdag"))

[1] 0.1748347

> (g.ida.cpdag <- ida(2,5, cov(dat),
                      myCPDAG, method = "global", type = "cpdag"))

[1] 0.1748347
```

3.2.2 `jointIda()`

The function `jointIda()` extends `ida()` by allowing a *set* of intervention variables (i.e., not just a single intervention variable).

Assuming observational data that are multivariate Gaussian and faithful to the true (but unknown) underlying causal DAG (without hidden variables), the

```
> if (require(Rgraphviz)) {
    ## plot the true and estimated graphs
    par(mfrow = c(1,2))
    plot(myDAG, main = "True DAG"); box(col="gray")
    plot(pc.fit, main = "Estimated CPDAG"); box(col="gray")
  }
```

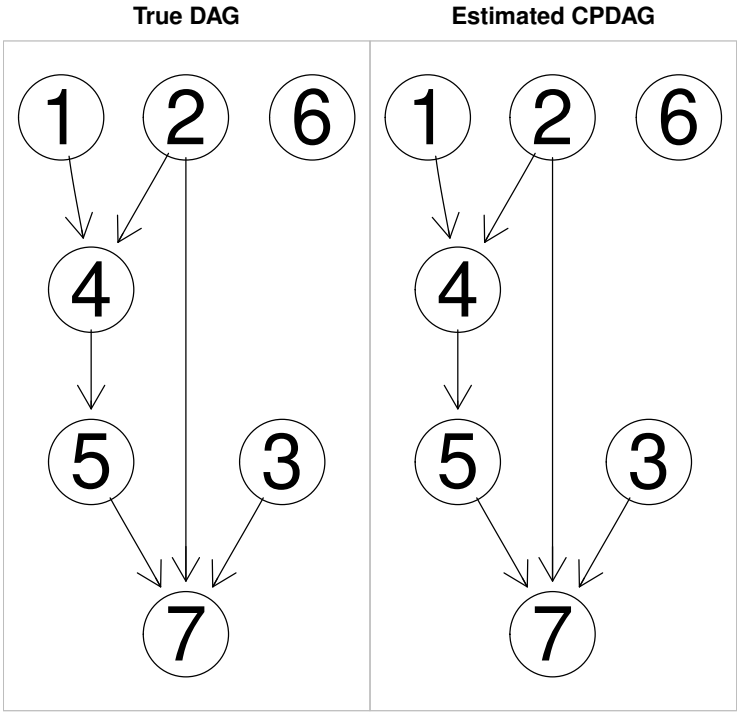

Figure 6: The true DAG (left) and the true CPDAG (right) in the example illustrating `ida()`.

function estimates the multiset of possible total joint effects of X on Y. The total joint effect of $X = (X_1, X_2)$ on Y is defined via Pearl's do-calculus as the vector $(E[Y|do(X_1 = x_1 + 1, X_2 = x_2)] - E[Y|do(X_1 = x_1, X_2 = x_2)], E[Y|do(X_1 = x_1, X_2 = x_2 + 1)] - E[Y|do(X_1 = x_1, X_2 = x_2)])$, with a similar definition for more than two variables. These values are equal to the partial derivatives (evaluated at (x_1, x_2)) of $E[Y|do(X = x_1', X_2 = x_2')]$ with respect to x_1' and x_2'. Moreover, under the Gaussian assumption, these partial derivatives do not depend on the values at which they are evaluated.

As with `ida()`, `jointIda()` needs an estimated CPDAG as input. It then constructs a collection of "jointly valid" parent sets of all intervention variables. For each set of jointly valid parent sets we apply RRC (recursive regressions for causal effects) or MCD (modifying the Cholesky decomposition) to estimate the total joint effect of X on Y from the sample covariance matrix.

When X is a single variable, `jointIda()` estimates the same quantities as `ida()`. When `graphEst` is a CPDAG, `jointIda()` yields correct multiplicities of the distinct elements of the resulting multiset (i.e., it matches `ida()` with `method="global"` up to a constant factor), while `ida()` with `method="local"` does not have this property. Like `idaFast()`, the effect on several target variables can be computed at the same time.

In the following example, we generate a DAG on six nodes and generate data from it (see Fig. 7).

```
> ## Generate DAG for simulating data
> p <- 6
> V <- as.character(1:p)
> edL <- list(
      "1" = list(edges=c(3,4), weights=c(1.1,0.3)),
      "2" = list(edges=c(6),  weights=c(0.4)),
      "3" = list(edges=c(2,4,6),weights=c(0.6,0.8,0.9)),
      "4" = list(edges=c(2),weights=c(0.5)),
      "5" = list(edges=c(1,4),weights=c(0.2,0.7)),
      "6" = NULL)
> myDAG <- new("graphNEL", nodes=V, edgeL=edL,
               edgemode="directed") ## true DAG
> myCPDAG <- dag2cpdag(myDAG) ## true CPDAG
> covTrue <- trueCov(myDAG) ## true covariance matrix
```

Then, we use `jointIda()` to estimate (using method `RCC`) the causal effect of an intervention at nodes 1 and 2 on the target variable 6. First, we use the true DAG and the true covariance matrix for illustration.

```
> ## Compute causal effect using true CPDAG and true cov. matrix
> (resExactDAG <- jointIda(x.pos = c(1,2), y.pos = 6,
                           mcov = covTrue,
                           graphEst = myDAG,
                           technique = "RRC"))
```

```
     [,1]
[1,] 0.99
[2,] 0.40
```

The result is a matrix representing the estimated possible total joint effects of X on Y. The number of rows equals the number of intervention variables. Thus, when intervening at both node 1 and node 2, a unit increase in node 1 leads to an increase of 0.99 in node 6, while a unit increase in node 2 leads to an increase of 0.40 in node 6.

Now we replace the true DAG by the true CPDAG. It is usually not possible anymore to estimate the causal effects uniquely. Thus, the result is a matrix representing the multiset containing the estimated possible total joint effects of X on Y. The number of rows equals the number of intervention variables. Each column represents a vector of possible joint causal effects.

```
> (resExactCPDAG <- jointIda(x.pos = c(1,2), y.pos = 6,
                             mcov = covTrue,
                             graphEst = myCPDAG,
                             technique = "RRC"))
```

```
     [,1]          [,2] [,3]
[1,] 0.99 -1.124101e-15 0.99
[2,] 0.40  4.000000e-01 0.40
```

In this example, the multisets contain three elements. The first and the third column coincide with our finding in the previous DAG example. The middle column shows new values. Without further information we cannot decide which of the elements of the multiset correspond to the true underlying causal system.

For building intuition, we will inspect the true underlying DAG and confirm the result: Node 2 is a parent node of node 6 and the weight is indeed 0.40. Thus, the causal effect of node 2 on node 6 is indeed 0.4. Now we compute the causal effect of node 1 on node 6. There are several possible directed paths from node 1 to node 6: 1-3-6, 1-3-2-6, 1-3-4-2-6 and 1-4-2-6. If node 1 was the only intervention variable, we would now compute the causal effects along all 4 paths and add them up. However, since we did a joint intervention on node 1 *and* node 2, the value of node 2 is fixed. Thus, changing the value of node 1 will have an effect on node 6 only over directed paths that do *not* include node 2. Thus, the only directed path is 1-3-6 with edge weights 1.1 and 0.9. Multiplying these two weights we get the causal effect along this path of 0.99, as was also produced with the function `jointIda()`.

However, the input of `jointIda()` is not the true DAG but the true CPDAG. The parent set of node 2 is unique in this CPDAG. However, the parent set of node 2 is not unique. At first sight, possible parent sets for node 2 are: empty set, only node 3, only node 5, both node 3 and 5. However, if both node 3 and node 5 would be parent sets of node 1, this would introduce a new v-structure in the graph. Thus, this parent set is not valid and we are left with three valid parent sets on of which coincides with the parent sets in the true DAG (only

```
> par(mfrow = c(1,2))
> plot(myDAG)  ; box(col="gray")
> plot(myCPDAG); box(col="gray")
```

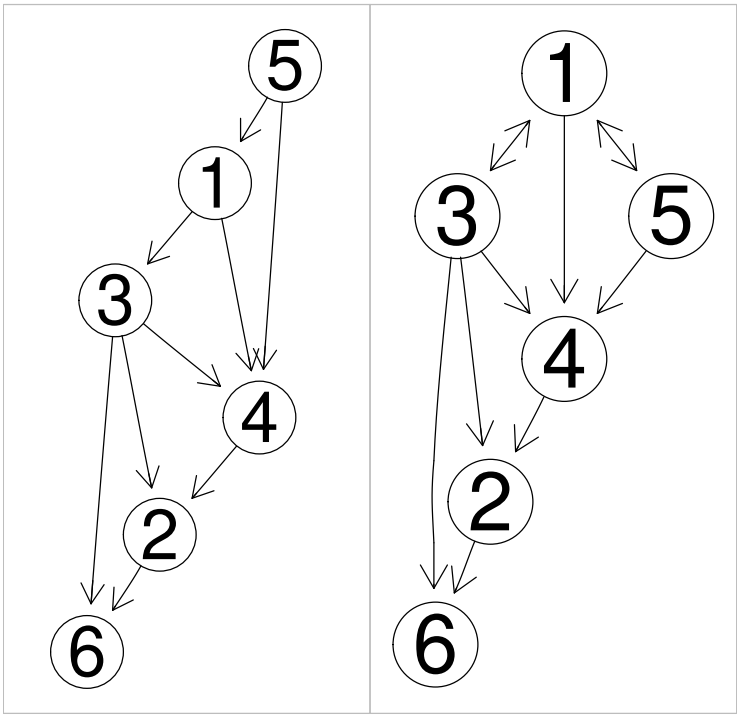

Figure 7: The true DAG (left) and the true CPDAG (right) in the example illustrating `jointIda()`.

node 5 is parent). For each of the three valid parent sets the same calculation as before yields the values in the three columns of the output.

In practice, both the CPDAG and the covariance matrix will be only estimates. Thus, both the estimated values and the multiplicities (number of columns of the result) are prone to error.

3.2.3 `backdoor()`

This function implements the Generalized Backdoor Criterion (GBC). The GBC is a generalization of Pearl's backdoor criterion (see Pearl [1993a]) defined for directed acyclic graphs (DAGs) for single interventions and single outcome variable to more general types of graphs (CPDAGs, MAGs, and PAGs) that describe Markov equivalence classes of DAGs with and without latent variables but without selection variables. For more details see Maathuis and Colombo [2015a].

The motivation to find a set W that satisfies the generalized backdoor criterion with respect to X and Y in the given graph relies on the result of the generalized backdoor adjustment that says: If a set of variables W satisfies the generalized backdoor criterion relative to X and Y in the given graph, then the causal effect of X on Y is identifiable and is given by: $P(Y|\text{do}(X = x)) = \sum_W P(Y|X, W) \cdot P(W)$. This result allows to write post-intervention densities (the one written using Pearl's do-calculus) using only observational densities estimated from the data.

This function can be called in the following way:

```
backdoor(amat, x, y, type = "pag", max.chordal = 10,
    verbose = FALSE)
```

where `amat` is the adjacency matrix of the given graph, `x` denotes the column position of the cause variable, `y` denotes the column position of the effect variable, and `mcov` is the covariance matrix of the original data.

The argument `type` specifies the type of graph of the given adjacency matrix in `amat`. If the input graph is a DAG (`type="dag"`), this function reduces to Pearl's backdoor criterion for single interventions and single outcome variable, and the parents of X in the DAG satisfies the backdoor criterion unless Y is a parent of X. Therefore, if Y is a parent of X, there is no set W that satisfies the generalized backdoor criterion relative to X and Y in the DAG and NA is output. Otherwise, the causal effect is identifiable and a set W that satisfies the generalized backdoor criterion relative to X and Y in the DAG is given. If the input graph is a CPDAG C (`type="cpdag"`), a MAG M, or a PAG P (with both M and P not allowing selection variables), this function first checks if the total causal effect of X on Y is identifiable via the generalized backdoor criterion (see Maathuis and Colombo [2015a], Theorem 4.1). If the effect is not identifiable, the output is NA. Otherwise, an explicit set W that satisfies the generalized backdoor criterion relative to X and Y in the given graph is found.

Note that if the set W is equal to the empty set, the output is NULL.

At this moment this function is not able to work with PAGs estimated using the `rfci` Algorithm.

It is important to note that there can be pair of nodes `x` and `y` for which there is no set W that satisfies the generalized backdoor criterion, but the total causal effect might be identifiable via some other technique.

To illustrate this function, we use the CPDAG displayed in Fig. 4, page 15 of Maathuis and Colombo [2015a]. The R-code below is used to generate a DAG `g` that belongs to the required equivalence class which is uniquely represented by the estimated CPDAG `myCPDAG`.

```
> p <- 6
> amat <- t(matrix(c(0,0,1,1,0,1, 0,0,1,1,0,1, 0,0,0,0,1,0,
                     0,0,0,0,1,1, 0,0,0,0,0,0, 0,0,0,0,0,0), 6,6))
> V <- as.character(1:6)
> colnames(amat) <- rownames(amat) <- V
> edL <- vector("list",length=6)
> names(edL) <- V
> edL[[1]] <- list(edges=c(3,4,6),weights=c(1,1,1))
> edL[[2]] <- list(edges=c(3,4,6),weights=c(1,1,1))
> edL[[3]] <- list(edges=5,weights=c(1))
> edL[[4]] <- list(edges=c(5,6),weights=c(1,1))
> g <- new("graphNEL", nodes=V, edgeL=edL, edgemode="directed")
> cov.mat <- trueCov(g)
> myCPDAG <- dag2cpdag(g)
> true.amat <- as(myCPDAG, "matrix")
> ## true.amat[true.amat != 0] <- 1
```

The DAG `g` and the CPDAG `myCPDAG` are shown in Fig. 8.

Now, we want to check if the effect of V_6 on V_3 in the given CPDAG is identifiable using `backdoor()` and if this is the case know which set W satisfies the generalized backdoor criterion. As explained in Example 4 in Maathuis and Colombo [2015a], the causal effect of V_6 on V_3 in the CPDAG `myCPDAG` is identifiable via the generalized backdoor criterion and there is a set $W = \{1, 2\}$ that satisfies the generalized backdoor criterion:

```
> backdoor(true.amat, 6, 3, type="cpdag")

[1] 1 2
```

3.2.4 `gac()`

The Generalized Adjustment Criterion (GAC) is a generalization of the Generalized Backdoor Criterion (GBC) of Maathuis and Colombo [2015a]: While GBC is sufficient but not necessary, GAC is both sufficient and necessary for DAGs, CPDAGs, MAGs and PAGs. Moreover, while GBC was originally only defined for single intervention and target variables, GAC also works with sets of target and/or intervention variables. For more details see Perković et al. [2018]. The Generalized Adjustment Criterion (GAC) is implemented in function `gac()`.

This function can be called in the following way:

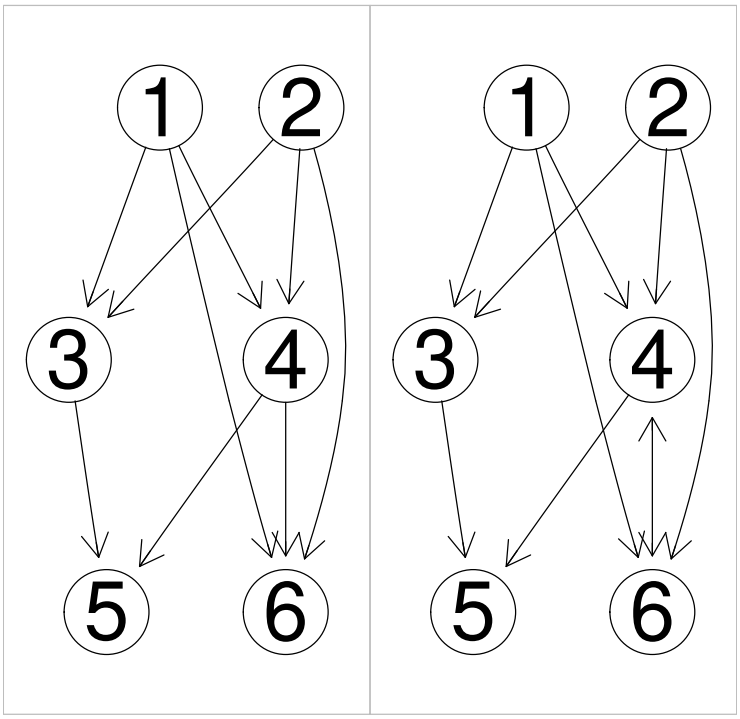

Figure 8: True DAG (left) and estimated CPDAG (right).

```
gac(amat, x, y, z, type = "pag")
```

where `amat` is the adjacency matrix of the given graph. `x` is a vector with all the (integer) positions of intervention nodes. `y` is a vector with all the (integer) positions of target nodes. `z` is a vector with all the (integer) positions of the adjustment set that should be checked with the GAC.

The output of `gac()` will be a list with three components: `gac` is `TRUE` if `z` satisfies the GAC relative to `(x,y)` in the graph represented by `amat` and `type`. `res` is a logical vector of length three indicating if each of the three conditions (0), (1) and (2) of GAC are true. `f` is a vector containing (integer) node positions of nodes in the forbidden set.

In the following example we consider the CPDAG shown in Fig. 9(a) and check whether the set consisting of node 2 and node 4 satisfies the GAC for estimating the causal effect from node 3 to node 6.

```
> mFig1 <- matrix(c(0,1,1,0,0,0, 1,0,1,1,1,0, 0,0,0,0,0,1,
                    0,1,1,0,1,1, 0,1,0,1,0,1, 0,0,0,0,0,0), 6,6)
> type <- "cpdag"
> x <- 3; y <- 6
> ## Z satisfies GAC :
> gac(mFig1, x,y, z=c(2,4),    type)

$gac
[1] TRUE

$res
[1] TRUE TRUE TRUE

$f
[1] 6
```

In the output we can see that all three conditions of GAC are satisfied and thus, GAC is satisfied. The forbidden set consists of node 6.

Function `gac()` can also be used on maximally oriented PDAGs. Such a graph might arise by adding orientational background knowledge to a CPDAG (see section 2.6). Details can be found in Perković et al. [2017]. As an illustration we reproduce Example 4.7b in Perković et al. [2017] (shown in Fig. 9(b)):

```
> mFig3a <- matrix(c(0,1,0,0, 0,0,1,1, 0,0,0,1, 0,0,1,0), 4,4)
> type <- "pdag"
> x <- 2; y <- 4
> ## Z does not satisfy GAC
> str( gac(mFig3a,x,y, z=NULL, type) )## not amenable rel. to (X,Y)

List of 3
 $ gac: logi TRUE
 $ res: logi [1:3] TRUE TRUE TRUE
 $ f  : num [1:2] 3 4
```

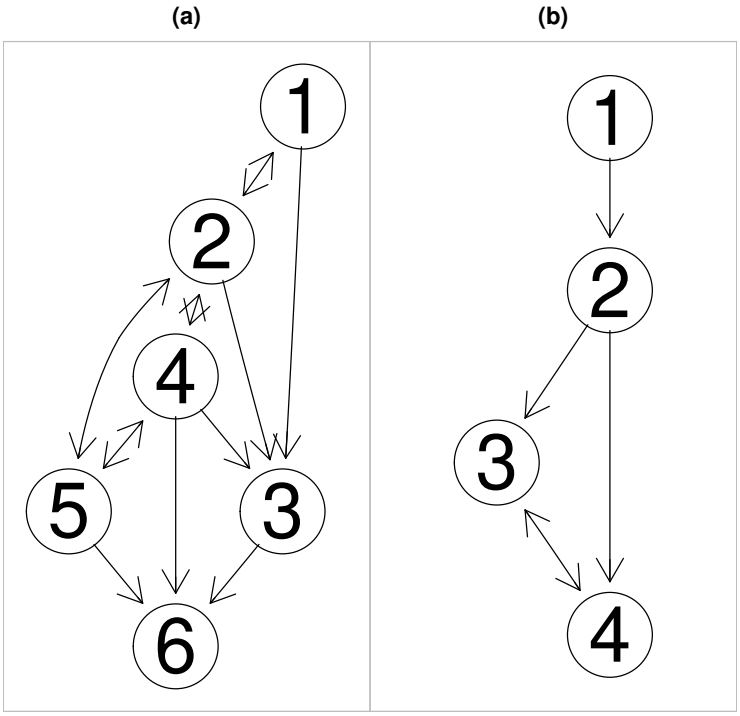

Figure 9: A CPDAG (left) and the PDAG from Example 4.7 (right) from Perković et al. [2017] used in the example illustrating `gac()`.

3.2.5 `adjustment()`

The previously discussed functions check whether a given set of variables is valid for adjustment or not. However, with the exception of `backdoor()`, they don't help finding a valid adjustment set in the first place. For finding valid adjustment sets the function `adjustment()` can be used. This function is an interface to the function `adjustmentSet()` from package `dagitty`. This function can be called in the following way:

```
>     showF(adjustment)

adjustment(amat, amat.type, x, y, set.type)
```

3.3 Summary of assumptions

3.4 Methods for covariate adjustment

IDA: No hidden or selection variables; all conditional expectations are linear; see Maathuis et al. [2009] for full details. Implemented in function `ida()`.

jointIDA: Same assumptions as IDA but can deal with several intervention variables at the same time. Implemented in function `jointIda()`.

Pearl's Backdoor Criterion (BC): No hidden or selection variables. Sound, but not complete. Implemented as a special case of GBC in function `gbc()`.

Generalized Backdoor Criterion (GBC): Allows for arbitrarily many hidden but no selection variables; works on DAG, CPDAG, MAG and PAG. Sound, but not complete. Implemented in function `gbc()`.

Generalized Adjustment Criterion (GAC): Allows for arbitrarily many hidden but no selection variables; works on DAG, CPDAG, MAG and PAG. Sound and complete. Implemented in function `gac()`.

4 Random DAG Generation

Simulation methods are essential to investigate the performance of estimation methods. For that reason, the `pcalg` package included from the beginning the function `randomDAG` to generate random DAGs. However, the method implemented there was restricted to Erdös-Renyi graphs. We now include the new function `randDAG()` which can sample random graphs from a much wider class.

Eight different random graph models are provided. The Erdös-Renyi Graph Model and some important extensions are available (Regular Random Graph Model, Geometric Random Graph Model, Bipartite Random Graph Model, Watts-Strogatz Random Graph Model and the Interconnected-Island Random Graph Model). Moreover, graph models with power law degree distributions are provided (Power-Law Random Graph Model and the Barabasi-Albert Random

Graph Model). The methods are based on the analogous `.game` functions in the `igraph` package.

As an option, individual parameters can be passed for all methods. Alternatively, the desired expected neighborhood size of the sampled graph can be specified and the individual parameters will be set automatically. Using the option `DAG`, the output can either be a DAG or the skeleton of a DAG. In contrast to the old function `randomDAG()`, the nodes in the DAG produced by the new function `randDAG()` are not topologically sorted.

In the following example we generate a random graph `dag1` according to the Erdös-Renyi random graph model (`method="er"`) and a random graph `dag2` according to the Power-Law random graph model (`method="power"`). In both cases, the number of nodes is $n = 100$ and the expected neighborhood size is $d = 3$.

```
> n <- 100; d <- 3; s <- 2
> myWgtFun <- function(m,mu,sd) { rnorm(m,mu,sd) }
> set.seed(42)
> dag1 <- randDAG(n=n, d=d, method = "er", DAG = TRUE)
> dag2 <- randDAG(n=n, d=d, method = "power", DAG = TRUE)
```

The average neighborhood sizes in `dag1` and `dag2` are 2.94 and 2.86, respectively. The maximum neighborhood sizes in `dag1` and `dag2`, however, are 9 and 42, respectively. Thus, as expected, in the power-law graph some nodes have a much larger neighborhood size than in the Erdös-Renyi graph.

We now expand the previous example by also generating edge weights. This is done using the arguments `weighted = TRUE` and `wFUN`. The argument `wFUN` takes a function for randomly generating the edge weights. For this example, function `myWgtFun` is defined and passed to argument `wFUN`. Function `myWgtFun` takes as first argument a number of edges `m` (this value will be automatically specified within the call of `randDAG` and therefore it need not be specified by the user) for which it returns a vector of length `m` containing the generated weights. Alternatively, `wFUN` can be a list consisting of the function in the first entry and of further arguments of the function in the additional entries. In this example, the edge weights are sampled independently from $N(0, s^2)$ where $s = 2$.

```
> n <- 100; d <- 3; s <- 2
> myWgtFun <- function(m,mu,sd) { rnorm(m,mu,sd) }
> set.seed(42)
> dag1 <- randDAG(n=n, d=d, method = "er", DAG = TRUE,
                  weighted = TRUE, wFUN = list(myWgtFun, 0, s))
> dag2 <- randDAG(n=n, d=d, method = "power", DAG = TRUE,
                  weighted = TRUE, wFUN = list(myWgtFun, 0, s))
```

Previous versions of package `pcalg` contained the functions `unifDAG()` and `unifDAG.approx()` for sampling DAGs uniformly from the space of all DAGs given a certain number of nodes. These functions were moved to the package `unifDAG`.

5 General Object Handling

5.1 A comment on design

The `pcalg` package is an organically growing collection of functions related to structure learning and causal inference. It captures several research results produced at the Seminar for Statistics, ETH Zürich and also implements several other common algorithms (e.g. for comparison). Given this history, we hope the user will forgive the lack of an overarching design.

Functionality centered around constraint based learning (`skeleton()`, `pc()`, `fci()`, `rfci()` and `fciPlus()`) is based on the S4 classes `pcAlgo` and `fciAlgo` (both inheriting from virtual class `gAlgo`) and the S3 class `amat`.

Functionality centered around score based learning (`gies()`, `ges()`, `simy()`) are based on the virtual reference classes `ParDAG` (for parametric causal models), `Score` (for scoring classes) and `EssGraph` (for interventional CPDAGs).

Functionality centered around covariate adjustment mainly follows functional programming.

5.2 Adjacency matrices

Two types of adjacency matrices are used in package `pcalg`: Type `amat.cpdag` for DAGs and CPDAGs and type `amat.pag` for MAGs and PAGs. See helpfile of `amatType` for coding conventions for the entries of the adjacency matrices. The required type of adjacency matrix is documented in the help files of the respective functions or classes.

Using the coercion `as(from, "amat")` one can extract such adjacency matrices as (S3) objects of `class "amat"`. We illustrate this using the estimated CPDAG from section 2.2.1:

```
> ## as(*, "amat") returns an adjacency matrix incl. its type
> as(pc.gmG8, "amat")

Adjacency Matrix 'amat' (8 x 8) of type 'cpdag':
       Author Bar Ctrl Goal V5 V6 V7 V8
Author      .   1    .    .  .  .  .  .
Bar         1   .    1    .  .  .  .  .
Ctrl        .   1    .    .  .  .  .  .
Goal        .   .    .    .  .  .  .  .
V5          .   1    .    .  .  .  .  .
V6          1   .    .    .  1  .  .  .
V7          .   .    .    .  .  1  .  .
V8          1   .    .    .  1  .  .  .
```

In some functions, more detailed information on the graph type is needed (i.e. DAG or CPDAG; MAG or PAG). Such information is passed in a separate argument. We illustrate this using the function `gac()` which, in this example, takes as input an adjacency matrix `m1` of type `amat.cpdag`. In addition to

that, the information that the input is actually a DAG is passed through the argument `type`:

```
> ## INPUT: Adjacency matrix of type 'amat.cpdag'
> m1 <- matrix(c(0,1,0,1,0,0, 0,0,1,0,1,0, 0,0,0,0,0,1,
                 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0), 6,6)
> ## more detailed information on the graph type needed by gac()
> str( gac(m1, x=1,y=3, z=NULL, type = "dag") )
```

```
List of 3
 $ gac: logi TRUE
 $ res: logi [1:3] TRUE TRUE TRUE
 $ f  : num [1:4] 2 3 5 6
```

5.3 Interface to package dagitty

The web-based software *DAGitty* offers functionality for analyzing causal graphical models. The R package `dagitty` offers an interface to *DAGitty*. In our package `pcalg` we offer the function `pcalg2dagitty()` for converting the adjacency matrix of type `amat.cpdag` or `amat.pag` into a `dagitty` object. Thus, it is easy to use the full functionality of `dagitty`. As an example, we convert an adjacency matrix to a dagitty object and use the function `is.dagitty()` from the `dagitty` package to verify that the resulting object is indeed a `dagitty` object.

```
> n <- nrow    (gmG8$x)
> V <- colnames(gmG8$x) # labels aka node names
> amat <- wgtMatrix(gmG8$g)
> amat[amat != 0] <- 1
> if(requireNamespace("dagitty")) {
      dagitty_dag1 <- pcalg2dagitty(amat,V,type="dag")
      dagitty::is.dagitty(dagitty_dag1)
  }
```

```
[1] TRUE
```

5.4 Methods for visualizing graph objects

The most flexible way to visualize graph objects is via the `Rgraphviz` package. In this package, several different edge marks are possible. Unfortunately, due to a persistent bug, the edgemarks are sometimes not placed at the end of an edge but at some other position along the edge. Nevertheless, for most purposes `Rgraphviz` will probably produce the best visualization of the graph. As an example, we plot in Fig. 10 a DAG and a PAG (which requires more complex edgemarks).

If the package `Rgraphviz` is not available, we provide the function `iplotPC()` as an interface to the `igraph` package for plotting `pcAlgo` objects (graphs with

```
> set.seed(42)
> p <- 4
> ## generate and draw random DAG :
> myDAG <- pcalg::randomDAG(p, prob = 0.4)
> myCPDAG <- dag2cpdag(myDAG)
> ## find skeleton and PAG using the FCI algorithm
> suffStat <- list(C = cov2cor(trueCov(myDAG)), n = 10^9)
> fci.fit <- fci(suffStat, indepTest=gaussCItest,
                 alpha = 0.9999, p=p, doPdsep = FALSE)
> if (require(Rgraphviz)) {
    par(mfrow = c(1,2))
    plot(myCPDAG); box(col="gray") ## CPDAG
    plot(fci.fit); box(col="gray") ## PAG
  }
```

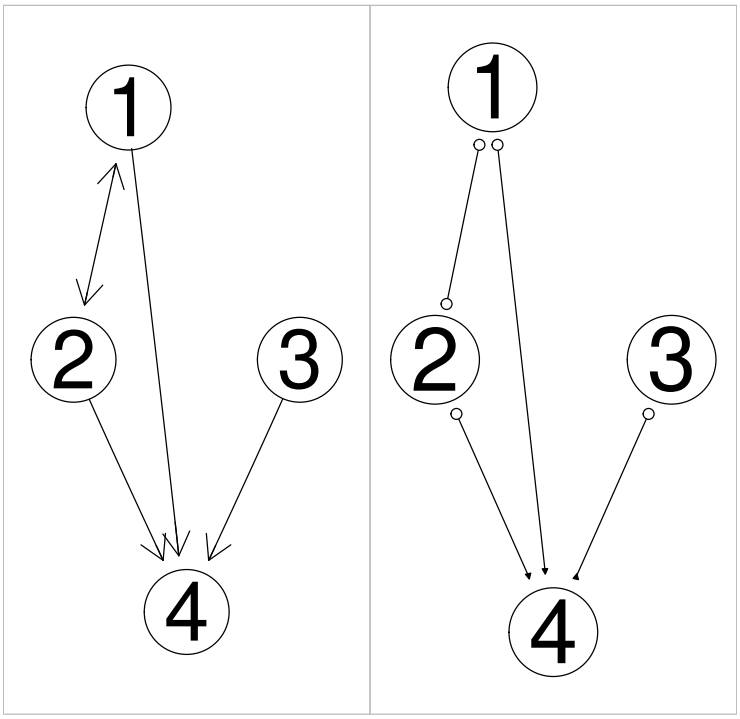

Figure 10: True causal DAG (left) and the corresponding PAG (right) visualized using package `Rgraphviz`.

more complex edge marks, e.g. circles, are currently not supported). As an example, we plot in Fig. 11 the result of calling the function `pc()` using the function `iplotPC()`:

```
> data(gmG)
> n <- nrow    (gmG8$ x)
> V <- colnames(gmG8$ x) # labels aka node names
> ## estimate CPDAG
> pc.fit <- pc(suffStat = list(C = cor(gmG8$x), n = n),
               indepTest = gaussCItest, ## indep.test: partial correlations
               alpha=0.01, labels = V, verbose = FALSE)
> if (require(igraph)) {
    par(mfrow = c(1,1))
    iplotPC(pc.fit)
  }
```

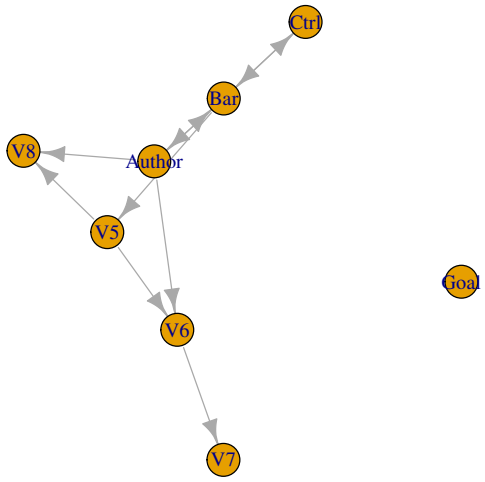

Figure 11: Visualizing with the `igraph` package: The Estimated CPDAG.

Finally, if neither `Rgraphviz` nor `igraph` are available, the estimated graph object can be converted to an adjacency matrix and inspected in the console. See the helpfile of `amatType` for coding details.

```
> as(pc.fit,  "amat") ## Adj. matrix of type  'amat.cpdag'

Adjacency Matrix 'amat' (8 x 8) of type 'cpdag':
       Author Bar Ctrl Goal V5 V6 V7 V8
```

```
Author      .   1    .    .  .  .  .  .
Bar         1   .    1    .  .  .  .  .
Ctrl        .   1    .    .  .  .  .  .
Goal        .   .    .    .  .  .  .  .
V5          .   1    .    .  .  .  .  .
V6          1   .    .    .  1  .  .  .
V7          .   .    .    .  .  1  .  .
V8          1   .    .    .  1  .  .  .
```

```
> as(fci.fit, "amat") ## Adj. matrix of type  'amat.pag'

Adjacency Matrix 'amat' (4 x 4) of type 'pag':
  1 2 3 4
1 . 1 . 2
2 1 . . 2
3 . . . 2
4 1 1 1 .
```

Alternatively, for `pcAlgo` objects also an edge list can be produced.

```
> showEdgeList(pc.fit) ## Edge list

Edge List:

Undirected Edges:
  Author  ---  Bar
  Bar  ---  Ctrl

Directed Edges:
  Author  -->  V6
  Author  -->  V8
  Bar  -->  V5
  V5  -->  V6
  V5  -->  V8
  V6  -->  V7
```

For graph objects it is possible to visualize only a sub-graph using function `plotSG()`.

References

D. M. Chickering. Learning equivalence classes of Bayesian-network structures. *J. Mach. Learn. Res.*, 2:445–498, 2002a.

D. M. Chickering. Optimal structure identification with greedy search. *J. Mach. Learn.*, 3(3):507–554, 2002b.

D. M. Chickering. Optimal structure identification with greedy search. *J. Mach. Learn. Res.*, 3:507–554, 2003.

T. Claassen, J. Mooij, and T. Heskes. Learning sparse causal models is not NP-hard. In *Proceedings UAI 2013*, 2013.

D. Colombo and M. H Maathuis. Order-independent constraint-based causal structure learning. *J. Mach. Learn. Res.*, 15:3741–3782, 2014.

D. Colombo, M. H. Maathuis, M. Kalisch, and T. S. Richardson. Learning High-Dimensional drected acyclic graphs with latent and selection variables. *Ann. Stat.*, 40(1):294–321, 2012.

A. Hauser and P. Bühlmann. Characterization and greedy learning of interventional Markov equivalence classes of directed acyclic graphs. *J. Mach. Learn. Res.*, 13:2409–2464, 2012.

A. Hauser and P. Bühlmann. Characterization and greedy learning of interventional Markov equivalence classes of directed acyclic graphs. *J. Mach. Learn.*, 13:2409–2464, 2012.

C. Heinze-Deml, M. H. Maathuis, and N. Meinshausen. Causal structure learning. *Ann. Rev. Stat. Appl.*, 5:371–391, 2018.

M. Kalisch and P. Bühlmann. Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm. *J. Mach. Learn.*, 8:613–636, 2007.

M. Kalisch and P. Bühlmann. Causal structure learning and inference: a selective review. *Qual. Tech. & Quant. Man.*, 11:3—21, 2014.

M. Kalisch, M. Mächler, D. Colombo, M.H. Maathuis, and P. Bühlmann. Causal inference using graphical models with the R package `pcalg`. *J. Stat. Softw.*, 47:1–26, 2012.

M. H. Maathuis and D. Colombo. A generalized backdoor criterion. *Ann. Stat.*, 43:1060–1088, 2015a.

M. H. Maathuis and D. Colombo. A generalized back-door criterion. *Ann. Stat.*, 43:1060–1088, 2015b.

M. H. Maathuis and M. Drton. Structure learning in graphical modeling. *Ann. Rev. Stat. Appl.*, 4:365–393, 2017.

M. H. Maathuis and P. Nandy. *A review of some recent advances in causal inference*. Handbook of Big Data. Chapman and Hall/CRC, Boca Raton, FL, 2016.

M. H. Maathuis, M. Kalisch, and P. Bühlmann. Estimating high-dimensional intervention effects from observational data. *Ann. Stat.*, 37:3133–3164, 2009.

Christopher Meek. Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI 1995)*, pages 403–418, 1995.

P. Nandy, A. Hauser, and M. H. Maathuis. Understanding consistency in hybrid causal structure learning. *Ann. Stat.*, 2018. To appear, available at arXiv:1507.02608.

J. Pearl. Comment: Graphical models, causality and intervention. *Stat. Sci.*, 8: 266–269, 1993a.

J. Pearl. Comment: Graphical models, causality and intervention. *Stat. Sci.*, 8: 266–269, 1993b.

E. Perković, J. Textor, M. Kalisch, and M. H. Maathuis. A complete generalized adjustment criterion. In *Proceedings UAI 2015*, pages 682–691, 2015.

E. Perković, M. Kalisch, and M. H. Maathuis. Interpreting and using CPDAGs with background knowledge. In *Proceedings UAI 2017*, 2017.

E. Perković, J. Textor, M. Kalisch, and M. H. Maathuis. Complete graphical characterization and construction of adjustment sets in Markov equivalence classes of ancestral graphs. *J. Mach. Learn. Res.*, 2018. To appear, available at arXiv:1606.06903.

J. Ramsey, J. Zhang, and P. Spirtes. Adjacency-Faithfulness and Conservative Causal Inference. In *Proceedings UAI 2006*, pages 401–408, 2006.

D. Rubin. Author's reply. *Stat. Med.*, 27:2741–2742, 2008.

G. Schwarz. Estimating the dimension of a model. *Ann. Stat.*, 6(2):461–464, 1978.

S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. Kerminen. A linear non-Gaussian acyclic model for causal discovery. *J. Mach. Learn. Res.*, 7:2003–2030, 2006.

I. Shpitser, T. J. VanderWeele, and J. Robins. On the validity of covariate adjustment for estimating causal effects. In *Proceedings UAI 2010*, pages 527–536, 2010a.

I. Shpitser, T. J. VanderWeele, and J. Robins. Appendum to *On the validity of covariate adjustment for estimating causal effects*. Personal communication, 2010b.

I. Shrier. Letter to the editor. *Stat. Med.*, 27:2740–2741, 2008.

T. Silander and P. Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings UAI 2006*, pages 445–452, 2006.

P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, second edition, 2000.

B. van der Zander, M. Liśkiewicz, and J. Textor. Constructing separators and adjustment sets in ancestral graphs. In *Proceedings UAI 2014*, pages 907–916, 2014.

J. Zhang. On the completeness of orientation rules for causal discovery in the presence of latent confounders and selection bias. *Art. Intel.*, 172:1873–1896, 2008.

6 Appendix A: Simulation study

In this section we give more details on the simulation study in chapter 5 of Perković et al. [2017]. We show how to reproduce Fig. 4 and Fig. 5 in that document. However, in this document we choose parameter settings so that the simulation study can be computed very quickly but the results are much less informative. Still, they show the same qualitative behavior as in the paper.

First, we set up the parameters. In the comments we indicate the parameter settings that were chosen in Perković et al. [2017]:

```
> possible_p <- c(seq(5,10,by=1))   # paper: possible_p = c(seq(20,100,by=10))
> possible_neighb_size <- c(1:3)    # paper: possible_neighb_size = c(3:10)
> num_settings <-10                 # paper: num_settings = 1000
> num_rep <- 2                      # paper: num_rep = 20
> pb <- seq(0,1,by=0.5)             # paper: pb = seq(0,1,by=0.2)
```

Next we define a helper function and run the simulation:

```
> ## helper function
> revealEdge <- function(c,d,s) {
    ## cpdag, dag, selected edges to reveal
    if (!anyNA(s)) { ## something to reveal
      for (i in 1:nrow(s)) {
        c[s[i,1], s[i,2]] <- d[s[i,1], s[i,2]]
        c[s[i,2], s[i,1]] <- d[s[i,2], s[i,1]]
      }
    }
    c
  }
> ## save results from each iteration in here:
> resFin <- vector("list", num_settings)
> ## run simulation
> for(r in 1:num_settings) {

    set.seed(r)
    ## Then we sample one setting:
    p <-  sample(possible_p,1)
    neigh <- sample(possible_neighb_size,1)
    prob <- round(neigh/(p-1),3)

    resFin[[r]] <- vector("list", num_rep)

    ## then for every setting selected we generate num_rep graphs
    for (i in 1:num_rep){

      ## get DAG
      isEmpty <-  1
```

```
while(isEmpty){
  g <- pcalg::randomDAG(p, prob)  ## true DAG
  cpdag <- dag2cpdag(g) ## true CPDAG

  ## get adjacency matrix of the CPDAG  and DAG
  cpdag.amat <- t(as(cpdag,"matrix"))
  dag.amat <- t(as(g,"matrix"))
  dag.amat[dag.amat != 0] <- 1

  ## only continue if the graph is not fully un-connected
  if (sum(dag.amat)!= 0){
    isEmpty <- 0
  }
}

## choose x and y
y <- NULL
while(is.null(y)){
  # choose x
  x <- sample(p,1)

  ## choose y as a node connected to x but not x <- y
  skeleton <- cpdag.amat + t(cpdag.amat)
  skeleton[skeleton == 2] <- 1
  connectt <- possDe(skeleton,x, type = "pdag")
  if (length(connectt) != 1) {
    pa.x <- which(dag.amat[x,]==1 & dag.amat[,x]==0)
    ## remove x and parents of x (in the DAG) from pos.y
    pos.y <- setdiff(setdiff(connectt, pa.x), x)
    if (length(pos.y)==1){
      y <- pos.y[1]
    } else if (length(pos.y) > 0) {
      y <- sample(pos.y, 1)
    }
  }
}

## calculate true effect:
true_effect <- causalEffect(g,y,x)

## sample data for ida
## need to set nData
nData <- 200
dat <- rmvDAG(nData, g) ## sample data from true DAG

## Resulting lists, of same length as 'pb' :
```

```
pdag.amat <-
adjust_set <-
result_adjust_set <-
ida_effects <- vector("list", length(pb))

## for each proportion of background knowledge
## find a PDAG and an adjustment set relative to (x,y) in this
## PDAG aditionally calculate the set of possible total
## causal effects of x on y using ida in this PDAG
for (j in 1:length(pb)){
  ## reveal proportion pb[j] of bg knowledge
  tmp <- ( (cpdag.amat + t(cpdag.amat))==2 ) &
    lower.tri(cpdag.amat)
  ude <- which(tmp, arr.ind = TRUE) ## undir edges
  nbg <- round(pb[j] * nrow(ude)) ## nmb of edges to reveal
  ## edges to reveal
  sele <- if (nbg>0) ude[sample(nrow(ude), nbg),,drop=FALSE] else NA
  ## reveal edges
  pdag.amat[[j]] <- revealEdge(cpdag.amat, dag.amat, sele)
  pdag.amat[[j]] <- addBgKnowledge(pdag.amat[[j]],
                                   checkInput = FALSE)

  ## find adjustment set (if it exists)
  adjust <- if(requireNamespace("dagitty")) {
              adjustment(pdag.amat[[j]],amat.type="pdag",x,y,
                         set.type="canonical")
          } else NULL
  adjust_set[[j]] <- if(length(adjust)) adjust$'1' else NA
  result_adjust_set[[j]] <- length(adjust) > 0
  ## ida
  ## convert to graph for ida()
  pdag.g <- as(t(pdag.amat[[j]]), "graphNEL")
  ida_effects[[j]] <- ida(x,y,cov(dat), graphEst = pdag.g,
                          method = "local", type = "pdag")

  ## for j = 1 that is when pdag.g == cpdag compare
  ## runtime of local method for CPDAGs vs. PDAGs
  if (j == 1){
    time.taken.ida <-
      system.time(ida(x,y,cov(dat), graphEst = pdag.g,
                      method = "local", type = "cpdag"))
    time.taken.bida <-
      system.time(ida(x,y,cov(dat), graphEst = pdag.g,
                      method = "local", type = "pdag"))
  }
}
```

```
      ## save the results
      resFin[[r]][[i]] <- list(seed=r, p=p, prob=prob, neigh=neigh,
                               pb=pb, i=i, nData=nData,
                               dag.amat=dag.amat,
                               pdag.amat=pdag.amat,
                               x=x, y=y,
                               adjust_set = adjust_set,
                               result_adjust_set = result_adjust_set,
                               true_effect = true_effect,
                               ida_effects = ida_effects,
                               time.taken.ida = time.taken.ida,
                               time.taken.bida = time.taken.bida)
    }
  }
```

Next we transform the output of the simulation into a data frame containing summary statistics:

```
> ## total number of unique cpdags = num_settings*num_rep graphs
> nn <- sum(sapply(resFin, length))
> ## make data frame with relevant summary info
> nBG <- length(pb)
> x <- rep(NA, nn*nBG)
> df1 <- data.frame(setting=x, g=x, p=x, neigh=x, pb=x,
                    resAdj = x, idaNum = x, idaRange = x,
                    timeIda = x, timeBida = x,
                    trueEff = x)
> ii <- 0
> for (i1 in 1:length(resFin)) { ## settings
    nLE <- length(resFin[[i1]])
    for (i2 in 1:nLE) { ## graphs per setting
      for (i3 in 1:nBG) { ## BGK
        ii <- ii + 1
        df1[ii,"setting"] <- i1 ## List index for setting
        df1[ii,"g"] <- i2 ## List index for graph within setting
        df1[ii,"p"] <- resFin[[i1]][[i2]]$p ## Nmb nodes in graph
        ## Ave size of neighborhood
        df1[ii,"neigh"] <- resFin[[i1]][[i2]]$neigh
        ## fraction of background knowledge
        df1[ii,"pb"] <- resFin[[i1]][[i2]]$pb[i3]
        ## true if adj set exists
        df1[ii,"resAdj"] <-
          resFin[[i1]][[i2]]$result_adjust_set[[i3]]
        ## nmb unique results of ida
        df1[ii,"idaNum"] <-
          length(unique(resFin[[i1]][[i2]]$ida_effects[[i3]]))
```

```
        ## range of results of ida
        df1[ii,"idaRange"] <-
          diff(range(resFin[[i1]][[i2]]$ida_effects[[i3]]))
        ## runtime for CPDAG using option "cpdag"
        df1[ii,"timeIda"] <-
          resFin[[i1]][[i2]]$time.taken.ida[[1]]
        ## runtime for CPDAG using option "pdag"
        df1[ii,"timeBida"] <-
          resFin[[i1]][[i2]]$time.taken.bida[[1]]
        df1[ii,"trueEff"] <-
          resFin[[i1]][[i2]]$true_effect
      }
    }
  }
```

Finally, we illustrate the result using plots. First, we reproduce a plot like Fig. 4 in Perković et al. [2017] in Fig. 12:

```
>   ## Fig 4 in paper: Fraction of identifiable effects
>   ## Fraction of identifiable effects: ALL EFFECTS
>   tm1 <- tapply(X=df1$resAdj, INDEX=as.factor(df1$pb), FUN = mean)
>   ts1 <- tapply(X=df1$resAdj, INDEX=as.factor(df1$pb),
                  FUN = function(x) sd(x)/sqrt(length(x)))
>   ## Fraction of identifiable effects: add means for
>   ## only NON-ZERO EFFECTS
>   dfNZ <- subset(df1, subset = (trueEff!=0) )
>   tm <- c(tm1,tapply(X=dfNZ$resAdj, INDEX=as.factor(dfNZ$pb),
                       FUN = mean))
>   ts <- c(ts1,tapply(X=dfNZ$resAdj, INDEX=as.factor(dfNZ$pb),
                       FUN = function(x) sd(x)/sqrt(length(x))))
>   dfID <- data.frame(pb = as.factor(names(tm)), fit = tm, se = ts,
                       TrueEffect =
                         as.factor(c(rep("All", length(tm)/2),
                                     rep("Non-zero", length(tm)/2))))
```

Then we reproduce a plot like Fig. 5 in Perković et al. [2017] in Fig. 13:

```
> ## use dfNU2: settings where effect is NOT unique given zero bg knowledge
> nn <- length(pb)
> idx <- rep(seq(1,nrow(df1), by = nn), each = nn) ## pb=0 rows
> nmbIda <- df1$idaNum[idx]
> dfNU2 <- df1[nmbIda > 1,]
> bnTmp <- cut(x=dfNU2$idaNum, breaks = c(0,1,2,3,4,1e9),
               labels = c("1", "2", "3", "4", "5+"))
> dfNU2$idaNumF <- factor(bnTmp, levels = levels(bnTmp)[5:1])
> df3 <- dfNU2[,c("pb", "idaNumF")]
> df3$idx <- 1:nrow(df3)
```

```
> if(require(ggplot2)) {
    k <- ggplot(dfID, aes(pb, fit, ymin = fit-se,
                          ymax = fit+se, col = TrueEffect))
    k + geom_pointrange() +
      xlab("Proportion of background knowledge") +
      ylab("Fraction of identifiable effects via adjustment") +
      theme(legend.position = c(0.9,0.1),
            axis.text=element_text(size = 14),
            axis.title = element_text(size = 14),
            legend.text=element_text(size = 14),
            legend.title=element_text(size = 14))
  }
```

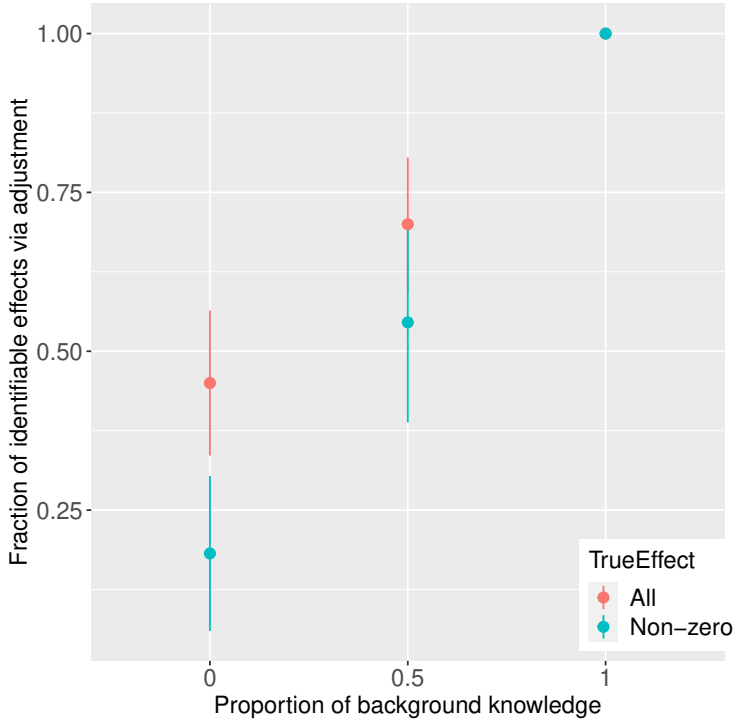

Figure 12: Conceptually reproducing Fig. 4 from Perković et al. [2017]. The parameter setting was simplified a lot in order to reduce runtime, but the qualitative result is still observable: As the proportion of background knowledge increases, the fraction of identifiable effects increases, too.

```
> df3N <- aggregate(idx ~ pb + idaNumF, data = df3, FUN = length)
> df3N$idaNumF <- droplevels(df3N$idaNumF)
```

```
> ggplot(df3N, aes(x = pb, y=idx, fill = idaNumF)) +
    geom_bar(stat = "identity") +
    ylab("Number of simulation settings") +
    xlab("Proportion of background knowledge")+
    theme(axis.text =  element_text(size = 14),
          axis.title=  element_text(size = 14),
          legend.text= element_text(size = 14),
          legend.title=element_text(size = 14)) +
    guides(fill=guide_legend(title="#effects"))
```

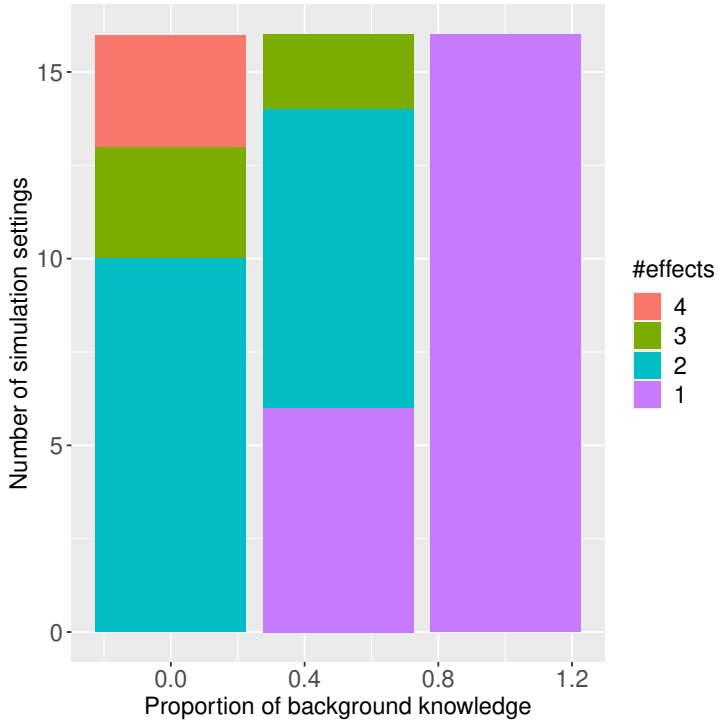

Figure 13: Conceptually reproducing Fig. 5 from Perković et al. [2017]. The parameter setting was simplified a lot in order to reduce runtime, but the qualitative result is still observable: As the proportion of background knowledge increases, the fraction simulation settings in which unambiguous identification of the causal effect is possible increases.