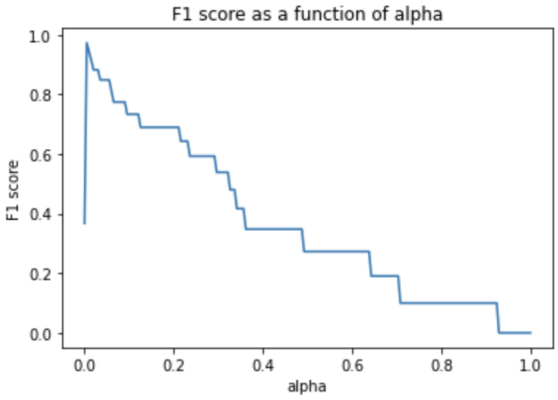
Causal inference project

Nicole Shukry, Caterina La Cava, Ali ElGuindy
scipers: 295798, 358741, 287238

June 2023

Part 1

1 The strategy implemented in the code aims to find the optimal value for the parameter α by evaluating the F1 score for different α values. It starts by dividing the range of α values (0 to 1) into 200 equally spaced values. For each α value, the code conducts hypothesis tests using the `ci_test` function and stores the results in a list. The F1 score is then calculated based on these results. This process is repeated for all α values, and the F1 scores are stored in a list. The F1 scores are plotted against the corresponding α values to visualize their relationship. The code identifies the α value that corresponds to the highest F1 score as the optimal α. Additionally, it determines the maximum F1 score achieved during the evaluation. The strategy involves systematically exploring different α values, evaluating the F1 scores, and selecting the α value that maximizes the F1 score to achieve the best model performance. The plot could be see below:




```
Optimal alpha: 0.005025125628140704
Max F1 Score: 0.972972972972973
```

Figure 1: Alpha vs F1 score with optimal values

2 Actually, it might happen in practice that the Markov boundaries are not symmetric. These asymmetries may arise due to variations in the relationships between variables or the presence of unobserved con-founders.

If it is the case: We conduct conditional independence tests, hence we could identify the parent variables for each variable. Then, for each node (variable) in the graph, we can establish connections between its parent variables.

By following this logic, we could actually know and capture the conditional dependencies without even the need to address any asymmetric Markov boundaries.

3 To resolve conflicts in v-structures caused by potential errors in CI test results, additional evidence from the data can be considered. One approach to resolving such conflicts is to utilize conditional independence tests with subsets of the data that are more informative or robust. This can help reduce the impact of sample noise and improve the accuracy of the CI test results. Furthermore, if conflicts arise between different v-structures, it may be necessary to prioritize or weigh the evidence from the data. This can be done by considering the strength of the statistical significance in the CI test results or by taking into account the size of the subsets used for testing. The idea is to give more weight to the CI test results that are based on larger or more reliable subsets of the data.Additionally, prior knowledge about the relationships between variables can help in making informed decisions and resolving conflicts based on logical consistency. In our approach, we have opted to address conflicts in v-structures. When a v-structure conflicts with another v-structure within the current set of v-structures, we choose to reject the second v-structure. This decision allows us to maintain consistency and resolve conflicts by prioritizing the preservation of a single v-structure rather than potentially conflicting multiple structures.

4 The GS algorithm is expected to be order-independent because the citest function, which is a crucial component of the algorithm, does not depend on the order of the columns in the data. The order of the columns does not affect the recovery of the Markov boundaries in the first phase of the GS algorithm, as it is calculated independently for each variable. Similarly, the neighbor discovery and v-structure extraction in the second phase are also independent for each node and potential v-structure. As said,by we prioritized the preservation of a single v-structure rather than potentially conflicting multiple structures which could affect the order.

5 As soon as a rule is satisfied, the code orients the edge without waiting for the next graph update. This allows for more efficient looping and handling of newly-oriented edges.

The code achieves this by making immediate modifications to the graph and updating the relevant variables. Specifically, when a rule is satisfied, the code orients the edge and updates the `existing_edges` set, which keeps count of all edges, as well as the `directed_edges` set, which keeps count of the directed edges.

By orienting the edge instantly, the code ensures that subsequent iterations of the loop can consider and handle the newly-oriented edges without delay. This approach reduces the need for additional iterations and provides a more efficient way to apply the orientation rules (consistent with our logic in resolving conflicts).

6

- Dataset 1:

  We faced challenges with the neighbor-finding component of the algorithm in part 2. As a result, the subsequent steps of our algorithm became significantly impaired and ineffective when applied to Dataset 1. Hence, we've performed a function called neighbors that firstly returns the node pairs that are neighbors in a list and the Markov boundary graph (moralized) and secondly a function `Step_2_calculations edges` in which we add manually the missing edges (we needed to establish the missing neighbor relationship to get the toy example). NB : In the python code there is a comment ###Run with these lines to see the algorithm working that should be commented/uncommented to see different outputs one with the dysfunction and the other for the right scenario. Find below the graphs obtained with right code (refer to the code to get the dysfunctional code by commenting the corespondent section of the code):

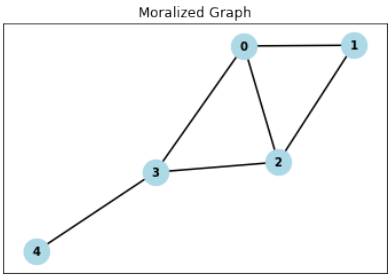


Figure 2: Moralized graph

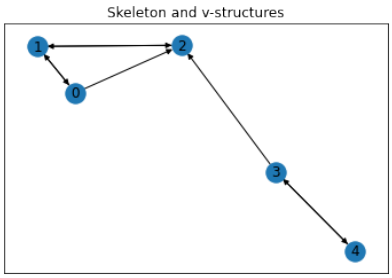


Figure 3: Skeleton+v structures

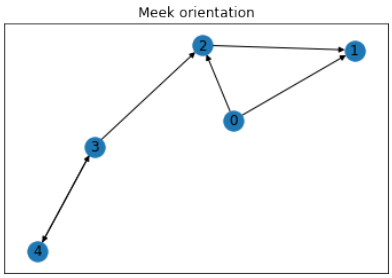


Figure 4: Meek orientations

- Dataset 2:

  No modifications. Plots are presented below :

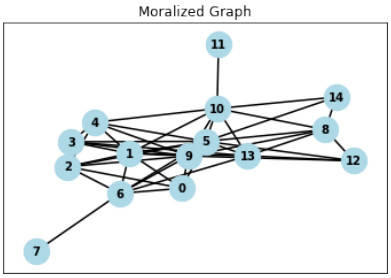


Figure 5: Moralized graph

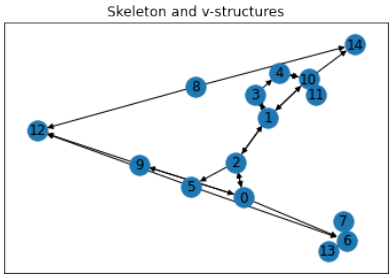


Figure 6: Skeleton+v structures

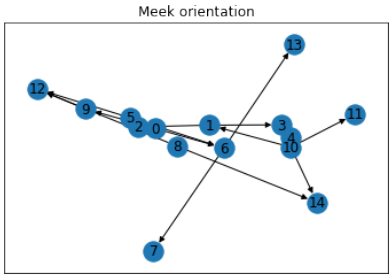


Figure 7: Meek orientations

=>Using the A2 matrix, we get a F1 score ≈ 0.76.

7 In selecting between the Grow-Shrink (GS) algorithm and the PC algorithm for causal discovery, it is crucial to consider multiple factors to ensure a comprehensive evaluation. First, the characteristics of the data should be examined, including whether it consists of continuous or mixed variable types. Additionally, the assumptions underlying each algorithm must be assessed, with GS assuming faithfulness and PC assuming a directed acyclic graph (DAG) and utilizing conditional independence tests. Scalability is another aspect to consider, as GS is known for its computational efficiency, making it suitable for large datasets, while PC's complexity grows with the number of variables. The availability of prior knowledge or domain expertise is also important, as GS can explore a wide range of potential graphs when little prior knowledge is available, whereas PC allows the incorporation of known causal relationships. Lastly, the specific goals of the causal analysis should be taken into account, as GS is focused on identifying causal relationships and discovering the underlying structure, whereas PC maximizes a specific score function. Considering these factors collectively will aid in selecting the most appropriate causal discovery method for a given scenario.

Part 2

1 We wrote Hhull such that it uses as subroutine other two functions: the first function finds the maximal c-component containing S, the second function finds all the ancestors in the subgraph that contains only nodes of the maximal c-component containing S. Hhull takes as input a graph and a set S, sets F equal to the set of nodes in the input graph and iteratively:

- constructs with the first function the maximal c-component containing S (F1),
- fills a set F2 with all the ancestors of S that are in the subgraph of F1,

- checks whether F2 equals F and if this condition is not satisfied assigns F2 to F, while if it is satisfied, it returns F.

Considering definition 4 in the text and that we started from F that contains the set of nodes of the graph in input, the function begins with a big set (that is F) that gradually decreases in cardinality to have in the end only the ancestors node of the set. In this way, the function will not treat set F smaller than the ancestors set and is able to return the maximal hedge.

2 To change both the function implemented for the exact and heuristic algorithm to make them able to handle cases where G[S] has multiple c-component we proceed in a similar way.
In the following we will call exact/heuristic algorithm 1 and exact/heuristic algorithm 2 to indicate respectively exact/heuristic algorithm that can only treat one c-component or multiple c-component.
For the exact algorithm 2, we:

- found the maximal c-component for the set of nodes in input (S),
- made a for loop in which each c-component in the maximal c-component was sent as input for exact,
- every time that the output of exact algorithm 1 is different from the empty set and from the set of nodes already found (starting from the empty set) , append the set and the cost (relative to the set) found in two different lists (A and min_cost),
- the output of the exact algorithm 2 is the set A and the min_cost.

For the heuristic algorithm 2, we:

- found the maximal c-components for the set of nodes in input (S),
- made a for loop in which each c-component in the maximal c-component was sent as input for exact/heuristic
- at the end of each loop append the cut_cost and the vertex_cut in two different lists (total_cost,vertex_cut_set),
- the output of heuristic algorithm 2 is the vertex_cut_set and the total_cost.

3 The goal of the heuristic algorithm is to provide a solution more quickly than the exact algorithm, even if it is not optimal. The main element which makes the algorithm faster is the minimum weight vertex cut problem. Because it can be reduced to a minimum-weight edge cut problem, we can reduce the complexity to polynomial instead of the worst case exponential computational complexity that we get using the exact algorithm.

We reduce the minimum-weight vertex cut problem to a minimum-weight edge cut problem the following way:

For every node n in the graph $\mathcal{H} = \text{Hhull}(\mathcal{S}, \mathcal{G})$ (as provided in the description of the heuristic algorithm), we create 2 nodes n_in and n_out, which we connect with an edge to which we give the initial weight of node n (note that nodes $\in S$ are given weight ∞). Then, considering the initial graph again, we proceed as follows:

- for every successor s of node n, a directed edge is created between n_out and s_in, with a given weight of ∞
- for every predecessor p of node n a directed edge is created between p_out and n_in, with a given weight of ∞
- we connect the source node x to all nodes in $Pa(S) \bigcap H$ and the sink node y to all nodes in $Pa(S)$, as described in the heuristic algorithm

Thus, we can solve a minimum-weight edge problem, using the Push–relabel maximum flow algorithm which has time complexity $O(V^2E)$ for E edges in the graph, V vertices. This algorithm is implemented in the function *minimum_cut* function of the NetworkX Python package. When running the function, the source node is taken to be x while the sink node is y.

Finally, we convert the the minimum-weight edge cut back to the minimum-weight vertex cut by adapting the output so that it gives the solution as a set of nodes n instead of n_in and n_out.

4 To generate random ADMGs we used the Erdos–Rényi model. There are two similar variants of this model:

- G(n,M) model: the output is a graph with n nodes and M edges that is chosen uniformly from a collection of graphs that contains all the possible graph with the same number of nodes and edges.
- G(n,p) model: n nodes are connected in the graph in output randomly, that is, every edge can be included in the graph with probability p independently from the others such that the probability of joining node j from node i is always p for every i,j.

We implemented the second version of the model above (G(n,p)), in which, as the probability p increases, going from 0 to 1, the model will more likely outputs random graph with more edges; due to this and also to the fact that we used the random graph as input for the exact algorithm, we set p=0.1. In fact, this algorithm can be difficult to run for large and dense graphs as said in the text. We set the number of nodes equal to 20 in order to not give as input too big graphs.
To test the exact and the heuristic algorithm on different random ADGM graphs, we made a for loop in which we repeated n=100 times the following steps:

1. generate a random graphs following the Erdos–Rényi model and setting p=0.1 and n=20; the model outputs a directed graph.
2. for each edge in the random graph, generate a random integer number from 0 to 10: when this number is bigger than 7 the edge will become a bidirected edge in the ADMG graph, while when is smaller or equal than 7 the edge will remain a normal directed edge in the ADMG graph; in this way 30% of the directed edges of the random graph will become bidirected edges in the ADGM graph. Iterating through the edges, if in some cases we have directed edges going from, say, $n1$ to $n2$ and vice versa, we remove the directed edges and replace them with a bidirected edge.
3. for each node, add a random integer cost between 1 and 5.
4. generate a random list of nodes: choose randomly an integer between 1 and m (to specify in the following) and take from the list of nodes of the Erdos–Rényi graph m random nodes. This will be our random set S, the input of the algorithms.
5. run the exact algorithm and keep track of the execution time and the cost of the intervention.
6. run the heuristic algorithm and keep track of the execution time and the cost of the intervention.

In the end we have n execution time and n costs stored in two lists both for the exact algorithm and the heuristic algorithm.
At first, in step 4 of the procedure, we chose m equal to 1; the results are given in figures 8 and 9.

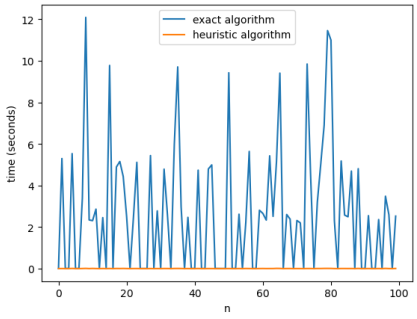


Figure 8: Comparison of execution times exact vs heuristic algorithm

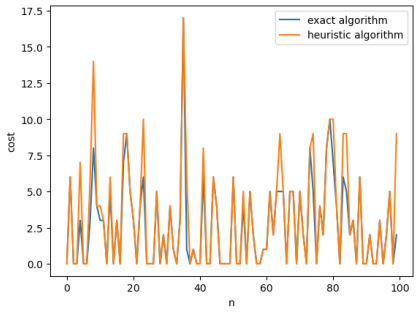


Figure 9: Comparison of intervention costs exact vs heuristic algorithm

As expected, the execution time of the exact algorithm is always bigger than the heuristic's. Looking at the graph of the costs, we can notice that most of the times the two lines completely overlap: this is a good

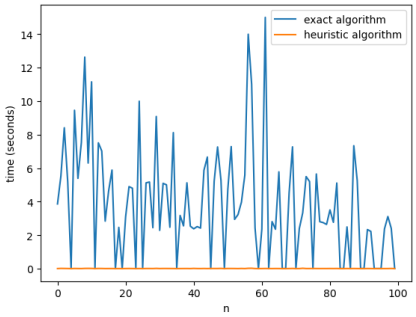


Figure 10: Comparison of execution times exact vs heuristic algorithm

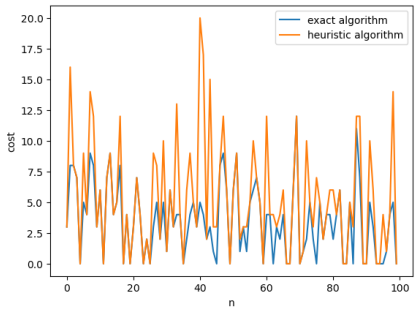


Figure 11: Comparison of intervention costs exact vs heuristic algorithm

result because it means that, approximately, the two algorithms return the same solution.
Then, in step 4 of the procedure, we chose m equal to 4 the results are given in figures 10 and 11.

When m=4 the algorithms could potentially receive in input a set of 4 random nodes between 0 to 19: since we treated quite big random graphs, in this case the 2 methods are on average slowly than when m=1 and also more "expensive". In fact from figure 11 the costs are bigger than before. Both execution time and cost depend on the random graph(number of nodes, density) and on the set S in input; we tried different combinations of the number of iterations and of the maximum number of nodes to give to set S. The results are reported on the "CodePart2" notebook.

5 Figure 12 reports the plot of a random ADMG for which the heuristic algorithm gave a sub-optimal solution: putting as input the set S = {'9','7'} the exact algorithm returns ([{'1', '13'}], 6) while the heuristic one ({'19', '1', '13'}, 10). Both the sets and the costs in output are different depending on the algorithm used and in particular the cost given by the exact algorithm is smaller than the one given by the sub-optimal algorithm (heuristic one).
In figure 13, we can observe, instead, the maximal c-component of the set S={'9','7'}.

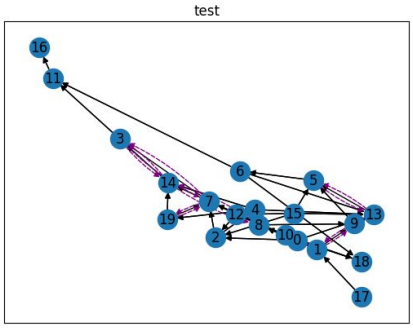


Figure 12: ADMG where the heuristic algorithm returns a sub-optimal solution

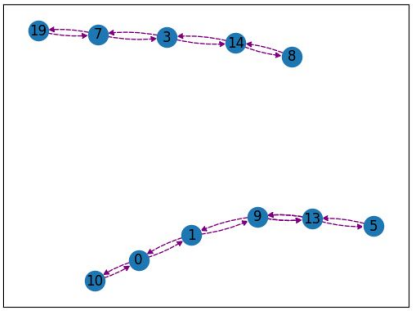


Figure 13: The maximal c-components of the set S={'9','7'}

It's difficult to understand in which way we can improve the heuristic algorithm because, depending on the graph in input (for example if it's too dense or not), the way in which the problem solved by the algorithm can be tackled in different ways. As an example, when the input of the heuristic algorithm is not a set of nodes in a c-component, it could happen that the algorithm finds an infinite capacity path; we tried to deal with his problem returning the empty set and a zero cost in such cases.