École Polytechnique Fédérale de Lausanne

Causal Inference - MGT-416

FINAL PROJECT

June 10, 2023

Pierre-Benoît Grimaux 302257
Ellie Turpin 296441
Valentin Peyron 301340

Contents

1 A Causal Discovery Algorithm

1.1 Tunning of the α parameter

The first part of this subsection is the optimization of the parameter α. To perform this optimization, a very useful function has been used, which is $nx.d\_separated(model, var1, var2, set\_vars)$. Given a graph as input, the function returns either 1 or 0 depending on whether $var1$ and $var2$ are d-separated or not with respect to S, where S is a set of variables. Using this function, the goal has been to find the parameter α such that the *ci_test* would be as close as possible to the output of the $nx.d\_separated$ function.

The strategy to find such an α parameter has been illustrated in the project through a dichotomy search. In order to perform it, a function that returns the global accuracy for every possible combination of the graph, given a specific value of α, has been constructed. This function returns a float between 0 and 1, transforming the problem into a maximization problem with respect to α.

The purpose of this report is not to explain how a dichotomy algorithm works, but in a nutshell, the dichotomy algorithm, or binary search, works by repeatedly dividing a sorted range in half to efficiently find a target value.

In the dichotomy algorithm, $\epsilon = 1.10^{-3}$ and $max\_iter = 30$. The resulting plots are presented in Annex [3.1].

The dichotomy results in the following parameter: $\boxed{\alpha \simeq 0.001}$. Another possible way to check if this parameter is coherent is to examine the evolution of the accuracy score as a function of the parameter. Again, this figure can be found in Annex [3.1].

The plot in Figure [3] supports the result obtained with the dichotomy, as it shows a maximum accuracy in the interval $[1.10^{-5}; 1.10^{-3}]$.

1.2 Markov boundary

Theoretically, the Markov boundary of a variable x is defined as the set of variables that directly depend on x when conditioning on its parents. The Markov property implies a symmetric relationship, meaning that if x is in the Markov boundary of y, then y must be in the Markov boundary of x.

However, in practice, there can be situations where the Markov boundaries are not symmetric for various reasons. One possible explanation is the failure of the faithfulness assumption, which assumes that the graphical model accurately represents the statistical interdependence among variables.

When the Markov boundaries are not symmetric, it means that there are conditional dependencies in one direction but not the other. This can occur due to unreported confounders, measurement errors, or insufficient data.

To capture the conditional interdependence between variables in such cases, the moralized graph is drawn. The moralized graph is created by adding an undirected edge between any two variables that have a common child in their Markov boundaries. This approach considers the existence of conditional dependencies and disregards the directionality of the edges.

In the project, the moralized graph was drawn by considering the Markov boundary of each vertex. If x is in the Markov boundary of y ($x \in Mb(y)$), an undirected edge was drawn between (x, y). The Markov boundary for each vertex was determined using a probabilistic approach.

As mentioned earlier, if x is in the Markov boundary of y ($x \in Mb(y)$) but y is not in the Markov boundary of x ($y \notin Mb(x)$), an edge was still drawn between (x, y).

1.3 V-Structure

This part was one of the more challenging aspects of the first problem. The strategy implemented to find each V-Structure was as follows:

Note that at this step, the input is the moralized graph obtained from the previous question. The first step was to find all the directed neighbors using the formula provided in the Markov equivalence class subsection. According to the formula, y is considered a direct neighbor of x if $x \not\perp\!\!\!\perp y | S$ for every $S \subseteq T$, where T is the smaller set in cardinality between $Mb(x) \setminus y$ and $Mb(y) \setminus x$.

It should be noted that for the first dataset, there was a problem because $x \not\perp\!\!\!\perp z | y$. Consequently, the pair (x, z), which are direct neighbors, would not be considered as direct neighbors. **However, since the original graph is known, a special condition was added in the code to obtain the actual graph.** For the second dataset, if such a problem occurs, it would not be observed since it was assumed, for this dataset, that the original graph is not known (”realistic case”). The authors of this report acknowledge that this approach may not work in real-life scenarios. However, it is assumed that the first dataset is used to develop the algorithm. Making a mistake in the second step of this algorithm would invalidate the usefulness of this dataset for the algorithm's conception, which would not make any sense.

Once all the direct neighbors were found, the goal was to identify all the candidates for a V-Structure. This was done by implementing simple conditions in an iteration through all the direct edges. The complexity of this task is $\theta(n^2)$, but it is not a significant issue since both datasets in this project are not very large. However, optimizing this function would be interesting to handle larger datasets.

Using this function, all the candidates for V-Structures were found. The task then was to determine which of them actually constituted a V-Structure. As discussed in the problem statement, the V-Structures might conflict with each other. This problem needed to be resolved to avoid having directed edges become undirected (consider the quadruplet $a - b - c - d$, where due to noise, $a \to b \leftarrow c$ could be seen as a V-Structure and $b \to c \leftarrow d$ could also be seen as a V-Structure. The algorithm would end up with $b \to c$ and $c \to b$). To avoid this issue, a very strong assumption was made. Without any knowledge of the graph or the meaning of the data, it is impossible for a human to determine the correct V-Structure. Therefore, the group decided to fix the first occurrence of a vertex in a V-Structure as the correct one. In other words, if there is a quadruplet $a - b - c - d$, and the algorithm identifies $a \to b \leftarrow c$ as a V-Structure, then $b - c - d$ cannot be a V-Structure. This is a strong assumption, as the probability of being correct in such a case is $p = \frac{1}{2}$. However, not considering the problem and adding an undirected edge would result in a probability of being correct of $p = 0$. Moreover, it would create a special structure: $a \to b - c \leftarrow d$, this would distort all subsequent steps. Ultimately, without knowledge of the data's meaning or the original graph, it is impossible to determine which of the two candidates is the correct one.

It is important to note that in theory, the skeleton V-Structure graph should contain the same number of edges as the original graph. However, due to the problem explained at the beginning of this subsection, certain direct neighbors may not be detected in the moralized graph. This results in a smaller number of edges in the skeleton V-Structure graph compared to the original graph.

Once the V-Structures are identified, the graph is reconstructed. In Python, the *networkx* library does not allow having directed and undirected edges in the same graph. Therefore, undirected edges are represented by bidirected edges, which are allowed in the library.

1.4 Order Dependent Algorithm ?

The output of the constraint-based causal discovery method implemented depends on the order in which the CI tests are performed, hence the implemented algorithm is order dependent. This dependence appears at different step of the algorithm. During the Grow phase, the order of the test condition $x \perp\!\!\!\perp y | M$ with $y \in V \setminus (M \cup \{x\})$ matters because the accuracy of the *ci_test* is not equal to one. As discussed previously, $x \not\!\perp\!\!\!\perp z | \{y\}$. In order to avoid this problem for the first datasaet, the for loop must be performed in the other way in order to test $x \perp\!\!\!\perp y | \{z\}$ before. Hence this step is order dependent. In a more general way, because the accuracy of the *ci_test* is not equal to one, everytime a for loop is performed with a condition the *ci_test* the algorithm becomes order dependent.
To make the algorithm order independent, one approach could be to perform multiple for loops, where the iteration vector is shuffled for each loop. The output after each loop is stored, and the algorithm selects the output that occurs most frequently across all the loops. This can potentially improve the results obtained. However, implementing this approach would significantly increase the complexity of the algorithm.

Please note that this specific approach has not been implemented in the provided code.

1.5 Implement Meek's orientation rules

Each of Meek's orientation rules has been implemented using two or three for loops. The complexity of each rule is then approximately $\theta(n^2)$ (R1, R2) or $\theta(n^3)$ (R3, R4). As discussed previously, due to sample noise, conflicting directions may arise. To resolve these conflicts, a "first come, first served" strategy was implemented.

The entire graph is examined to apply the first rule, R1. Then, the second rule, R2, is applied. Next, the third rule, R3, is applied, followed by the fourth rule, R4. In other words, if a direction is assigned to the edge $a - b$ using rule R2, it cannot be changed by rules R3 and R4. To ensure that no edges are overlooked, a while loop is implemented. While changes are made to the graph, the graph will be checked again.

1.6 Outcome and Discussion

All the requested figures are available in Annex [3.2]. In the case of the first dataset, the correct graphs were obtained because a condition was implemented to enforce the correctness of the V-Structure.

For the second dataset, it is noteworthy that the maximally oriented graph obtained closely resembles the original graph. However, due to noise, some edges have disappeared. The original graph had 24 edges, the moralized graph had 36 edges, and the maximally oriented graph had 21 edges.

In conclusion, the optimal method between PC and GS depends on the dataset. For a small dataset, the Greedy Search (GS) approach might be feasible. However, for a larger dataset, the PC Algorithm implemented in this project appears to be more scalable.

2 Experimental Design for Causal Effect Identification

2.1 Why H_hull returns the maximal hedge

We are considering the case where the partitioning of $\mathcal{G}_{[S]}$ to its maximal c-component comprises only one part (but not necessarily all the nodes of S). *H_Hull* returns the maximal hedge F formed for $Q[S]$ in a given graph $\mathcal{G}$. As a recall, F **forms a hedge** for $Q[S]$ if $S \subsetneq F$, F is the set of ancestors of S in $\mathcal{G}_{[F]}$ and $\mathcal{G}_{[F]}$ is a c-component. F is a maximal hedge for $Q[S]$ is it is the smallest possible set which is a hedge for $Q[S]$. We look at the algorithm *H_Hull*. The function is an iterative loop. F is initialized to be the set of all vertices of $\mathcal{G}$. It is progressively tightened until the smallest set that meets the hedge conditions. First iteration : $F1$ is the maximal c-component in $\mathcal{G}_{[F]}$ (full graph) that contains S. So $F \subseteq F1$. The second condition to meet is that F is the set of ancestors of S in $\mathcal{G}_{[F]}$. As $F \subseteq F1$ we set F2 as the set of ancestors of S in $\mathcal{G}_{[F1]}$. So $F \subseteq F2$. $F2$ is the second biggest candidate for F (the first initialization of F being the first one). If $F2 = F$, F is the smallest possible hedge for $Q[S]$ and the function returns F. If $F2 \neq F$, F is not the smallest possible hedge for $Q[S]$, $F2$ is a smaller candidate and F is assigned to be $F2$. The the loop iterate for F being $F2$, until no smaller hedge set is found. We implemented our *H_Hull* function exactly as in the algorithm provided n the paper. We added a condition to check that S is in the graph $\mathcal{G}$. If it is not the case, the execution stops.

2.2 Heuristic algorithm: Minimum weight vertex cut

In order to solve the minimum-weight vertex cut problem for the heuristic algorithm, we transformed the problem into a minimum-weight edge-cut problem. Thus allowing us to use a mas-flow/min-cut Algorithm. More precisely, from the nodes found using Hhull, we created the undirected graph $\mathcal{H}$ using the condition given in the algorithm. Then from this graph, we created a weighted graph. The weighted graph was constructed based on the following rules: First, for each node neither in S nor in source or target (basically x et y) we created a node *in* and a node *out*, so we doubled them. Then the weight of each element is assigned to the edges from node *in* to node *out*. Finally, the edges from one node to another one which is not is double (ex: t to v and not t*in* to t*out*) the weight associated with the edge is infinite. Thus using a min-cut algorithm on this weighted graph, we can obtain the min-cut. And Following the papers from Sina Akbari, will give us the nodes we have to interact on in order to have an identifiable probability.

2.3 Comparison of Algo.1 versus Heuristic

In the following, we are interested in comparing the exact algorithm (Alg. 1) to the heuristic algorithm in terms of run time and optimality of output. To do so, we generated random ADMGs over a range of graph sizes from 2 to 30 nodes (number without the source and the target). More precisely, the ADMGs were represented as 2 graphs one directed and one undirected. Those were created randomly, using a probability of 0.5 for two vertices to have an edge. The cost where also created randomly with costs between 1 and 50 for each node not in S. Finally, we considered two case for S one with only one node and the other one with a few. The Nodes in S were chosen randomly. Finally, we saw that if the random graph where such that pa(S)∩H $= \emptyset$, with $pa(S)$ the parents of S and H the set of nodes returned by Hhull, there was no solution. Indeed, in this case x was

not connected to the rest of the graph, thus the graph is not of interest. Thus to encounter this we verified that pa(S)∩H != ∅ before running min-cut.
To compare the two algorithms, we first compare the run time of these two algorithms, the result are presented in Fig.6 and Fig.7. The result is clearly showing that Algo_1 is less effective than heuristic. In Fig.6, it even seems as if heuristic was immediate because the difference in run time is so great. And the more the nodes the worst the run time for Algo_1, of course, it depends on the graph created, but we can clearly see the run time growing. Secondly, we can look at the optimality of output to compare both algorithm. The Fig.8 shows the number of different nodes in the cut returned by both Algorithm. As Algo_1 is the exact solution, this allows us to see how wrong heuristic algo is. As for the run time, the error on heuristic is growing as the number of nodes grows. But for 28 nodes for example, we can see that the solution is exact. So This informs us that heuristic is not exact, but it gives us a good approximation, and as it is much faster than Algo_1 this allows to get an approximation for a huge number of nodes or a lot of connections.

We also include a graph that gives us a wrong cut using heuristic, the graph is presented in fig.9. The result given by the algorithm are Algo_1: {1} and heuristic: {0,1}.

PS: Note that for the Heuristic algorithm, there is 2 functions you can use. There is heuristic_plus(n, type) and heuristic_algo(G_U,G_B,S,C). Where n is the number of nodes, type is either if S has one (type = 0) or more nodes, G_U the directed graph, G_B the bidirected graph, C the costs/weights of nodes not in S, and S. The first function runs heuristic on a random ADMs of size n such that min_cut does not give a nx.NetworkXUnbounded and pa(S)∩H != ∅. So the function keep creating random ADMs if the condition are not met. The second function is in itself the heuristic algo to whom you give an ADMs, costs and S. But it will return an empty set() if the ADMs is not well defined. If the ADMs is such that he gives one of the two cases described above.

2.4 Modification for multiple c-component

If $\mathcal{G}_{[S]}$ consists of several maximal c-components, we have to treat each c-component separately. Let's say that $S_1, ..., S_k$ is the partitioning of $\mathcal{G}_{[S]}$ into its maximal c-components. We have to solve the minimum cost intervention problem for every S_i, $i = 1, 2, ..., k$. We created the function *find_subsets_S_cc* to compute $S_1, ..., S_k$ and put it in an array of sets. We consider the Algorithm given on the paper sheet. We initialize an empty array of set, *min_cost*, that is going to store the minimum cost intervention set for every S_i. We iterate lines 3 to 17 for every S_i. We replace the "return ∅" and "return A" by "*min_cost.append(∅) ; break*" and "*min_cost.append(A) ; break*" respectively.

We also created the function *set_of_sets_WMHS* to create the set to input in line 14 of the Algorithm, as described in the Lemma 1. As a reminder, $\mathbf{F}_i = \{F_{i,1}, \ldots, F_{i,m_i}\}$ is the set of all hedges formed for $Q[S_i]$ in $\mathcal{G}$, for every $1 \leq i \leq k$. Then we have to solve the MWHS problem for the sets :

$$\{(F_{1,1} \backslash S_1), \ldots, (F_{1,m_1} \backslash S_1), (F_{2,1} \backslash S_2), \ldots, (F_{k,m_k} \backslash S_k)\}$$

Given $\mathbf{F}$ and S_i, *set_of_sets_WMHS* returns this array of sets. At the end, we transform *min_cost* in a set to have the intervention set with minimum cost.

3 Annexe

3.1 Figures for the tunning of α

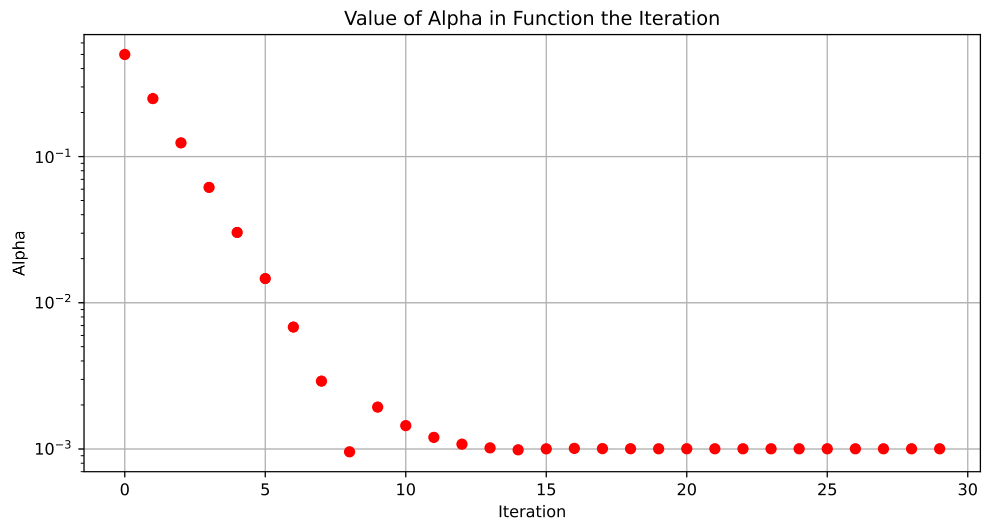

Figure 1: Tracking the Variation of Alpha with Iterations

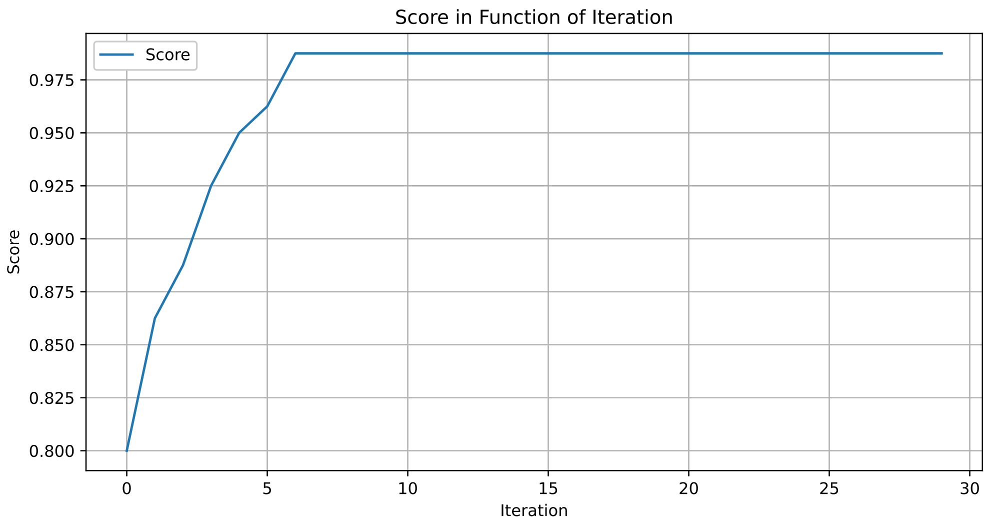

Figure 2: Analyzing the Accuracy Score's Progression with Iterations: Well-classified Combinations Over Total Combinations

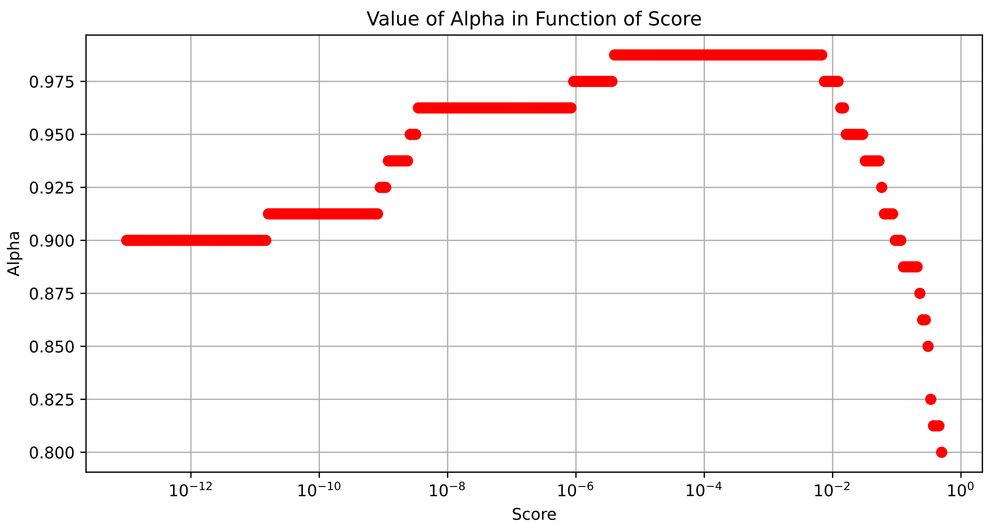

Figure 3: Examining the Relationship between Accuracy Score and Alpha: Variation of Accuracy with Different Alpha Values

3.2 Outcome of the Algorithm partie 1

3.2.1 First Dataset

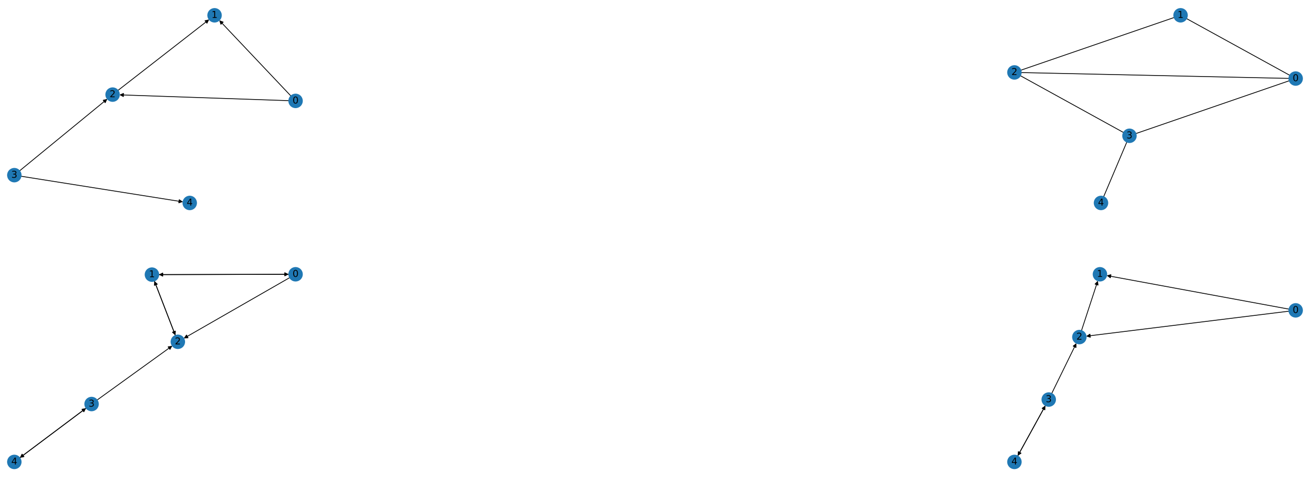

Figure 4: Evolution of the Graph for the First Dataset In the upper left, the original graph is displayed. The upper right shows the moralized graph. The lower left presents the V-Structure Skeleton graph, while the lower right exhibits the maximally oriented graph. Please note that in the two lower graphs, undirected edges are represented by bidirectional edges.

3.2.2 Second Dataset

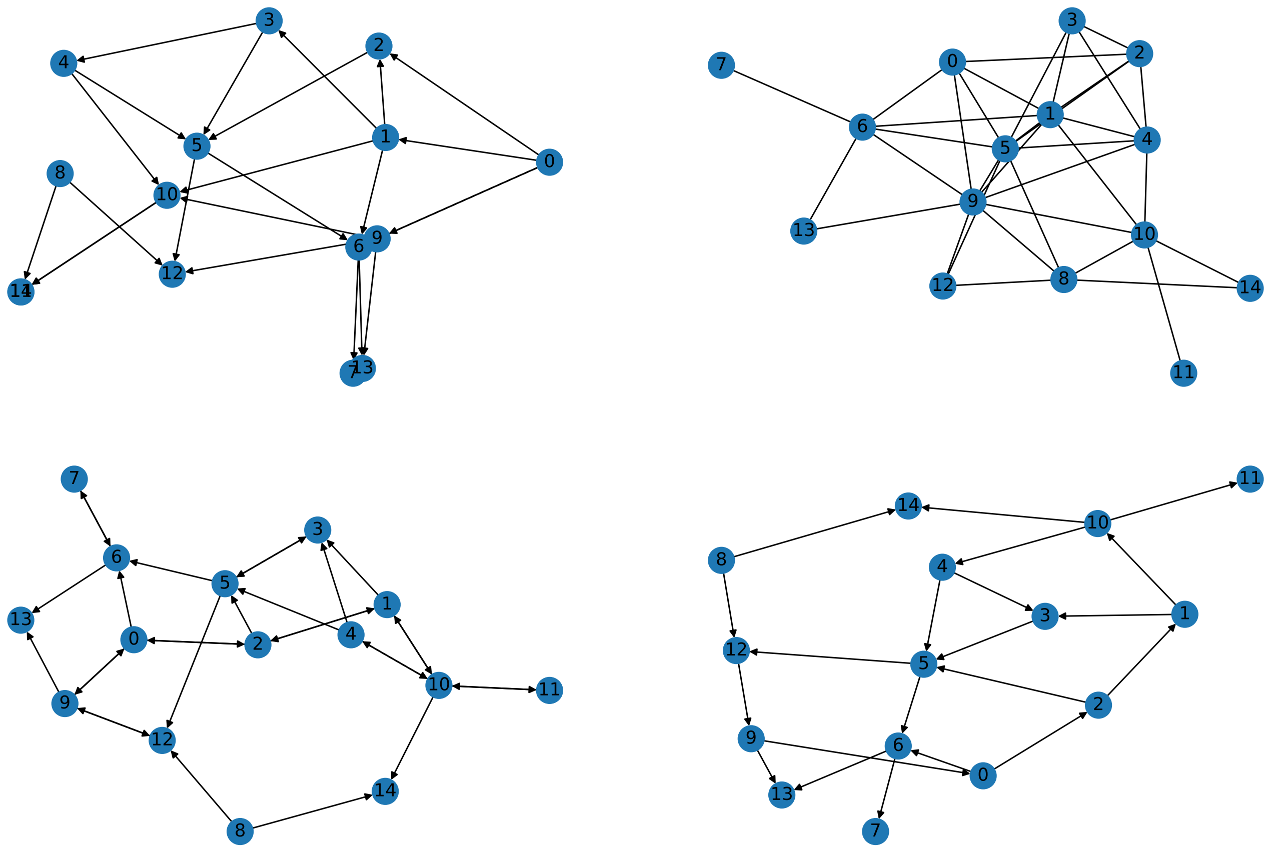

Figure 5: Evolution of the Graph for the Second Dataset In the upper left, the original graph is displayed. The upper right shows the moralized graph. The lower left presents the V-Structure Skeleton graph, while the lower right exhibits the maximally oriented graph. Please note that in the two lower graphs, undirected edges are represented by bidirectional edges.

3.3 Part 2: Comparison of the algorithm and sub-optimal solution for heuristic

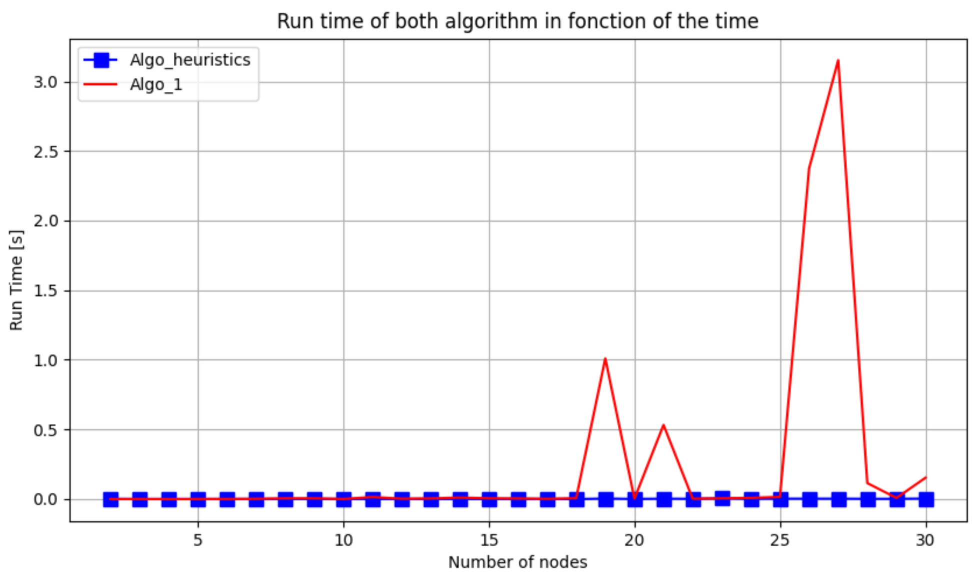

Figure 6: Comparison of the run time of both algorithm, heuristic and Algo-1, for different size of graph (from 2-30). This number of nodes does not count the source and the target.

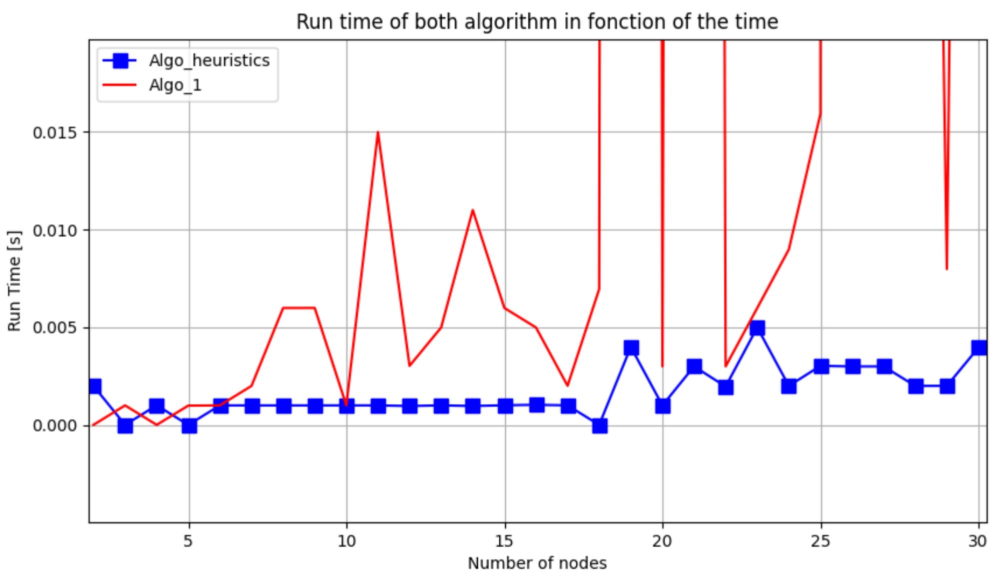

Figure 7: Zoom on the Figure 6 allowing to see the run time of heuristic, which is much smaller than Algo-1 .

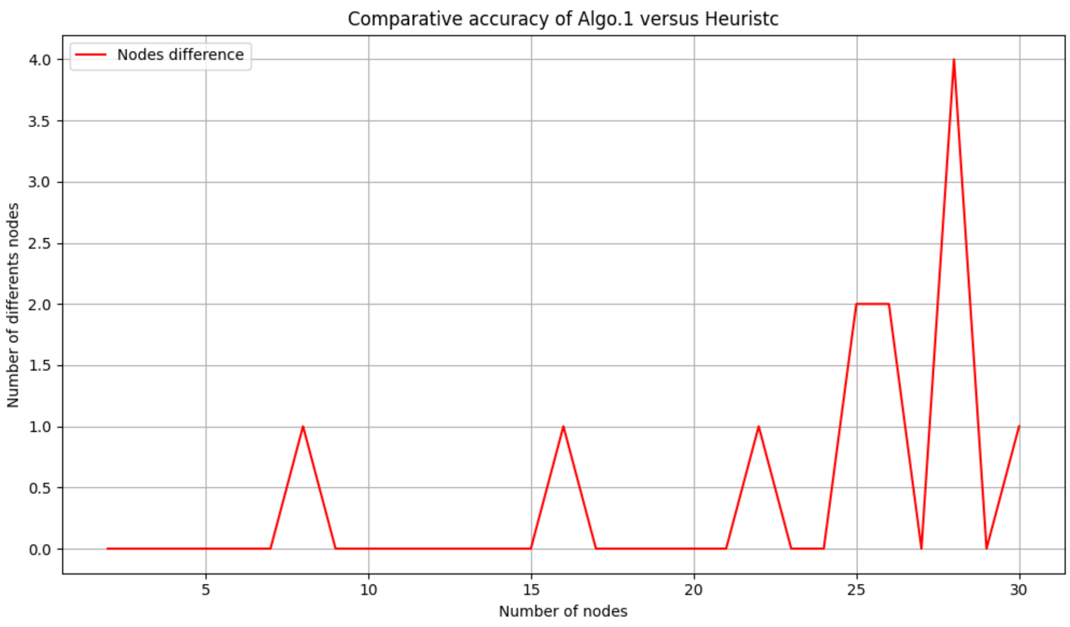

Figure 8: Accuracy of the heuristic algorithm in comparison with the exact solution (algo-1).

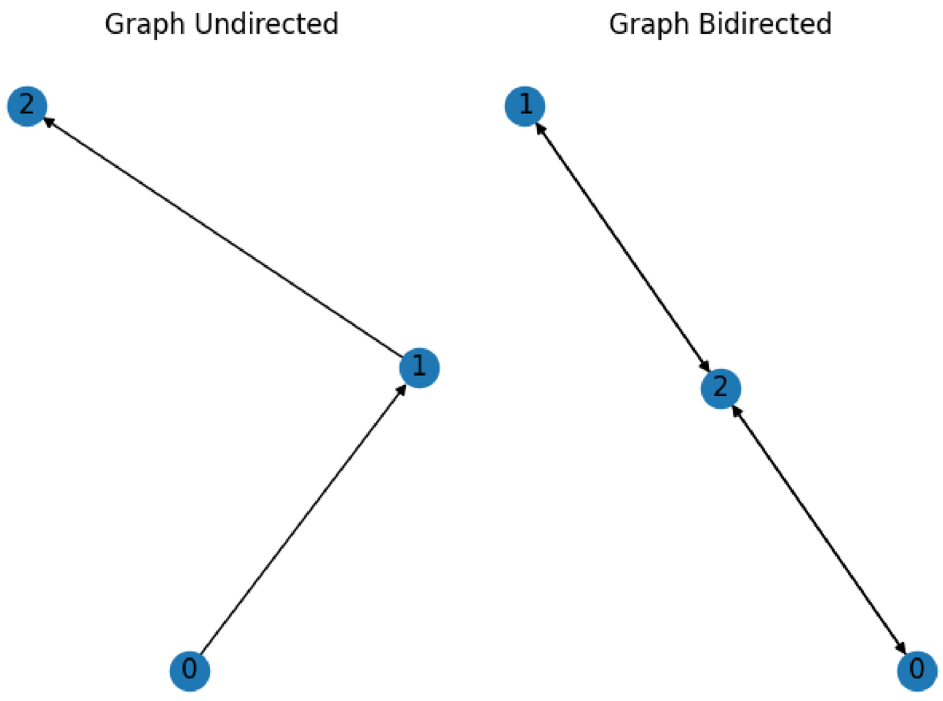

Figure 9: Graph which gives us a sub-optimal solution with the heuristic algorithm for 3 nodes. The results are: Algo_1: {1} and heuristic: {0,1}