Implementation of K2 Procedure to find the most probable belief-network structures

Alessio Bocini

Abstract

This document presents an implementation of the K2 procedure for learning Bayesian network structures, as required for the Artificial Intelligence course. The algorithm follows the assumptions and structure proposed by Cooper and Herskovits, including the original scoring function. The implementation is demonstrated using a Python-based approach and evaluated on a sample dataset.

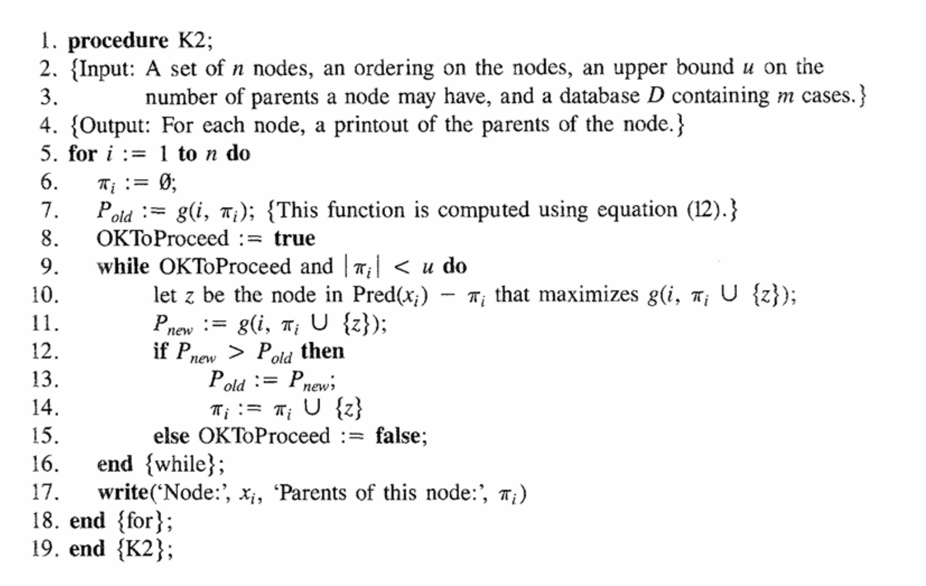
Table of Contents

1 Introduction

The K2 procedure is an heuristic-search method, proposed by Cooper and Herskovits [1], used in Bayesian network structure learning to determine the most likely network from a dataset of variables. It applies a greedy search approach to build the structure by adding parent nodes to each variable, optimizing based on a scoring function.
The algorithm evaluates possible parent sets for each variable, incrementally

adding nodes that maximize the likelihood until no further improvement is possible. It simplifies the complex task of network learning by focusing on maximizing the posterior probability of the structure given the data, while adhering to a predefined node ordering (in particular topological ordering).

All the assumptions from their method proposed were actually followed in this implementation.
The pseudocode from the paper was provided as following:



```
procedure K2;
{Input: A set of $n$ nodes, an ordering on the nodes, an upper bound $u$ on the
        number of parents a node may have, and a database $D$ containing $m$ cases.}
{Output: For each node, a printout of the parents of the node.}
for $i$ := 1 to $n$ do
    $\pi_i := \emptyset$;
    $P_{old} := g(i, \pi_i)$; {This function is computed using equation (12).}
    OKToProceed := true
    while OKToProceed and $|\pi_i| < u$ do
        let $z$ be the node in $\text{Pred}(x_i) - \pi_i$ that maximizes $g(i, \pi_i \cup \{z\})$;
        $P_{new} := g(i, \pi_i \cup \{z\})$;
        if $P_{new} > P_{old}$ then
            $P_{old} := P_{new}$;
            $\pi_i := \pi_i \cup \{z\}$
        else OKToProceed := false;
    end {while};
    write('Node:', $x_i$, 'Parents of this node:', $\pi_i$)
end {for};
end {K2};
```

Figure 1: K2 Pseudocode

Next, the scoring function:

$$g(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}!,$$

Figure 2: Cooper-Herskovits Score in the Paper

Also, in order to easily understand the protagonists of these previous pseudocodes, it was presented this Theorem from the same paper[1]:

Theorem 1. *Let Z be a set of n discrete variables, where a variable xi in Z*

has r_i possible value assignments: $(v_{i1}, ., v_{ir_i})$. Let D be a database of m cases, where each case contains a value assignment for each variable in Z.
Let Bs denote a belief-network structure containing just the variables in Z.
Each variable x_i in Bs has a set of parents, which we represent with a list of variables π_i, Let W_{ij} denote the jth unique instantiation of$\pi_i$relative to D. Suppose there are q_i such unique instantiations of π_i. Define N_{ijk} to be the number of cases in D in which variable x_i has the value v_i and π_i is instantiated as W_{ij}.
Let $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$
Given the definitions above, the joint probability of a belief network structure B_s and a dataset D is proportional to:

$$P(B_s, D) = P(B_s) \prod_{i=1}^{n} \prod_{j=1}^{q_i} \left[\frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}! \right]$$

2 Implementation

In order to implement the K2 procedure as faithfully as possible to the original algorithm proposed by Cooper and Herskovits, it was necessary to analyze and develop each core function individually:

1. A function to collect all unique instantiations of parent sets, used to compute q_i and the set W_{ij}.
2. A function to retrieve the j-th unique instantiation W_{ij}.
3. A function to compute Pred(x_i), which returns the set of variables that precede x_i in the predefined node ordering [1].
4. A function to compute N_{ijk}, the number of cases in which $x_i = v_{ik}$ and its parents are instantiated as W_{ij}.
5. A function to compute $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$.
6. The scoring function $g(i, \pi_i)$ used to evaluate parent sets.
7. Finally, the implementation of the K2 algorithm itself.

The K2 procedure was implemented using `pandas.DataFrame` structures for efficient manipulation and analysis of tabular data.

A `DataFrame` is a two-dimensional, size-mutable, and heterogeneous data structure with labeled axes (rows and columns), commonly used in data analysis and manipulation tasks in Python.
It is important to note that in this implementation, the database D is represented by a DataFrame named `data`, and the collection of discrete variables Z is implicitly defined as the set of columns in this DataFrame.

2.1 unique_instantiations

This function finds the unique instantiations (i.e., combinations of actual values) of the parent nodes π_i for a given node x_i, based on the dataset. If no parents are defined for the node, the function simply returns an empty collection.

The implementation leverages the pandas DataFrame by selecting the columns corresponding to the parents, sorting them to preserve the node ordering, and removing duplicate rows to obtain the unique value combinations, which correspond to the set W_{ij} with size q_i.

The logic reflects the intent of the original algorithm without unnecessary complexity, and its implementation can be found in the provided source code.

2.2 j_th_unique_instantiation (W_{ij})

This function retrieves the j-th unique instantiation from the set of combinations of parent values for node x_i.
It builds on the output of *unique_instantiations*, returning the instantiation at the specified index j. It is conceptually important to connect the implementation to the formal notation W_{ij} as defined in the original K2 algorithm.

2.3 Pred(x_i)

This function returns the set of nodes that precede x_i in the given node ordering. In the context of the K2 algorithm, this is essentialo respect the topological constraints imposed on parent s telection. It ensures that only earlier nodes in the ordering can be considered as potential parents of x_i.

The implementation is straightforward and operates by slicing the list of nodes up to the position of x_i.

2.4 Nijk(x_i, j, k, π_i)

This function computes N_{ijk}, which is defined as the number of cases in the dataset where variable x_i takes its k-th value, and its parent set π_i is instantiated as the j-th unique combination W_{ij}.

The implementation works by first retrieving the j-th instantiation using the `j_th_unique_instantiation` function.
Then, it filters the dataset to select rows where the parent variables match this instantiation, and finally counts how many of those rows have $x_i = v_{ik}$.

This count is essential for computing the Bayesian score $g(i, \pi_i)$.

2.5 Nij(x_i, π_i, j)

This function computes N_{ij}, defined as the total number of cases in the dataset where the parents of node x_i are instantiated as the j-th unique combination W_{ij}, regardless of the value of x_i.

In practice, the function loops through all possible values v_{ik} of x_i, and for each, it computes N_{ijk} and accumulates the result.

This total count is used as the denominator in the scoring function $g(i, \pi_i)$.

2.6 cooper_herskovits_score(x_i, π_i) ($g(x_i, \pi_i)$)

This function computes the Cooper-Herskovits score $g(x_i, \pi_i)$ for a node x_i and its current set of parent nodes π_i. The score is a key component of the K2 algorithm, used to evaluate how well a given parent configuration explains the data, based on Bayesian principles.

The formula from the original paper is shown below:

$$g(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}!,$$

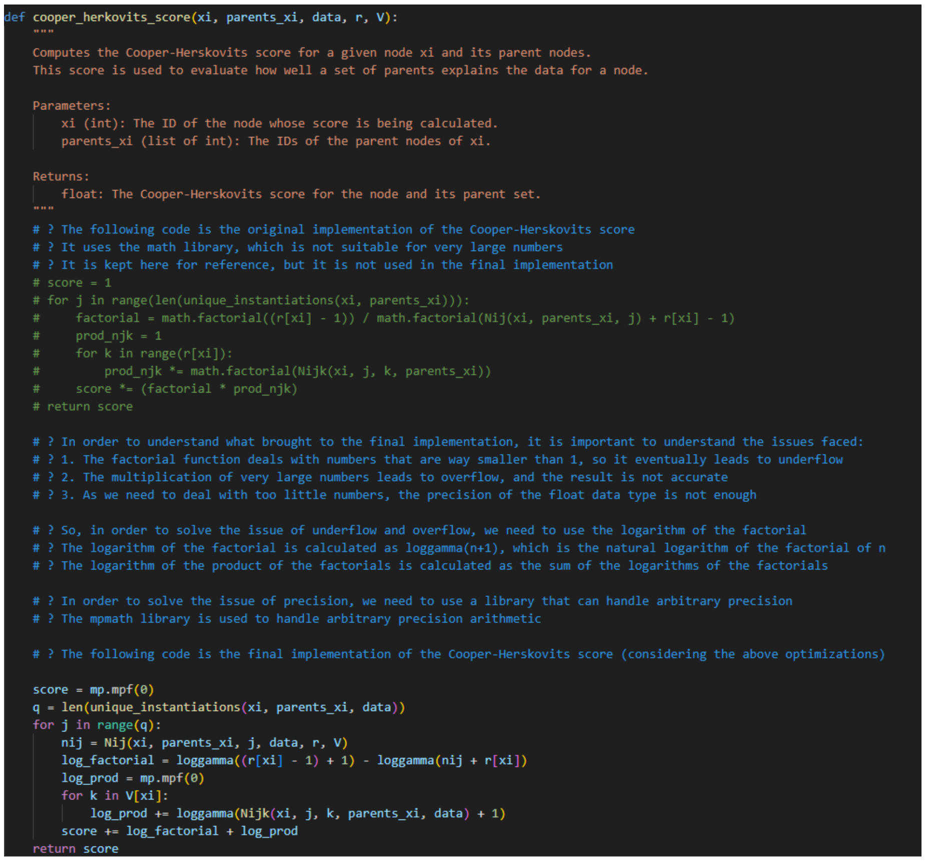
Figure 3: Cooper-Herskovits score formula from the original paper

The implementation iterates over each unique parent instantiation W_{ij} and over all possible values v_{ik} of x_i, using factorials to compute the product terms as described in the formula. The quantities N_{ijk} and N_{ij} are used to count matching cases from the dataset.

The core logic is summarized in the pseudocode below:

Algorithm 1 cooper_herskovits_score

```
1: function COOPER_HERSKOVITS_SCORE($x_i, \pi_i$)
2:     $g \leftarrow 1$
3:     $q_i \leftarrow$ number of unique instantiations of $\pi_i$
4:     $r_i \leftarrow$ number of possible values of $x_i$
5:     for $j = 1$ to $q_i$ do
6:         $N_{ij} \leftarrow \mathrm{NIJ}(x_i, \pi_i, j)$
7:         $term \leftarrow \frac{(r_i-1)!}{(N_{ij}+r_i-1)!}$
8:         for $k = 1$ to $r_i$ do
9:             $N_{ijk} \leftarrow \mathrm{NIJK}(x_i, \pi_i, j, k)$
10:            $term \leftarrow term \cdot N_{ijk}!$
11:        end for
12:        $g \leftarrow g \cdot term$
13:    end for
14:    return $g$
15: end function
```



```python
def cooper_herkovits_score(xi, parents_xi, data, r, V):
    """
    Computes the Cooper-Herskovits score for a given node xi and its parent nodes.
    This score is used to evaluate how well a set of parents explains the data for a node.

    Parameters:
        xi (int): The ID of the node whose score is being calculated.
        parents_xi (list of int): The IDs of the parent nodes of xi.

    Returns:
        float: The Cooper-Herskovits score for the node and its parent set.
    """
    # ? The following code is the original implementation of the Cooper-Herskovits score
    # ? It uses the math library, which is not suitable for very large numbers
    # ? It is kept here for reference, but it is not used in the final implementation
    # score = 1
    # for j in range(len(unique_instantiations(xi, parents_xi))):
    #     factorial = math.factorial((r[xi] - 1)) / math.factorial(Nij(xi, parents_xi, j) + r[xi] - 1)
    #     prod_njk = 1
    #     for k in range(r[xi]):
    #         prod_njk *= math.factorial(Nijk(xi, j, k, parents_xi))
    #     score *= (factorial * prod_njk)
    # return score

    # ? In order to understand what brought to the final implementation, it is important to understand the issues faced:
    # ? 1. The factorial function deals with numbers that are way smaller than 1, so it eventually leads to underflow
    # ? 2. The multiplication of very large numbers leads to overflow, and the result is not accurate
    # ? 3. As we need to deal with too little numbers, the precision of the float data type is not enough

    # ? So, in order to solve the issue of underflow and overflow, we need to use the logarithm of the factorial
    # ? The logarithm of the factorial is calculated as loggamma(n+1), which is the natural logarithm of the factorial of n
    # ? The logarithm of the product of the factorials is calculated as the sum of the logarithms of the factorials

    # ? In order to solve the issue of precision, we need to use a library that can handle arbitrary precision
    # ? The mpmath library is used to handle arbitrary precision arithmetic

    # ? The following code is the final implementation of the Cooper-Herskovits score (considering the above optimizations)

    score = mp.mpf(0)
    q = len(unique_instantiations(xi, parents_xi, data))
    for j in range(q):
        nij = Nij(xi, parents_xi, j, data, r, V)
        log_factorial = loggamma((r[xi] - 1) + 1) - loggamma(nij + r[xi])
        log_prod = mp.mpf(0)
        for k in V[xi]:
            log_prod += loggamma(Nijk(xi, j, k, parents_xi, data) + 1)
        score += log_factorial + log_prod
    return score
```

Figure 4: Code snippet of cooper_herskovits_score implementation

Motivation for using the `loggamma` Function

When computing the Cooper-Herskovits score for a node X_i, the classical formula involves products and factorials of potentially very large numbers:

$$\text{score}_i = \prod_{j=1}^{q_i} \left[\frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}! \right]$$

These factorials grow extremely fast, infact for moderate values such as $n > 200$, directly computing $n!$ can cause *overflow*.
On the other hand, ratios involving large factorials in the denominator can lead to *underflow*.

To avoid these numerical issues, the formula was transformed into the logarithmic domain.

Using the identity:

$$\log(n!) = \log(\Gamma(n+1))$$

the score can be rewritten as:

$$\log(\text{score}_i) = \sum_{j=1}^{q_i} \left[\log\Gamma(r_i) - \log\Gamma(N_{ij} + r_i) + \sum_{k=1}^{r_i} \log\Gamma(N_{ijk} + 1) \right]$$

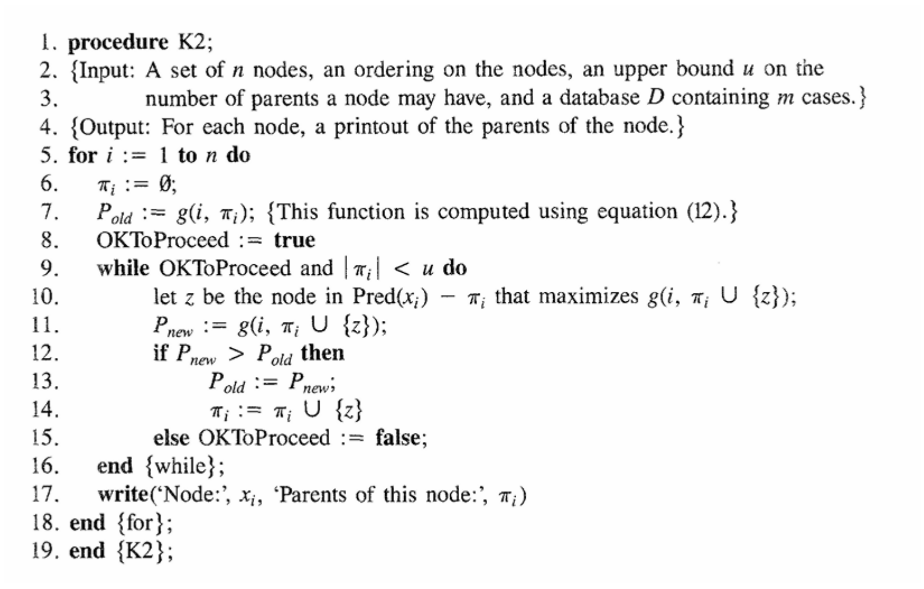
This formulation offers two major advantages:

- **Large numbers become manageable**: products become sums, and factorials become logarithms. This prevents both overflow and underflow even with large datasets.
- **Improved precision**: by using the `mpmath` library, we can compute with arbitrary precision, avoiding critical rounding errors that might impact model comparisons.

Then, `loggamma` enables the exact same theoretical computation, but in a numerically stable, precise, and robust way.

2.7 K2 Procedure

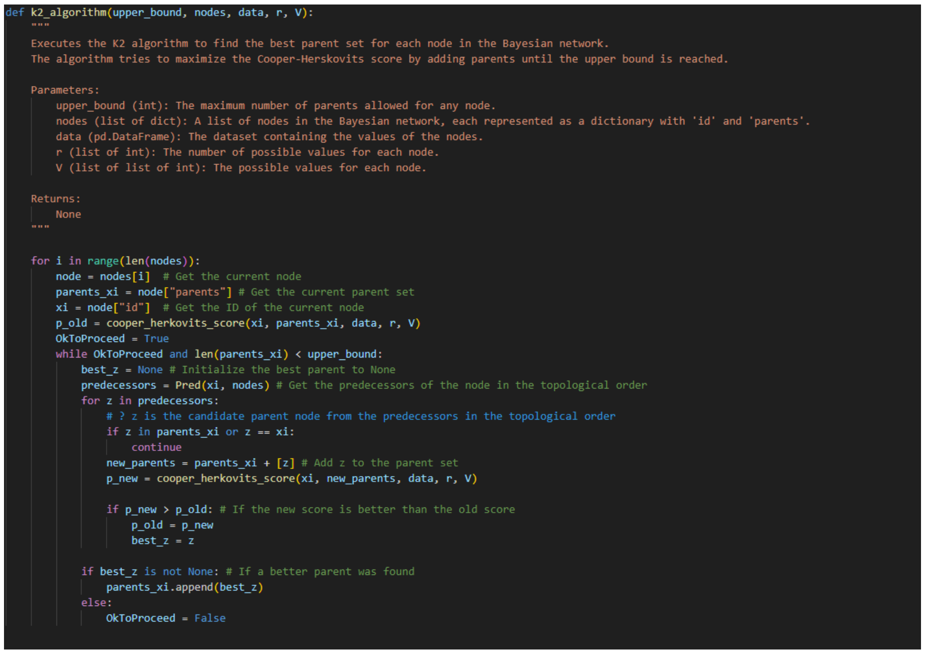
The K2 algorithm is the core procedure used to construct the Bayesian network structure. It performs a greedy search over possible parent sets for each node x_i, maximizing the score $g(x_i, \pi_i)$ under the constraint of a predefined node ordering.
Below is the original pseudocode from the paper:



```
procedure K2;
{Input: A set of $n$ nodes, an ordering on the nodes, an upper bound $u$ on the
        number of parents a node may have, and a database $D$ containing $m$ cases.}
{Output: For each node, a printout of the parents of the node.}
for $i$ := 1 to $n$ do
    $\pi_i := \emptyset$;
    $P_{old} := g(i, \pi_i)$; {This function is computed using equation (12).}
    OKToProceed := true
    while OKToProceed and $|\pi_i| < u$ do
        let $z$ be the node in $\mathrm{Pred}(x_i) - \pi_i$ that maximizes $g(i, \pi_i \cup \{z\})$;
        $P_{new} := g(i, \pi_i \cup \{z\})$;
        if $P_{new} > P_{old}$ then
            $P_{old} := P_{new}$;
            $\pi_i := \pi_i \cup \{z\}$
        else OKToProceed := false;
    end {while};
    write('Node:', $x_i$, 'Parents of this node:', $\pi_i$)
end {for};
end {K2};
```

Figure 5: Original K2 pseudocode from the Cooper-Herskovits paper

The implementation follows this logic, using Pandas data structures and looping through the variables in the specified order. Parent sets are incrementally expanded if doing so increases the score.



```python
def k2_algorithm(upper_bound, nodes, data, r, V):
    """
    Executes the K2 algorithm to find the best parent set for each node in the Bayesian network.
    The algorithm tries to maximize the Cooper-Herskovits score by adding parents until the upper bound is reached.

    Parameters:
        upper_bound (int): The maximum number of parents allowed for any node.
        nodes (list of dict): A list of nodes in the Bayesian network, each represented as a dictionary with 'id' and 'parents'.
        data (pd.DataFrame): The dataset containing the values of the nodes.
        r (list of int): The number of possible values for each node.
        V (list of list of int): The possible values for each node.

    Returns:
        None
    """

    for i in range(len(nodes)):
        node = nodes[i]  # Get the current node
        parents_xi = node["parents"] # Get the current parent set
        xi = node["id"]  # Get the ID of the current node
        p_old = cooper_herkovits_score(xi, parents_xi, data, r, V)
        OkToProceed = True
        while OkToProceed and len(parents_xi) < upper_bound:
            best_z = None # Initialize the best parent to None
            predecessors = Pred(xi, nodes) # Get the predecessors of the node in the topological order
            for z in predecessors:
                # ? z is the candidate parent node from the predecessors in the topological order
                if z in parents_xi or z == xi:
                    continue
                new_parents = parents_xi + [z] # Add z to the parent set
                p_new = cooper_herkovits_score(xi, new_parents, data, r, V)

                if p_new > p_old: # If the new score is better than the old score
                    p_old = p_new
                    best_z = z

            if best_z is not None: # If a better parent was found
                parents_xi.append(best_z)
            else:
                OkToProceed = False
```

Figure 6: Snippet of K2 implementation

3 Functional and Unit Testing

To ensure the correctness of the K2 algorithm's core components, a structured set of unit tests was implemented using Python's `unittest` framework. The testing focused on the foundational functions involved in computing the score and managing parent configurations.

Tested Functions

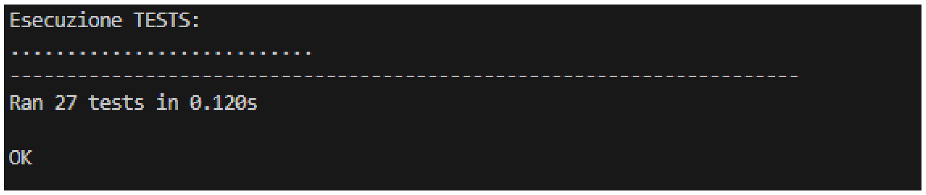
The following key functions were tested extensively:

- `unique_instantiations(xi, parents_xi, data)`
  Returns all unique value combinations for a given set of parent nodes. Tested under cases of no parents, single/multiple parents, duplicated rows, and uniform columns.

- `Nijk(xi, j, k, parents_xi, data)`
  Counts how many times node X_i takes value v_k while its parents match the j-th instantiation. Tested across all combinations of parents and node values.

- `Nij(xi, parents_xi, j, data, r, V)`
  Computes marginal counts across the different parent configurations. Tests covered cases with increasing parent set size and rare value instantiations.

- `Pred(xi, nodes)`
  Returns all nodes that precede X_i in the ordering. This function was validated for boundary and random cases to ensure consistency with the ordering constraints.
- Consistency checks for the value sets V and the corresponding cardinality vector r.



```
Esecuzione TESTS:
...........................
----------------------------------------------------------------------
Ran 27 tests in 0.120s

OK
```

Figure 7: Result unit tests

Omitted Tests

Two important functions were not unit tested directly:

- `cooper_herkovits_score(xi, parents_xi)`
  This function was not tested in isolation due to the difficulty in manually predicting the exact output values, especially for non-trivial configurations. Instead, correctness was verified indirectly through the convergence behavior observed in large-scale experiments on benchmark networks.
- `k2_procedure(ordering)`
  Testing the full K2 procedure via unit test would have been uninformative: This function was assessed qualitatively through its ability to reconstruct known structures under increasing data sizes.

4 Findings: Empirical Behavior of the K2 Procedure

To evaluate the empirical behavior of the K2 algorithm, it was tested the structure learning procedure on two benchmark Bayesian networks: the **Child** network and the **Water** network.
For each, the goal was to assess how the inferred structure differs from the expected ground-truth structure as a function of the number of samples.

Child Network

It was evaluated the Child network using sample sizes of 1000, 5000, 50000, and 100000.

For each configuration, we computed the absolute structural difference between the learned and expected DAGs and recorded the frequency of each difference level.
These frequencies were normalized to yield relative ratios.

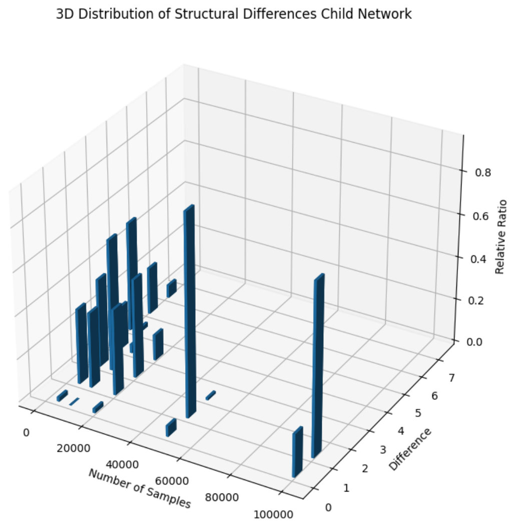


Figure 8: ***Child network*** *- Distribution of structural differences by sample size.*

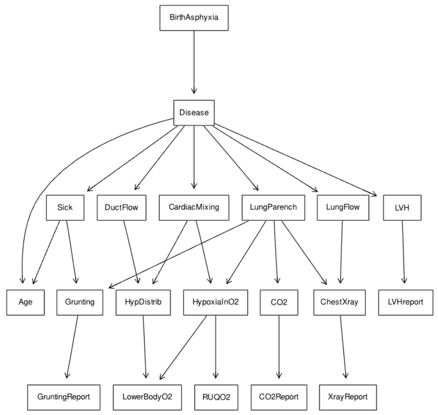


Figure 9: ***Child network*** *- Expected DAG structure used as ground truth.*

As illustrated in *Figure 8*, with smaller sample sizes, the structural differences are more widely distributed, peaking at a difference of 4. As the sample size in-

creases to 5000, the distribution becomes more concentrated around differences of 1 and 2. At 50000 and 100000 samples, the learned structures almost always differ by at most one edge, with some peak at difference zero.
This confirms the expected convergence of the K2 procedure to the expected network structure, displayed in *Figure 9*, given a sufficiently great sample size as data.

Water Network

A similar experiment was conducted on the Water network, using a broader range of sample sizes: 1000, 5000, 10000, 20000, 50000, 100000, and 500000.

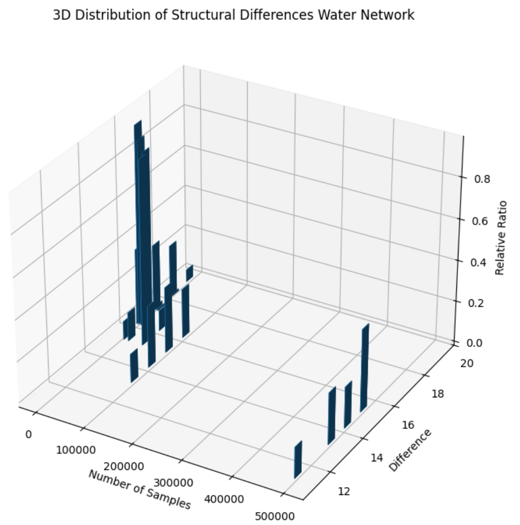


Figure 10: ***Water network*** *- Distribution of structural differences by sample size.*

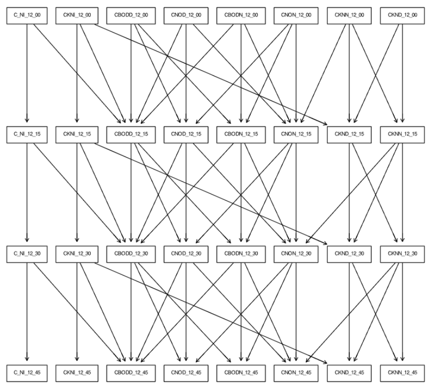


Figure 11: ***Water network*** *- Expected DAG structure used as ground truth.*

Also in this case, as the sample size increases, the distribution of differences tightens, though more gradually than in the Child network. At lower sample sizes, differences between 16 and 19 are common.
As the sample size increases to 100000 and beyond, the distribution shifts toward lower differences (11 to 15), indicating convergence but at a slower rate. This slower convergence may be due to the increased complexity of the Water network or suboptimal node ordering.

Although it was not possible to reach zero structural difference for the Water network within the tested sample sizes, the trend clearly suggests that, asymptotically, the learned structure will converge to the expected one as more data becomes available.

Interpretation

The results we observed align well with what the theory promises: if the node ordering is correct, the conditional dependencies are appropriately defined for the use case and we have enough data, the K2 algorithm does a great job at getting close to the true structure of the network. This was particularly evident in the Child network, where increasing the number of samples led to near-perfect reconstruction.

One key detail that helped throughout these experiments was the use of the `loggamma`-based scoring function.
It made the calculations faster and much more stable, especially with larger datasets where traditional factorials would have caused serious numerical problems.

Overall, while there's always room for improvement, I'm confident in both the implementation and the algorithm itself.

Version History

- **April 2024** – Initial draft of the K2 implementation and basic experiments.
- **September 2024** – Refactored code and introduced unit tests to validate each component of the algorithm.
- **November 2024** – Full refactoring of the implementation and introduction of the `loggamma` function, enabling execution on the Water network.
- **March 2025** – Executed repeated experimental trials across multiple sample sizes.
- **April 2025** – Finalized implementation, documented findings, and completed the final report.

References

[1] Gregory F. Cooper and Edward Herskovits. *A Bayesian Method for the Induction of Probabilistic Networks from Data.* Machine Learning, 1992.

[2] David Barber. *Bayesian Reasoning and Machine Learning.* Cambridge University Press, 2012.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 2020.