

Bayesian Network Learning: Ordering-Search + K2

Alexander Burns

AA 228: Decision Making Under Uncertainty · Project 1
A06690908

October 9th 2026

Abstract

Bayesian structure learning is an NP-hard problem creating difficulties as the number of variables n grows. Techniques like dynamic programming are well suited for problems with $n \leq 26$ variables since they can obtain a globally optimal structure, but are not efficient for large datasets because their complexity exponentially grows [1] with increasing n . Alternative approaches are required to obtain near-optimal structures of Bayesian networks for a range of dataset sizes. The algorithm proposed combines randomized restarts, ordering-based searches, and the *K2* algorithm to learn near-optimal structures for $n = 8, 13, 50$ datasets in runtimes of $t = 2, 8, 126$ seconds.

1 Algorithm Overview

The *K2* algorithm in combination with the *Bayesian score* scoring function is chosen due to its ability to scale well to datasets with n in the order of tens. *K2* is a greedy algorithm which determines the best suited parent node as it parses through a predefined variable ordering. As a result, *K2* does not guarantee global optimality and therefore requires a large number of executions on randomized variable orderings to obtain a near-optimal solution. I chose to combine *K2* with both *randomized restarts* and *ordering-based searches* to reach near-optimal solutions more efficiently. A total of 1000 random restarts are performed for each dataset. Code is included in the Appendix.

1.1 Improved K2 Algorithm

Improvements made to the *K2* algorithm defined in Chapter 5 of "Algorithms for Decision Making" [2] include vectorizing the *Bayesian score* computation and the storage of computed scores.

Vectorizing Count and Prior Matrix Computation. The *K2* algorithm detailed in Algorithm 5.2 [2] relies on the *Bayesian score* computation of Algorithm 5.1, which in turn calls two computationally expensive functions (Algorithms 4.1 and 4.2), one to compute the counts M and another to compute the prior α . Because these functions return the corresponding matrices for the entire network, they fail to exploit the *Bayesian score*'s inherent decomposable property, resulting in operations repeating unnecessarily. When *K2* evaluates a candidate parent p_j of node $node_i$, only the local score of $node_i$ changes, so computing the score of the remaining structure is computationally inefficient. Therefore, the *Bayesian score* is evaluated per node given its parents, and then accumulated afterwards to determine the total score of the entire network.

Since the score is now computed at the node level, M is built using vectorized operations instead of nested loops over the recorded observations. Additionally, since a uniform α is assumed, then α is

simply a matrix containing all ones of size dimensions(M).

Storing Previously Computed Scores. For each restart, the computed cost of a node given its sorted parents — defined as (node, parents) — is stored so that it can be retrieved later instead of being recomputed. This proved to be especially beneficial for the larger $n = 50$ variable dataset, since the ordering-based search detailed in §1.2 evaluates the $K2$ algorithm for orderings that differ by only one swap, meaning the rest of the structure remains the same.

1.2 Ordering Based Searches

$K2$'s predefined variable ordering means that node i may only have node j as a parent if $j < i$, so the score of the learned network greatly depends on the ordering provided. Because potential orderings grow factorially as $n!$, hill climbing is implemented to locally optimize the predefined ordering [3] as suggested by [1].

A set of $n - 1$ swaps could be performed by choosing to swap a variable v_i with its neighbor v_{i+1} . The ordering-based search implemented starts from the current ordering, performs a swap, evaluates the *Bayesian score* with $K2$, and immediately accepts the swap if it improves on the previous best score — otherwise ignoring it. This process is repeated until the *Bayesian score* no longer improves when adjacent variables are swapped, signaling that the local optimum ordering has been achieved.

1.3 Parallelizing Random Restarts

While hill climbing reduces the dependency of the algorithm on the predefined ordering, it does not eliminate it entirely since it can only yield a local optimum variable ordering. Therefore, by randomly permuting initial orderings across available CPU cores through `Joblib`, a large number of independent ordering-based searches utilizing $K2$ can be executed in parallel, with the structure of the highest *Bayesian score* returned as the final result.

2 Results

Table 1 summarizes key metrics from the learned structures, and Figs. 1 to 3 show the resulting graphs. The runtime presented includes 1000 randomized restarts of the algorithm.

2.1 Key Metrics

Table 1: Key metrics of the learned Bayesian network structures.

Dataset	Nodes	Bayesian score	Runtime (s)
Small	8	−3,794.86	2.19
Medium	13	−96,316.39	7.92
Large	50	−465,514.29	125.54

2.2 Learned Directed Acyclic Graphs

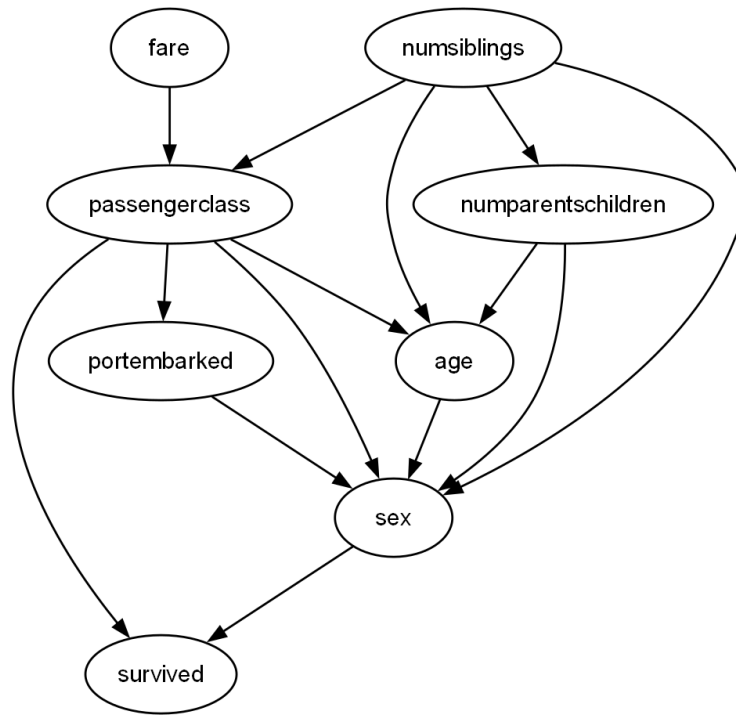


Figure 1: Learned Bayesian network for the small dataset.

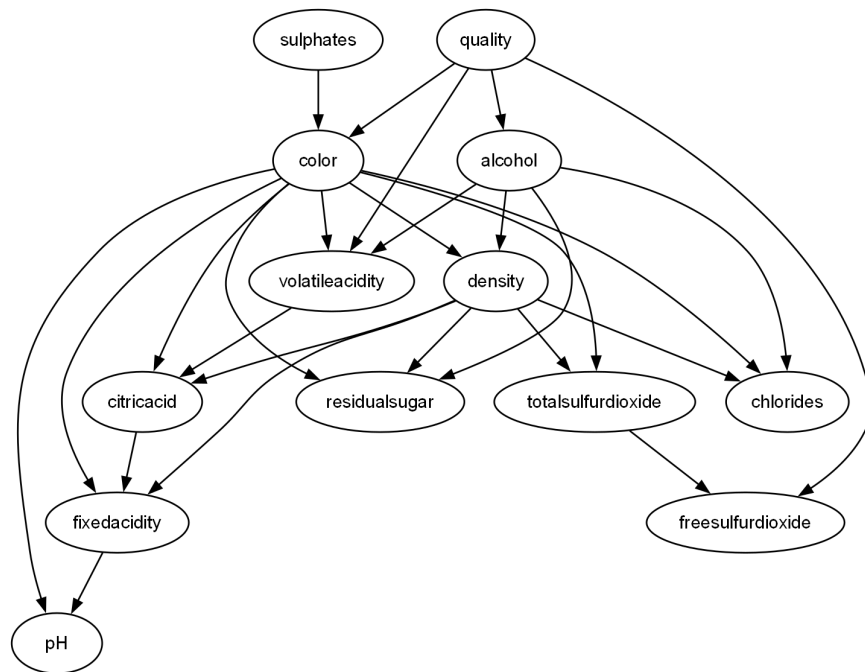


Figure 2: Learned Bayesian network for the medium dataset.

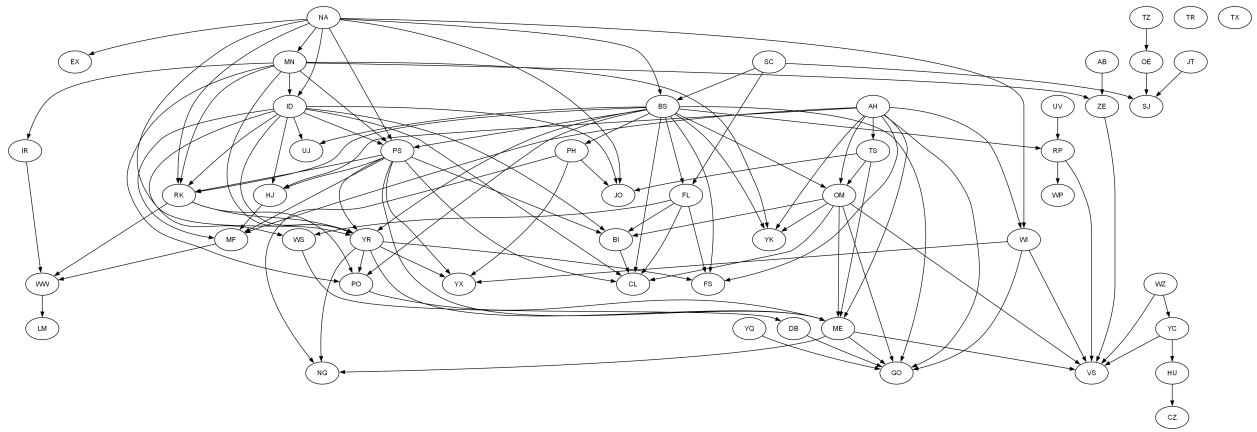


Figure 3: Learned Bayesian network for the large dataset.

AI Policy

AI (Claude) was used to assist with learning. I asked Claude questions about the Order-Based Search and K2 algorithms to provide example scenarios to help build my understanding. Claude was also used to suggest alternatives to Matlab and Julia commands that I am more accustomed to, as well as explain the syntax of commands I implemented. AI was not used to write code directly or structure the implementation of my algorithm. AI was used to learn how to create tables, figures, references, and other general formatting in LaTeX.

References

- [1] Yang Zhou. Structure learning of probabilistic graphical models: A comprehensive survey. 2011. URL <https://arxiv.org/abs/1111.6925>.
- [2] Mykel J. Kochenderfer, Tim A. Wheeler, and Kyle H. Wray. *Algorithms for Decision Making*. MIT Press, 2022. URL <https://algorithmsbook.com/files/dm.pdf>.
- [3] Marc Teyssier and Daphne Koller. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005. URL <https://arxiv.org/abs/1207.1429>.

Appendix

Code 1: K2 Algorithm Code (Python)

```

1  # === Alexander Burns / AA 228 Decision Making Under Uncertainty / Project 1 / Bayesian
   ↳ Structure Learning ===
2  """K2 Bayesian Structure Learning in combination with Ordering-based search https://arxiv.org/
   ↳ pdf/1111.6925 https://arxiv.org/pdf/1207.1429 https://algorithmsbook.com/files/dm.pdf
3  --> Uses hill climbing to determine local optimum of ordering
4  --> Uses randomized restarts parallelized across cpu cores
5  --> Uses vectorization instead of nested for loops for counts matrix
6  --> Uses memory to save computed costs for node + parent combinations for reference"""
7
8  # === Imports ===
9  import sys
10 import time
11
12 import graphviz
13 import networkx as nx
14 import numpy as np
15 import pandas as pd
16 import scipy.special as sps
17 from joblib import Parallel, delayed
18 from tqdm import tqdm
19
20
21 # === General Helper Functions ===
22 def read_csv(filename):
23     """Read the CSV data file and create the data structure"""

```

```

24     data = pd.read_csv(filename, sep=",", header=0)
25     return data
26
27
28 def save_results(outputfilename, vars, G, score, dt):
29     """Write the learned DAC to a .gph file, save the DAC figure as a png, and save the score
    ↪ and runtime"""
30
31     # == Write the learned DAC to a .gph file ==
32     H = nx.relabel_nodes(G, dict(enumerate(vars))) # Convert nodes from index to labels
33     nx.write_edgelist(H, outputfilename + ".gph", delimiter=",", data=False)
34
35     # == Save the learned DAC figure as a png ==
36     dot = graphviz.Digraph(format="png")
37     dot.attr(dpi="200")
38     dot.attr("node", fontname="Arial", fontsize="11")
39     dot.attr("edge", arrowsize="0.75")
40
41     # = Label nodes with variable names =
42     for node in G.nodes:
43         dot.node(str(node), str(vars[node]))
44     for u, v in G.edges:
45         dot.edge(str(u), str(v))
46
47     # = Write the DAC to the png =
48     dot.render(outputfilename, cleanup=True)
49
50     # == Write the Learned DAC Score ==
51     with open(outputfilename + ".score", "w") as f:
52         f.write(f"{score}")
53
54     # == Write the time it took for the algorithm to finish ==
55     with open(outputfilename + ".time", "w") as f:
56         f.write(f"{dt}")
57
58
59 # === Math Helper Functions ===
60 def bayesian_score_component(M, alpha):
61     """Identical to Ch 5 Algorithm 5.1"""
62     p_r = np.sum(sps.gammaln(alpha + M) - sps.gammaln(alpha)) # Across rows
63     p_c = np.sum(
64         sps.gammaln(np.sum(alpha, axis=1)) - sps.gammaln(np.sum(alpha + M, axis=1))
65     ) # Across columns
66
67     p = p_r + p_c # total component score
68
69     return p
70
71
72 def stats_priors_single_node(node, parents, r_all, D):
73     """Based on Ch 4 Algorithm 4.1 & 4.2: Extracts the counts and statistics from a dataset D
    ↪ given a node and parent combination.
74     Improvements are:
75     1. Vectorizes determining the count matrix and prior matrix to avoid having to have nested
    ↪ for loops
76     2. Stores each computed node parent combination into memory so that it can be pulled later
    ↪ on instead of having to recompute,
77     which really proved to be beneficial on the large dataset since I kept recomputing the
    ↪ score of a highly similar structure"""

```

```

78
79     # === Determine q and r ===
80     r = r_all[node] # Grab the number of values this node can take on
81     if len(parents) == 0:
82         q = 1
83     else:
84         q = np.prod(
85             r_all[list(parents)]
86         ) # Grab r for all parent nodes and multiply together
87
88     # === Compute the Count Matrix (qi x ri) ===
89     if len(parents) == 0:
90         D_node = D[:, node]
91
92     if len(parents) != 0:
93         # = Instead of looping over all possible observations -> Analyze all at once =
94         o_all = tuple(D[:, parent] for parent in parents) # store all observations
95         D_node_orig = D[:, node] # For this node (same as above)
96         D_node = D_node_orig + r * np.ravel_multi_index(
97             o_all, r_all[list(parents)]
98         ) # add new observation count to original
99
100     M_flat = np.bincount(
101         D_node, minlength=q * r
102     ) # create the count matrix in a 1D shape
103     M = M_flat.reshape(q, r) # convert from the list in one dimension to matrix
104
105     # === Compute The Prior Matrix (qi x ri of ones) ===
106     alpha = np.ones((q, r))
107
108     return M, alpha
109
110
111 # === Main Functions ===
112 def compute_cost_single_node(G, D, node, r_all, saved_costs):
113     """Equivalent to Chapter 5 Algorithm 5.1 but only computes the score for a single node.
114     This is done to avoid looping over all nodes every time we want to update the score, which
115     creates the potential to take advantage of the bayesian score being inherently
116     ↪ decomposable."""
117
118     # === Grab Init Values ===
119     parents = list(G.predecessors(node))
120
121     # === Check if node + parent combo has been computed before ===
122     current_entry = (node, tuple(sorted(parents)))
123     if current_entry in saved_costs:
124         return saved_costs[current_entry]
125
126     # === Determine Count and Prior Matrix ===
127     M, alpha = stats_priors_single_node(node, parents, r_all, D)
128
129     # === Compute the cost ===
130     cost = bayesian_score_component(M, alpha)
131     saved_costs[current_entry] = cost # save computed cost to memory for future
132
133     return cost
134
135 def K2_fit(D, r_all, G, ordering, saved_costs):

```

```

136 """Ch 5 Algorithm 5.2: K2 search of the space of DAGs using a specified variable ordering.
137 K2 greedily adds the next parent that maximally increases the bayesian score function.
138
139 Variable Definitions:
140 - r_all: number of values all nodes can take on
141 - ordering: nodes will only have parents preceeding the node
142 - i: the current node
143 - j: the current potential parent of node i
144 - k: counts the # of nodes preceeding the current node to use as options for parents
145 """
146
147 cost_cumulative = compute_cost_single_node(
148     G, D, ordering[0], r_all, saved_costs
149 ) # cost of just first node
150 for k, i in enumerate(ordering[1:], start=1):
151     cost = compute_cost_single_node(G, D, i, r_all, saved_costs)
152     while True:
153         cost_best, j_best = -np.inf, None
154         for j in ordering[:k]:
155             if not G.has_edge(j, i):
156                 G.add_edge(j, i)
157                 cost_new = compute_cost_single_node(G, D, i, r_all, saved_costs)
158                 if cost_new > cost_best:
159                     cost_best, j_best = cost_new, j
160                 G.remove_edge(j, i)
161         if cost_best > cost:
162             cost = cost_best
163             G.add_edge(j_best, i)
164         else:
165             break
166     cost_cumulative += cost
167 return G, cost_cumulative
168
169
170 def single_eval(vars, D, r_all, order, saved_costs):
171     """Executes a single evaluation of the K2 fit from a defined ordering"""
172
173     # = Initialize a directed acyclic graph (DAG) =
174     G = nx.DiGraph()
175     G.add_nodes_from(range(len(vars)))
176
177     # = Execute K2 Algorithm =
178     G, Score = K2_fit(D, r_all, G, order, saved_costs)
179
180     return G, Score
181
182
183 def ordering_search(vars, D, r_all, order):
184     """Ordering search using hill climb (Adjacent Variable Swaps https://arxiv.org/pdf/1207.1429)"""
185
186     # === Evaluate 'Baseline' Score ===
187     saved_costs = {} # Empty since no node + parent combos have been computed yet
188     G, Score = single_eval(vars, D, r_all, order, saved_costs)
189
190     # === Perform the Variable Swap ===
191     score_increased = True
192     while (
193         score_increased

```

```

194 ): # Repeats variable ordering swaps as long as the bayesian score improves
195     score_increased = False
196     for i in range(len(order) - 1): # iterate across n - 1 variables
197         order_curr = list(order)
198
199         # = Swap the variables =
200         order_curr[i + 1], order_curr[i] = order_curr[i], order_curr[i + 1]
201
202         # = Evaluate New Score =
203         G_curr, Score_curr = single_eval(vars, D, r_all, order_curr, saved_costs)
204
205         # = Compare New Score to Previous Best Score =
206         if Score_curr > Score:
207             # Update order for next run
208             Score, G, order = Score_curr, G_curr, order_curr
209             score_increased = True
210
211     return G, Score, order
212
213
214 def main():
215     # === Initialize ===
216     tic = time.perf_counter()
217
218     if len(sys.argv) != 4:
219         raise Exception(
220             "usage: python project1.py <infile>.csv <outfile>.gph n-restarts"
221         )
222
223     inputfilename = sys.argv[1]
224     outputfilename = sys.argv[2]
225     n_restarts = int(sys.argv[3]) # Number of Randomized Restarts to Execute
226
227     # = Read CSV data file =
228     D_pandas = read_csv(inputfilename)
229     D_numpy = D_pandas.to_numpy() - 1 # Convert pandas data frame to numpy
230     vars = D_pandas.columns
231
232     # === Compute Helper Variables to Speed Up Computation ===
233     # - Originally had to compute r every time we were finding count matrix but this can be
234     ↪ computed once at the start =
235     r_all = np.array(
236         [D_pandas[v].max() for v in vars]
237     ) # number of values all nodes can take on
238
239     # === Execute K2 Algorithm with ordering search for n_iter random restarts (Parallelized)
240     ↪ ===
241     orderings = [np.random.permutation(len(vars)) for _ in range(n_restarts)]
242
243     # = Note: I chose to use tqdm to monitor the progress of the different processes running =
244     results = list(
245         tqdm(
246             Parallel(n_jobs=-1, return_as="generator_unordered")(
247                 delayed(ordering_search)(vars, D_numpy, r_all, o) for o in orderings
248             ),
249             total=n_restarts,
250             desc="Randomized Restarts",
251         )
252     )

```

```
251
252     # = Grab Best Results (Maximum Score) =
253     toc = time.perf_counter()
254     G_best, Score_best, Order_best = max(results, key=lambda t: t[1])
255     print(f"Best score: {Score_best}")
256
257     # === Generate Output File & Plot ===
258     save_results(outputfilename, vars, G_best, Score_best, toc - tic)
259
260
261 if __name__ == "__main__":
262     main()
```