Learning Bayesian Network Structures

Brandon Malone

Much of this material is adapted from Heckerman 1998

Many of the images were taken from the Internet

February 14, 2014

Learning Bayesian Network Structures

Suppose we have a dataset $\mathcal{D}$.

H3K27me3	H2AK126su	H4AK5ac	H2AS1ph	H3K27ac	Transcription	Count
Present	Absent	Present	Absent	Absent	Inactive	25
Absent	Absent	Present	Present	Absent	Active	10
Present	Absent	Absent	Present	Absent	Inactive	15
Present	Absent	Present	Present	Absent	Active	5
Absent	Present	Absent	Absent	Present	Active	15
Absent	Present	Present	Absent	Absent	Inactive	5
Absent	Absent	Absent	Absent	Present	Active	20
Present	Present	Present	Absent	Absent	Inactive	10

What structure $\mathcal{N}$ best explains a dataset $\mathcal{D}$?

Learning Bayesian Network Structures

Suppose we have a dataset $\mathcal{D}$.

H3K27me3	H2AK126su	H4AK5ac	H2AS1ph	H3K27ac	Transcription	Count
Present	Absent	Present	Absent	Absent	Inactive	25
Absent	Absent	Present	Present	Absent	Active	10
Present	Absent	Absent	Present	Absent	Inactive	15
Present	Absent	Present	Present	Absent	Active	5
Absent	Present	Absent	Absent	Present	Active	15
Absent	Present	Present	Absent	Absent	Inactive	5
Absent	Absent	Absent	Absent	Present	Active	20
Present	Present	Present	Absent	Absent	Inactive	10

What structure $\mathcal{N}$ best explains a dataset $\mathcal{D}$?

We will use **scoring functions** to rate each network. The one with the best score is "better."

Greedy Hill Climbing

Suppose we have some network structure, which is a DAG.

We can define its **neighborhood** in **DAG-space** as all networks we can reach by applying an **operator**.

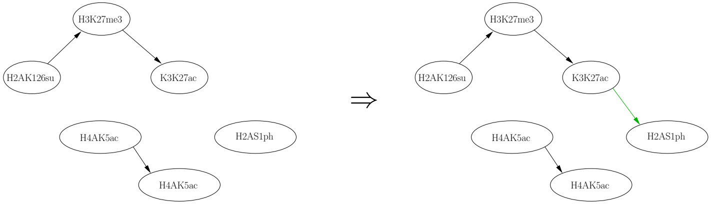
Operators

- Add an edge
- Delete an edge
- Reverse an edge

At each search step, we find the best neighbor and move to it.

Operator: Add an edge

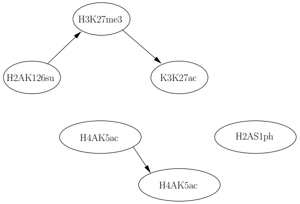
We add one edge which is missing from the structure.



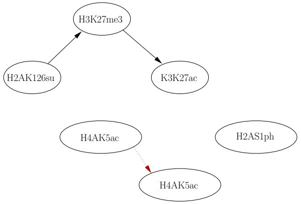
The resulting structure must still be a DAG.

Operator: Delete an edge

We remove one existing edge from the structure.



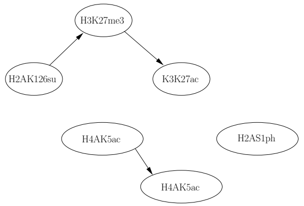
$\Rightarrow$



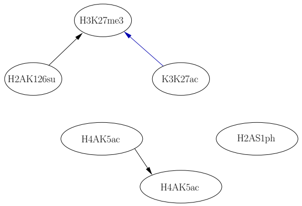
The resulting structure is guaranteed to still be a DAG.

Operator: Reverse an edge

We flip one existing edge in the structure.



$\Rightarrow$

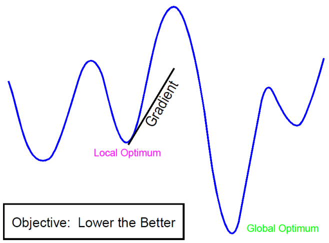


The resulting structure must still be a DAG.

Why do we need this operator?

Escaping local optima

Greedy hill climbing can "get stuck" in local optima.



Two simple strategies can be effective for escaping.

- **TABU list**. Do not revisit recently seen structures.
- **Random restarts**. Apply some operators at random when at a local optimum.

Can we guarantee anything?

Efficient implementations

During each search step, we have $O\left(n^2\right)$ operators.

How many values change from one step to the next?

Which counts do we need?

Efficient implementations

During each search step, we have $O\left(n^2\right)$ operators.

How many values change from one step to the next?

> Not many, due to decomposability.
> So we can effectively cache family scores.

Which counts do we need?

Efficient implementations

During each search step, we have $O\left(n^2\right)$ operators.

How many values change from one step to the next?

> Not many, due to decomposability.
> So we can effectively cache family scores.

Which counts do we need?

> Only those relevant for the scores we need.
> These can be efficiently calculated using an AD-tree.

Greedy hill climbing algorithm

```
procedure GreedyHillClimbing(initial structure, $\mathcal{N}_{init}$, dataset $\mathcal{D}$, scoring function $s$, stopping criteria $\mathcal{C}$ )
    $\mathcal{N}^* \leftarrow \mathcal{N}_{init}$, $\mathcal{N}' \leftarrow \mathcal{N}^*$, $tabu \leftarrow \{\mathcal{N}^*\}$
    while $\mathcal{C}$ is not satisfied do
        $\mathcal{N}'' \leftarrow \arg\max_{\mathcal{N} \in neighborhood(\mathcal{N}') and \mathcal{N} \notin tabu} s(\mathcal{N})$
        if $s(\mathcal{N}') > s(\mathcal{N}'')$ then                    ▷ Check for local optimum
            $\mathcal{N}'' \leftarrow random(\mathcal{N}')$              ▷ Apply random operators
        end if
        if $s(\mathcal{N}'') > s(\mathcal{N}^*)$ then                  ▷ Check for new best
            $\mathcal{N}^* \leftarrow \mathcal{N}''$
        end if
        $tabu \leftarrow tabu \cup \mathcal{N}'$
        $\mathcal{N}' \leftarrow \mathcal{N}''$                          ▷ Move to the neighbor
    end while
    return $\mathcal{N}^*$
end procedure
```

Dynamic Programming

Greedy hill climbing can be efficiently implemented, but it offers no guarantees about the quality of the network it finds.

Score-based structure learning is NP-complete (reduction from feedback arc set).

In practice, dynamic programming can effectively learn provably optimal networks with up to about 30 variables.

Exploiting scoring function decomposability

Observation: All BNs are DAGs, and all DAGs have a topological order.

Observation: Most scoring functions are decomposable.

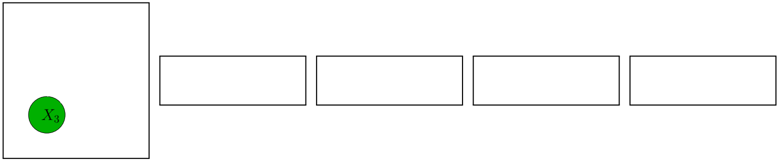
$$Score(\mathcal{N} : \mathcal{D}) = \sum_{i}^{n} Score(X_i, PA_i : \mathcal{D})$$

The relationships among X_i's parents does not matter.

We will typically omit $\mathcal{D}$ and refer to $Score(X, PA_X)$ as **local scores**.

Dynamic programming decomposition

Suppose we are given an ordering: X_3, X_2, X_1, X_4.



We can select optimal parents as follows.

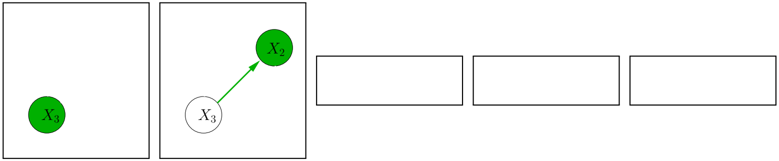
$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

$$PA_X = \arg\min_{PA \in \mathbf{U}} Score(X, PA)$$

U is the set of variables which precede X in the order.

Dynamic programming decomposition

Suppose we are given an ordering: X_3, X_2, X_1, X_4.



We can select optimal parents as follows.

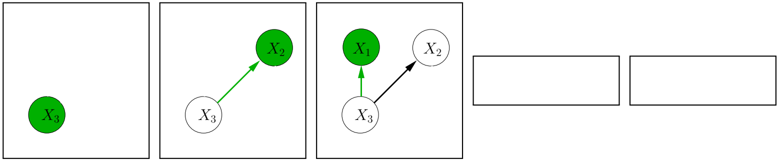
$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

$$PA_X = \arg\min_{PA \in \mathbf{U}} Score(X, PA)$$

U is the set of variables which precede X in the order.

Dynamic programming decomposition

Suppose we are given an ordering: X_3, X_2, X_1, X_4.



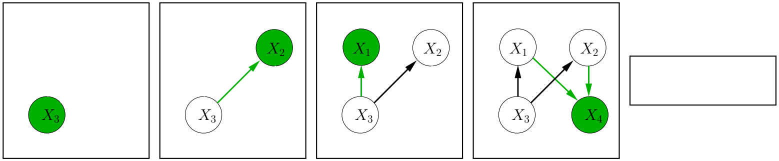
We can select optimal parents as follows.

$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$
$$PA_X = \arg\min_{PA \in \mathbf{U}} Score(X, PA)$$

$\mathbf{U}$ is the set of variables which precede X in the order.

Dynamic programming decomposition

Suppose we are given an ordering: X_3, X_2, X_1, X_4.



We can select optimal parents as follows.

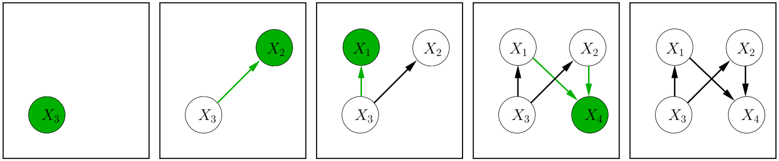
$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

$$PA_X = \arg\min_{PA \in \mathbf{U}} Score(X, PA)$$

$\mathbf{U}$ is the set of variables which precede X in the order.

Dynamic programming decomposition

Suppose we are given an ordering: X_3, X_2, X_1, X_4.



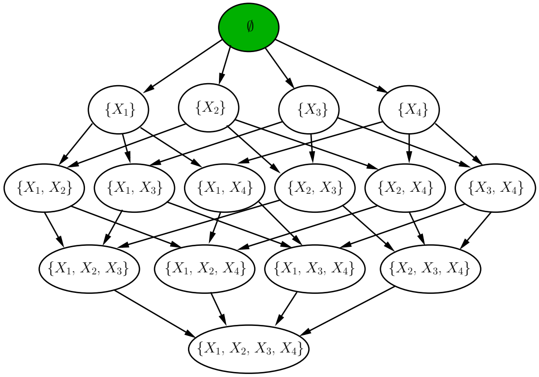
We can select optimal parents as follows.

$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

$$PA_X = \arg\min_{PA \in \mathbf{U}} Score(X, PA)$$

$\mathbf{U}$ is the set of variables which precede X in the order.

Dynamic programming graph



A path from $\emptyset$ to $\{X_1, X_2, X_3, X_4\}$ induces an order on the variables.

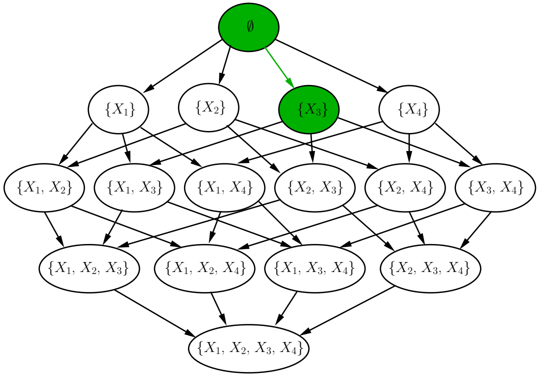
Based on the order, we select optimal parents. This associates a cost with each edge.

Observation: The path taken to reach a node does not affect the later choices.

Each node $\mathbf{U}$ contains an optimal subnetwork and $Score(\mathbf{U})$.

Following an edge means adding a leaf to the subnetwork.

Dynamic programming graph



Each node **U** contains an optimal subnetwork and *Score*(**U**).

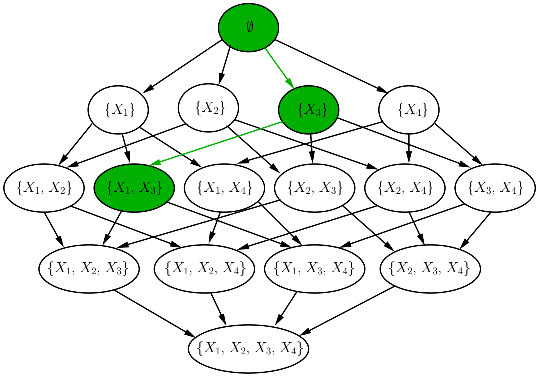
Following an edge means adding a leaf to the subnetwork.

A path from $\emptyset$ to $\{X_1, X_2, X_3, X_4\}$ induces an order on the variables.

Based on the order, we select optimal parents. This associates a cost with each edge.

Observation: The path taken to reach a node does not affect the later choices.

Dynamic programming graph



A path from $\emptyset$ to $\{X_1, X_2, X_3, X_4\}$ induces an order on the variables.

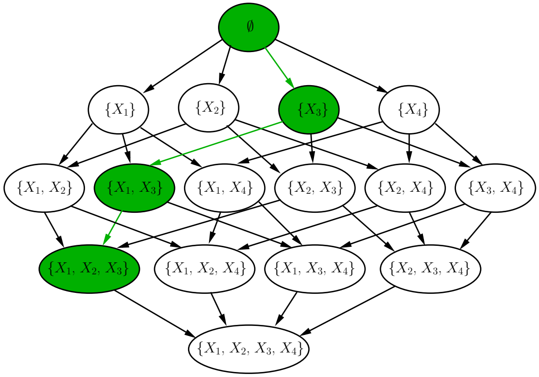
Based on the order, we select optimal parents. This associates a cost with each edge.

Observation: The path taken to reach a node does not affect the later choices.

Each node $\mathbf{U}$ contains an optimal subnetwork and $Score(\mathbf{U})$.

Following an edge means adding a leaf to the subnetwork.

Dynamic programming graph



Each node **U** contains an optimal subnetwork and *Score*(**U**).

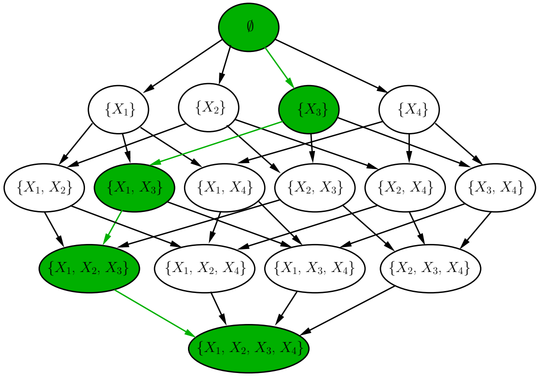
Following an edge means adding a leaf to the subnetwork.

A path from $\emptyset$ to $\{X_1, X_2, X_3, X_4\}$ induces an order on the variables.

Based on the order, we select optimal parents. This associates a cost with each edge.

Observation: The path taken to reach a node does not affect the later choices.

Dynamic programming graph



A path from $\emptyset$ to $\{X_1, X_2, X_3, X_4\}$ induces an order on the variables.

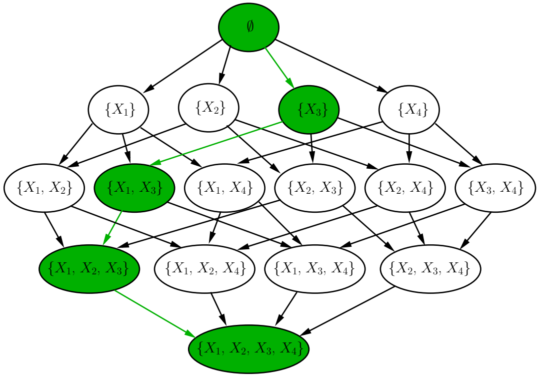
Based on the order, we select optimal parents. This associates a cost with each edge.

Observation: The path taken to reach a node does not affect the later choices.

Each node $\mathbf{U}$ contains an optimal subnetwork and $Score(\mathbf{U})$.

Following an edge means adding a leaf to the subnetwork.

Dynamic programming graph



The cost of a path is equal to the sum of the costs of its edges.

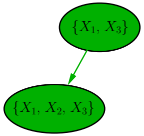
The cost of the shortest path corresponds to the optimal network.

We can explore the graph, e.g., in breadth-first order to find the shortest path to each node.

Each node $\mathbf{U}$ contains an optimal subnetwork and $Score(\mathbf{U})$.

Following an edge means adding a leaf to the subnetwork.

Dealing with local scores

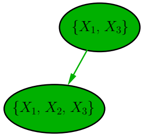


The cost of an edge from $\mathbf{U}$ to $\mathbf{U} \cup X$ is defined as follows.

$$Cost(\mathbf{U}, \mathbf{U} \cup X) := BestScore(X, \mathbf{U})$$
$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

How can we efficiently calculate this value?

Dealing with local scores



The cost of an edge from $\mathbf{U}$ to $\mathbf{U} \cup X$ is defined as follows.

$$Cost(\mathbf{U}, \mathbf{U} \cup X) := BestScore(X, \mathbf{U})$$
$$BestScore(X, \mathbf{U}) = \min_{PA \in \mathbf{U}} Score(X, PA)$$

How can we efficiently calculate this value?

Precompute "all" local scores for X and sort them.

Linearly scan when given a new query parent set $\mathbf{U}$.

(Fancier tricks tend to be much slower.)

Pruning Theorem

Say $PA \subset PA'$ and $Score(X, PA) < Score(X, PA')$. Then PA' is not optimal.

Dynamic programming algorithm

```
procedure EXPAND(node $\mathbf{U}$, sorted family scores $BestScore$)
    for each $leaf$ in $\mathbf{V} \setminus \mathbf{U}$ do
        $newScore \leftarrow Score(\mathbf{U}) + BestScore(leaf, \mathbf{U})$
        if $newScore < Score(\mathbf{U} \cup leaf)$ then
            $Score(\mathbf{U} \cup leaf) \leftarrow newScore$
        end if
    end for
end procedure

procedure MAIN(variables $\mathbf{V}$, sorted family scores $BestScore$)
    $Score(\emptyset) \leftarrow 0$
    for layer $l = 0$ to $|\mathbf{V}|$ do
        for each node $\mathbf{U}$ such that $|\mathbf{U}| = l$ do
            expand($\mathbf{U}$, $BestScore$)
        end for
    end for
    return $Score(\mathbf{V})$
end procedure
```

Recap

During this part of the course, we have discussed:

- A greedy hill climbing algorithm for structure learning
- A dynamic programming algorithm which guarantees to find the optimal structure

Next in probabilistic models

We will return to parameter learning.

- Expectation-maximization for learning parameters in mixture models
- (If time permits) Estimating parameters in topic models