

# Constraint Satisfaction Problems

## Search and Look-ahead

Albert-Ludwigs-Universität Freiburg



UNI  
FREIBURG

**Stefan Wöfl, Christian Becker-Asano, and Bernhard Nebel**

November 17, 2014

- Enforcing consistency is one way of solving constraint networks: Globally consistent networks can easily be solved in polynomial time.
- However, enforcing global consistency is costly in time and space: it not only takes exponential time to compute an equivalent globally consistent network, but also exponential space to store it.
- Thus, it is usually advisable to only enforce **local** consistency (e. g., arc consistency or path consistency), and compute a solution through **search** through the remaining possibilities.

State Spaces

Backtracking

Look-ahead  
strategies

# 1 State Spaces



State Spaces

Backtracking

Look-ahead  
strategies

The fundamental abstractions for search are **state spaces**.  
They are defined in terms of:

- **states**, representing a partial solution to a problem (which may or may not be extensible to a full solution)
- an **initial state** from which to search for a solution
- **goal states** representing solutions
- **operators** that define how a new state can be obtained from a given state

## Definition (state space)

A **state space** is a 4-tuple  $\mathcal{S} = \langle S, s_0, S_*, O \rangle$ , where

- $S$  is a finite set of **states**,
- $s_0 \in S$  is the **initial state**,
- $S_* \subseteq S$  is the set of **goal states**, and
- $O$  is a finite set of **operators**, where each operator  $o \in O$  is a partial function on  $S$ , i. e.  $o : S' \rightarrow S$  for some  $S' \subseteq S$ .

We say that an operator  $o$  is **applicable** in state  $s$  if  $o(s)$  is defined.

**Search** is the problem of finding a sequence of operators that transforms the initial state into a goal state.

## Definition (solution of a state space)

Let  $\mathcal{S} = \langle S, s_0, S_*, O \rangle$  be a state space, and let  $o_1, \dots, o_n \in O$  be an operator sequence.

Inductively define result states  $r_0, r_1, \dots, r_n \in S \cup \{\text{invalid}\}$ :

- $r_0 := s_0$
- For  $i \in \{1, \dots, n\}$ , if  $o_i$  is applicable in  $r_{i-1}$ , then  $r_i := o_i(r_{i-1})$ . Otherwise,  $r_i := \text{invalid}$ .

The operator sequence is a **solution** iff  $r_n \in S_*$ .

State Spaces

Backtracking

Look-ahead  
strategies

- State spaces can be depicted as **state graphs**: labeled directed graphs where states are vertices and there is a directed arc from  $s$  to  $s'$  with label  $o$  iff  $o(s) = s'$  for some operator  $o$ .
- There are many classical algorithms for finding solutions in state graphs, e. g. **depth-first search**, **breadth-first search**, **iterative deepening search**, or heuristic algorithms like  $A^*$ .
- These algorithms offer different trade-offs in terms of runtime and memory usage.

The state spaces for constraint networks usually have two special properties:

- The search graphs are **trees** (i. e., there is exactly one path from the initial state to any reachable search state).
- All **solutions** are **at the same level** of the tree.

Due to these properties, variations of **depth-first search** are usually the method of choice for solving constraint networks.

We will now define state spaces for constraint networks.



## Definition (unordered search space)

Let  $N = \langle V, D, C \rangle$  be a constraint network.

The **unordered search space** of  $N$  is the following state space:

- **states**: partial solutions of  $N$  (i. e., consistent assignments)
- **initial state**: the empty assignment  $\emptyset$
- **goal states**: solutions of  $N$
- **operators**: for each  $v_i \in V$  and  $d \in D_i$ , one operator  $o_{v_i=d}$  as follows:
  - $o_{v_i=d}$  is applicable in those states  $s$  where  $v_i$  is not defined and  $s \cup \{(v_i, d)\}$  is consistent
  - $o_{v_i=d}(s) = s \cup \{(v_i, d)\}$

State Spaces

Backtracking

Look-ahead  
strategies

The unordered search space formalizes the systematic construction of solutions, by consistently extending partial solutions until a solution is found.

- Later on, we will consider alternative (non-systematic) search techniques.



In practice, one will only search for solutions in **subspaces** of the complete unordered search space:

- Consider a state  $s$  where  $v_i \in V$  has not been assigned a value. If no solution can be reached from **any** successor state for the operators  $o_{v_i=d}$  ( $d \in D_i$ ), then no solution can be reached from  $s$ .
- There is **no point** in trying operators  $o_{v_j=d'}$  for other variables  $v_j \neq v_i$  in this case!
- Thus, it is sufficient to consider operators for **one particular unassigned variable** in each search state.
- How to decide which variable to use is an important issue. Here, we first consider **static variable orderings**.

State Spaces

Backtracking

Look-ahead  
strategies

Let  $N = \langle V, D, C \rangle$  be a constraint network.

## Definition (variable ordering)

A **variable ordering** of  $N$  is a permutation of the variable set  $V$ .  
We write variable orderings in sequence notation:  $v_1, \dots, v_n$ .

## Definition (ordered search space)

Let  $\sigma = v_1, \dots, v_n$  be a variable ordering of  $N$ .

The **ordered search space** of  $N$  along ordering  $\sigma$  is the state space obtained from the unordered search space of  $N$  by restricting each operator  $o_{v_i=d_i}$  to states  $s$  with  $|s| = i - 1$ .

- In other words, in the initial state, only  $v_1$  can be assigned, then only  $v_2$ , then only  $v_3, \dots$

State Spaces

Backtracking

Look-ahead  
strategies

# The importance of good orderings



State Spaces

Backtracking

Look-ahead  
strategies

- All ordered search spaces for the same constraint network contain the same set of solution states.
- However, the **total number of states** can vary dramatically between different orderings.
- The size of a state space is a (rough) measure for the hardness of finding a solution, so we are interested in small search spaces.
- One way of measuring the quality of a state space is by counting the number of **dead ends**: the fewer, the better.

## Definition (dead end)

A **dead end** of a state space is a state which is not a goal state and in which no operator is applicable.

- In an ordered search space, a dead end is a partial solution that cannot be consistently extended to the next variable in the ordering.
- In the unordered search space, a dead end is a partial solution that cannot be consistently extended to **any** of the remaining variables.

In both cases, this partial solution cannot be part of a solution.

State Spaces

Backtracking

Look-ahead  
strategies

## Definition (backtrack-free)

A state space is called **backtrack-free** if it contains no dead ends.

A constraint network  $N$  is called **backtrack-free** along variable ordering  $\sigma$  if the ordered search space of  $N$  along  $\sigma$  is backtrack-free.

- Backtrack-free networks are the ideal case for search algorithms.
- Constraint networks are rarely backtrack-free along any ordering in the way they are specified naturally.
- However, constraint networks can be **reformulated** (replaced with an equivalent constraint network) to reduce the number of dead ends.
- One way of doing this is by enforcing a local consistency property like arc consistency or path consistency, which leads to a **tighter** network.

State Spaces

Backtracking

Look-ahead  
strategies



## Lemma

*Let  $N$  and  $N'$  be equivalent constraint networks.*

*If  $N'$  is at least as tight as  $N$ , then*

- *the unordered search space of  $N'$  has at most as many dead ends as the unordered search space of  $N$ , and*
- *the ordered search space of  $N'$  along any ordering  $\sigma$  has at most as many dead ends as the ordered search space of  $N$  along the same ordering  $\sigma$ .*

## Proof.

For every dead end of  $N'$  (in either kind of state space), the same assignment is a state in the state space for  $N$  which has at least one dead end as a descendant. □

State Spaces

Backtracking

Look-ahead  
strategies

## Lemma

*Let  $N$  be a constraint network.*

*The following three statements are equivalent:*

- *The unordered search space of  $N$  is backtrack-free.*
- *The ordered search space of  $N$  is backtrack-free along each ordering  $\sigma$ .*
- *$N$  is globally consistent.*

- Replacing constraint networks by tighter, equivalent networks is a powerful way of reducing dead ends.
- However, one can go much further by also tightening constraints **during search**, for example by enforcing local consistency **for a given partial instantiation**.
- We will consider such search algorithms soon.
- In general, there is a trade-off between reducing the number of dead ends and the overhead for consistency reasoning.

## 2 Backtracking



State Spaces

**Backtracking**

Look-ahead  
strategies

Backtracking traverses the search space of partial instantiations in a depth-first manner in two phases:

- **forward phase**: variables are selected in sequence; the current partial solution is extended by assigning a consistent value to the next variable (if possible)
- **backward phase**: if no consistent instantiation for the current variable exists, we return to the previous variable.

# Backtracking: Example

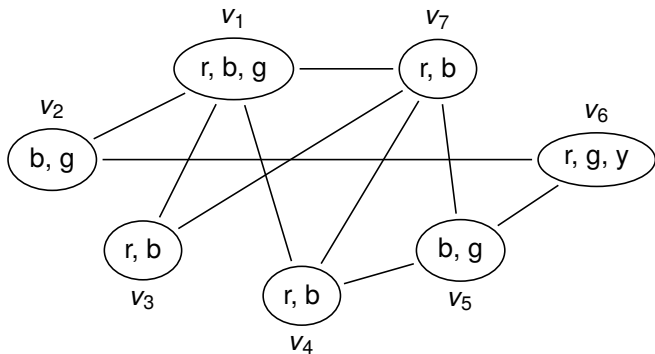


Consider the constraint network defined by the following coloring problem:

State Spaces

Backtracking

Look-ahead  
strategies



# Backtracking: Example

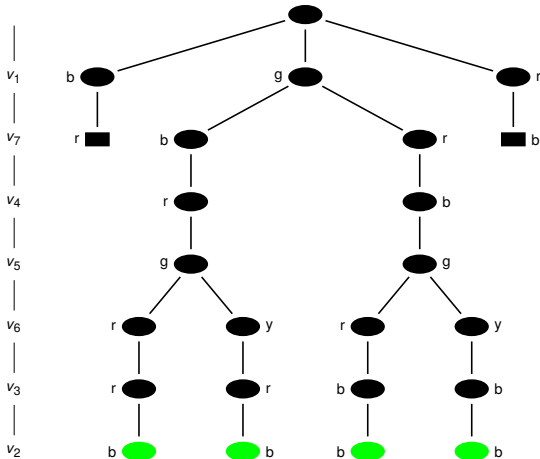


On this example we apply the backtracking algorithm by using the variable ordering:  $v_1, v_7, v_4, v_5, v_6, v_3, v_2$ , and we obtain:

State Spaces

Backtracking

Look-ahead  
strategies



# Backtracking algorithm (recursive version)



State Spaces

Backtracking

Look-ahead  
strategies

**Backtracking**( $N, a$ ):

*Input:* a constraint network  $N = \langle V, D, C \rangle$  and  
a partial solution  $a$  of  $N$   
(initially: the empty instantiation  $a = \{ \}$ )

*Output:* a solution of  $N$  or “inconsistent”

**if**  $a$  is not locally consistent with  $N$ :

**return** “inconsistent”

**if**  $a$  is defined for all variables in  $V$ :

**return**  $a$

select **some variable**  $v_i$  for which  $a$  is not defined

**for** **each value**  $d$  from  $D_j$ :

$a' := a \cup \{ (v_i, d) \}$

$a'' \leftarrow \text{Backtracking}(N, a')$

**if**  $a''$  is not “inconsistent”:

**return**  $a''$

**return** “inconsistent”



# Backtracking algorithm (recursive version 2)



## Backtracking( $N, a$ ):

*Input:* a constraint network  $N = \langle V, D, C \rangle$  and  
a partial solution  $a$  of  $N$   
(initially: the empty instantiation  $a = \{ \}$ )

*Output:* a solution of  $N$  or “inconsistent”

**if**  $a$  is defined for all variables in  $V$ :

**return**  $a$

select **some variable**  $v_i$  for which  $a$  is not defined

**for each value**  $d$  from  $D_i$ :

$a' := a \cup \{(v_i, d)\}$

**if**  $a'$  is locally consistent with  $N$ :

$a'' \leftarrow \text{Backtracking}(N, a')$

**if**  $a''$  is not “inconsistent”:

**return**  $a''$

**return** “inconsistent”

State Spaces

Backtracking

Look-ahead  
strategies

**Enumeration** The variable  $v$  is instantiated in turn to each value in its domain. First  $v = d_1$ , then  $v = d_2$ , etc.

**Binary choice points** The variable  $v$  is instantiated to some value in its domain. Assuming the value 1 is chosen in our example, two branches are generated and the constraints  $v = d_1$  and  $v \neq d_1$  are posted, respectively.

**Domain splitting** The domain of the variable  $v$  is split in two parts. For instance, with a domain of size 4: choose first  $v = \{d_1, d_2\}$ , then  $v = \{d_3, d_4\}$

Those are identical when constraints are binary. For this lecture, we will only consider the **enumeration** branching strategy.

State Spaces

Backtracking

Look-ahead  
strategies

# 3 Look-ahead strategies



State Spaces

Backtracking

Look-ahead  
strategies

- Backtracking suffers from **thrashing**: partial solutions that cannot be extended to a full solution may be reprocessed several times (always leading to a dead end in the search space)
  - **Idea**: Improve (practical) performance by
    - preprocessing the search space underneath the currently selected variable
    - improving (in a dynamic way) the search strategy
- ⇒ two schemes (related to the two phases of backtracking search), namely **look-ahead** and **look-back** strategies

- **Look-ahead:** invoked when next variable or next value is selected. For example:
  - Which variable should be instantiated next?  
↪ prefer variables that impose tighter constraints on the rest of the search space
  - Which value should be chosen for the next variable?  
↪ maximize the number of options for future assignments
- **Look-back:** invoked when the backtracking step is performed after reaching a dead end. For example:
  - How deep should we backtrack?  
↪ avoid irrelevant backtrack points (by analyzing reasons for the dead end and **jumping back** to the source of failure)
  - How can we learn from dead ends?  
↪ record reasons for dead ends as new constraints so that the same inconsistencies can be avoided at later stages of the search

State Spaces

Backtracking

Look-ahead  
strategies

## **LookAhead**( $N, a$ ):

*Input:* a constraint network  $N = \langle V, D, C \rangle$  and a partial solution  $a$  of  $N$  (initially: the empty instantiation  $a = \{ \}$ )

*Output:* a solution of  $N$  or “inconsistent”

**SelectValue**( $v_i, D_i, a, N$ ): procedure that selects and deletes a consistent value  $d \in D_i$ ; returns  $d$  and a refinement of  $N$ ;  
returns “null”, if all  $a \cup \{ (v_i, d) \}$  are inconsistent

**if**  $a$  is defined for all variables in  $V$ :

**return**  $a$

select **some** variable  $v_i$  for which  $a$  is not defined

$N' \leftarrow N, D'_i \leftarrow D_i$  // (work on a copy)

**while**  $D'_i$  is non-empty

$d, N'' \leftarrow \text{SelectValue}(v_i, D'_i, a, N')$

**if**  $d$  is not “null”:

$a' \leftarrow \text{LookAhead}(N'', a \cup \{ (v_i, d) \})$

**if**  $a'$  is not “inconsistent”:

**return**  $a'$

**return** “inconsistent”

- 1 **Forward Checking**: propagate the effect of a value-selection to each single non-instantiated variables
- 2 **Partial Look-Ahead** ...
- 3 **Full Look-Ahead** ...
- 4 **Real Full Look-Ahead**: enforce full arc consistency on the future variables after each assignment to the current variable

**SelectValue-ForwardChecking**( $v_i, D'_i, a, N$ ):

---

```
select and delete  $d$  from  $D'_i$ 
for each  $v_j$  sharing a constraint with  $v_i$  for which  $a$  is not defined
     $D'_j \leftarrow D_j$  // (work on a copy)
    for each value  $d' \in D'_j$ 
        if not consistent( $a \cup \{(v_i, d), (v_j, d')\}$ )
            remove  $d'$  from  $D'_j$ 
    if any future  $D'_j$  is empty // ( $v_i \mapsto x$  leads to a dead end)
        return "null"
 $D_j \leftarrow D'_j$  // (propagate all future  $D_j$ )
return  $d$ 
```

State Spaces

Backtracking

Look-ahead  
strategies





**SelectValue-RealFullLookAhead**( $v_i, D'_i, a, N$ ):

---

select and delete  $d$  from  $D'_i$

$D'_j \leftarrow D_j$  (for all non-assigned  $v_j \neq v_i$  work on a copy)

**repeat**

**for** each  $v_j$  ( $j \neq i$ ) for which  $a$  is not defined

**for** each  $v_k$  ( $k \neq i, j$ ) for which  $a$  is not defined

**for** each value  $d' \in D'_j$

**if** there is no value  $d'' \in D'_k$  such that

          consistent( $a \cup \{(v_i, d), (v_j, d'), (v_k, d'')\}$ )

          remove  $d'$  from  $D'_j$

**until** no value was removed

**if** any future  $D'_j$  is empty // ( $v_i \mapsto d$  leads to a dead end)

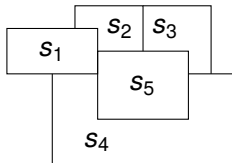
**return** "null"

$D_j \leftarrow D'_j$  // (propagate all future  $D_j$ )

**return**  $d$

# Look-ahead example (no look-ahead)

## Example



	Red	Blue	Green
S <sub>1</sub>			
S <sub>2</sub>			
S <sub>3</sub>			
S <sub>4</sub>			
S <sub>5</sub>			

Initial State

# Example: Look-ahead with forward checking

## Example

State Spaces

Backtracking

Look-ahead  
strategies

	Red	Blue	Green
$s_1$			
$s_2$			
$s_3$			
$s_4$			
$s_5$			

Initial State

	Red	Blue	Green
$s_1$	$O$		
$s_2$			
$s_3$			

# Example: Real full look-ahead

## Example

State Spaces

Backtracking

Look-ahead  
strategies

	Red	Blue	Green
$s_1$			
$s_2$			
$s_3$			
$s_4$			
$s_5$			

Initial State

	Red	Blue	Green
$s_1$	$O$		
$s_2$			
$s_3$			

- 1 Forward checking:  $\mathcal{O}(e \cdot k^2)$ ,
- 2 ...
- 3 Real full look-ahead (also known as MAC): with AC3-variant in  $\mathcal{O}(e \cdot k^3)$ ,

where  $k$  is the cardinality of the largest domain and  $e$  is the number of constraints.

## Remark

Keeping the balance between pruning the search space and cost of look-ahead

Good tradeoffs are nowadays:

- Forward checking
- Real full look-ahead

State Spaces

Backtracking

Look-ahead  
strategies

*Dynamic look-ahead value orderings*: estimate likelihood that a non-rejected value leads to a solution. For example:

- **MinConflicts (MC)**: prefer a value that removes the smallest number of values from the domains of future variables
- **MaxDomainSize (MD)**: prefer a value that ensures the largest minimum domain sizes of future variables (i.e., calculate  $n_d := \min_{v_j} |D'_j|$  after assigning  $v_i \mapsto d$ , and  $n_{d'}$  for  $v_i \mapsto d'$ , respectively; if  $n_d > n_{d'}$ , then prefer  $v_i \mapsto d$ )

- Backtracking and LookAhead leave the choice of variable ordering open.
- Ordering greatly affects performance.  
     $\rightsquigarrow$  exercises

State Spaces

Backtracking

Look-ahead  
strategies

We distinguish

- **Dynamic ordering:**
  - In each state, decide **independently** which variable to assign to next.
  - Can be seen as search in a subspace of the unordered search space.
- **Static ordering:**
  - A variable ordering  $\sigma$  is fixed in advance.
  - Search is conducted in the ordered search space along  $\sigma$ .

Common heuristic:

## Fail-first

Always select a variable whose remaining domain has a minimal number of elements.

- intuition: few subtrees  $\rightsquigarrow$  small search space
- extreme case: only one value left  $\rightsquigarrow$  no search  
 $\Rightarrow$  compare [Unit Propagation](#) in DPLL procedure
- Should be combined with a constraint propagation technique such as Forward Checking or Arc Consistency.

State Spaces

Backtracking

Look-ahead  
strategies



Static variable orderings...

- lead to **no overhead** during search
- but are **less flexible** than dynamic orderings

In practice, they are often very good if chosen properly.

Popular choices:

- **Max-cardinality ordering**
- **Min-width ordering**
- **Cycle cutset ordering**

State Spaces

Backtracking

Look-ahead  
strategies

## Max-cardinality ordering

- 1 Start with an arbitrary variable.
- 2 Repeatedly add a variable such that the number of constraints whose scope is a subset of the set of added variables is maximal. Break ties arbitrarily.

↪ for the other two ordering strategies, we first need to lay some foundations

## Definition (ordered graph)

Let  $G = \langle V, E \rangle$  be a graph.

An **ordered graph** for  $G$  is a tuple  $\langle V, E, \sigma \rangle$ , where  $\sigma$  is an ordering (permutation) of the vertices in  $V$ .

As usual, we use sequence notation for the ordering:

$$\sigma = v_1, \dots, v_n.$$

We write  $v \prec v'$  if  $v$  precedes  $v'$  in  $\sigma$ .

The **parents** of  $v \in V$  in the ordered graph are the neighbors that precede it:  $\{u \in V \mid u \prec v, \{u, v\} \in E\}$ .

## Definition (width)

The **width** of a vertex  $v$  of an ordered graph is the number of parents of  $v$ .

The **width** of an ordered graph is the maximal width of its vertices.

The **width** of a graph  $G$  is the minimal width of all ordered graphs for  $G$ .

## Theorem

*A graph with at least one edge has width 1 iff it is a forest (i.e., iff it contains no cycles).*

## Proof.

A graph with at least one edge has at least width 1.

( $\Rightarrow$ ): If a graph has a cycle consisting of vertices  $C$ , then in any ordering  $\sigma$ , one of the vertices in  $C$  will appear last. This vertex will have width at least 2. Thus, the width of the ordering cannot be 1.

( $\Leftarrow$ ): Consider a graph  $\langle V, E \rangle$  with no cycles. In every connected component, pick an arbitrary vertex; these are called root nodes. Construct ordered graph  $\langle V, E, \sigma \rangle$  by putting root nodes first in  $\sigma$ , then nodes with distance 1 from a root node, then distance 2, 3, etc. This ordered graph has width 1. □

State Spaces

Backtracking

Look-ahead  
strategies

To find solutions to constraint networks, we are interested in the **width of the primal constraint graph**.

- The width of a graph is a (rough) **difficulty measure**.
  - For width 1, we can make this more precise (next slide).
  - In general, there is a provable relationship between solution effort and a closely related measure called **induced width**.
- The ordering that leads to an ordered graph of minimal width is usually a good static variable ordering.

## Theorem

*Let  $N$  be a constraint network whose primal constraint graph has width 1. Then  $N$  can be solved in polynomial time.*

**Note:** Such a constraint network must be binary, as constraints of higher arity  $\geq 3$  induce cycles in the primal constraint graph.

## Lemma

*Let  $N$  be an **arc-consistent** (normalized) constraint network whose primal constraint graph has width 1, and where all variable domains are non-empty. Then  $N$  is backtrack-free along any ordering with width 1.*

State Spaces

Backtracking

Look-ahead  
strategies



## Proof of the lemma.

Let  $N$  be such a constraint network, and let  $\sigma = v_1, \dots, v_n$  be a width-1 ordering for  $N$ . We must show that all partial solutions of the form  $\{v_1 \mapsto d_1, \dots, v_i \mapsto d_i\}$  for  $0 \leq i < n$  can be consistently extended to variable  $v_{i+1}$ .

Since  $\sigma$  has width 1, the width of  $v_{i+1}$  is 0 or 1.

- **$v_{i+1}$  has width 0:** There is no constraint between  $v_{i+1}$  and any assigned variable, so any value in the (non-empty) domain of  $v_{i+1}$  is a consistent extension.
- **$v_{i+1}$  has width 1:** There is exactly one variable  $v_j \in \{v_1, \dots, v_i\}$  with a constraint between  $v_j$  and  $v_{i+1}$ . For every choice  $(v_j \mapsto d_j)$ , there must be a consistent choice  $(v_{i+1} \mapsto d_{i+1})$  because of arc consistency.

State Spaces

Backtracking

Look-ahead  
strategies





## Proof of the theorem.

We can enforce arc consistency and compute a width 1 ordering in polynomial time. If the resulting network has any empty variable domains, it is trivially unsolvable. Otherwise, by the lemma, it can be solved in polynomial time by the Backtracking procedure.  $\square$

**Remark:** Enforcing full arc consistency is actually not necessary; a limited form of consistency is sufficient. (We do not discuss this further.)

## Min-width ordering

Select a variable ordering such that the resulting ordered constraint graph has minimal width among all choices.

**Remark:** Can be computed efficiently by a greedy algorithm:

- 1 Choose a vertex  $v$  with minimal degree and remove it from the graph.
- 2 Recursively compute an ordering for the remaining graph, and place  $v$  after all other vertices.

# Static variable orderings: Cycle cutset ordering



State Spaces

Backtracking

Look-ahead  
strategies

## Definition (cycle cutset)

Let  $G = \langle V, E \rangle$  be a graph.

A **cycle cutset** for  $G$  is a vertex set  $V' \subseteq V$  such that the subgraph induced by  $V \setminus V'$  has no cycles.

## Cycle cutset ordering

- 1 Compute a (preferably small) cycle cutset  $V'$ .
- 2 First order all variables in  $V'$  (using any ordering strategy).
- 3 Then order the remaining variables, using a width-1 ordering for the subnetwork where the variables in  $V'$  are removed.

- If the network is binary and the search algorithm enforces arc consistency after assigning to the cutset variables, no further search is needed at this point.

↪ runtime  $\mathcal{O}(k^{|V|} \cdot p(\|N\|))$  for some polynomial  $p$

- However, finding **minimum** cycle cutsets is NP-hard.
- Even finding **approximate solutions** is provably hard.
- However, in practice good cutsets can usually be found.

State Spaces

Backtracking

Look-ahead  
strategies



Rina Dechter.

Constraint Processing,

Chapters 4 and 5, Morgan Kaufmann, 2003