Constraint Satisfaction Problems

Search and Lookahead

Bernhard Nebel and **Stefan Wölfl**

based on a slideset by
Malte Helmert and Stefan Wölfl
(summer term 2007)

Albert-Ludwigs-Universität Freiburg

November 2/4, 2009

Constraint Satisfaction Problems

November 2/4, 2009 — Search and Lookahead

Search and Lookahead

- 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: As much space as $\Omega(k^n)$ may be required to represent an equivalent globally consistent network in the case of n variables with domain size k.
- 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: Informally

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

State Spaces: Formally

Definition (state space)

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

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $S_\star \subseteq S$ 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. $o : S' \to S$ for some $S' \subseteq S$.

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

Search

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

Definition (solution of a state space)

Let $\mathcal{S} = \langle S, s_0, S_\star, O\rangle$ 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, $r_i :=$ invalid.

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

Search Graphs and Search Algorithms

- 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.

State Spaces for Constraint Networks

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.

Unordered Search Space

Definition (unordered search space)

Let $\mathcal{C} = \langle V, \mathrm{dom}, C \rangle$ be a constraint network.
The unordered search space of $\mathcal{C}$ is the following state space:

- states: partial solutions of $\mathcal{C}$ (i. e., consistent assignments)
- initial state: the empty assignment $\emptyset$
- goal states: solutions of $\mathcal{C}$
- operators: for each $v \in V$ and $a \in \mathrm{dom}(v)$, one operator $o_{v=a}$ as follows:
  - $o_{v=a}$ is applicable in those states s where v is not defined and $s \cup \{(v \mapsto a)\}$ is consistent
  - $o_{v=a}(s) = s \cup \{(v \mapsto a)\}$

Unordered Search Space: Intuition

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.

Unordered Search Space: Discussion

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

- Consider a state s where $v \in V$ has not been assigned a value. If no solution can be reached from any successor state for the operators $o_{v=a}$ ($a \in \mathrm{dom}(v)$), then no solution can be reached from s.
- There is no point in trying operators $o_{v'=a'}$ for other variables $v' \neq v$ 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.

Ordered Search Spaces

Let $\mathcal{C} = \langle V, \mathrm{dom}, C \rangle$ be a constraint network.

Definition (variable ordering)

A variable ordering of $\mathcal{C}$ 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 $\mathcal{C}$.
The ordered search space of $\mathcal{C}$ along ordering σ is the state space obtained from the unordered search space of $\mathcal{C}$ by restricting each operator $o_{v_i=a_i}$ 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, ...

The Importance of Good Orderings

- 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.

Dead Ends

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.

Backtrack-Free Search Spaces

Definition (backtrack-free)

A state space is called backtrack-free if it contains no dead ends.
A constraint network $\mathcal{C}$ is called backtrack-free along variable ordering σ if the ordered search space of $\mathcal{C}$ along σ is backtrack-free.

Backtrack-Free Networks: Discussion

- 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 enforcing a local consistency property like arc consistency or path consistency, which leads to a tighter network.

Constraint Tightness and Dead Ends

Lemma

Let $\mathcal{C}$ and $\mathcal{C}'$ be equivalent constraint networks.
If $\mathcal{C}'$ is at least as tight as $\mathcal{C}$, then

- *the unordered search space of $\mathcal{C}'$ has at most as many dead ends as the unordered search space of $\mathcal{C}$, and*
- *the ordered search space of $\mathcal{C}'$ along any ordering σ has at most as many dead ends as the ordered search space of $\mathcal{C}$ along the same ordering σ.*

Proof.

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

Global Consistency and Dead Ends

Lemma

Let $\mathcal{C}$ be a constraint network.
The following three statements are equivalent:

- *The unordered search space of $\mathcal{C}$ is backtrack-free.*
- *The ordered search space of $\mathcal{C}$ is backtrack-free along each ordering σ.*
- *$\mathcal{C}$ is globally consistent.*

Reducing Dead Ends Further

- 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.

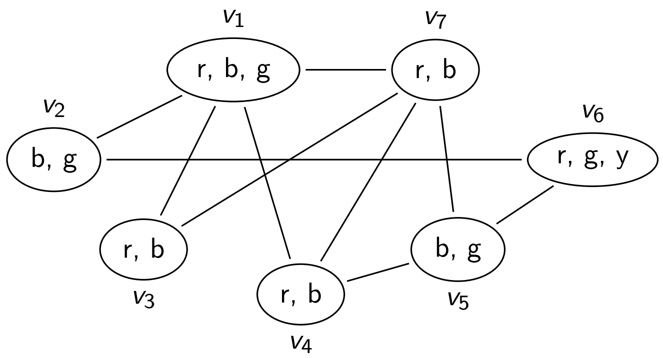
Backtracking

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.

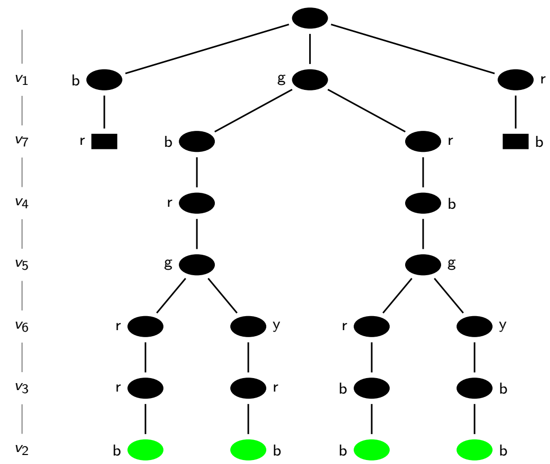
Backtracking: Example

Consider the constraint network defined by the following coloring problem:



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:



Backtracking Algorithm (Recursive Version)

Backtracking$(\mathcal{C}, a)$:

Input: a constraint network $\mathcal{C} = \langle V, D, C \rangle$ and a partial solution a of $\mathcal{C}$ (possible: the empty instantiation $a = \{\}$)
Output: a solution of $\mathcal{C}$ or "inconsistent"

if a is defined for all variables in V:
  return a
else select a variable v_i for which a is not defined
  $D'_i \leftarrow D_i$
  while D'_i is non-empty
    select and delete a value x from D'_i
    $a' := a \cup \{v_i \mapsto x\}$
    if a' is consistent:
      $a'' \leftarrow \text{Backtracking}(\mathcal{C}, a')$
      if a'' is not "inconsistent":
        return a''
  return "inconsistent"

Improvements of Backtracking

- 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 and Look-Back

- Look-ahead: invoked when next variable or next value is selected. For example:
  - Which variable should be instantiated next?
    $\rightsquigarrow$ prefer variables that impose tighter constraints on the rest of the search space
  - Which value should be chosen for the next variable?
    $\rightsquigarrow$ 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?
    $\rightsquigarrow$ 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?
    $\rightsquigarrow$ record reasons for dead ends as new constraints so that the same inconsistencies can be avoided at later stages of the search

Backtracking with Look-Ahead

LookAhead($\mathcal{C}, a$):

Input: a constraint network $\mathcal{C} = \langle V, D, C \rangle$ and
a partial solution a of $\mathcal{C}$
(possible: the empty instantiation $a = \{\,\}$)
Output: a solution of $\mathcal{C}$ or "inconsistent"
SelectValue($v_i, a, \mathcal{C}$): procedure that selects and deletes a consistent value $x \in D_i$; side-effect: $\mathcal{C}$ is refined; returns 0, if all $a \cup \{v_i \mapsto x\}$ are inconsistent

```
if a is defined for all variables in V:
    return a
else select a variable v_i for which a is not defined
    C' ← C, D'_i ← D_i      // (work on a copy)
    while D'_i is non-empty
        x, C' ← SelectValue(v_i, a, C')
        if x ≠ 0:
            a' ← LookAhead(C', a ∪ {v_i ↦ x})
            if a' is not "inconsistent":
                return a'
    return "inconsistent"
```


SelectValue-ForwardChecking

SelectValue-ForwardChecking($v_i, a, \mathcal{C}$):

```
select and delete x from D_i
for each v_j (i ≠ j) for which a is not defined
    D'_j ← D_j          // (work on a copy)
    for each value y ∈ D'_j
        if not consistent(a ∪ {v_i ↦ x, v_j ↦ y})
            remove y from D'_j
    if D'_j is empty  // (v_i ↦ x leads to a dead end)
        return 0
    else D_j ← D'_j   // (propagate refined D_j)
return x
```


SelectValue-ArcConsistency

SelectValue-ArcConsistency($v_i, a, \mathcal{C}$):

```
select and delete x from D_i
repeat
    for each v_j (j ≠ i) for which a is not defined
        D'_j ← D_j          // (work on a copy)
        for each v_k (k ≠ i, j) for which a is not defined
            for each value y ∈ D'_j
                if there is no value z ∈ D_k such that
                        consistent(a ∪ {v_i ↦ x, v_j ↦ y, v_k ↦ z})
                    remove y from D'_j
        if D'_j is empty  // (v_i ↦ x leads to a dead end)
            return 0
        else D_j ← D'_j   // (propagate refined D_j)
until no value was removed
return x
```

SelectValue-FullLookAhead

SelectValue-FullLookAhead($v_i, a, \mathcal{C}$):

```
select and delete x from D_i
for each v_j (j ≠ i) for which a is not defined
    D'_j ← D_j        // (work on a copy)
    for each v_k (k ≠ i, j) for which a is not defined
        for each value y ∈ D'_j
            if there is no value z ∈ D_k such that
                    consistent(a ∪ {v_i ↦ x, v_j ↦ y, v_k ↦ z})
                remove y from D'_j
    if D'_j is empty  // (v_i ↦ x leads to a dead end)
        return 0
    else D_j ← D'_j   // (propagate refined D_j)
return x
```


Further SelectValue Functions

Consistency-based strategies:

- MaintainingArcConsistency (MAC): perform full arc consistency each time after a domain value for v_i has been rejected
- PartialLookAhead (PLA): ...

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 $n_x := \min_{v_j} |D'_j|$ after assigning $v_i \mapsto x$, and n_y for $v_i \mapsto y$, respectively; if $n_x > n_y$, then prefer $v_i \mapsto x$)

Choosing a Variable Order

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

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 σ is fixed in advance.
  - Search is conducted in the ordered search space along σ.

Dynamic Variable Orderings

Common heuristic:

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

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

Static Variable Orderings

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

Static Variable Orderings: Max-Cardinality Ordering

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.

$\rightsquigarrow$ for the other two ordering strategies, we first need to lay some foundations

Ordered Graphs

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 σ 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 $v \prec v'$ iff v precedes v' in σ.

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

Width of a Graph

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.

Graphs of Width 1

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 σ, 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 σ, then nodes with distance 1 from a root node, then distance 2, 3, etc. This ordered graph has width 1. □

Significance of Width

For finding 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.

Constraint Graphs with Width 1

Theorem

Let $\mathcal{C}$ be a constraint network whose primal constraint graph has width 1. Then $\mathcal{C}$ can be solved in polynomial time.

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

Lemma

Let $\mathcal{C}$ be an arc-consistent constraint network whose primal constraint graph has width 1, and where all variable domains are non-empty. Then $\mathcal{C}$ is backtrack-free along any ordering with width 1.

Constraint Graphs with Width 1 (ctd.)

Proof of the lemma.

Let $\mathcal{C}$ be such a constraint network, and let $\sigma = v_1, \dots, v_n$ be a width-1 ordering for $\mathcal{C}$. We must show that all partial solutions of the form $\{v_1 \mapsto a_1, \dots, v_i \mapsto a_i\}$ for $0 \leq i < n$ can be consistently extended to variable v_{i+1}.
Since σ 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 $(v_j \mapsto a_j)$, there must be a consistent choice $(v_{i+1} \mapsto a_{i+1})$ because of arc consistency.

□

Constraint Graphs with Width 1 (ctd.)

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. □

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

Static Variable Orderings: Min-Width Ordering

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

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.

Cycle Cutsets: Remarks

- 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.
  - $\rightsquigarrow$ runtime $\mathcal{O}(k^{|V'|} \cdot p(\|\mathcal{C}\|))$ 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.

Literature

Rina Dechter.
Constraint Processing,
Chapters 4 and 5, Morgan Kaufmann, 2003