Introduction to constraint satisfaction problems

We now return to the idea of problem solving by search and examine it from a slightly different perspective.

Aims:

- To introduce the idea of a *constraint satisfaction problem (CSP)* as a general means of representing and solving problems by search.
- To look at the basic *backtracking algorithm* for solving CSPs.
- To look at some basic heuristics for solving CSPs.

Reading: Russell and Norvig, chapter 5.

Constraint satisfaction problems

The search scenarios examined so far seem in some ways unsatisfactory.

- States were represented using an *arbitrary* and *problem-specific* data structure.
- Heuristics, similarly, were problem-specific.

Constraint satisfaction problems

CSPs *standardise* the manner in which states and goal tests are represented.

- As a result we can devise *general purpose* algorithms and heuristics.
- The form of the goal test can tell us about the structure of the problem.
- Consequently it is possible to introduce techniques for decomposing problems.
- We can also try to understand the relationship between the *structure* of a problem and the *difficulty of solving it*.

Constraint satisfaction problems

We have:

- A set of n *variables* $V_1, V_2, \ldots, V_n$.
- For each V_i, and *domain* D_i specifying the values that V_i can take.
- A set of m *constraints* $C_1, C_2, \ldots, C_m$.

Each constraint C_i involves a set of variables and specifies an allowable collection of values.

- A *state* is an assignment of specific values to some or all of the variables.
- An assignment is *consistent* if it violates no constraints.
- An assignment is *complete* if it gives a value to every variable.

A *solution* is a consistent and complete assignment.

Formulation of CSPs as standard search problems

Clearly a CSP can be formulated as a search problem in the familiar sense:

- **Initial state**: $\{\}$—no variables are assigned.
- **Successor function**: assigns value(s) to currently unassigned variable(s) provided constraints are not violated.
- **Goal**: reached if all variables are assigned.
- **Path cost**: constant c per step.

In addition:

- The tree is limited to depth n so depth-first search is usable.
- We don't mind what path is used to get to a solution, so it is feasible to allow every state to be a complete assignment whether consistent or not. (Local search is a possibility.)

Varieties of CSP

The simplest possible CSP will be *discrete* with *finite domains* and we will concentrate on these.

1. Discrete CSPs with *infinite domains*:
   - will need a *constraint language*. For example
   $$V_3 \leq V_{10} + 5$$
   - Algorithms are available for integer variables and linear constraints.
   - There is *no algorithm* for integer variables and nonlinear constraints.
2. Continuous domains:
   - Using linear constraints defining convex regions we have *linear programming*.
   - This is solvable in polynomial time in n.

Types of constraint

We will concentrate on *binary constraints*.

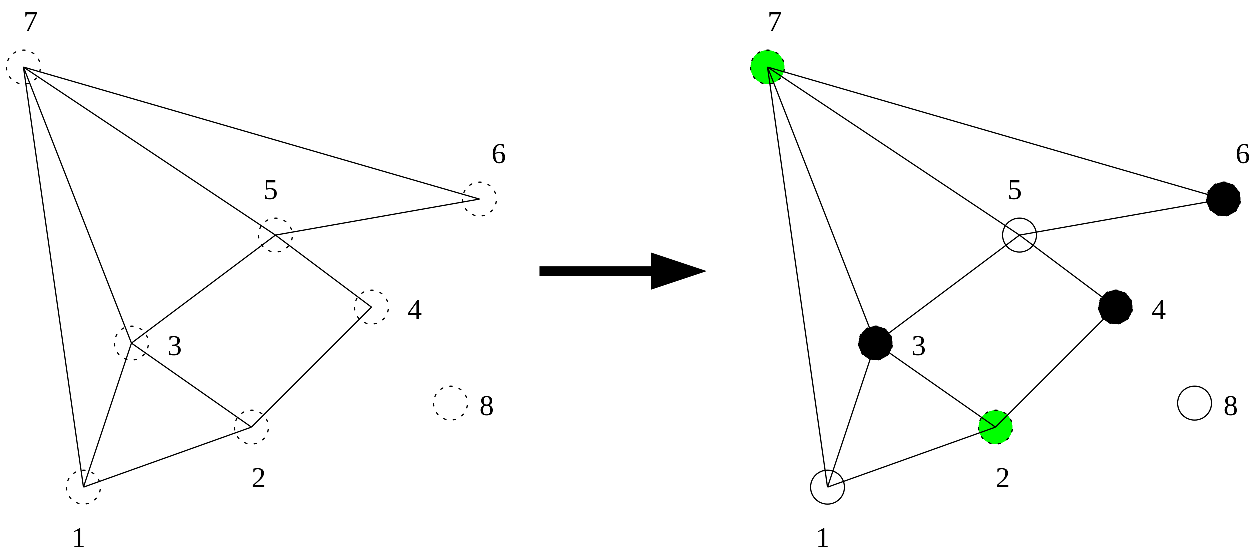
- *Unary constraints* can be removed by adjusting the domains.
- *Higher-order constraints* applying to three or more variables can certainly be considered, but...
- ...when dealing with finite domains they can always be converted to sets of binary constraints by introducing extra *auxiliary variables*.

It is also possible to introduce *preference constraints* in addition to *absolute constraints*.

We may sometimes also introduce an *objective function*.

Example

We will use the problem of colouring the nodes of a graph as an example.



We have three colours and directly connected nodes should have different colours.

Example

This translates easily to a CSP formulation:

- The variables are the nodes

$$V_i = \text{node } i$$

- The domain for each variable contains the values black, white and green (or grey on the printed handout)

$$D_i = \{B, W, G\}$$

- The constraints enforce the idea that directly connected nodes must have different colours. For example, for 1 and 2 the constraints specify

$$(B, W), (B, G), (W, B), (W, G), (G, B), (G, W)$$

Backtracking search

Consider what happens if we try to solve a CSP using a simple technique such as *breadth-first search*.

The branching factor is nd at the first step, for n variables each with d possible values.

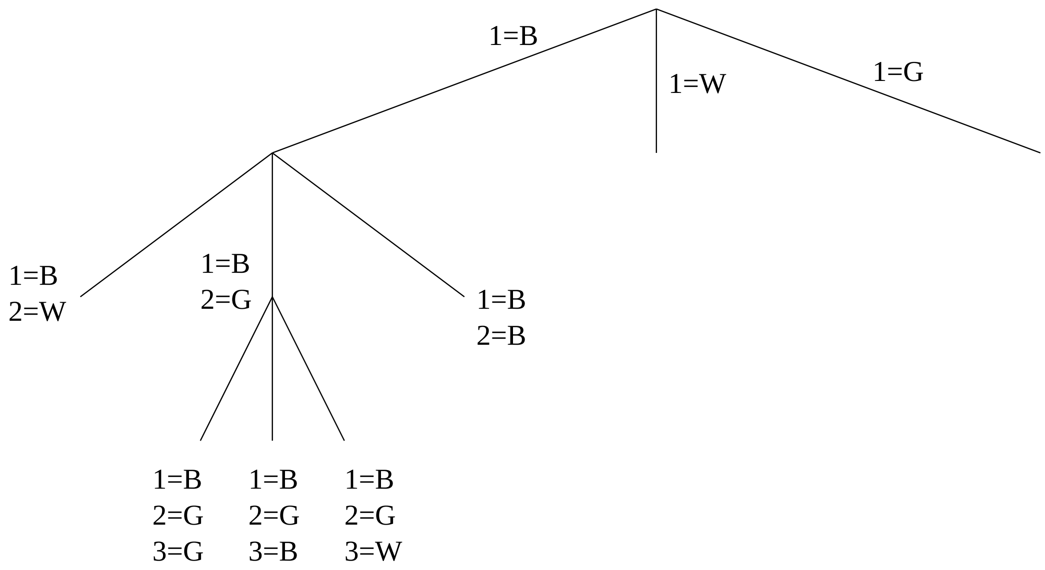
$$\left.\begin{array}{ll} \text{Step 2:} & (n-1)d \\ \text{Step 3:} & (n-2)d \\ & \vdots \\ \text{Step } n\text{:} & 1 \end{array}\right\} \qquad \text{Number of leaves} = nd \times (n-1)d \times \cdots \times 1 = n!d^n$$

BUT: only d^n assignments are possible.

The order of assignment doesn't matter, and we should assign to one variable at a time.

Backtracking search

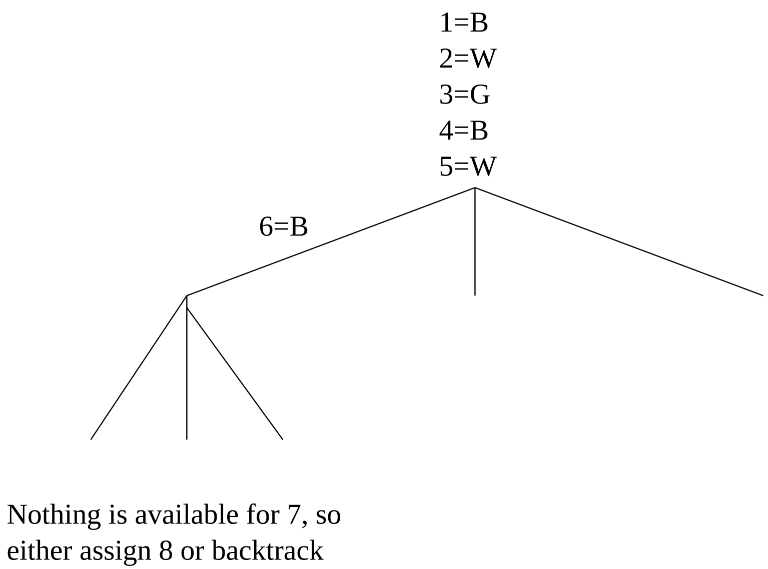
The search now looks something like this...

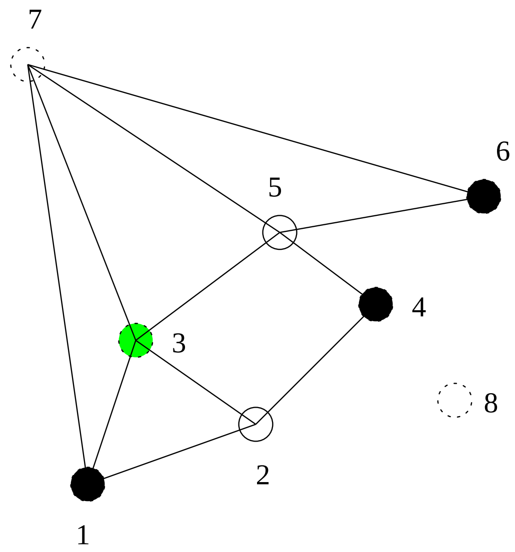


...and new possibilities appear.

Backtracking search

Backtracking search searches depth-first, assigning a single variable at a time, and backtracking if no valid assignment is available.





Rather than using problem-specific heuristics to try to improve searching, we can now explore heuristics applicable to *general* CSPs.

Backtracking search

```
result backtrack(problem)
{
  return bt ([],problem);
}

result bt(assignment_list problem)
{
  if (assignment_list is complete)
    return assignment_list;
  next_var = get_next_var(assignment_list, problem);
  for (every value in order_variables(next_var, assignment_list, problem))
  {
    if (value is consistent with assignment_list)
    {
      add "next_var=value" to assignment_list;
      solution = bt(assignment_list, problem);
      if (solution is not "fail")
        return solution;
      remove "next_var=value" from assignment_list;
    }
  }
  return "fail";
}
```

Backtracking search: possible heuristics

There are several points we can examine in an attempt to obtain general CSP-based heuristics:

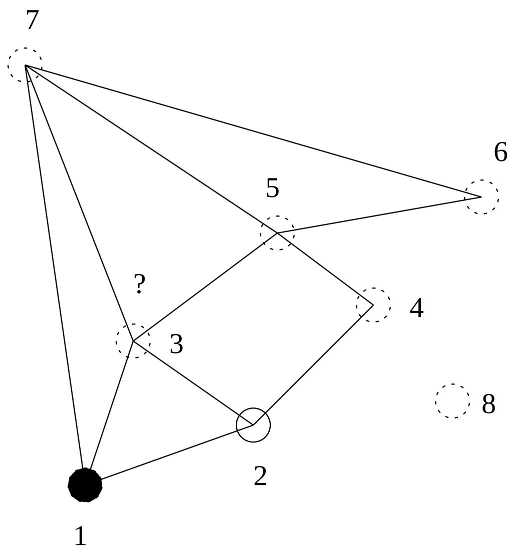
- In what order should we try to assign variables?
- In what order should we try to assign possible values to a variable?

Or being a little more subtle:

- What effect might the values assigned so far have on later attempted assignments?
- When forced to backtrack, is it possible to avoid the same failure later on?

Heuristics I: Choosing the order of variable assignments and values

Say we have $1 = B$ and $2 = W$

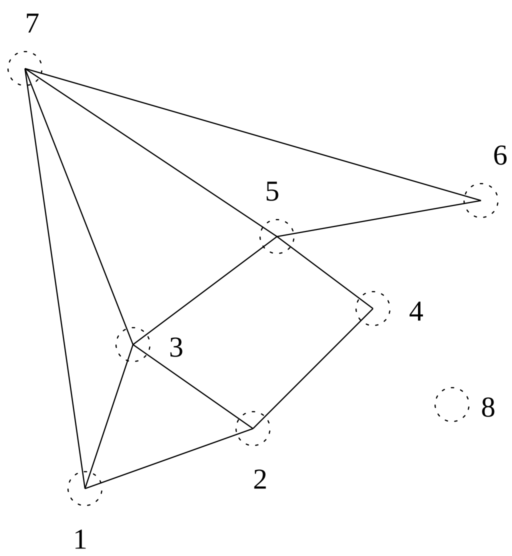


At this point there is only one possible assignment for 3, whereas the others have more flexibility. Assigning such variables *first* is called the *minimum remaining values (MRV)* heuristic. (Alternatively, the *most constrained variable* or *fail first* heuristic.

Heuristics I: Choosing the order of variable assignments and values

How do we choose a variable to begin with?

The *degree heuristic* chooses the variable involved in the most constraints on as yet unassigned variables.



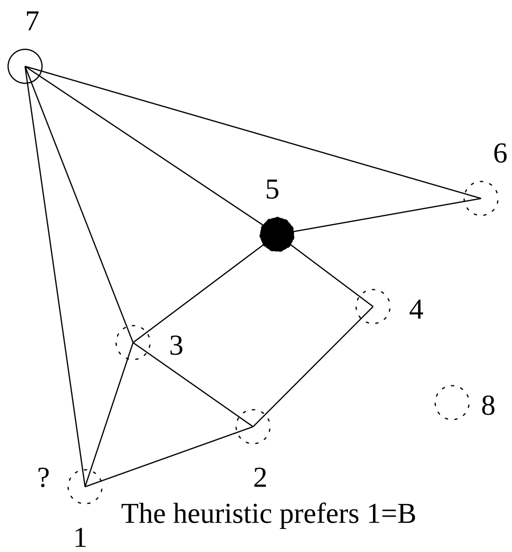
Start with 3, 5 or 7.

MRV is usually better but the degree heuristic is a good tie breaker.

Heuristics I: Choosing the order of variable assignments and values

Once a variable is chosen, in what order should values be assigned?

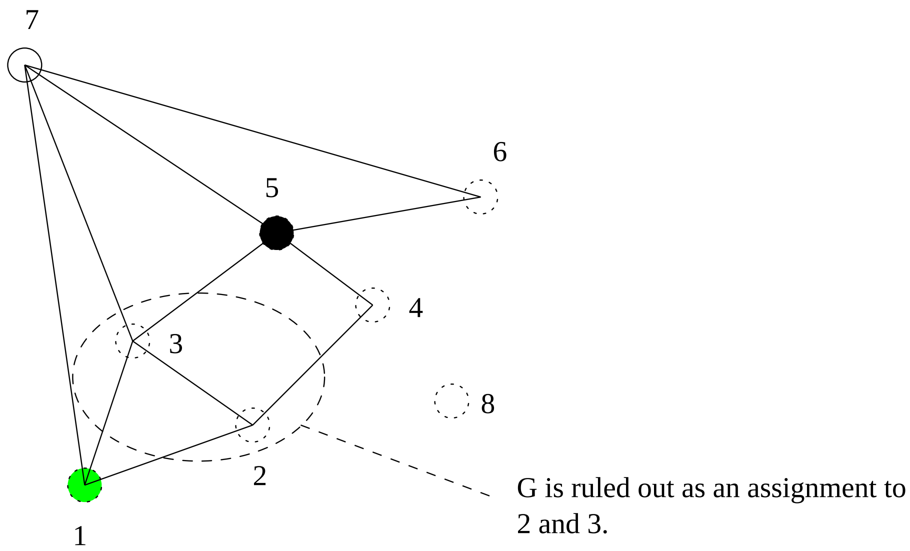
The *least constraining value* heuristic chooses first the value that leaves the maximum possible freedom in choosing assignments for the variable's neighbours.



Choosing $1 = G$ is bad as it removes the final possibility for 3.

Heuristics II: forward checking and constraint propagation

Continuing the previous slide's progress, now add $1 = G$.



Each time we assign a value to a variable, it makes sense to delete that value from the collection of *possible assignments to its neighbours*. This is called *forward checking*. It works nicely in conjunction with MRV.

Heuristics II: forward checking and constraint propagation

We can visualise this process as follows:

	1	2	3	4	5	6	7	8
Start	BWG	BWG	BWG	BWG	BWG	BWG	BWG	BWG
$2 = B$	WG	$= B$	WG	WG	BWG	BWG	BWG	BWG
$3 = W$	G	$= B$	$= W$	WG	BG	BWG	BG	BWG
$6 = B$	G	$= B$	$= W$	WG	G	$= B$	G	BWG
$5 = G$	G	$= B$	$= W$	W	$= G$	$= B$	!	BWG

At the fourth step, 7 has no possible assignments left.

However, we could have detected a problem a little earlier...

Heuristics II: forward checking and constraint propagation

...by looking at step three.

- At step three, 5 can be G only and 7 can be G only.
- But 5 and 7 are connected.
- So we can’t progress, and this hasn’t been detected.
- Ideally we want to do *constraint propagation*.

Trade-off: time to do the search, against time to explore constraints.

Constraint propagation

Arc consistency:

Consider a constraint as being *directed*. For example $4 \rightarrow 5$.

In general, say we have a constraint $i \rightarrow j$ and currently the domain of i is D_i and the domain of j is D_j.

$i \rightarrow j$ is *consistent* if

$$\forall d \in D_i, \exists d' \in D_j \text{ such that } i \rightarrow j \text{ is valid}$$

Example:

In step three of the table, $D_4 = \{W, G\}$ and $D_5 = \{G\}$.

- $5 \to 4$ in step three of the table is consistent.
- $4 \to 5$ in step three of the table is not consistent.

$4 \to 5$ can be made consistent by deleting G from D_4.

Enforcing arc consistency

We can enforce arc consistency each time a variable i is assigned.

- We need to maintain a collection of arcs to be checked.
- Each time we alter a domain, we may have to include further arcs in the collection.

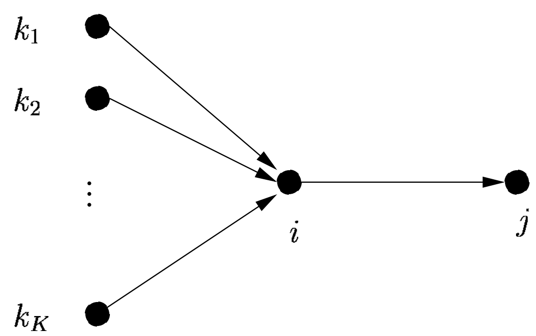
This is because if $i \to j$ is inconsistent, resulting in a deletion from D_i, we may as a consequence make some arc $k \to i$ inconsistent.

Enforcing arc consistency

Why is this?

- $i \to j$ inconsistent means removing a value from D_i.
- $\exists d \in D_i$ such that there is no valid $d' \in D_j$.
- So delete $d \in D_i$.

However some $d'' \in D_k$ may only previously have been pairable with d.



We need to continue until all consequences are taken care of.

Enforcing arc consistency

Complexity:

- A binary CSP with n variables can have $O(n^2)$ directional constraints $i \rightarrow j$.
- Any $i \rightarrow j$ can be considered at most d times where $d = \max_k |D_k|$ because only d things can be removed from D_i.
- Checking any single arc for consistency can be done in $O(d^2)$.

So the complexity is $O(n^2 d^3)$.

Note: this setup includes 3SAT.

Consequence: we can't check for consistency in polynomial time. Which suggests this doesn't guarantee to find all inconsistencies.

The AC-3 algorithm

```
new_domains AC-3 (problem)
{
  queue to_check = all arcs i->j;
  while (to_check is not empty)
  {
    i->j = next(to_check);
    if (remove_inconsistencies(Di,Dj))
    {
      for (each k that is a neighbour of i)
        add k->i to to_check;
    }
  }
}
```

The AC-3 algorithm

```
bool remove_inconsistencies (domain1, domain2)
{
  bool result = false;
  for (each d in domain1)
  {
    if (no d’ in domain2 valid with d)
    {
      remove d from domain1;
      result = true;
    }
  }
  return result;
}
```

A more powerful form of consistency

We can define a stronger notion of consistency as follows:

Given:

- Any $k-1$ variables and,
- any consistent assignment to these.

Then:

- We can find a consistent assignment to any kth variable.

This is known as k-*consistency*.

A more powerful form of consistency

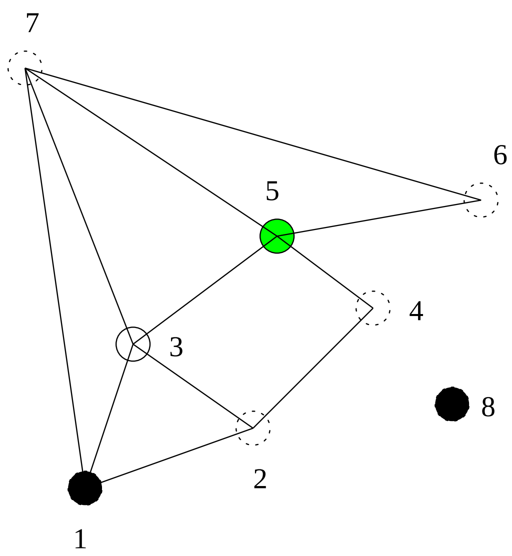
Strong k-consistency requires the we be k-consistent, $k-1$-consistent *etc* as far down as 1-consistent.

If we can demonstrate strong n-consistency (where as usual n is the number of variables) then an assignment can be found in $O(nd)$.

Unfortunately, demonstrating strong n-consistency will be worst-case exponential.

Backjumping I

The basic backtracking algorithm backtracks to the most recent assignment. This is known as *chronological backtracking*. It is not always the best policy:



Say we've done $1 = B$, $3 = W$, $5 = G$ and $8 = B$ and now we want to do 7. This isn't possible so we backtrack, however re-assigning 8 clearly doesn't help.

Backjumping I

Backjumping backtracks to the *conflict set*, which in this case is $\{7\}$:

$$\text{conflict}(x) = \text{set of currently assigned variables connected to } x$$

This can be done by accumulating the sets conflict(x) as we make assignments.

Backjumping I

If forward checking is in operation it can be used to find conflict sets.

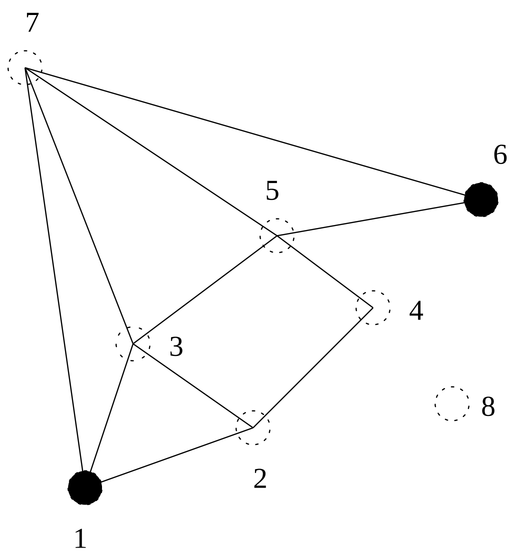
Say we're assigning to x, say $x = v$:

- Forward checking removes v from the D_i of all x_i connected to x.
- Then x needs to be added to conflict(x_i).
- If the *last* member of D_i is ever removed then we need to add *all* of conflict(x_i) to conflict(x).

In fact, use of forward checking turns out to make backjumping redundant.

Backjumping II

In the current example, only two assignments are needed to doom the process:



Next we can assign $8, 3, 7$ and 4, but then 5 fails.

This can never work because 1 and 6 prevent us from getting an assignment for $3, 7, 4$ and 5.

Backjumping II

In this example $\{3, 7, 4, 5\}$ as a *collection* are prevented by 1 and 6 from having an assignment.

We can redefine conflict(x) to be the collection of preceding variables causing x and any subsequent variables not to have a valid set of assignments.

Using the new concept for conflict(x) gives us *conflict-directed backjumping*:

When backtracking from x' to x:

$$\text{conflict}(x) = \text{conflict}(x) \cup (\text{conflict}(x') - x)$$

so that the causes of failure *after* x are maintained.