Artificial Intelligence I

Dr Sean Holden

Notes on *constraint satisfaction problems (CSPs)*

Constraint satisfaction problems (CSPs)

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

- States were represented using an *arbitrary* and *problem-specific* data structure.
- Heuristics were also *problem-specific*.
- It would be nice to be able to *transform* general search problems into a *standard format*.

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

Constraint satisfaction problems (CSPs)

By standardising like this we benefit in several ways:

- We can devise *general purpose* algorithms and heuristics.
- We can look at general methods for exploring the *structure* of the problem.
- Consequently it is possible to introduce techniques for *decomposing* problems.
- We can try to understand the relationship between the *structure* of a problem and the *difficulty of solving it*.

Note: another method of interest in AI that allows us to do similar things involves transforming to a *propositional satisfiability* problem. We'll see an example of this in AI II.

Introduction to constraint satisfaction problems

We now return to the idea of problem solving by search and examine it from this new 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 a *backtracking algorithm* for solving CSPs.
- To look at some *general heuristics* for solving CSPs.
- To look at *more intelligent ways of backtracking*.

Reading: Russell and Norvig, chapter 5.

Constraint satisfaction problems

We have:

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

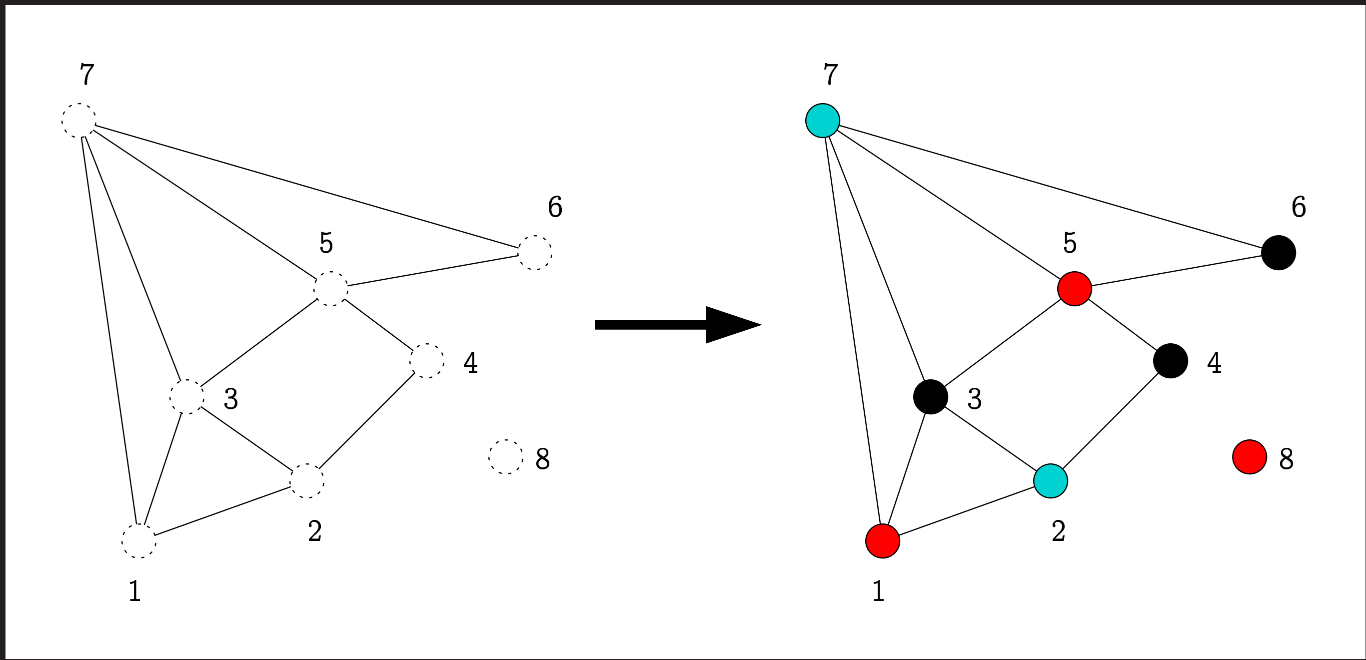
Each constraint C_i involves a set of variables and specifies an *allowable collection of values*.

- 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.

Example

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



Each node corresponds to a *variable*. 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, red and cyan

$$D_i = \{B, R, C\}$$

- The constraints enforce the idea that directly connected nodes must have different colours. For example, for variables V_1 and V_2 the constraints specify

$$(B, R), (B, C), (R, B), (R, C), (C, B), (C, R)$$

- Variable V_8 is unconstrained.

Different kinds of CSP

This is an example of the simplest kind of CSP: it is *discrete* with *finite domains*. We will concentrate on these.

We will also concentrate on *binary constraints*; that is, constraints between *pairs of variables*.

- Constraints on single variables—*unary constraints*—can be handled by adjusting the variable's domain. For example, if we don't want V_i to be *red*, then we just remove that possibility from D_i.
- *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*.

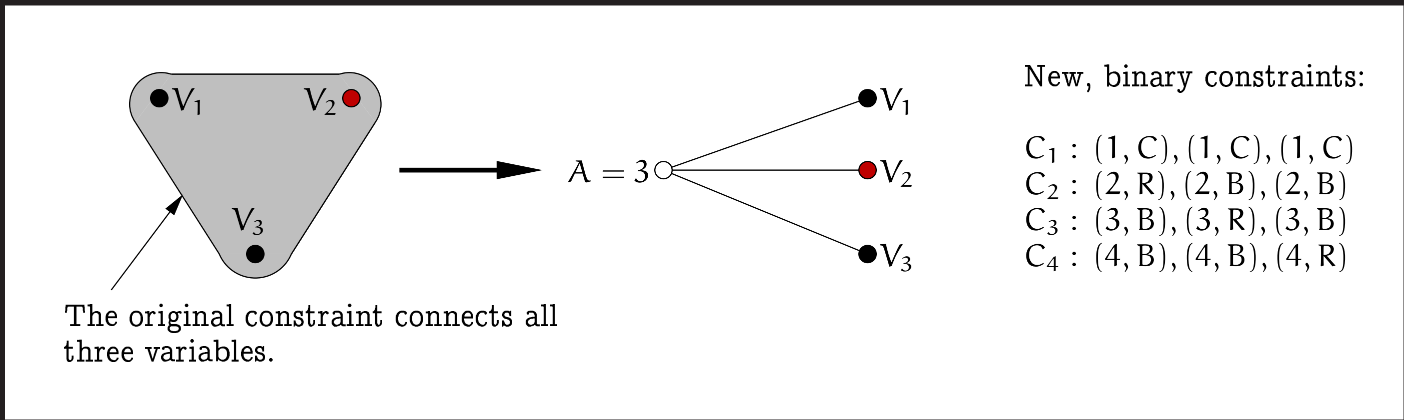
How does that work?

Auxiliary variables

Example: three variables each with domain $\{B, R, C\}$.

A single constraint

$$(C, C, C), (R, B, B), (B, R, B), (B, B, R)$$



Introducing auxiliary variable A with domain $\{1, 2, 3, 4\}$ allows us to convert this to a set of binary constraints.

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.

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

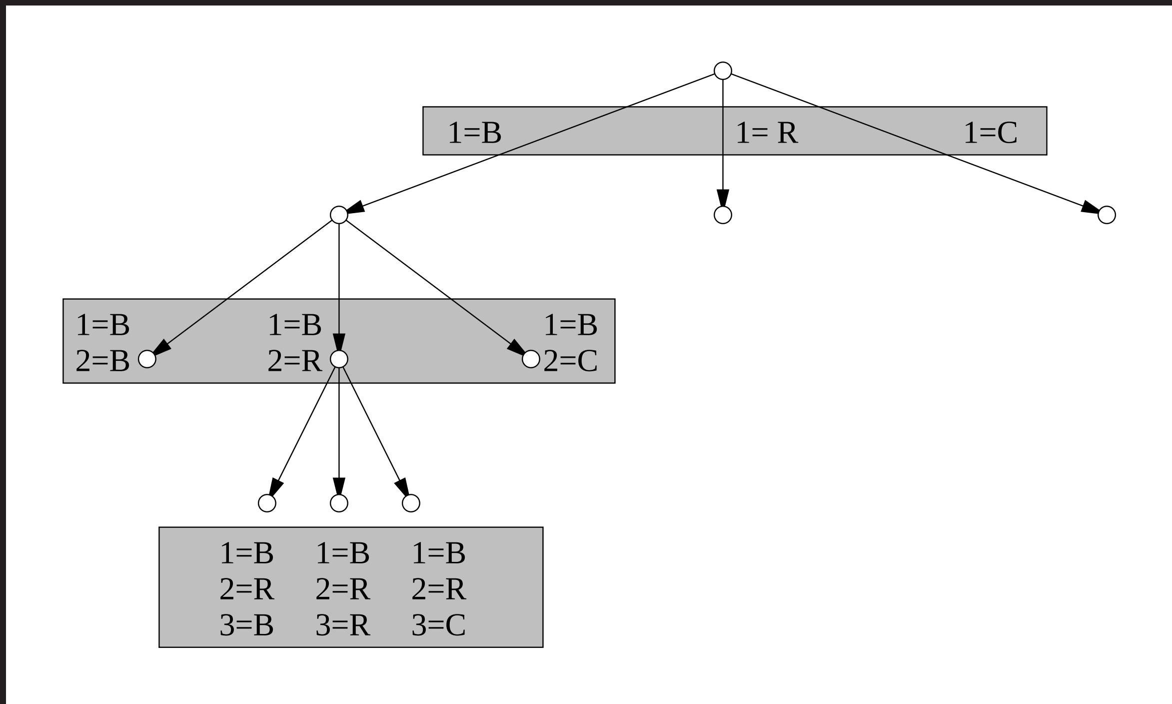
BUT: only d^n assignments are possible.

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

Backtracking search

Using the graph colouring example:

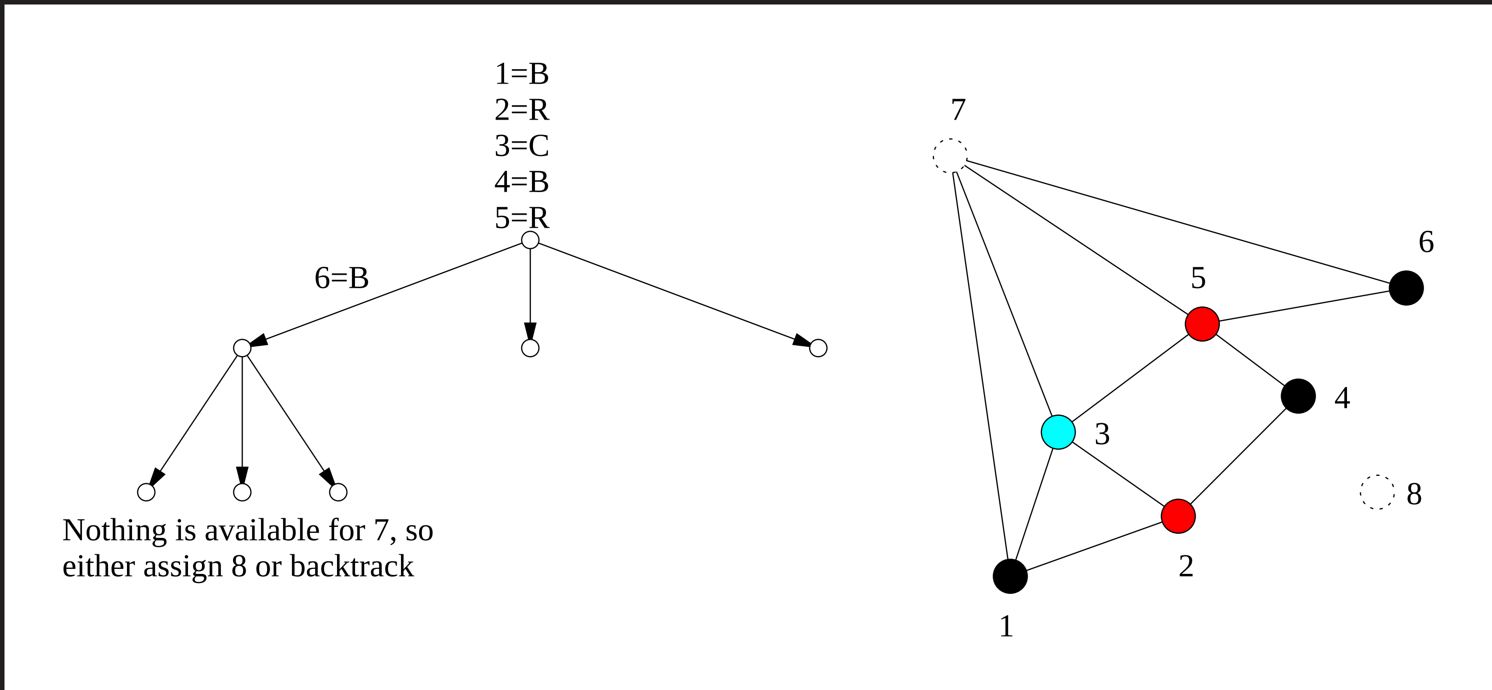
The search now looks something like this...



...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(assignmentList, problem)
{
  if (assignmentList is complete)
    return assignmentList;
  nextVar = getNextVar(assignmentList, problem);
  for (every value v in orderVariables(nextVar, assignmentList, problem))
  {
    if (v is consistent with assignmentList)
    {
      add "nextVar = v" to assignmentList;
      solution = bt(assignmentList, problem);
      if (solution is not "fail")
        return solution;
      remove "nextVar = v" from assignmentList;
    }
  }
  return "fail";
}
```

Backtracking search: possible heuristics

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

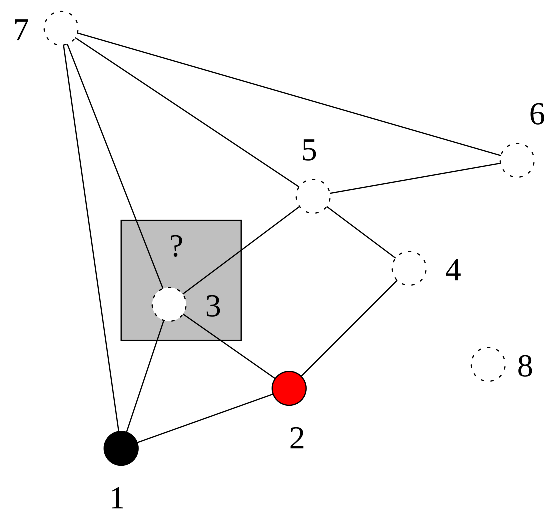
- In what order should we try to *assign variables*?
- In what order should we try to *assign possible values* to a variable?

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 $1 = \mathrm{B}$ and $2 = \mathrm{R}$



At this point there is *only one possible assignment* for 3, whereas the others have more flexibility.

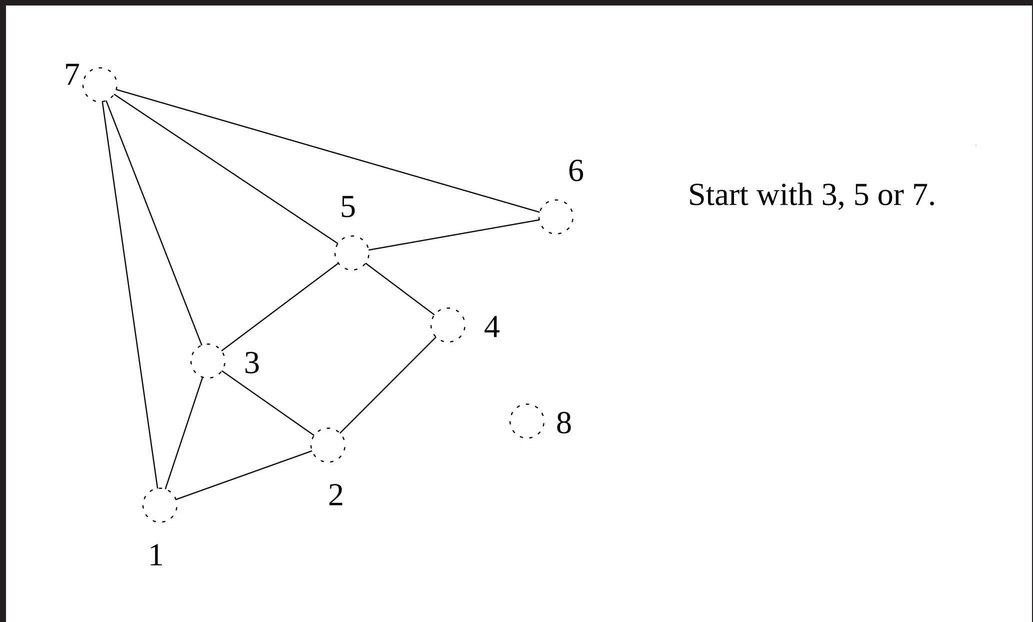
Assigning such variables *first* is called the *minimum remaining values (MRV)* heuristic.

(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?

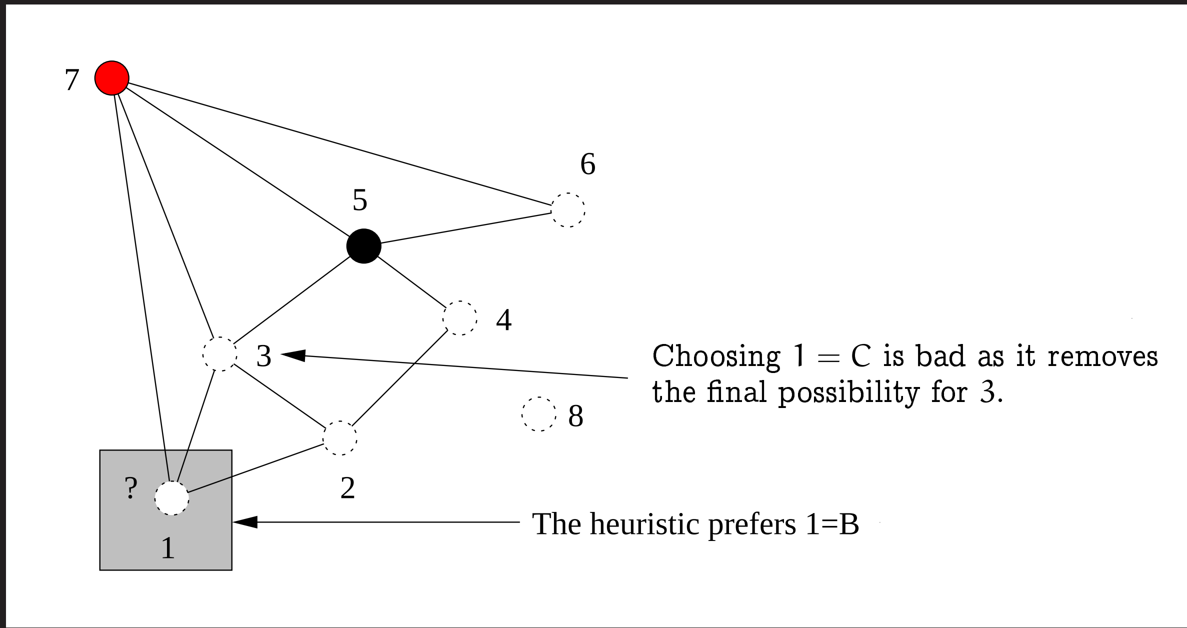
The *degree heuristic* chooses the variable involved in the most constraints on as yet unassigned variables.



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

Heuristics I: Choosing the order of variable assignments and values

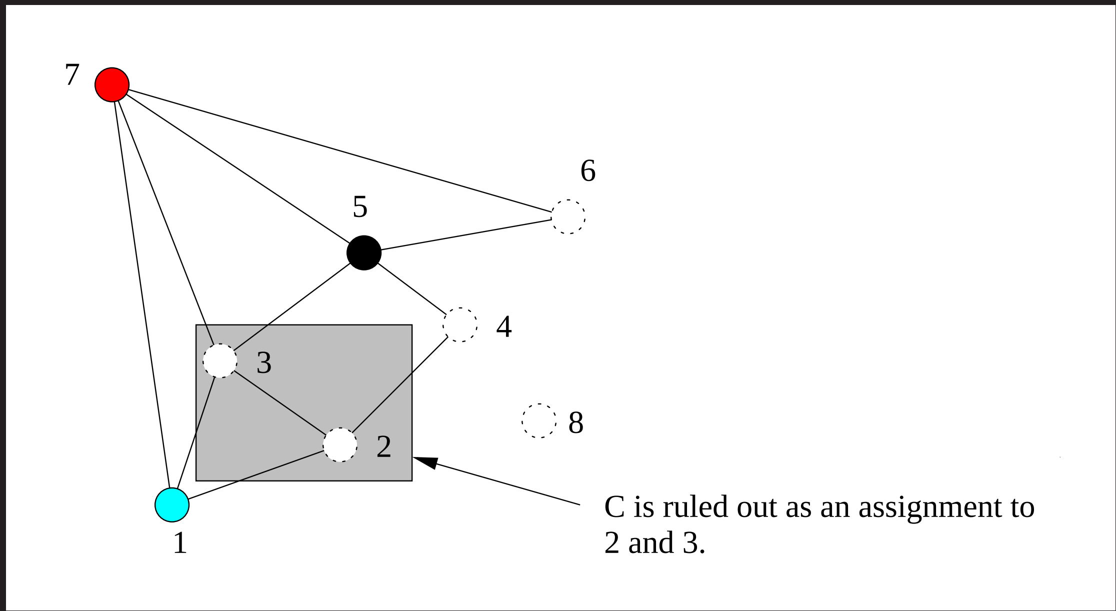
Once a variable is chosen, in *what order should values be assigned*?



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

Heuristics II: forward checking and constraint propagation

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



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	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
$2 = B$	RC	$= B$	RC	RC	BRC	BRC	BRC	BRC
$3 = R$	C	$= B$	$= R$	RC	BC	BRC	BC	BRC
$6 = B$	C	$= B$	$= R$	RC	C	$= B$	C	BRC
$5 = C$	C	$= B$	$= R$	R	$= C$	$= B$	!	BRC

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.

	1	2	3	4	5	6	7	8
Start	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
$2 = B$	RC	$= B$	RC	RC	BRC	BRC	BRC	BRC
$3 = R$	C	$= B$	$= R$	RC	BC	BRC	BC	BRC
$6 = B$	C	$= B$	$= R$	RC	C	$= B$	C	BRC
$5 = C$	C	$= B$	$= R$	R	$= C$	$= B$	!	BRC

- At step three, 5 can be C only and 7 can be C only.
- But 5 and 7 are connected.
- So we can't progress, but 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 $4 \longrightarrow 5$.

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

$i \longrightarrow j$ is *consistent* if

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

Constraint propagation

Example:

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

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

$4 \longrightarrow 5$ can be made consistent by deleting C from D_4.

Or in other words, regardless of what you assign to i you'll be able to find something valid to assign to j.

Enforcing arc consistency

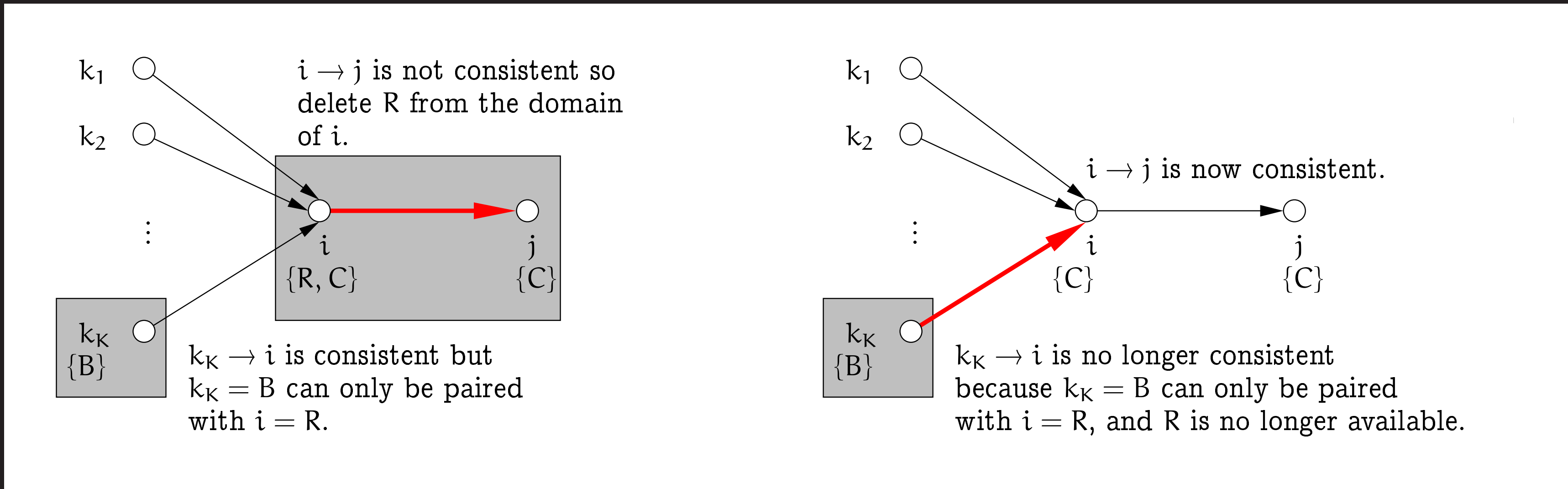
We can enforce arc consistency each time a variable i is assigned.

- 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 $i \longrightarrow j$ is inconsistent resulting in a deletion from D_i we may as a consequence make some arc $k \longrightarrow i$ inconsistent.

Why is this?

Enforcing arc consistency



- $i \to j$ 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 have been pairable with d.

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

The AC-3 algorithm

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

Bool removeInconsistencies (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;
}
```

Enforcing arc consistency

Complexity:

- A binary CSP with n variables can have $O(n^2)$ directional constraints $i \longrightarrow j$.
- Any $i \longrightarrow j$ 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.

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*.

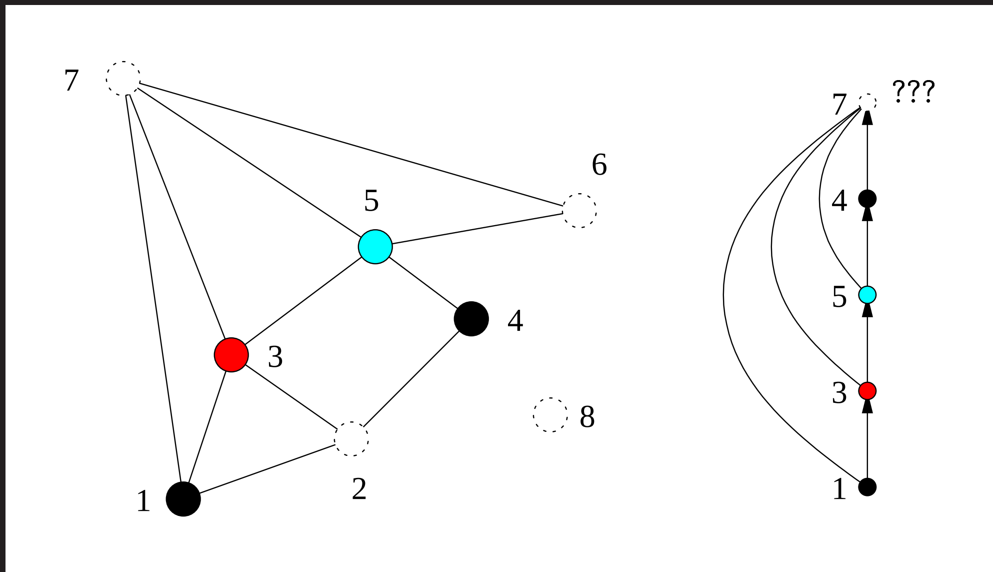
Strong k-*consistency* requires the we be k-consistent, $k-1$-consistent *etc* as far down as 1-consistent.

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

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 assigned $1 = B$, $3 = R$, $5 = C$ and $4 = B$ and now we want to assign something to 7. This isn't possible so we backtrack, however re-assigning 4 clearly doesn't help.

Backjumping

With some careful bookkeeping it is often possible to *jump back multiple levels* without sacrificing the ability to find a solution.

We need some definitions:

- When we set a variable V_i to some value $d \in D_i$ we refer to this as the *assignment* $A_i = (V_i \leftarrow d)$.
- A *partial instantiation* $I_k = \{A_1, A_2, \ldots, A_k\}$ is a *consistent* set of assignments to the first k variables...
- ... where *consistent* means that no constraints are violated.

Henceforth we shall assume that variables are assigned in the order $V_1, V_2, \ldots, V_n$ when formally presenting algorithms.

Gaschnig's algorithm

Gaschnig's algorithm works as follows. Say we have a partial instantiation I_k:

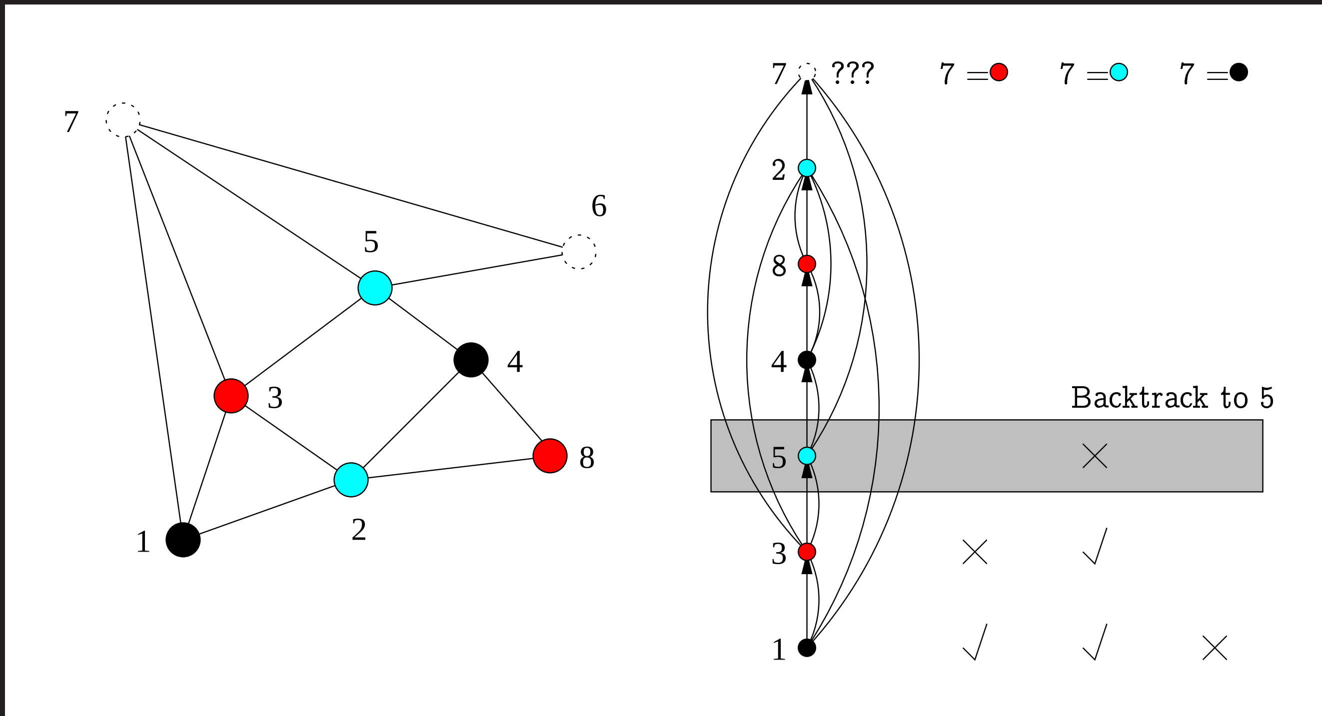
- When choosing a value for V_{k+1} we need to check that any candidate value $d \in D_{k+1}$, is consistent with I_k.
- When testing potential values for d, we will generally discard one or more possibilities, because they conflict with some member of I_k
- We keep track of the *most recent assignment* A_j for which this has happened.

Finally, if *no* value for V_{k+1} is consistent with I_k then we backtrack to V_j.

If there are no possible values left to try for V_j then we backtrack *chronologically*.

Gaschnig's algorithm

Example:



If there's no value left to try for 5 then backtrack to 3 and so on.

Graph-based backjumping

This allows us to jump back multiple levels *when we initially detect a conflict*.

Can we do better than chronological backtracking *thereafter*?

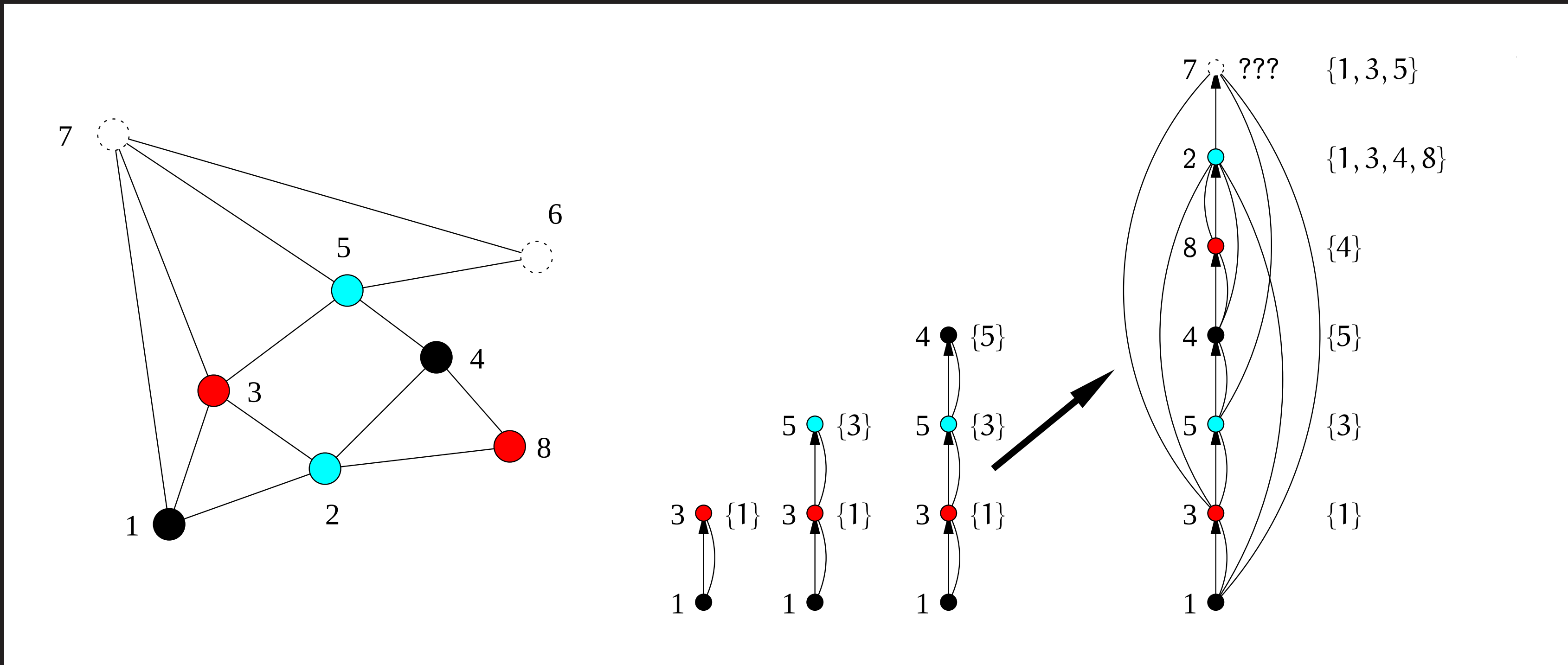
Some more definitions:

- We assume an ordering $V_1, V_2, \ldots, V_n$ for the variables.
- Given $V' = \{V_1, V_2, \ldots, V_k\}$ where $k < n$ the *ancestors* of V_{k+1} are the members of V' connected to V_{k+1} by a constraint.
- The *parent* $P(V)$ of V_{k+1} is its most recent ancestor.

The ancestors for each variable can be accumulated as assignments are made.

Graph-based backjumping backtracks to the *parent* of V_{k+1}.

Graph-based backjumping



At this point, backjump to the *parent* for 7, which is 5.

Backjumping and forward checking

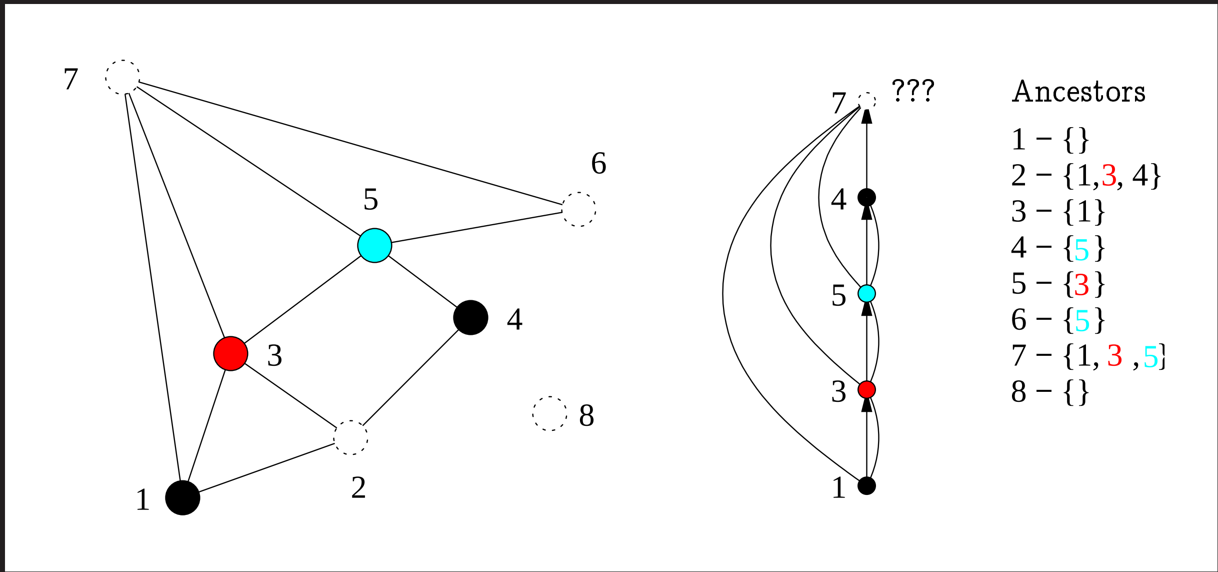
If we use *forward checking*: say we're assigning to V_{k+1} by making $V_{k+1} = d$:

- Forward checking removes d from the D_i of *all* V_i connected to V_{k+1} by a constraint.
- When doing graph-based backjumping, we'd also add V_{k+1} to the ancestors of V_i.

In fact, use of forward checking can make some forms of backjumping *redundant*.

Note: there are in fact many ways of combining *constraint propagation* with *backjumping*, and we will not explore them in further detail here.

Backjumping and forward checking

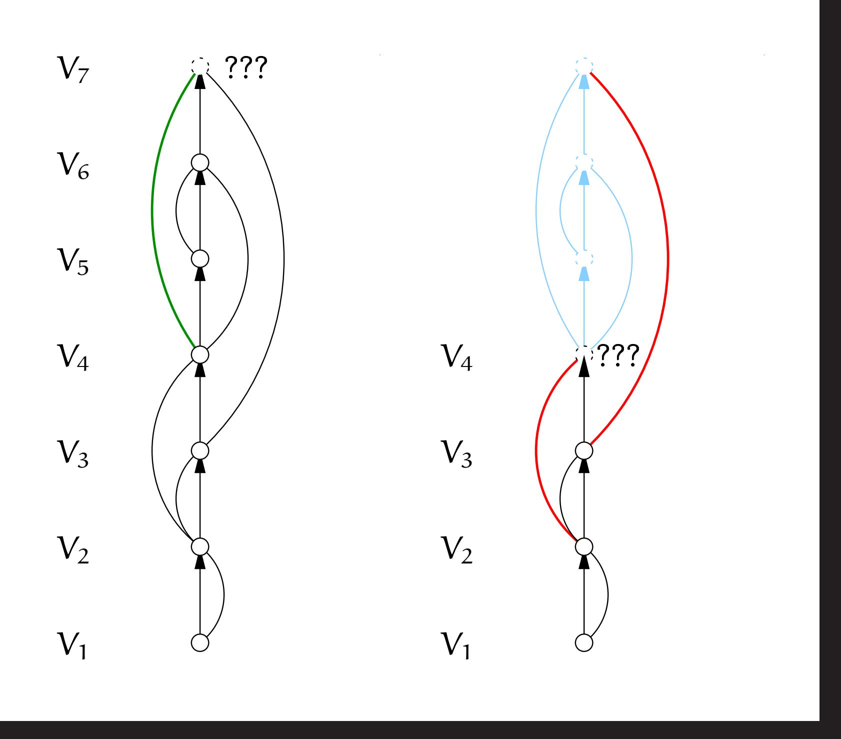


	1	2	3	4	5	6	7	8
Start	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
1 = B	= B	RC	RC	BRC	BRC	BRC	RC	BRC
3 = R	= B	C	= R	BRC	BC	BRC	C	BRC
5 = C	= B	C	= R	BR	= C	BR	!	BRC
4 = B	= B	C	= R	BR	= C	BR	!	BRC

Forward checking finds the problem *before backtracking does*.

Graph-based backjumping

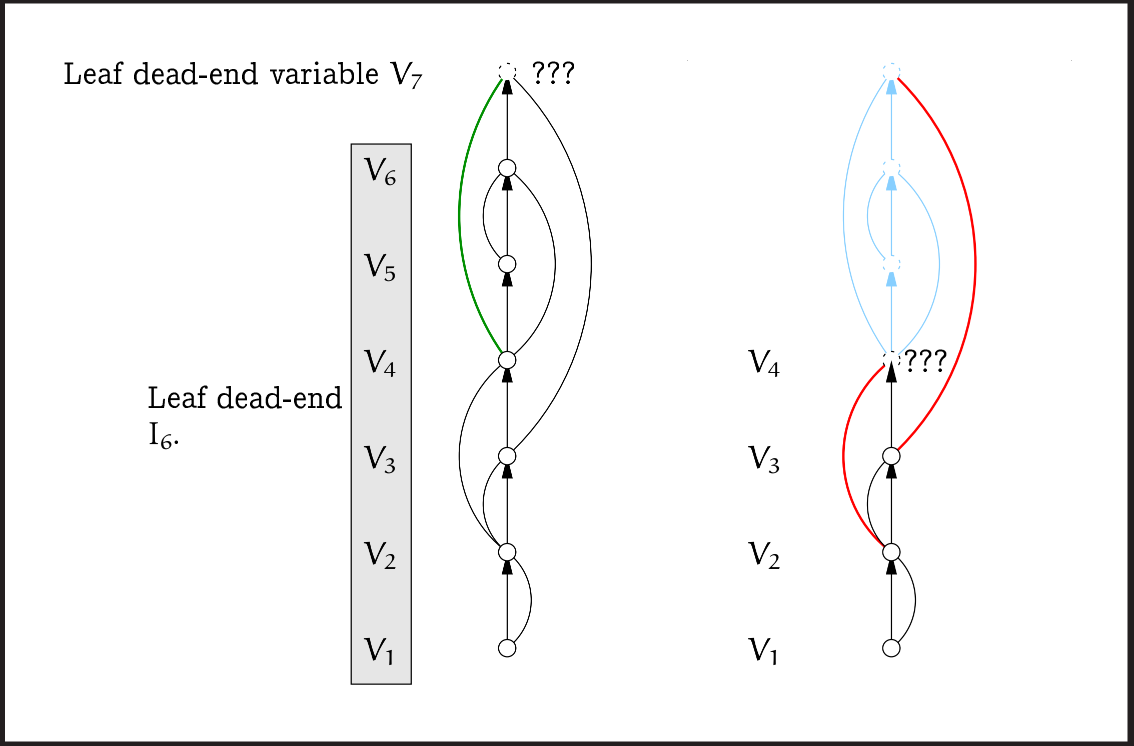
We're not quite done yet though. What happens when *there are no assignments left for the parent we just backjumped to*?



Backjumping from V_7 to V_4 is fine. However we shouldn't then just backjump to V_2, because changing V_3 could fix the problem at V_7.

Graph-based backjumping

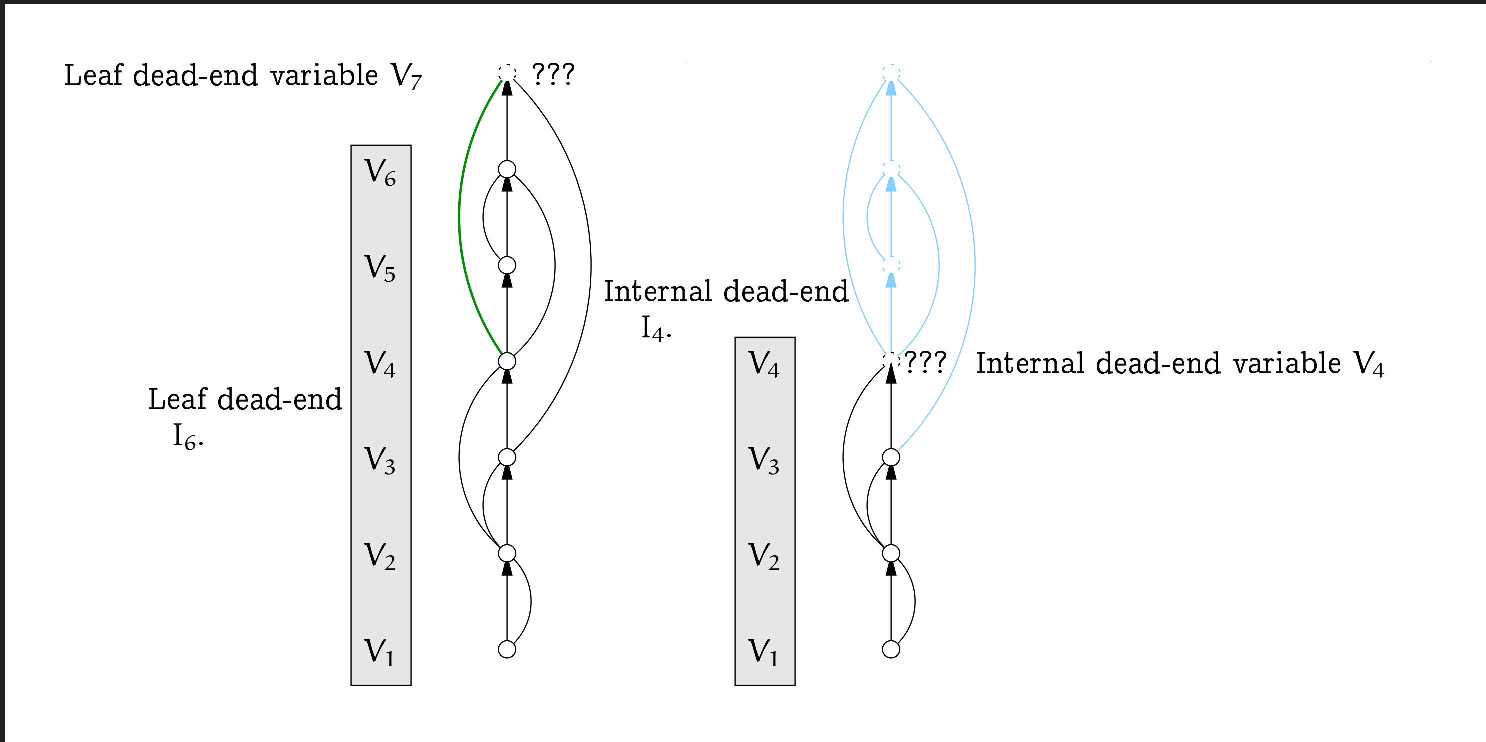
To describe an algorithm in this case is a little involved.



Given an instantiation I_k and V_{k+1}, if there is no consistent $d \in D_{k+1}$ we call I_k a *leaf dead-end* and V_{k+1} a *leaf dead-end variable*.

Graph-based backjumping

Also



If V_i was backtracked to from a later leaf dead-end and there are no more values to try for V_i then we refer to it as an *internal dead-end variable* and call I_{i-1} an *internal dead-end*.

Graph-based backjumping

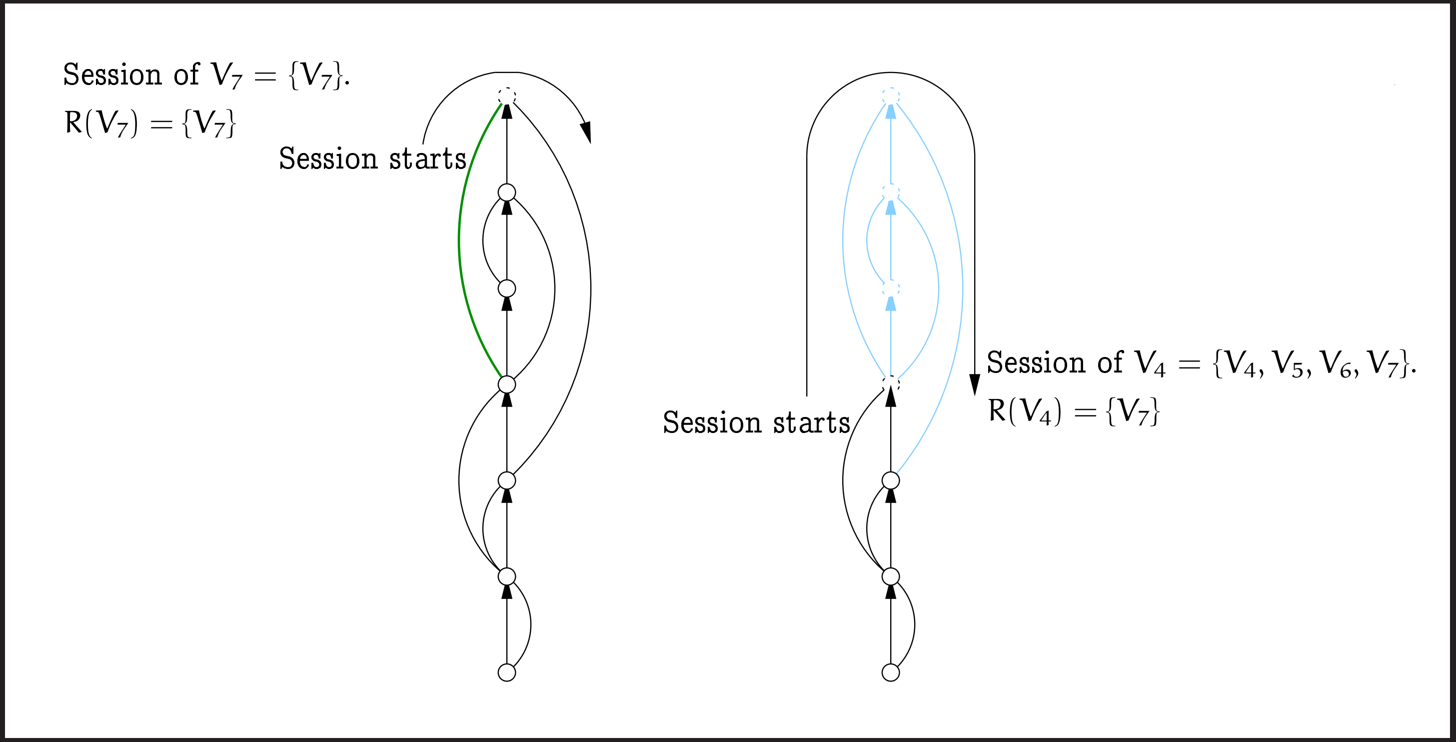
To keep track of exactly where to jump to we also need the definitions:

- The *session* of a variable V begins when the search algorithm visits it and ends when it backtracks through it to an earlier variable.
- The *current session* of a variable V is the set of all variables visiting during its session.
- In particular, the current session for any V contains V.
- The *relevant dead-ends for the current session* $\mathsf{R}(V)$ for a variable V are:
  1. If V is a leaf dead-end variable then $\mathsf{R}(V) = \{V\}$.
  2. If V was backtracked to from a dead-end V' then $\mathsf{R}(V) = \mathsf{R}(V) \cup \mathsf{R}(V')$.

And we're not done yet...

Graph-based backjumping

Example:



As expected, the relevant dead-end for V_4 is $\{V_7\}$.

Graph-based backjumping

One more bunch of definitions before the pain stops. Say V_k is a dead-end:

- The *induced ancestors* $\text{ind}(V_k)$ of V_k are defined as

$$\text{ind}(V_k) = \{V_1, V_2, \ldots, V_{k-1}\} \cap \left(\bigcup_{V \in R(V_k)} \text{ancestors}(V) \right)$$

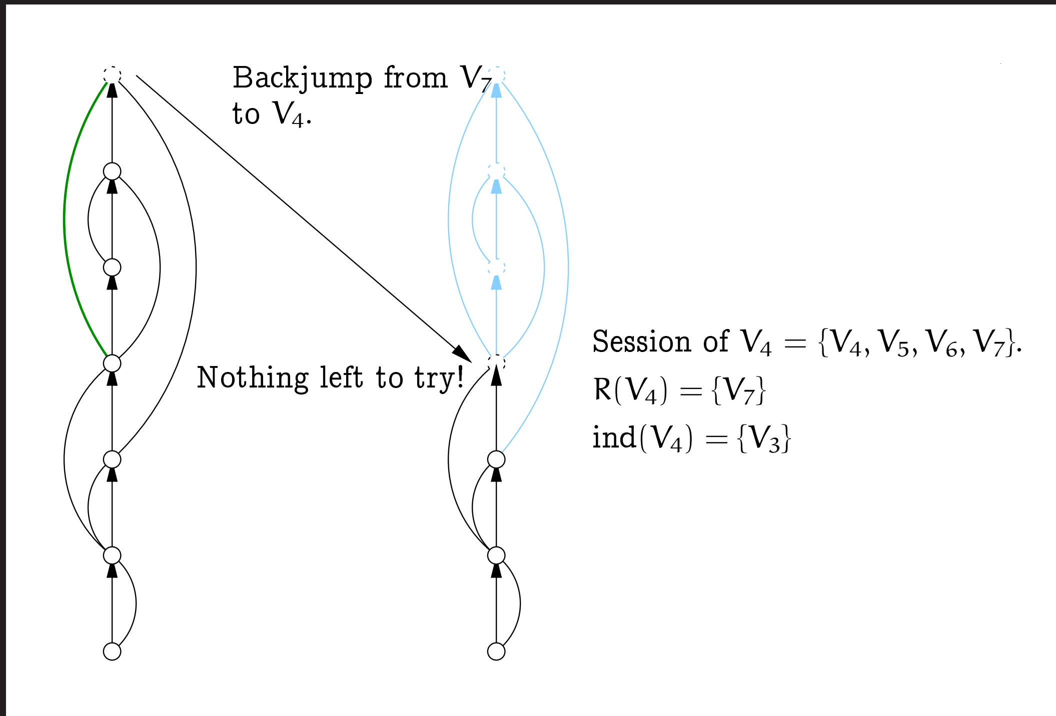
- The *culprit* for V_k is the most recent $V' \in \text{ind}(V_k)$.

Note that these definitions depend on $R(V_k)$.

FINALLY: graph-based backjumping *backjumps to the culprit*.

Graph-based backjumping

Example:



As expected, we back jump to V_3 instead of V_2. Hooray!

Conflict-directed backjumping

Gaschnig's algorithm and graph-based backjumping can be *combined* to produce *conflict-directed backjumping*.

We will not explore conflict-directed backjumping in this course.

For considerable further detail on algorithms for CSPs see:

"Constraint Processing," Rina Dechter. Morgan Kaufmann, 2003.

Varieties of CSP

We have only looked at *discrete* CSPs with *finite domains*. These are the simplest. We could also consider:

1. Discrete CSPs with *infinite domains*:
   - We 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.
3. We can introduce *preference constraints* in addition to *absolute constraints*, and in some cases an *objective function*.