Artificial Intelligence

Games (adversarial search)

Reading: AIMA chapter 5.

Solving problems by search: playing games

How might an agent act when *the outcomes of its actions are not known* because an *adversary is trying to hinder it*?

- This is essentially a more realistic kind of search problem because we do not know the exact outcome of an action.
- This is a common situation when *playing games*: in chess, draughts, and so on an opponent *responds* to our moves.

Game playing has been of interest in AI because it provides an *idealisation* of a world in which two agents act to *reduce* each other's well-being.

We now look at:

- How game-playing can be modelled as *search*.
- The *minimax algorithm* for game-playing.
- Some problems inherent in the use of minimax.
- The concept of $\alpha - \beta$ *pruning*.

Playing games: search against an adversary

Despite the fact that games are an idealisation, game playing can be an excellent source of hard problems. For instance with chess:

- The average branching factor is roughly 35.
- Games can reach 50 moves per player.
- So a rough calculation gives the search tree 35^{100} nodes.
- Even if only different, legal positions are considered it's about 10^{40}.

So: in addition to the uncertainty due to the opponent:

- We can't make a complete search to find the best move...
- ... so we have to act even though we're not sure about the best thing to do.

And chess isn't even very hard: *Go* is *much* harder...

Note: yes, more advanced learning-based methods have conquered chess and Go, but that's an entirely different approach with its own pros and cons.

Perfect decisions in a two-person game

Say we have two players. Traditionally, they are called *Max* and *Min* for reasons that will become clear.

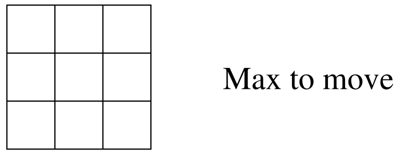
- We'll use *noughts and crosses* as an initial example.
- *Max* moves first.
- The players alternate until the game ends.
- At the end of the game, prizes are awarded. (Or punishments administered—EVIL ROBOT is starting up his favourite chainsaw...)

This is exactly the same game format as chess, Go, draughts and so on.

Perfect decisions in a two-person game

Games like this can be modelled as search problems as follows:

- There is an *initial state*.



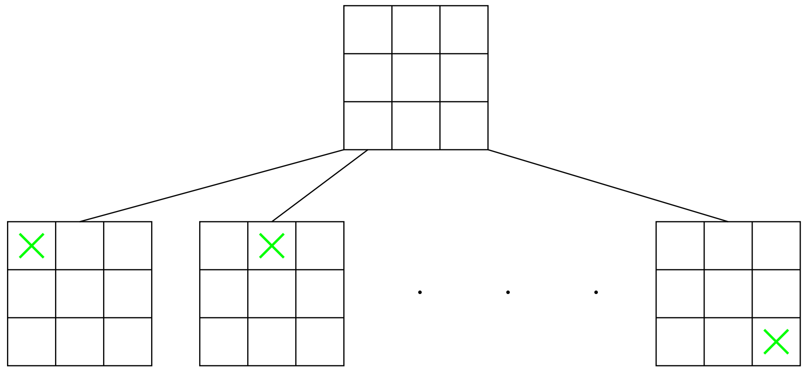
- There is a set of *operators*. Here, *Max* can place a cross in any empty square, or *Min* a nought.
- There is a *terminal test*. Here, the game ends when three noughts or three crosses are in a row, or there are no unused spaces.
- There is a *utility* or *payoff* function. This tells us, numerically, what the outcome of the game is.

This is enough to model the entire game.

Perfect decisions in a two-person game

We can *construct a tree* to represent a game.

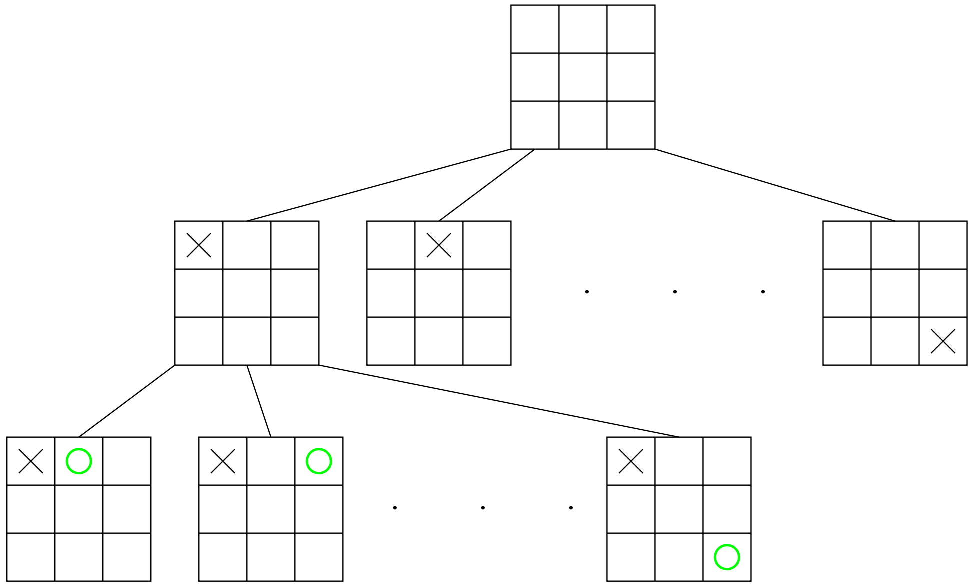
From the initial state *Max* can make nine possible moves:



Then it's *Min*'s turn...

Perfect decisions in a two-person game

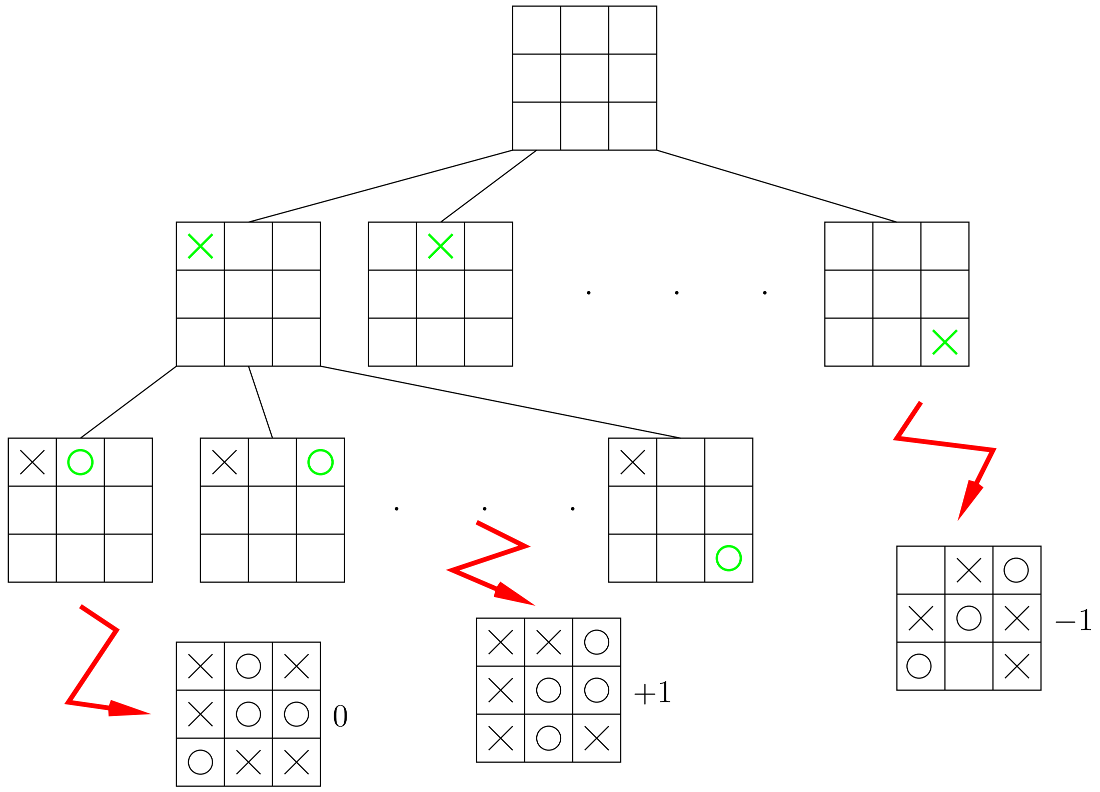
For each of *Max*'s opening moves *Min* has eight replies:



And so on...

This can be continued to represent *all* possibilities for the game.

Perfect decisions in a two-person game

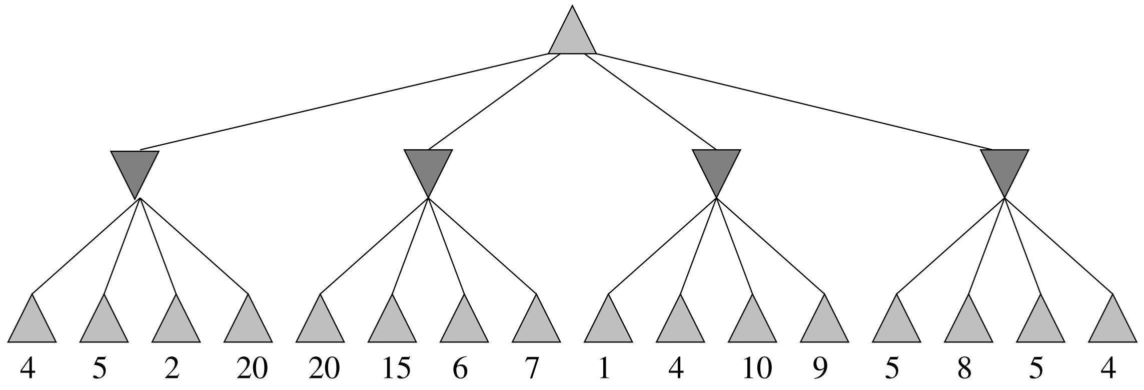


At the leaves a player has won or there are no spaces. Leaves are *labelled* using the utility function.

Perfect decisions in a two-person game

How can *Max* use this tree to decide on a first move?

Consider a much simpler tree:



Labels on the leaves denote utility.

High values are preferred by Max.

Low values are preferred by Min.

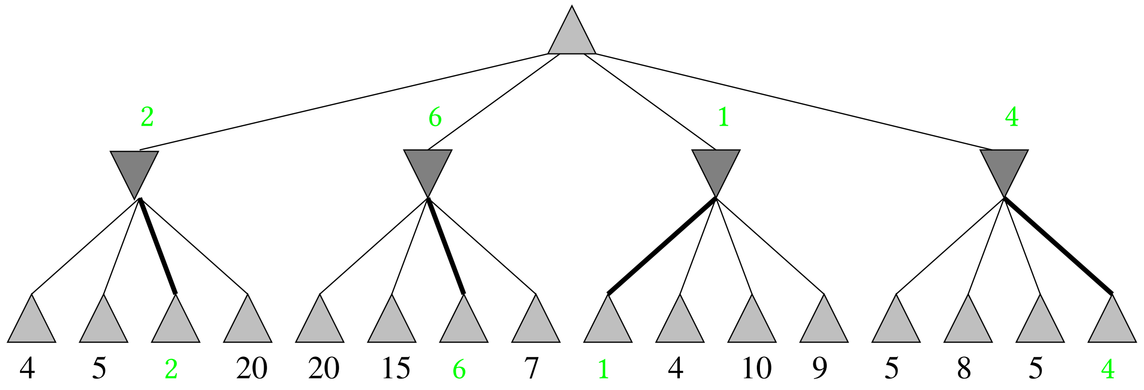
If *Max* is rational he will play to reach a position with the *biggest utility possible*

But if *Min* is rational she will play to *minimise* the utility available to *Max*.

The minimax algorithm

There are two moves: *Max* then *Min*. Game theorists would call this one move, or two *ply* deep.

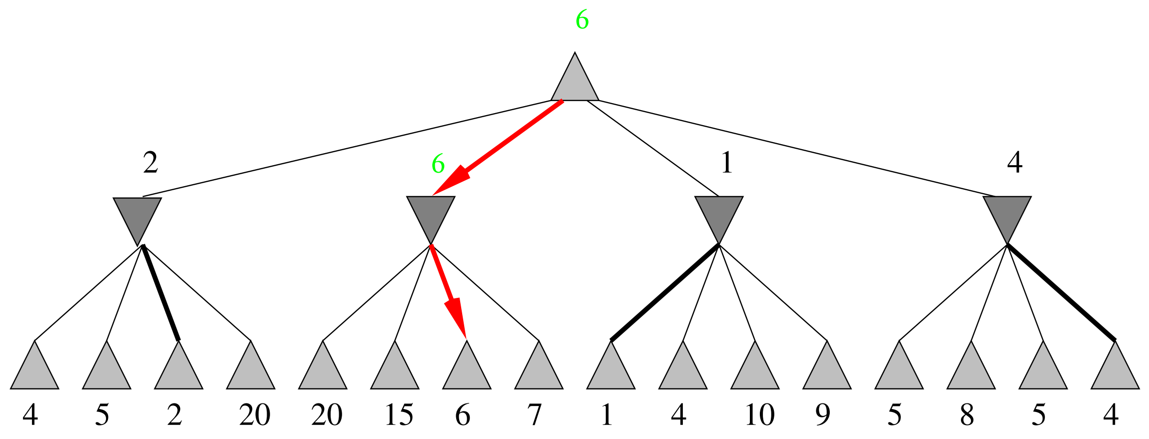
The *minimax algorithm* allows us to infer the best move that the current player can make, given the utility function, by working backward from the leaves.



As *Min* plays the last move, she *minimises* the utility available to *Max*.

The minimax algorithm

Moving one further step up the tree:



We can see that *Max*'s best opening move is move 2, as this leads to the node with highest utility.

The minimax algorithm

In general:

- Generate the complete tree and label the leaves according to the utility function.
- Working from the leaves of the tree upward, label the nodes depending on whether *Max* or *Min* is to move.
- If *Min* is to move label the current node with the *minimum* utility of any descendant.
- If *Max* is to move label the current node with the *maximum* utility of any descendant.

If the game is p ply and at each point there are q available moves then this process has (surprise, surprise) $O(q^p)$ time complexity and space complexity linear in p and q.

Making imperfect decisions

We need to avoid searching all the way to the end of the tree.

So:

- We generate only part of the tree: instead of testing whether a node is a leaf we introduce a *cut-off* test telling us when to stop.
- Instead of a utility function we introduce an *evaluation function* for the evaluation of positions for an incomplete game.

The evaluation function attempts to measure the expected utility of the current game position.

Making imperfect decisions

How can this be justified?

- This is a strategy that humans clearly sometimes make use of.
- For example, when using the concept of *material value* in chess.
- The effectiveness of the evaluation function is *critical*…
- … but it must be computable in a reasonable time.
- (In principle it could just be done using minimax.)

The importance of the evaluation function can not be understated—it is probably the most important part of the design.

The evaluation function

Designing a good evaluation function can be extremely tricky:

- Let's say we want to design one for chess by giving each piece its material value: pawn = 1, knight/bishop = 3, rook = 5 and so on.
- Define the evaluation of a position to be the difference between the material value of black's and white's pieces

$$\text{eval}(\text{position}) = \sum_{\text{black's pieces } p_i} \text{value of } p_i - \sum_{\text{white's pieces } q_i} \text{value of } q_i$$

This seems like a reasonable first attempt. Why might it go wrong?

- Until the first capture the evaluation function gives 0, so in fact we have a *category* containing many different game positions with equal estimated utility.
- For example, all positions where white is one pawn ahead.

So in fact this seems highly naïve ...

The evaluation function

We can try to *learn* an evaluation function.

- For example, using material value, construct a *weighted linear evaluation function*

  $$\text{eval}(\text{position}) = \sum_{i=1}^{n} w_i f_i$$

  where the w_i are *weights* and the f_i represent *features* of the position—in this case, the value of the ith piece.
- Weights can be chosen by allowing the game to play itself and using *learning* techniques to adjust the weights to improve performance.

However in general

- Here we probably want to give *different evaluations* to *individual positions*.
- The design of an evaluation function can be highly *problem dependent* and might require significant *human input and creativity*.

$\alpha - \beta$ pruning

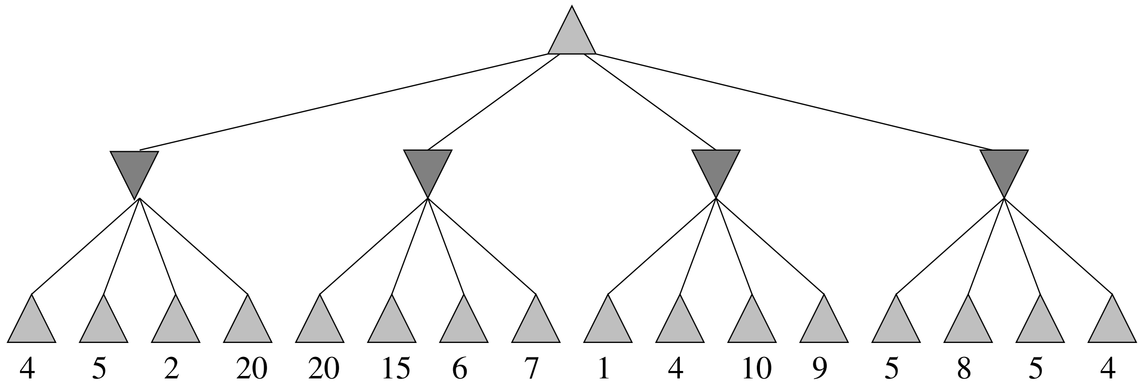
Even with a good evaluation function and cut-off test, the time complexity of the minimax algorithm makes it impossible to write a good chess program without some further improvement.

- Assuming we have 150 seconds to make each move, for chess we would be limited to a search of about 3 to 4 ply whereas...
- ...even an average human player can manage 6 to 8.

Luckily, it is possible to prune the search tree *without affecting the outcome* and *without having to examine all of it*.

$\alpha - \beta$ pruning

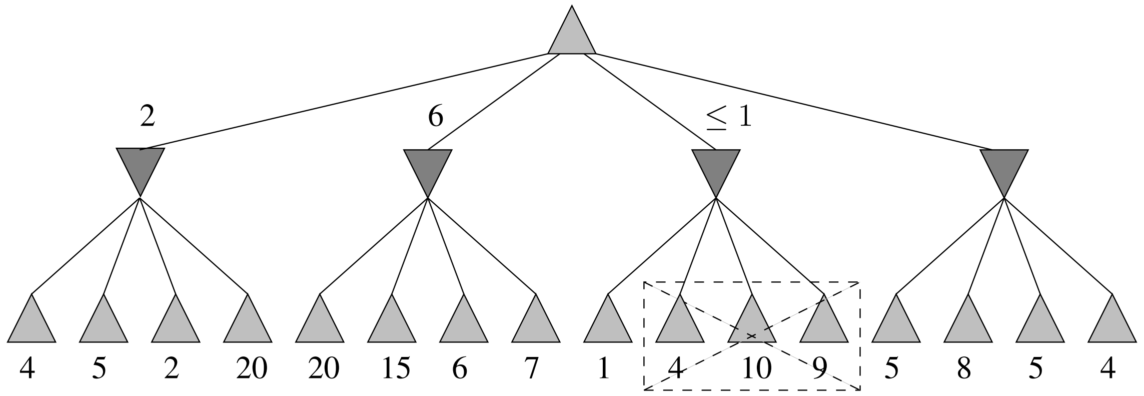
Returning for a moment to the earlier, simplified example:



The search is depth-first and left to right.

$\alpha - \beta$ pruning

The search continues as previously for the first 8 leaves.

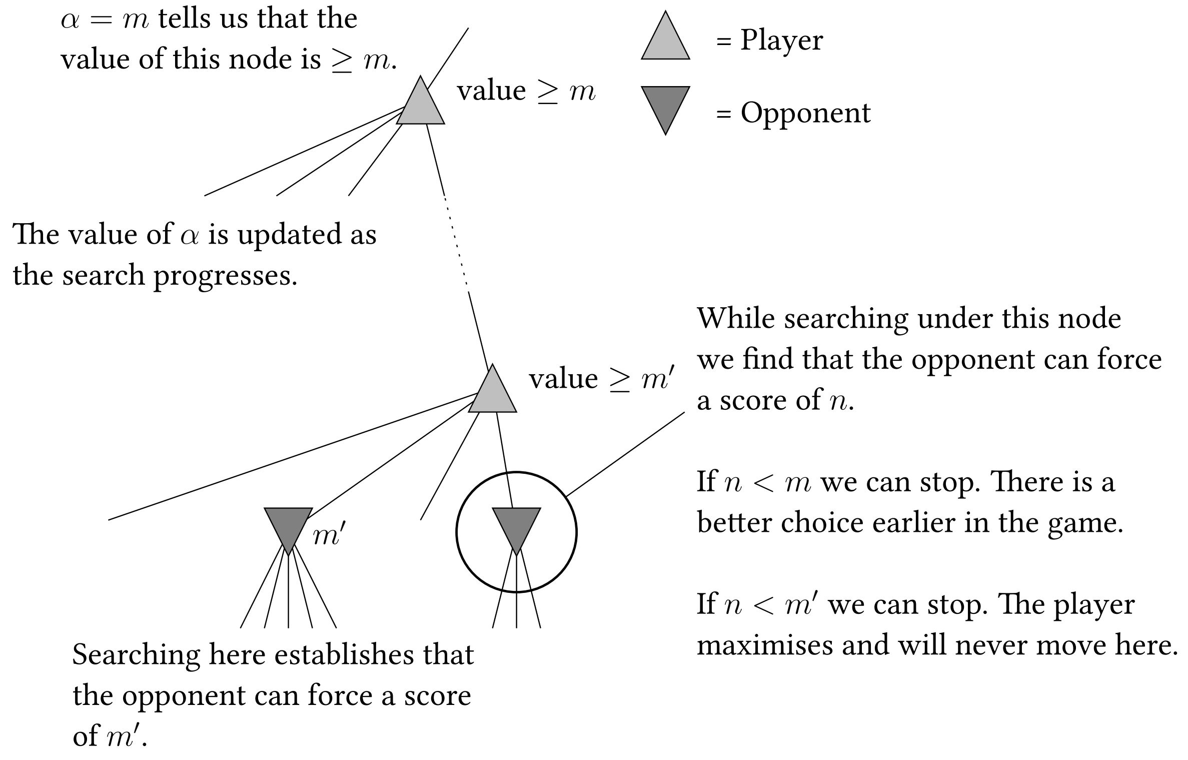


Then we note: if *Max* plays move 3 then *Min* can reach a leaf with utility at most 1.

So: *we don't need to search any further under Max's opening move* 3. This is because the search has *already established* that *Max* can do better by making opening move 2.

$\alpha - \beta$ pruning in general

Remember that this search is *depth-first*. We're only going to use knowledge of *nodes on the current path*.



$\alpha = m$ tells us that the value of this node is $\geq m$.
value $\geq m$
▲ = Player
▼ = Opponent
The value of α is updated as the search progresses.
value $\geq m'$
While searching under this node we find that the opponent can force a score of n.
m'
If $n < m$ we can stop. There is a better choice earlier in the game.
If $n < m'$ we can stop. The player maximises and will never move here.
Searching here establishes that the opponent can force a score of m'.

So: once you've established that n is sufficiently small, you don't need to explore any more of the corresponding node's children.

$\alpha - \beta$ pruning in general

The situation is exactly analogous if we *swap player and opponent* in the previous diagram.

The search is depth-first, so we're only ever looking at *one path through the tree*.

We need to keep track of the values α and β where

$$\alpha = \text{the } \textit{highest} \text{ utility seen so far on the path for } \textit{Max}$$

$$\beta = \text{the } \textit{lowest} \text{ utility seen so far on the path for } \textit{Min}$$

Assume *Max begins*. Initial values for α and β are

$$\alpha = -\infty$$

and

$$\beta = +\infty.$$

$\alpha - \beta$ pruning in general

So: we start with the function call

$$\texttt{player}(-\infty, +\infty, \texttt{root})$$

The following function implements the procedure suggested by the previous diagram:

```
function player($\alpha, \beta, n$)
    if cutoff($n$) then
        return eval($n$);
    value = $-\infty$;
    for each successor $n'$ of $n$ do
        value = max(value, opponent($\alpha, \beta, n'$));
        if value $> \beta$ then
            return value;
        if value $> \alpha$ then
            $\alpha$ = value;
    return value;
```

$\alpha - \beta$ pruning in general

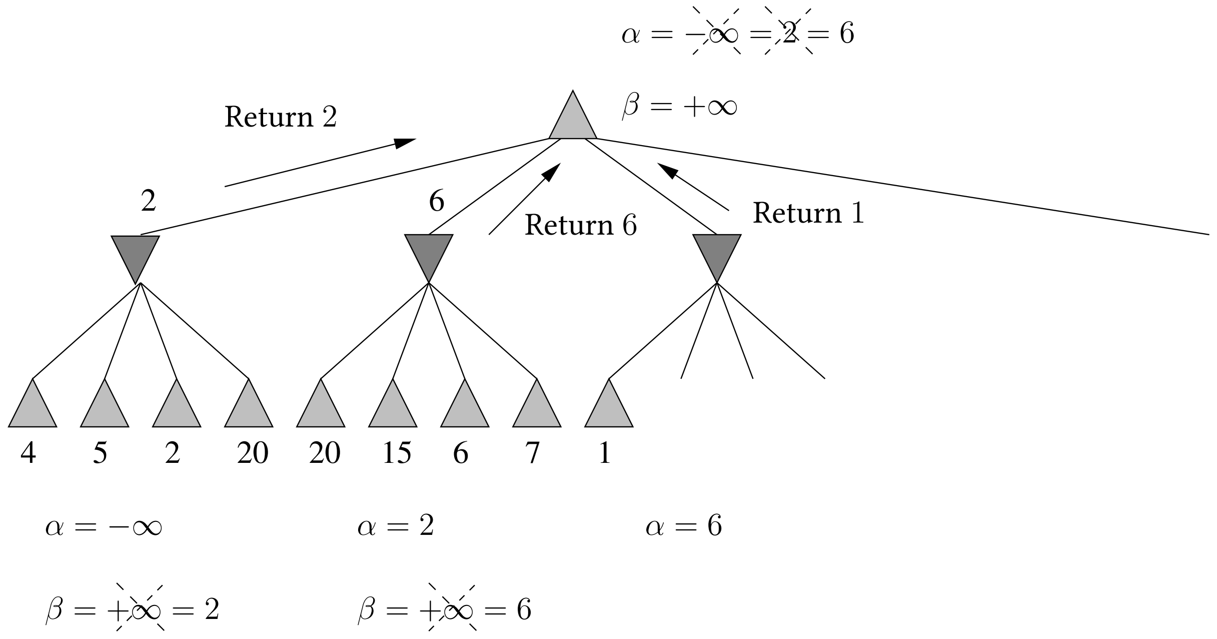
The function `opponent` is exactly analogous:

```
function opponent($\alpha, \beta, n$)
    if cutoff($n$) then
        return eval($n$);
    value = $\infty$;
    for each successor $n'$ of $n$ do
        value = min(value, player($\alpha, \beta, n'$));
        if value $< \alpha$ then
            return value;
        if value $< \beta$ then
            $\beta$ = value;
    return value;
```

Note: the semantics here is that parameters are passed to functions *by value*.

$\alpha - \beta$ pruning in general

Applying this to the earlier example and keeping track of the values for α and β you should obtain:



$\alpha = \cancel{-\infty} = \cancel{2} = 6$
$\beta = +\infty$
Return 2
Return 6
Return 1
2
6
4
5
2
20
20
15
6
7
1
$\alpha = -\infty$
$\beta = \cancel{+\infty} = 2$
$\alpha = 2$
$\beta = \cancel{+\infty} = 6$
$\alpha = 6$

How effective is $\alpha - \beta$ pruning?

(Warning: the theoretical results that follow are somewhat idealised.)

A quick inspection should convince you that the *order* in which moves are arranged in the tree is critical.

So, it seems sensible to try good moves first:

- If you were to have a perfect move-ordering technique then $\alpha - \beta$ pruning would be $O(q^{p/2})$ as opposed to $O(q^p)$.
- Consequently the branching factor would effectively be $\sqrt{q}$ instead of q.
- We would therefore expect to be able to search ahead *twice as many moves as before*.

However, this is not realistic: if you had such an ordering technique you'd be able to play perfect games!

How effective is $\alpha - \beta$ pruning?

If moves are arranged at random then $\alpha - \beta$ pruning is:

- $O((q/\log q)^p)$ asymptotically when $q > 1000$ or...
- ...about $O(q^{3p/4})$ for reasonable values of q.

In practice *simple ordering techniques* can get *close to the best case*. For example, if we try captures, then threats, then moves forward *etc.*

Alternatively, we can implement an *iterative deepening* approach and use the order obtained at one iteration to drive the next.

A further optimisation: the transposition table

Finally, note that many games correspond to *graphs* rather than *trees* because the same state can be arrived at in different ways.

- This is essentially the same effect we saw in heuristic search: recall *graph search* versus *tree search*.
- It can be addressed in a similar way: store a state with its evaluation in a hash table—generally called a *transposition table*—the first time it is seen.

The transposition table is essentially equivalent to the *closed list* introduced as part of graph search.

This can vastly increase the effectiveness of the search process, because we don't have to evaluate a single state multiple times.

Constraint satisfaction problems (CSPs)

Reading: AIMA chapter 6.

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. 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*.

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*.

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—and of the application of CSPs—when we discuss *planning*.

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* $C_1, C_2, \ldots, C_m$.

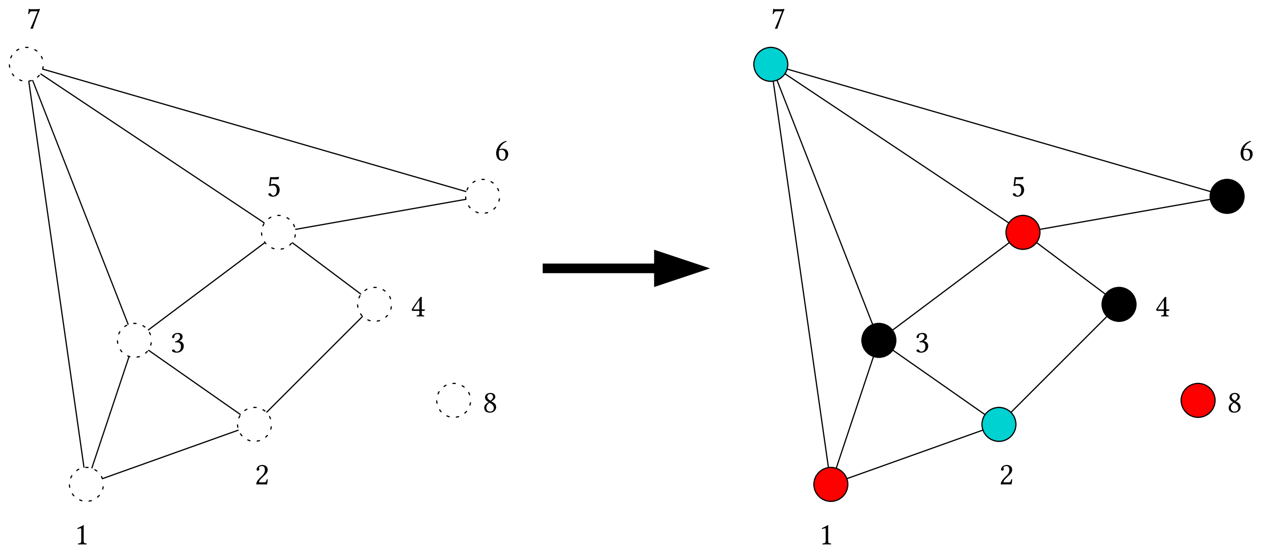
Each constraint C_i involves a set of variables and specifies an *allowable collection of values*.

- 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*.

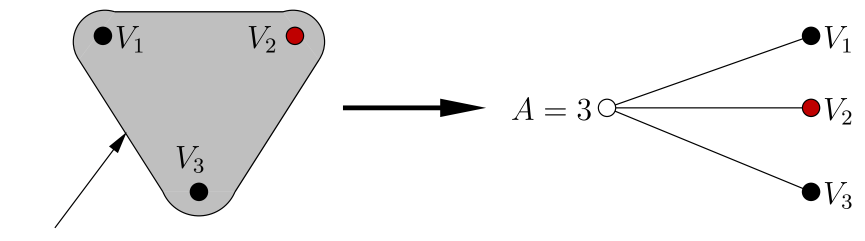
How does that work?

Auxiliary variables

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

A single constraint

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



The original constraint connects all three variables.

New, binary constraints:

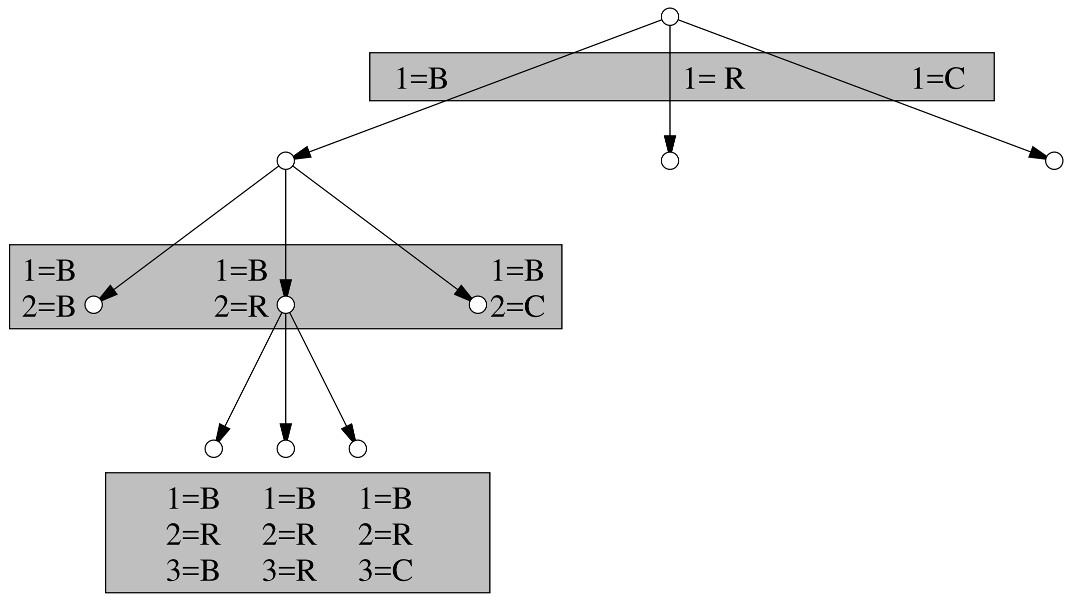
$(A = 1, V_1 = C), (A = 1, V_2 = C), (A = 1, V_3 = C)$
$(A = 2, V_1 = R), (A = 2, V_2 = B), (A = 2, V_3 = B)$
$(A = 3, V_1 = B), (A = 3, V_2 = R), (A = 3, V_3 = B)$
$(A = 4, V_1 = B), (A = 4, V_2 = B), (A = 4, V_3 = R)$

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

Backtracking search

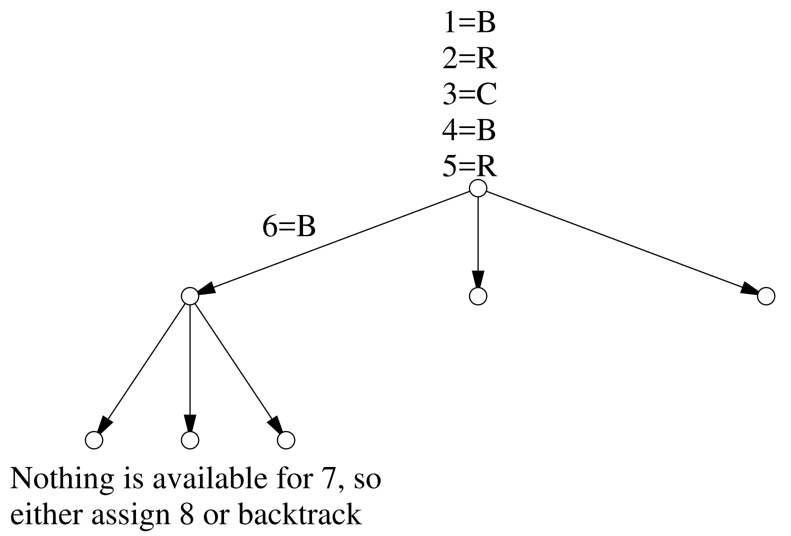
Backtracking search now takes on a very simple form: search depth-first, assigning a single variable at a time, and backtrack if no valid assignment is available.

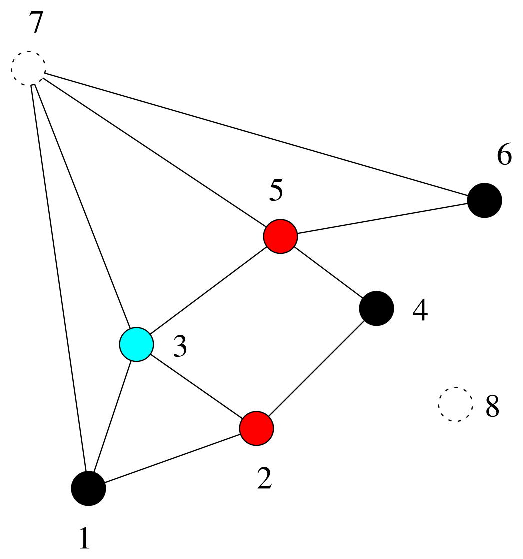
Using the graph colouring example, the search now looks something like this...



...and new possibilities appear.

Backtracking search





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

Backtracking search

Starting with:

$$\texttt{backtrack}([\,], \texttt{problemDescription})$$

```
function backTrack(assignmentList, problemDescription)
    if assignmentList is complete then
        return SOME assignmentList;
    nextVar = getNextVariable(assignmentList, problemDescription);
    for each $v$ in orderValues(nextVar, assignmentList, problemDescription) do
        if $v$ is consistent with assignmentList then
            add “nextVar = $v$” to assignmentList;
            solution = backTrack(assignmentList, problemDescription);
            if solution is not FAIL then
                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:

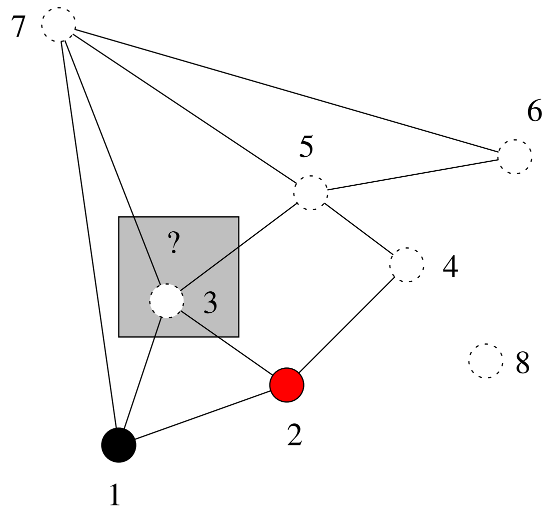
- In what order should we try to *assign variables*?
- In what order should we try to *assign possible values* to a variable?

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*?
- Can we try to force the search in a successful direction (remember the use of *heuristics*)?
- Can we try to force *failures/backtracks* to occur quickly?

Heuristics I: Choosing the order of variable assignments and values

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



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

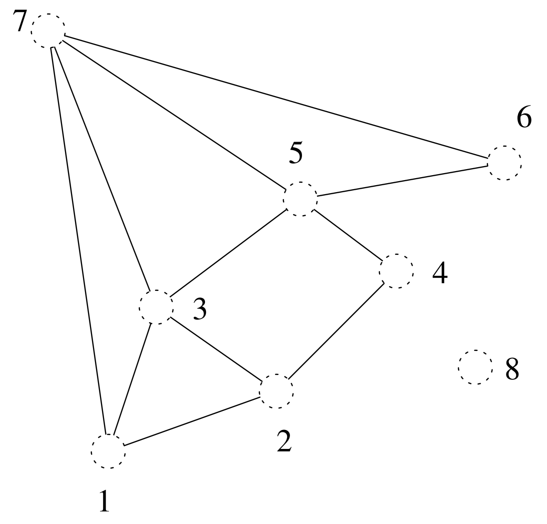
Assigning such variables *first* is called the *minimum remaining values (MRV)* heuristic.

(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.

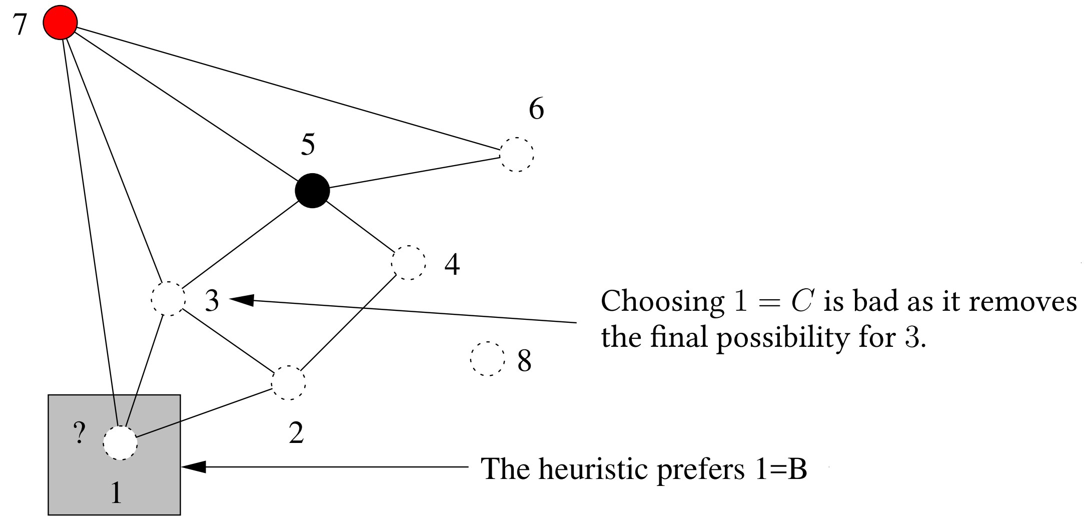


Start with 3, 5 or 7.

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

Heuristics I: Choosing the order of variable assignments and values

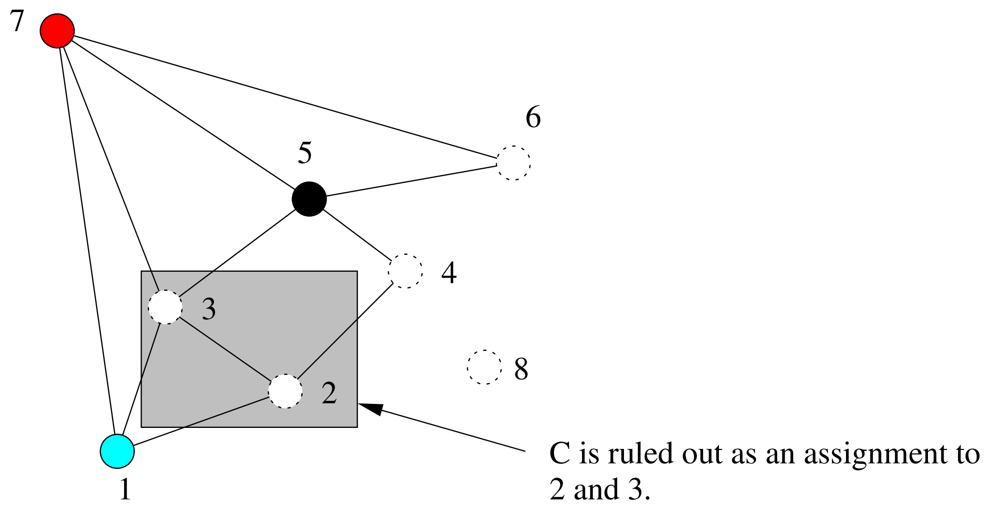
Once a variable is chosen, in *what order should values be assigned*?


7
6
5
4
3
8
2
?
1
Choosing $1 = C$ is bad as it removes the final possibility for 3.
The heuristic prefers 1=B

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 $4 \to 5$.

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

$i \to j$ is *consistent* if

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

Example:

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

- $5 \to 4$ 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 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

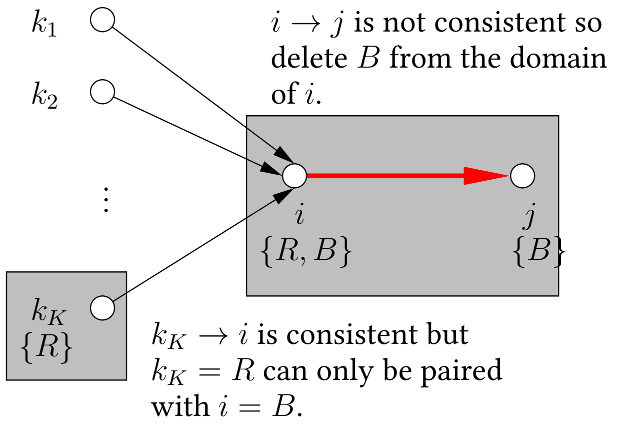
We can enforce arc consistency each time a variable i is assigned.

- 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.

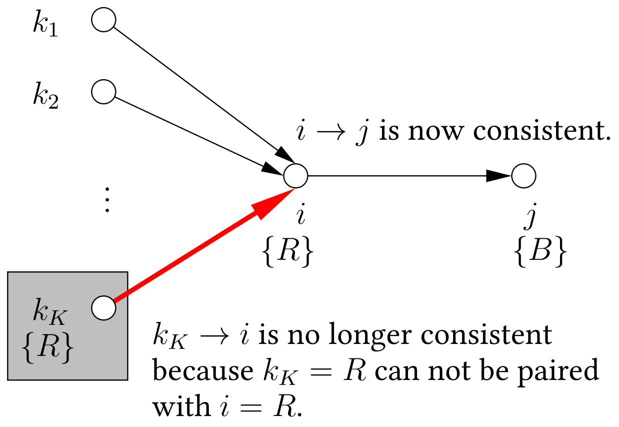
This is because if $i \rightarrow j$ is inconsistent resulting in a deletion from D_i we may as a consequence make some arc $k \rightarrow i$ inconsistent.

Why is this?

Enforcing arc consistency



k_1
k_2
$i \to j$ is not consistent so delete B from the domain of i.
i
$\{R, B\}$
j
$\{B\}$
k_K
$\{R\}$
$k_K \to i$ is consistent but $k_K = R$ can only be paired with $i = B$.



k_1
k_2
$i \to j$ is now consistent.
i
$\{R\}$
j
$\{B\}$
k_K
$\{R\}$
$k_K \to i$ is no longer consistent because $k_K = R$ can not be paired with $i = R$.

- $i \to j$ 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

```
function AC-3(problemDescription)
    Queue toCheck = [ all arcs $i \to j$ ];
    while toCheck is not empty do
        $i \to j$ = next(toCheck);
        if removeInconsistencies($D_i$, $D_j$) then
            for each $k$ that is a neighbour of $i$ do
                add $k \to i$ to toCheck;
```

```
function removeInconsistencies($D_1$, $D_2$)
    Bool result = FALSE;
    for each $d \in D_1$ do
        if no $d' \in D_2$ valid with $d$ then
            remove $d$ from $D_1$;
            result = TRUE;
    return result;
```

Enforcing arc consistency

Complexity:

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

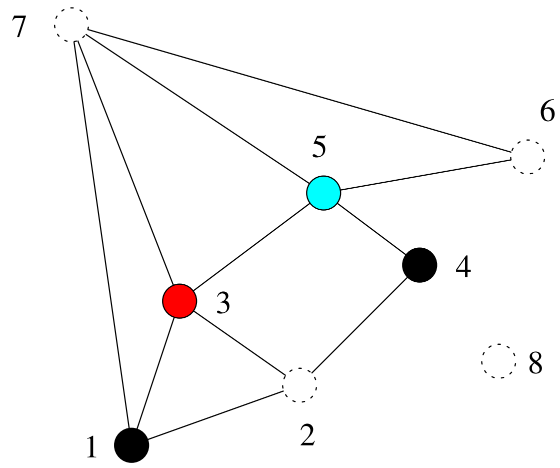
Strong k-consistency requires the we be k-consistent, $k - 1$-consistent *etc* as far down as 1-consistent.

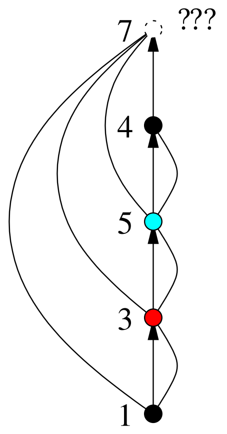
If we can demonstrate strong n-consistency (where as usual n is the number of variables) then an assignment can be found in $O(nd)$.

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.
- Conversely, I_k *conflicts* with some variable V if no value for V is consistent with I_k.

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.

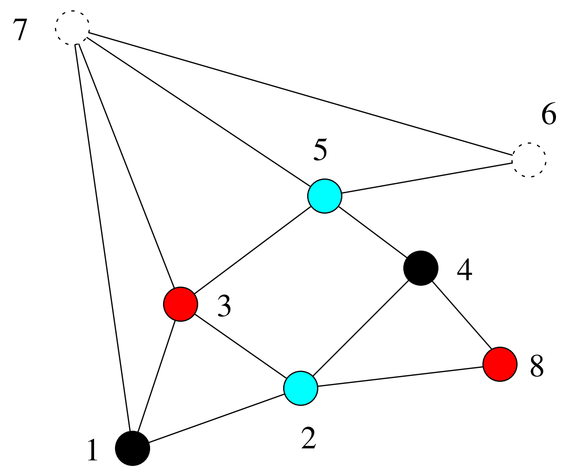
More formally: if I_k conflicts with V_{k+1} we backtrack to V_j where

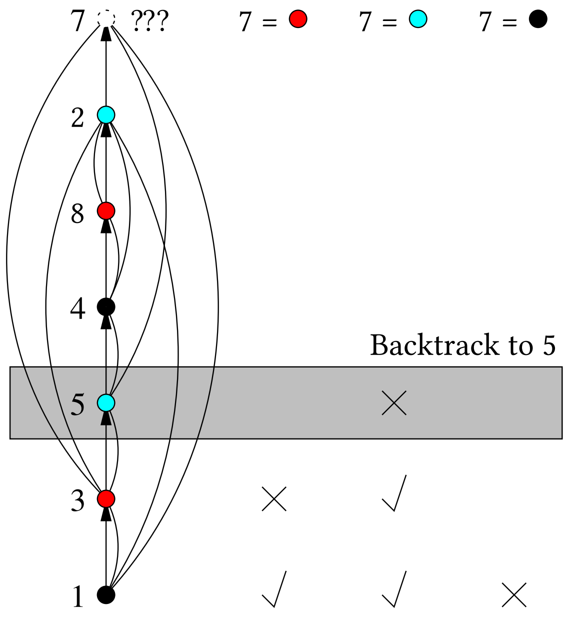
$$j = \min\{j \leq k | I_j \text{ conflicts with } V_{k+1}\}.$$

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:

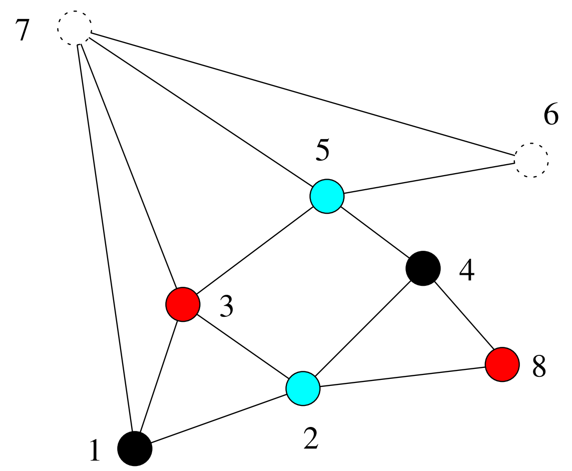
- We assume an ordering $V_1, V_2, \ldots, V_n$ for the variables.
- Given $V' = \{V_1, V_2, \ldots, V_k\}$ where $k < n$ the *ancestors* of V_{k+1} are the members of V' connected to V_{k+1} by a constraint.
- The *parent* $P(V_{k+1})$ of V_{k+1} is its most recent ancestor.

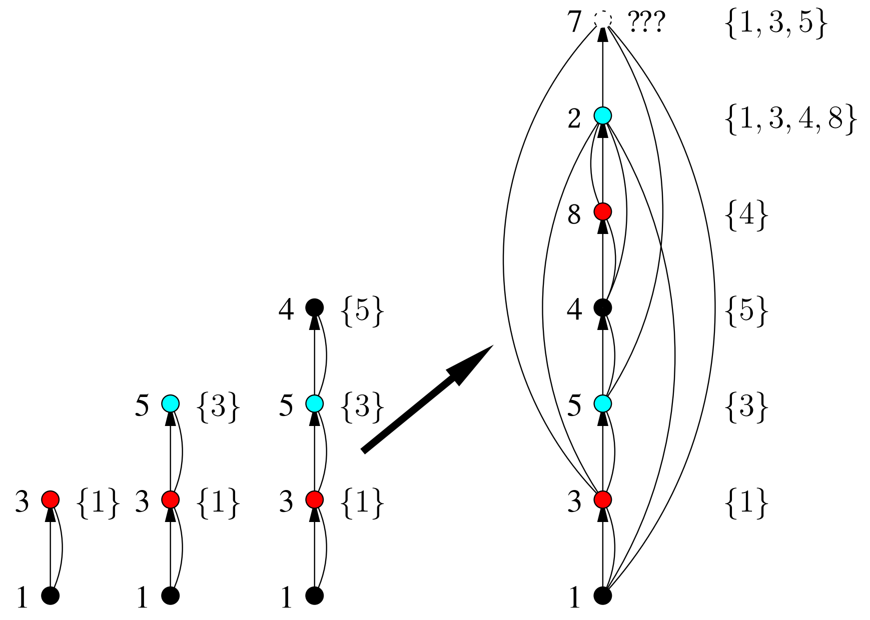
The ancestors for each variable can be accumulated as assignments are made.

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

Note: Gaschnig's algorithm uses *assignments* whereas graph-based backjumping uses *constraints*.

Graph-based backjumping





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

Backjumping and forward checking

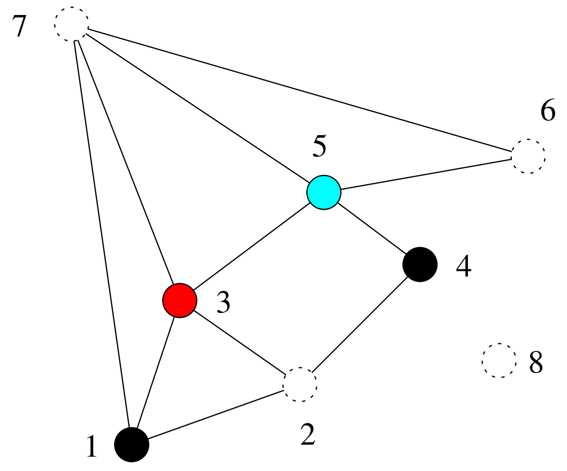
If we use *forward checking*: say we're assigning to V_{k+1} by making $V_{k+1} = d$:

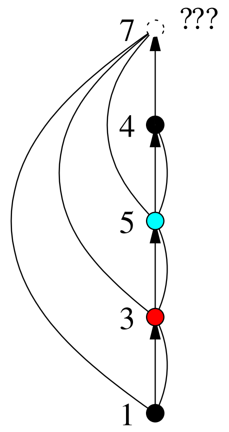
- Forward checking removes d from the D_i of *all* V_i connected to V_{k+1} by a constraint.
- When doing graph-based backjumping, we'd also add V_{k+1} to the ancestors of V_i.

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





Ancestors

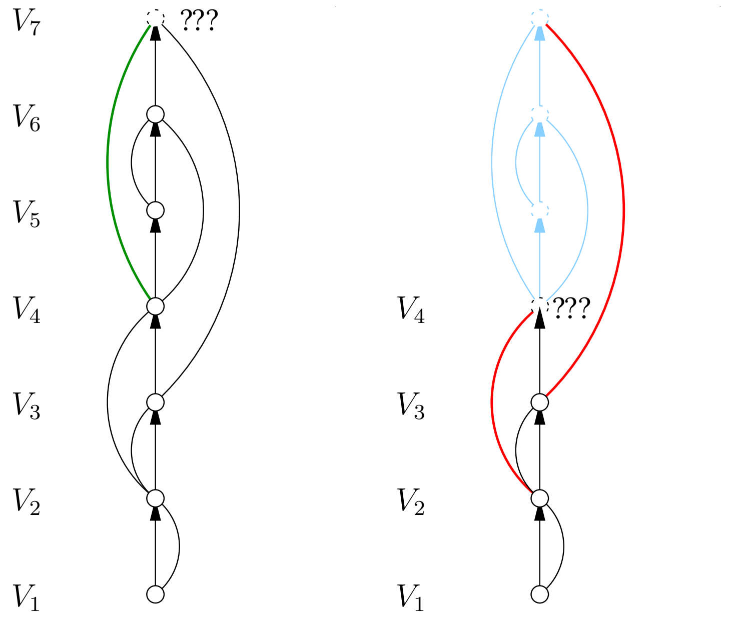
1 − { }
2 − {1,3, 4}
3 − {1}
4 − {5}
5 − {3}
6 − {5}
7 − {1, 3 ,5}
8 − { }

	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

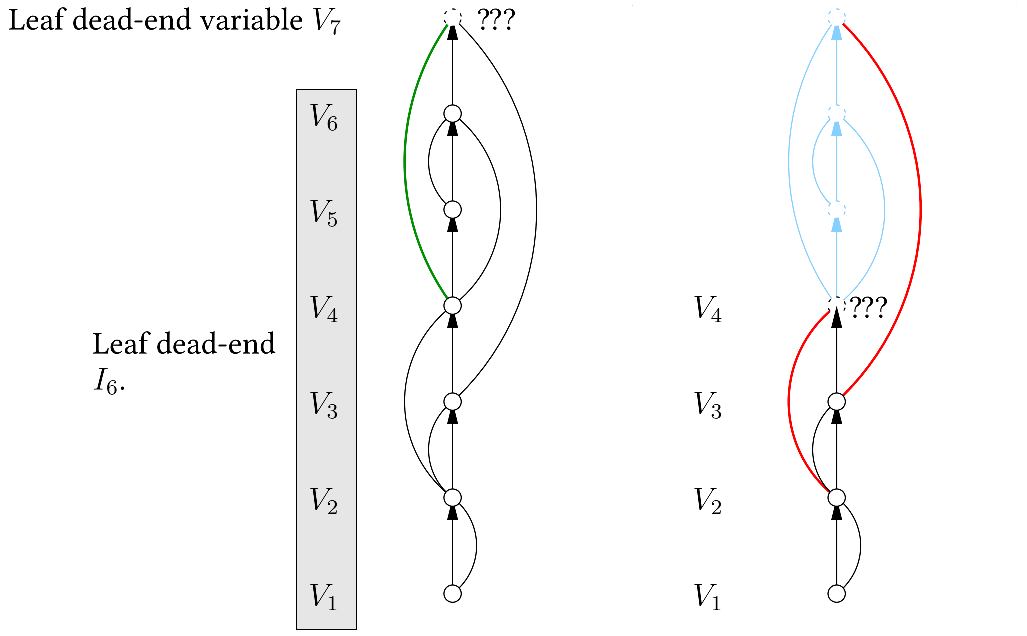
We're not quite done yet though. What happens when *there are no assignments left for the parent we just backjumped to*?



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

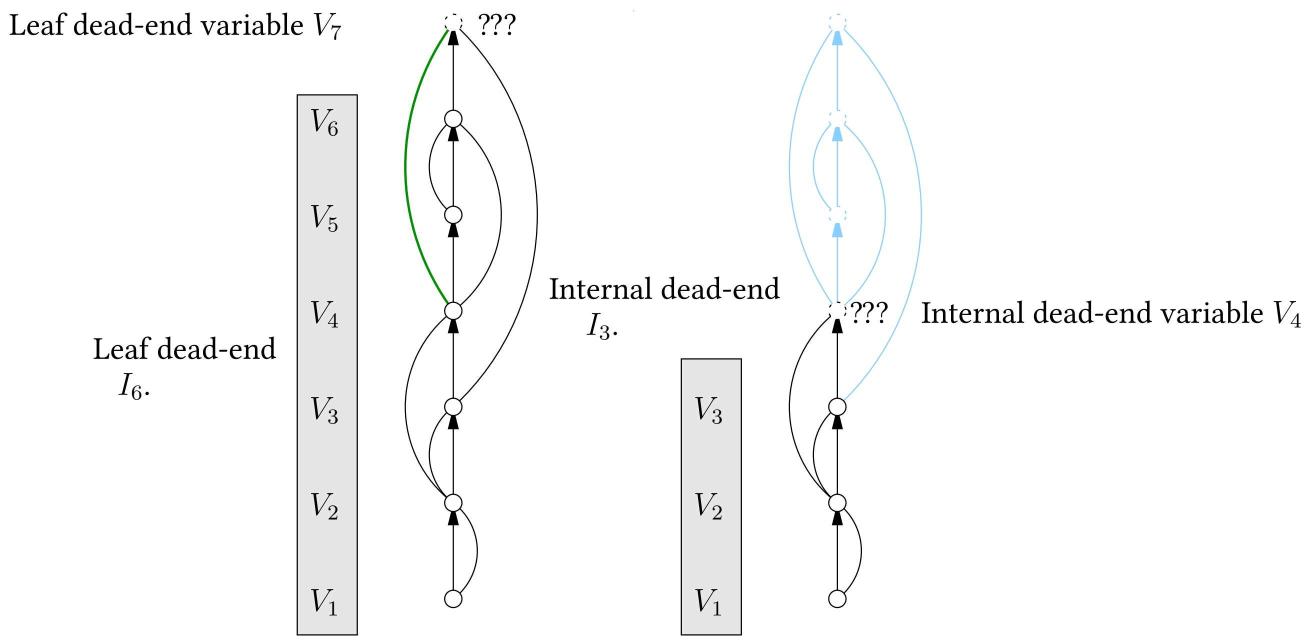
To describe an algorithm in this case is a little involved.



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

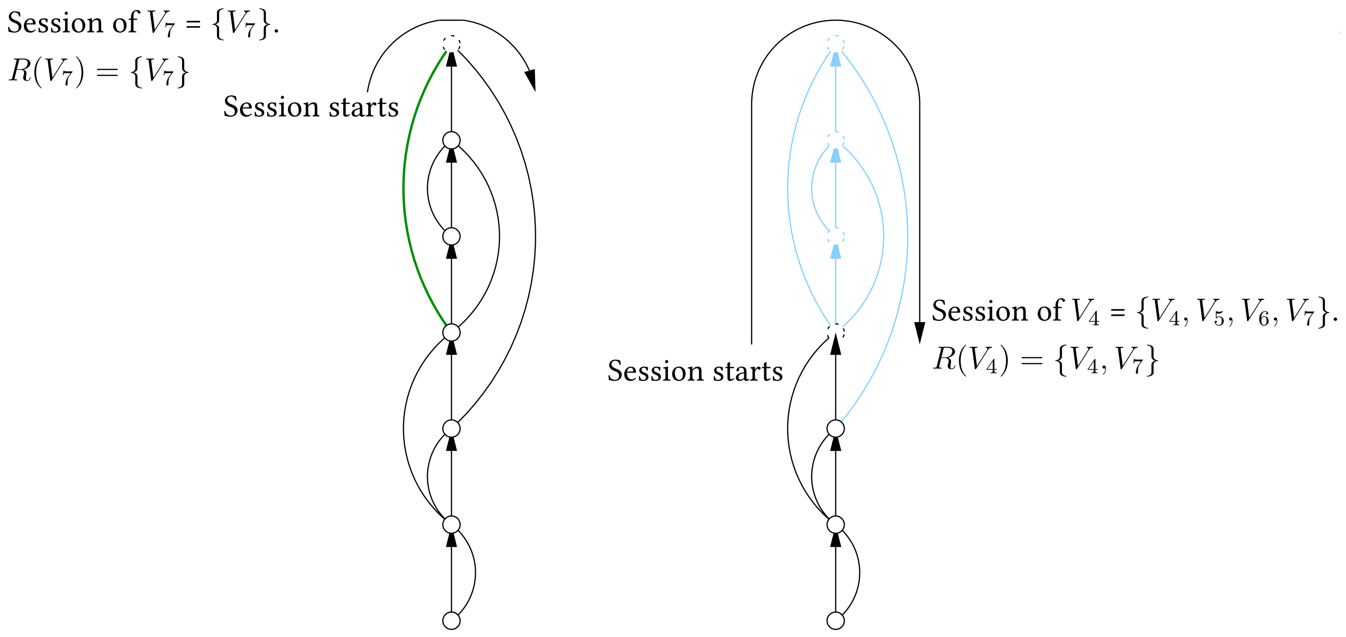
To keep track of exactly where to jump to we also need the definitions:

- 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* $R(V)$ for a variable V are:
  1. $R(V)$ is initialized to $\{V\}$ when V is first visited.
  2. If V is a leaf dead-end variable then $R(V) = \{V\}$.
  3. If V was backtracked to from a dead-end V' then $R(V) = R(V) \cup R(V')$.

And we're not done yet...

Graph-based backjumping

Example:



Session of V_7 = $\{V_7\}$.
$R(V_7) = \{V_7\}$
Session starts
Session starts
Session of V_4 = $\{V_4, V_5, V_6, V_7\}$.
$R(V_4) = \{V_4, V_7\}$

As expected, the relevant dead-ends for V_4 are $\{V_4\}$ and $\{V_7\}$.

Graph-based backjumping

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

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

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

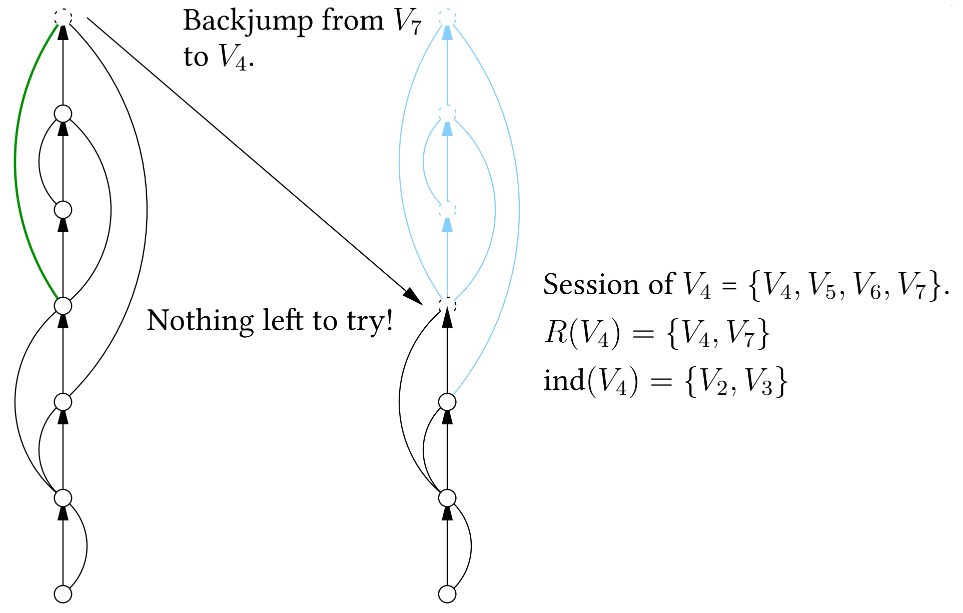
- The *culprit* for V_k is the most recent $V' \in \mathrm{ind}(V_k)$.

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

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

Graph-based backjumping

Example:



Backjump from V_7 to V_4.
Nothing left to try!
Session of V_4 = $\{V_4, V_5, V_6, V_7\}$.
$R(V_4) = \{V_4, V_7\}$
$\text{ind}(V_4) = \{V_2, V_3\}$

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

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.

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*.