General Search Strategies
Look Ahead

Chapter 5 @ Constraint Processing by Rina Dechter

Motivation

- No matter how much we reason about a problem, after some consideration we are left with choices
  - The only way to proceed is trial & error or guess & testing
- In the context of CSP, search (usually) corresponds to backtracking
  - Extend a partial solution by assigning values to variables
- Backtracking takes place when a dead-end occurs
  - Requires linear space but exponential time

YouTube video

- https://www.youtube.com/watch?v=X6m0DXt95bs

The state space

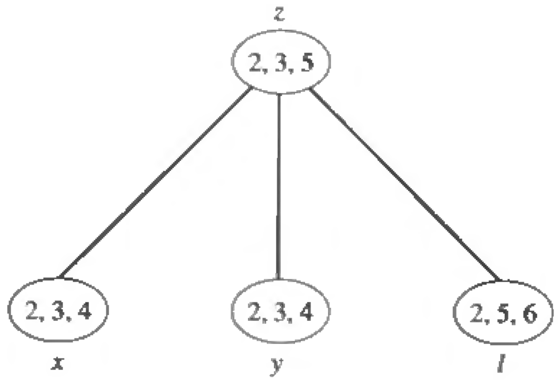
- Definition of state space with 4 elements (S,O,S_0,S_g)
  - A set S of states
  - A set O of operators that map states to states
  - An initial state $S_0 \in S$
  - A set $S_g \subseteq S$ of goal states
- State space can be represented with a directed graph, where nodes represent states and a directed arc from s_i to s_j means that there is an operator transforming s_i into s_j
- Find a solution = sequence of operators that transforms the initial state into a goal state

The search space

- A tree of all partial solutions
- A partial solution: value assignment $(a_1,\ldots,a_j)$ satisfying all relevant constraints
- The size of the underlying search space depends on
  - Variable ordering
  - Level of consistency

Search space: example (I)

- (a) constraint graph
- 4 variables: z, x, y, l
- Domains in the figure
- Constraints R
  - z evenly divides _
- (b) ordering (z,x,y,l)
- (c) ordering (x,y,l,z)

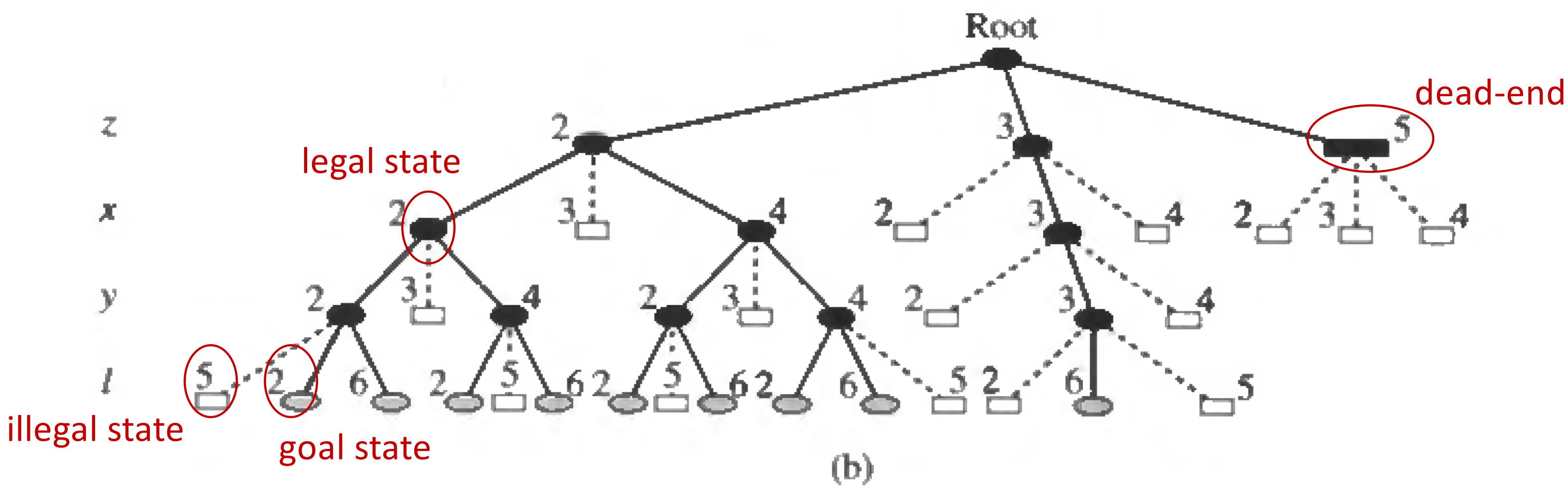


(a)

You have
5 minutes!

Search space: example (II)

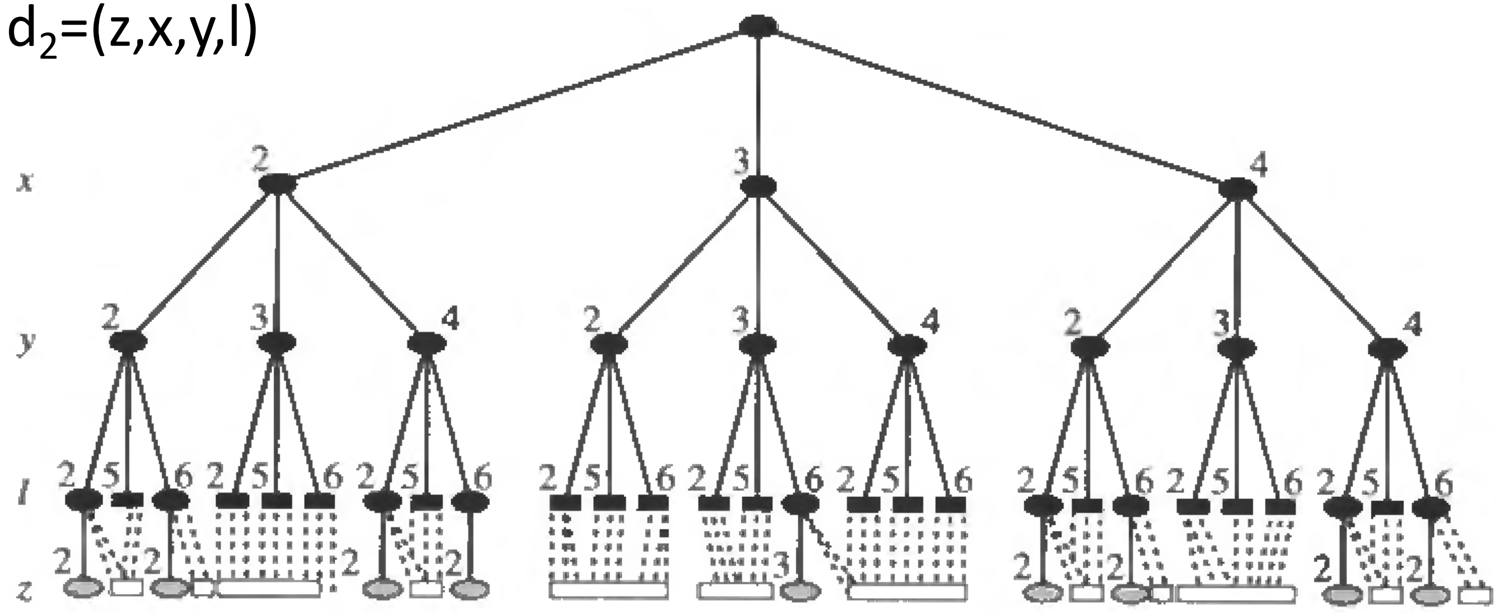
- (b) ordering d_1=(z,x,y,l)



Hollow boxes connected by broken lines represent illegal states that correspond to failed instantiation attempts.

Search space: example (III)

- (c) ordering d_2=(z,x,y,l)



(c)

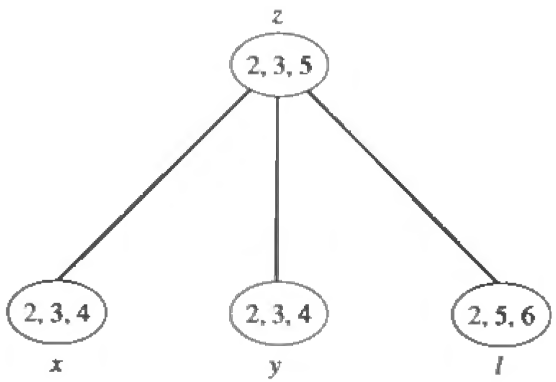
Variable ordering

- Different search spaces depending on the variable ordering
- Previous example with different orderings
  - 20 legal states vs. 48 legal states
  - 1 dead-end vs 18 dead-end leaves
- Ordering may be left to the search rather than fixed in advance

Consistency level: arc-consistency

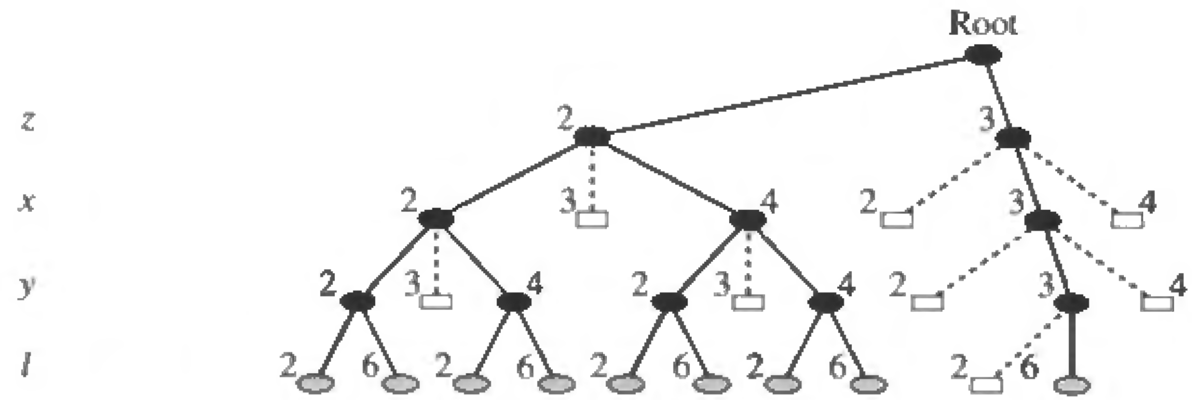
You have
3 minutes!

- Reduces the search space
  - Set of constraints R replaced by an equivalent but tighter set of constraints R'
- E.g. apply arc-consistency before search
  - $D_z = \{2,3\}$
  - $D_x = \{2,3,4\}$
  - $D_y = \{2,3,4\}$
  - $D_l = \{2,6\}$



(a)

Consistency level: search for d_1 after AC

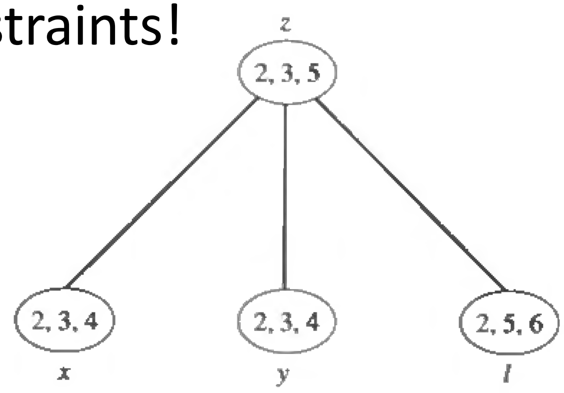


Search space contains only solution paths ☺

Consistency level: path-consistency

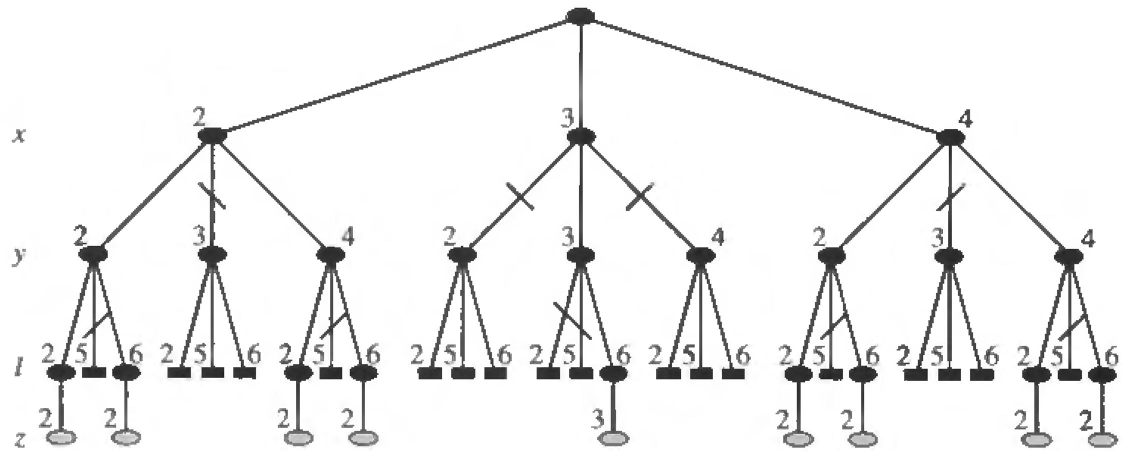
You have
3 minutes!

- The network R is not path consistency…
- For example, the assignment x=2 and y=3 is consistent
  - But cannot be extended to any value of z
- How to enforce path-consistency? With new constraints!
  - R'_{zx} = {(2,2),(2,4),(3,3)}
  - R'_{zy} = {(2,2),(2,4),(3,3)}
  - R'_{zl} = {(2,2),(2,6),(3,6)}
  - R'_{xy} = {(2,2),(2,4),(4,2),(4,4),(3,3)}
  - R'_{xl} = {(2,2),(2,6),(4,2),(4,6),(3,6)}
  - R'_{yl} = {(2,2),(2,6),(4,2),(4,6),(3,6)}



(a)

Consistency level: search for d_2 after PC



Theorem ☺

Let $\mathcal{R}'$ be a tighter network than $\mathcal{R}$, where both represent the same set of solutions. For any ordering d, any path appearing in the search graph derived from $\mathcal{R}'$ also appears in the search graph derived from $\mathcal{R}$. •

- Should we make the representation of a problem as explicit as possible by inference before search???
  - Inference is costly…
  - … and may not cut significantly the search space

Trade-off: R vs R’ with d_1

- Set of constraints R
  - Exactly one constraint tested for each node generated
- New set of constraints R’ (after path-consistency)
  - An explicit constraint for every pair of variables
  - Level 1: one constraint check, Level 2: two constraint checks, etc.
- Overall: ≈ 20 vs ≈ 40 constraint checks [does not pay-off!]

Trade-off: R vs R’ with d_2

- Set of constraints R
  - Constraint checks only at the fourth level, requiring as many as three constraint checks per node
- New set of constraints R’ (after path-consistency)
  - Constraint checks in each of the first three levels (one for first, two for second and three for third)
  - Only 9 nodes at level 4, each requiring 3 constraint checks
- Overall: ≈ 80 vs ≈ 50 constraint checks [pays-off!]

Reminder: backtrack-free network

(backtrack-free network)

A network R is said to be *backtrack-free* along ordering d if every leaf node in the corresponding search graph is a solution. •

AC + d_1 → search space contains only solution paths ☺

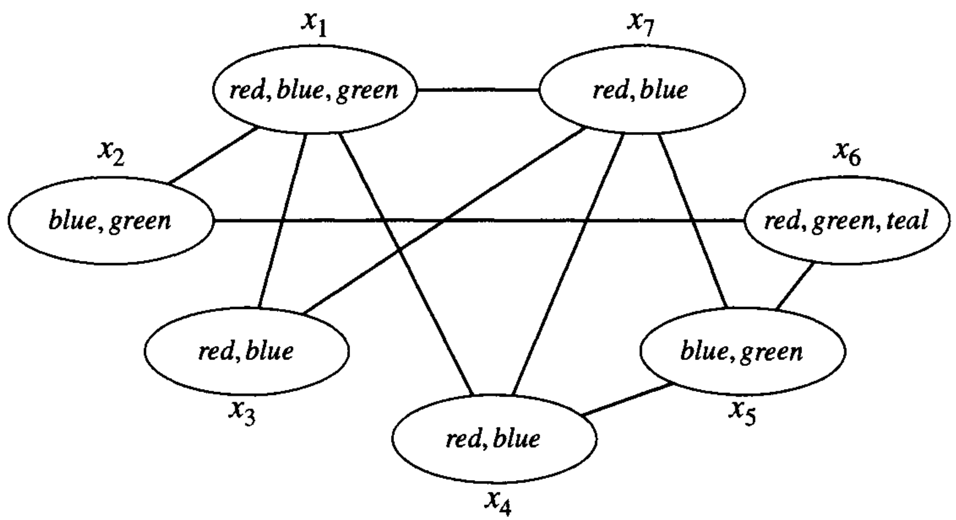
Backtracking

- Traverses the state space of partial instantiations in a depth-first way
- Backtracking has two phases: forward and backward
- Backtracking forward phase
  - Variables are selected in sequence
  - Current partial solution is extended
- Backtracking backward phase
  - When no consistent solution exists
  - The algorithm returns to the previous variable assigned

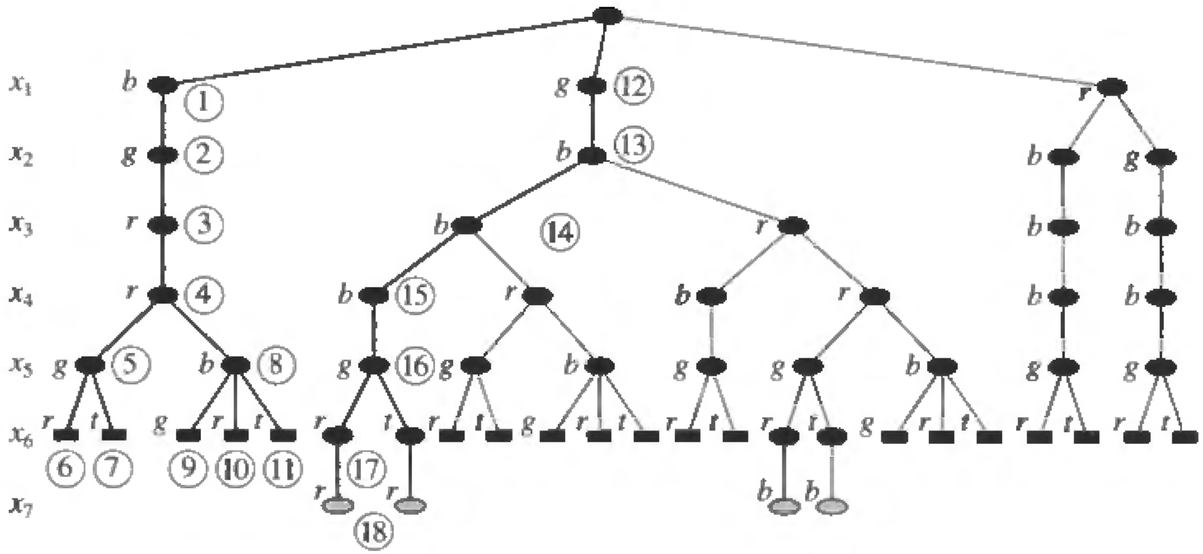
Backtracking: example

You have
3 minutes!

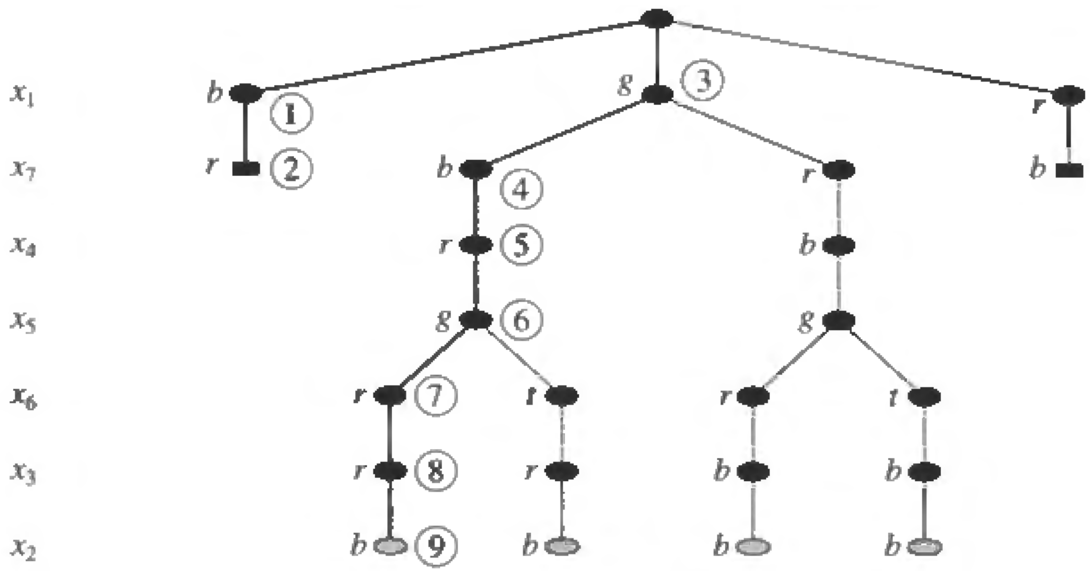
- Consider this graph coloring problem
- Variable ordering $d_1 = x_1, x_2, x_3, x_4, x_5, x_6, x_7$
- Variable ordering $d_1 = x_1, x_7, x_4, x_5, x_6, x_3, x_2$
- Color ordering (blue,green,red,teal)
- Consider only legal states



Backtracking for d_1



Backtracking for d_2



```
procedure BACKTRACKING
Input: A constraint network ℛ = (X, D, C).
Output: Either a solution, or notification that the network is inconsistent.
    i ← 1                                   (initialize variable counter)
    D'_i ← D_i                              (copy domain)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECT-VALUE
        if x_i is null                      (no value was returned)
            i ← i - 1                       (backtrack)
        else
            i ← i + 1                       (step forward)
            D'_i ← D_i
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1,...,x_n}
end procedure

subprocedure SELECT-VALUE   (return a value in D'_i consistent with a⃗_{i-1})
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        if CONSISTENT (a⃗_{i-1}, x_i = a)
            return a
    end while
    return null                             (no consistent value)
end procedure
```

Improvements to backtracking

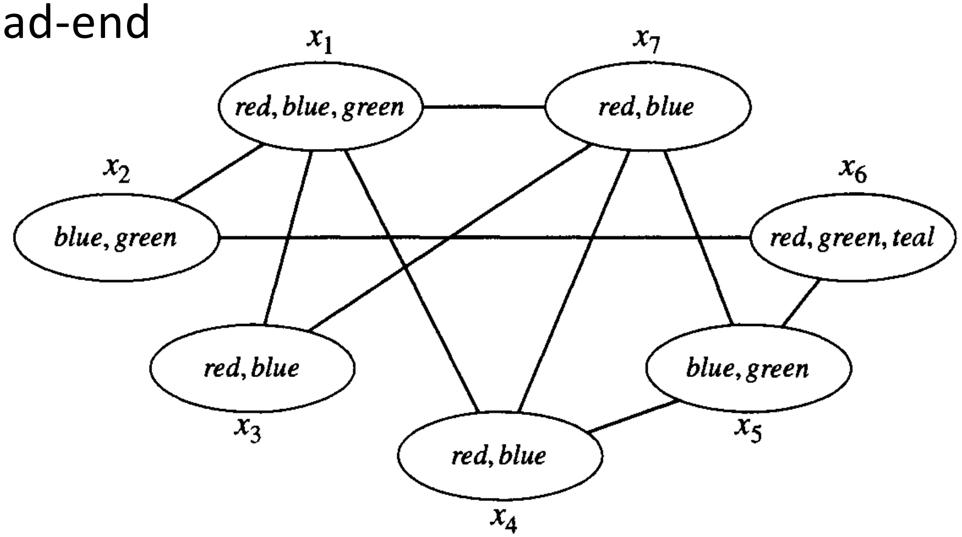
- Backtracking suffers from thrashing
  - Repeatedly rediscovering the same inconsistencies / partial successes
  - Consequence of NP- completeness (?)
- Backmarking keeps track of where past consistency tests failed
  - Does not prune the search but replaces consistency checks with table lookups
- Preprocessing may reduce the search space
- Dynamically improving the punning power
  - Look-ahead schemes (going forward) – this chapter
  - Look-back schemes (going back) – next chapter

Look-ahead schemes

- Decide which variable to assign next
  - Assuming the order is not predetermined
  - In general, first assign variables that maximally constraint the rest of the search space
  - I.e. select the most highly constrained variable with the least number of viable values
- Decide which value to assign to the next variable
  - Assign the value that maximizes the number of options available for future assignments

Look-ahead schemes: example

- Assume x_1 is first in the ordering, x_1 = red
  - Look-ahead notes incompability of value red for x_3, x_4, x_7
  - More extensive look-ahead notes that x_3 and x_7 are connected and left with incompatible values
  - So assigning red to x_1 will lead to a dead-end
    - The same for x_1= blue



```
procedure GENERALIZED-LOOK-AHEAD
Input: A constraint network R = (X, D, C).
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n                (copy all domains)
    i ← 1                                   (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECT-VALUE-XXX
        if x_i is null                      (no value was returned)
            i ← i - 1                       (backtrack)
            reset each D'_k, k > i, to its value before x_i was last instantiated
        else
            i ← i + 1                       (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure
```

Look-ahead for value selection

- Forward-checking
- Arc-consistency look-ahead
- Full and partial look-ahead
- Dynamic look-ahead value orderings

Forward checking

- Propagates the effect of a tentative value selection to each future variable, separately
- If the domain of a future variable becomes empty, the search backtracks

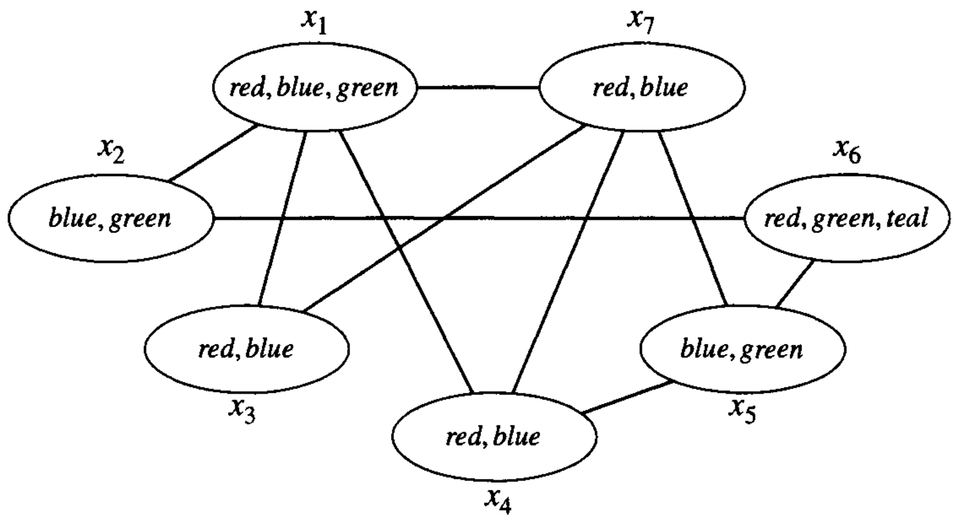
Forward checking: algorithm

procedure SELECT-VALUE-FORWARD-CHECKING
    while D_i' is not empty
        select an arbitrary element $a \in D_i'$, and remove a from D_i'
        for all k, $i < k \leq n$
            for all values b in D_k'
                if $\boxed{\text{not CONSISTENT}(\vec{a}_{i-1}, x_i = a, x_k = b)}$
                    $\boxed{\text{remove } b \text{ from } D_k'}$
            end for
            if $\boxed{D_k' \text{ is empty}}$ ($x_i = a$ leads to a dead-end don't select a)
            reset each D_k', $i < k \leq n$ to value before a was selected
            else
            return a
    end while
    return null (no consistent value)
end procedure

Forward checking: example

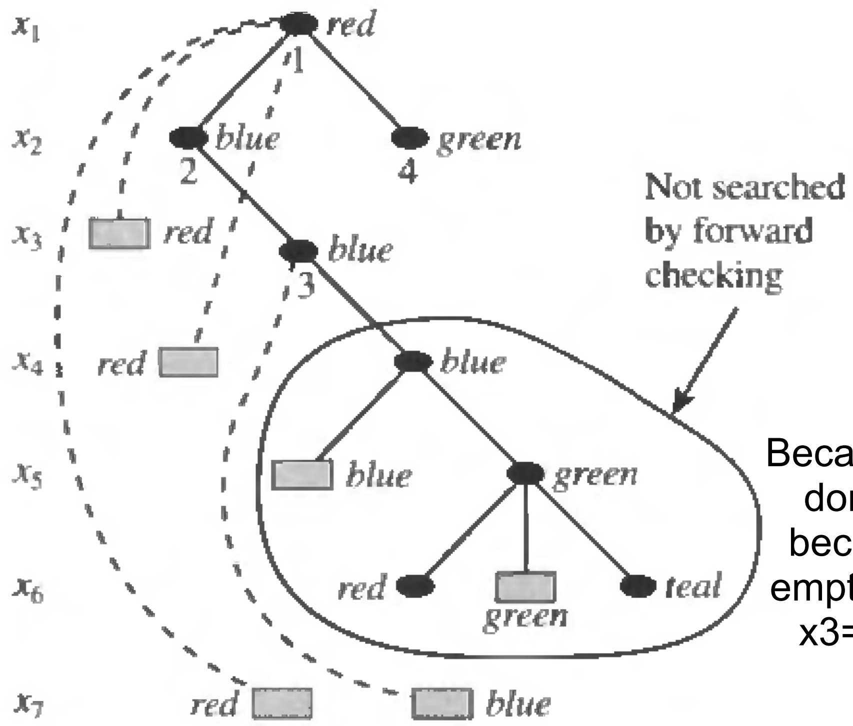
You have
3 minutes!

- Start assigning x_1=red



Forward checking: example

- Dotted lines connect values with future values that are filtered out



Because x7 domain becomes empty after x3=blue

Arc-consistency look-ahead

- Enforce full arc-consistency on all uninstantiated variables following each tentative value assignment to the current variable
  - Stronger than forward-checking
  - Contains a loop implementing AC-1

    But more efficient AC implementations may be considered!

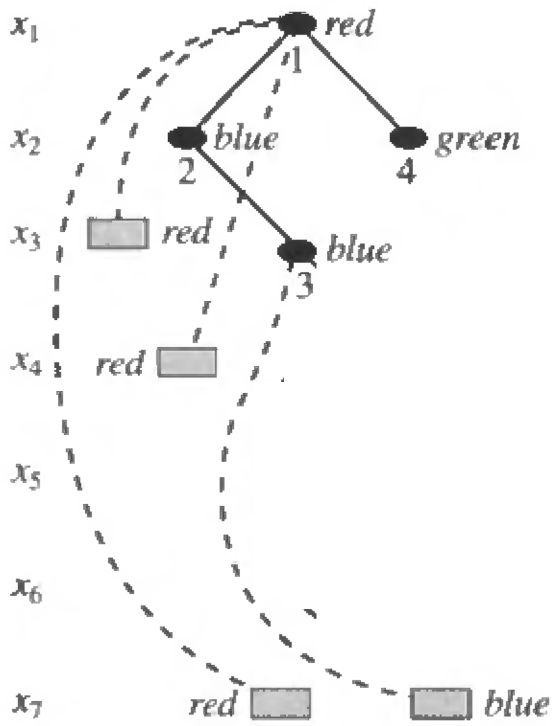
```
subprocedure SELECT-VALUE-ARC-CONSISTENCY

    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        repeat
        removed-value ← false
            for all j, i < j ≤ n
                for all k, k ≠ j, i < k ≤ n
                    for each value b in D'_j
                        if there is no value c ∈ D'_k such that
                                CONSISTENT(a⃗_{i-1}, x_i = a, x_j = b, x_k = c)
                            remove b from D'_j
                            removed-value ← true
                    end for
                end for
            end for
        until removed-value = false
        if any future domain is empty   (don't select a)
            reset each D'_j, i < j ≤ n, to value before a was selected
        else
            return a
    end while
    return null                          (no consistent value)
end procedure
```

Full and partial look-ahead

- More powerfull than forward-checking
- Less powerfull than full arc-consistency
- Full look-ahead makes a single pass through the future variables
  - Repeat until lines are removed from pseudo-code
- Partial look-ahead applies directional arc consistency
  - Future variables are only compared with those variables following them

Partial look-ahead: example

- Partial look-ahead initially considers x_1=red
  - Shrink the domains of x_3, x_4, x_7
  - Process x_4 against x_7; domain of x_4 becomes empty
  - Reject red for x_1
  - The same happens with x_1= blue



Dynamic look-ahead value orderings

- Min-conflicts heuristic
  - Chooses the value in the current variable that removes the smallest number of values from the domain of future variables
- Max-domain-size heuristic
  - Chooses the value in the current variable that creates the largest minimum domain size in the future variables

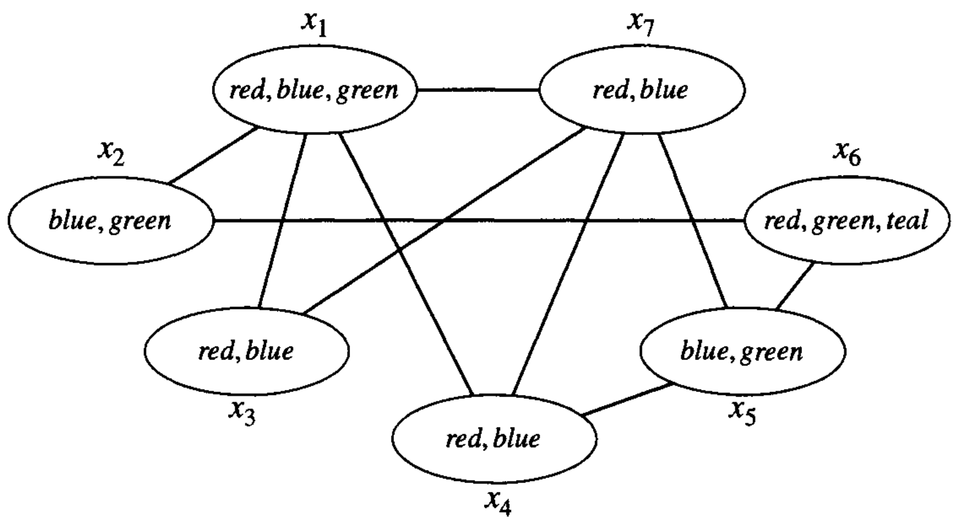
Look-ahead for variable ordering

- Fail-first: select the next variable the one likely to constraint the remainder of the search space the most
  - Usually the variable with the smallest number of viable values
- Dynamic variable ordering (DVO) with forward checking (DVFC)
  - Select variable with domain of minimal size

Look-ahead for variable ordering: example

You have
3 minutes!

- All variables have domain size 2+
- Apply DVFC
- Pick x_7 = blue
- FC reduces domains of x_3, x_4, x_5, x_1
- Pick x_3 = red
- FC reduces domain of x_1
- Pick x_4 = red
- Pick x_2 = blue
- Pick x_5 = green
- Pick x_6 = red/teal



YouTube videos

- https://www.youtube.com/watch?v=TXJ-k9ljDo0
- https://www.youtube.com/watch?v=4LAMiuNd6LI
- https://www.youtube.com/watch?v=YTfcDTsyQHU

Summary

- Introduced backtracking search for solving CSP
- Focused on enhacements that use look-ahead algorithms
  - Look-ahead for value selection
    - Forward checking
    - Arc-consistency look-ahead
    - Full and partial look-ahead
    - Dynamic value orderings
  - Look-ahead for variable ordering
    - Heuristics


That's all Folks!