Constraint Satisfaction Problems (CSP)

- Class of problems solvable using search methods.
- Problems have a well-defined format: <V,D,C>:
  - V is a set of variables.

$$\{X_1, X_2, X_3, \cdots, X_n\}$$

  - D is a set of domains for each variable; usually finite. E.g., {true,false}, {red,green,blue}, [0,10].

$$X_i \in D_i = \{d_{i1}, d_{i2}, \cdots\}$$

  - C is a set of constraints. Each constraint specifies allowable combinations of variables.

$$\{C_1, C_2, C_3, \cdots, C_m\}$$

Constraint Satisfaction Problems (CSP)

- An ASSIGNMENT of values to ALL variables that does NOT violate any constraints is said to be CONSISTENT.
- GOAL is to find a CONSISTENT ASSIGNMENT (if one exists).
- If a GOAL does not exist, perhaps we can say why (i.e., proof of INCONSISTENCY).

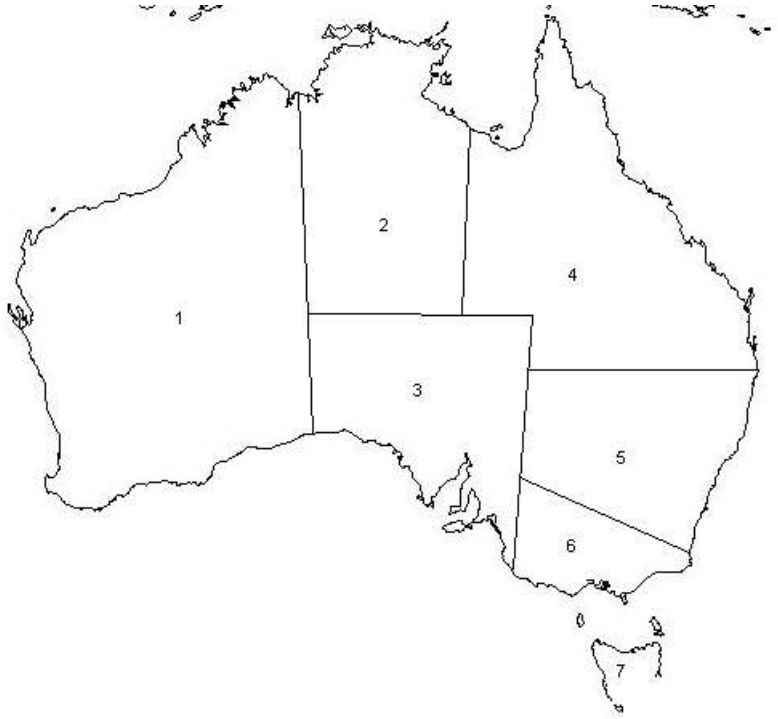
Applications of CSP

- Many real-world problems can be formulated as CSPs, e.g.:
  - VLSI Computer-Aided Design.
  - Optimization.
  - Model Checking/Formal Verification.
  - Planning, Deduction (Theorem Proving).
  - Timetabling/Scheduling.
  - Etc.

Example (Map Coloring)

- Given a map, can the territories (or provinces, states, etc) be colored with k-colors such that no two adjacent territories have the same color?

E.g., Color a map of Australia with 3 colors (red, green, blue)



- Variables:
  - territory1 (X_1), territory2 (X_2), etc.
- Domains:
  - $D_i = \{R,G,B\}$
- Constraints: no two adjacent territories have the same color.
  - E.g., $X_1 \neq X_2$, $X_1 \neq X_3$, etc.

Example (Class Scheduling)

- Given a list of courses to be taught, classrooms available, time slots, and professors who can teach certain courses, can classes be scheduled?

- Variables: Courses offered (C_1, … , C_i) , classrooms (R_1, … , R_j), time (T_1, … , T_k).

- Domains:
  - DC_i = {professors who can teach course i}
  - DR_j = {room numbers}
  - DT_k = {time slots}

- Constraints:
  - Maximum 1 class per room in each time slot.
  - A professor cannot teach 2 classes in the same time slot.
  - A professor cannot teach more than 2 classes.

Approaches for CSPs

- Two main approaches for solving CSPs:
  - Repair/Improvement-based methods.
  - Tree search methods.

Repair/Improvement for CSP

- Each variable is always assigned a value in its domain.
- Check constraints.
  - If consistent assignment stop (solution found).
  - If inconsistent, change a variable to reduce the number of violated constraints (or increase the number of violations the least).
- Do this a lot.
- Result is an INCOMPLETE ALGORITHM.
  - Some sort of iterative improvement.
  - Will explore search space, but perhaps not all of it.
  - Might not find a solution even if one exists.
- WILL NOT talk more about these methods … but will look at a quick example.

Tree Search for CSP

- Use an INCREMENTAL problem formulation:
  - Initial state is NO variables assigned.
  - Pick variable and assign a value provided no conflicts (violated constraints).
  - If conflict, backtrack. If no conflict, pick and assign next variable.
- Goal is at depth n for an n-variable problem.
- The results is a DFS-type algorithm, and inherently depth limited.
- Algorithm is COMPLETE!!!
  - It will expand the entire (finite) search space if necessary.
  - It will find a consistent assignment if one exists.
- Example …

Backtracking Search

Backtracking search simple and systematic:

- Start with no assignment.
- Pick a variable and assign it. Repeat.
- If we hit a dead end, backtrack up the search tree until we find a variable that can have its value set to something different.
- Backing all the way up the tree to the root, and finding no more values means NO SOLN.

DFS that chooses one variable at a time.

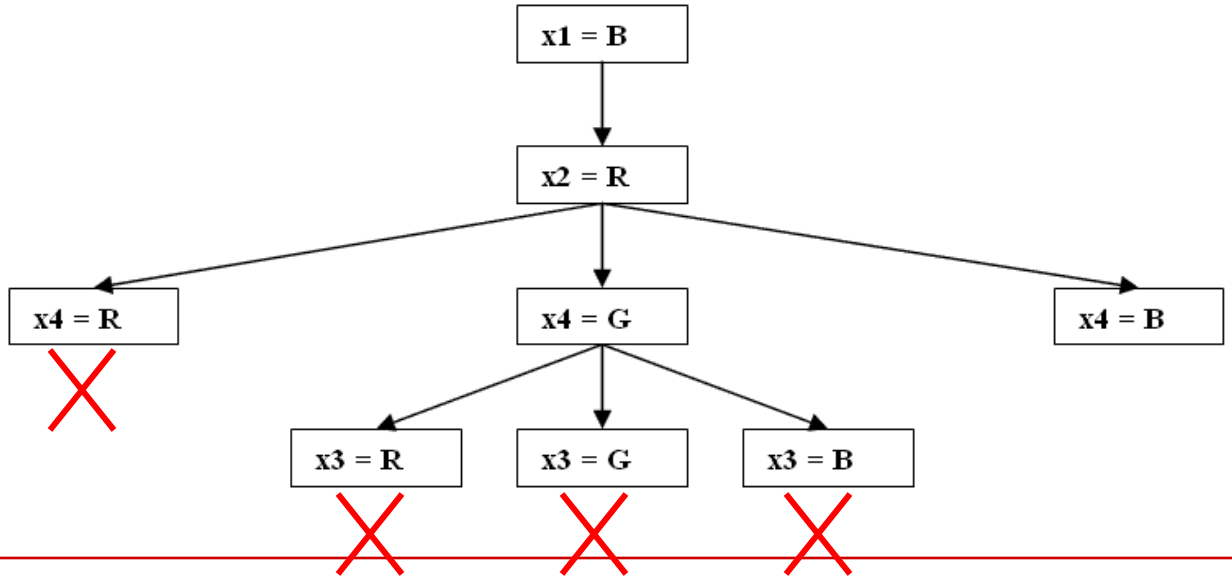
Pseudo-code for Backtracking Search

```
1. BACKTRACK_SEARCH(csp)  {
2.       return BACKTRACK_DFS({},csp);
3. }

4. BACKTRACK_DFS(assignment,csp) {
5.       if ((i = PICK_VAR(assignment,csp)) == NO_UNASSIGNED_VAR) {
6.                return TRUE;
7.       } // X_i is the chosen variable.
8.       for each x in D_i {
9.                assignment += {X_i = x};
10.               if (CONSISTENT(csp)) {
11.                       if (BACKTRACK_DFS(assignment,csp) == TRUE) {
12.                               return TRUE;
13.                       }
14.               }
15.               assignment -= {X_i = x};
16.      }
17.      return FALSE; // failure
18.}
```

Example of Backtracking Search (Map coloring)

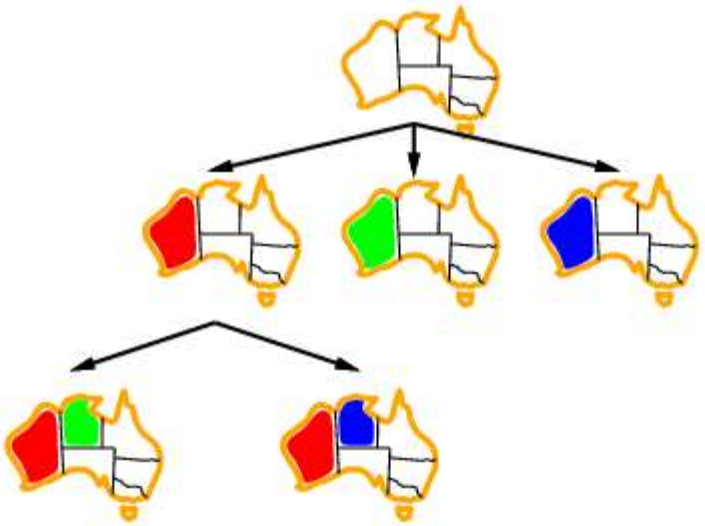
- Consider the map of Australia again.
- Assume we assign variables in the following order: $\{X_1, X_2, X_4, X_3, \ldots\}$, and values in the order {R,G,B}
- Also, assume we have two additional constraints on X_1: $X_1 \neq R$ and $X_1 \neq G$.
- A backtracking search will produce this search tree.
  - **NOTE:** X_3 has no consistent assignment when $\{X_1,X_2,X_4\} = \{B,R,G\}$.



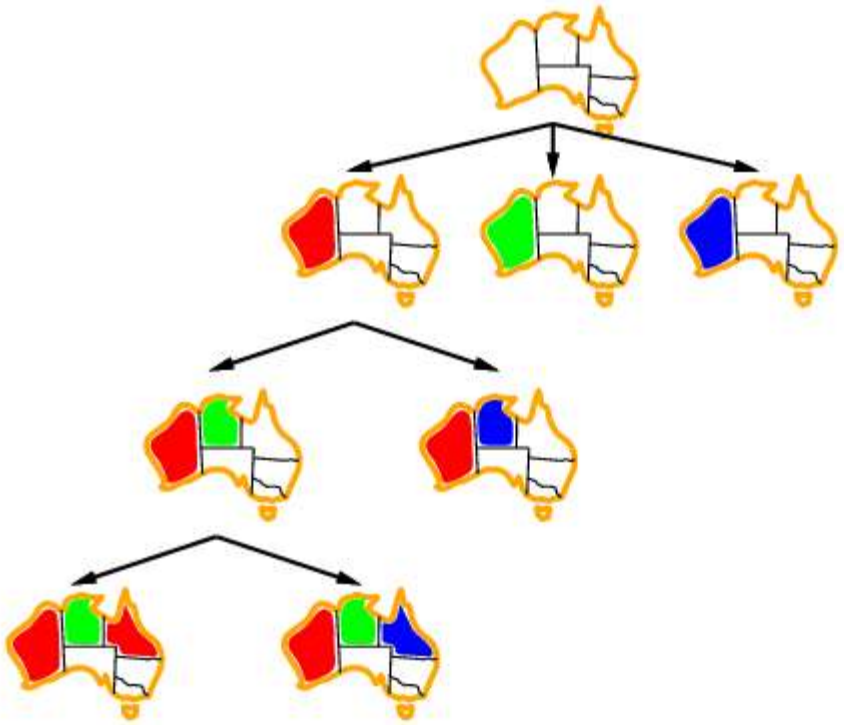
Backtracking example

Backtracking example

Backtracking example



Backtracking example



Issues in Backtracking Search for CSP

- How do we chose the next variable and its value?
- Can we prune the search space, and thereby search less?
- Can we **learn from our mistakes**, i.e., bad variable selections?

Variable and Value Selection

- Selecting variables and assigning values using a static list is not always the most efficient approach.
  - Difficult to make the "right" choice for picking and setting the next variable.

HEURISTICS can help here, e.g.,

- "Minimum remaining values" heuristic – choose the variable with the smallest number of remaining values in its domain.
  - Also called "most constrained variable" heuristic
- "Degree heuristic" – choose the variable that is part of the most remaining unsatisfied constraints.
  - Useful to select first variable to assign.
- "Least-constraining-value" heuristic – once a variable is chosen, choose its value as the one that rules out the fewest choices for neighboring variables.
  - Keeps maximum flexibility for future variable assignments.

Improving Backtracking Search (Forward Checking)

- Plain backtracking search is not very efficient or intelligent.
  - It does not use information available in the constraint set and assigned variables.

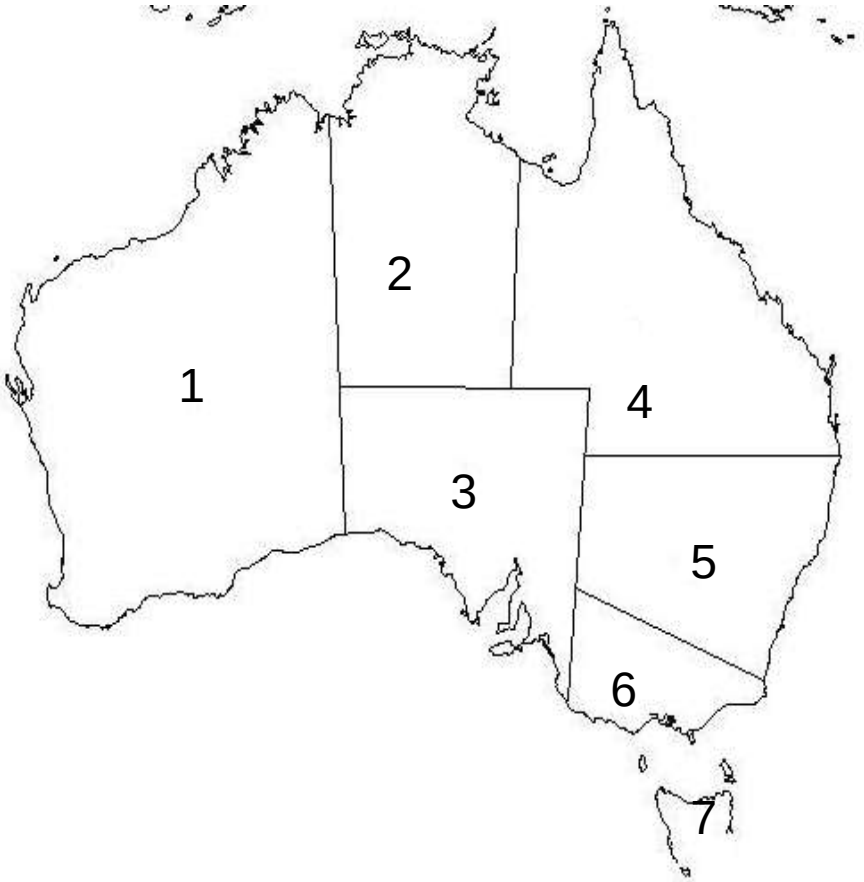
- When we assign a variable, its value and constraints can be used to PRUNE the domains of FUTURE VARIABLES.

  - If the domain of a FUTURE VARIABLE becomes empty, we know we can backtrack.

  - No need to explore the subtree below the current variable/value pair.

Forward Checking Example (Map Coloring)

- Consider coloring Australia with red, green and blue {R,G,B}.
- Assume decisions are X_1 = R and X_4 = G.



		X1 = R	X4 = G
D1	R,G,B	R	R
D2	R,G,B	G,B	B
D3	R,G,B	G,B	B
D4	R,G,B	R,G,B	G
D5	R,G,B	R,G,B	R,B
D6	R,G,B	R,G,B	R,G,B
D7	R,G,B	R,G,B	R,G,B

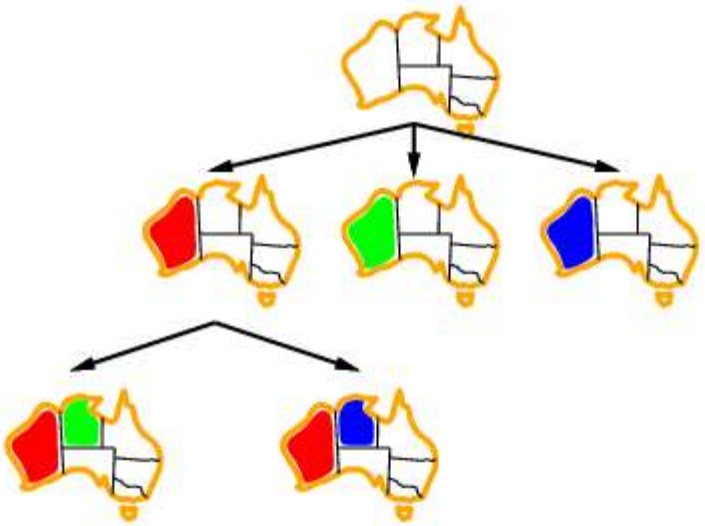
variable settings

reduced domains with forward checking

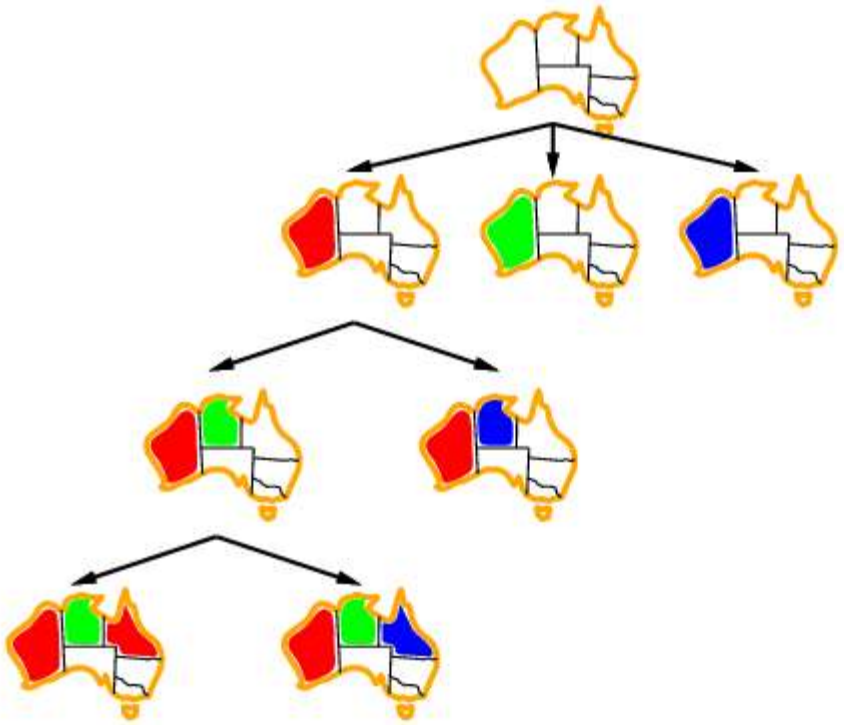
Backtracking example

Backtracking example

Backtracking example



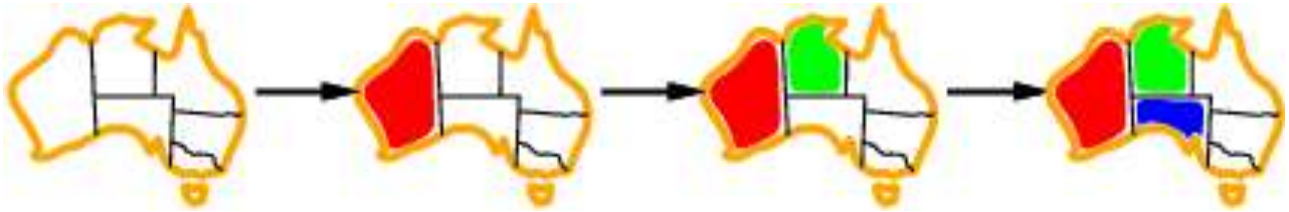
Backtracking example



Improving backtracking efficiency

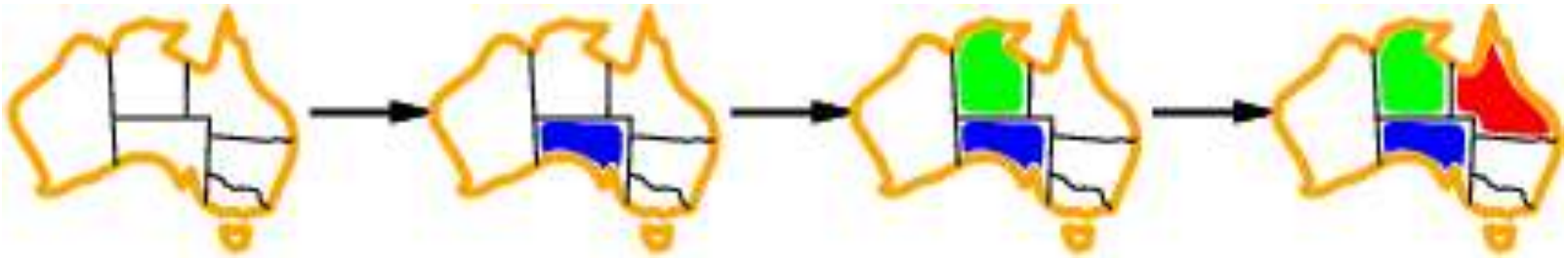
- *General-purpose* methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

Most constrained variable

- Most constrained variable:
  choose the variable with the fewest legal values
- a 

Most constraining variable

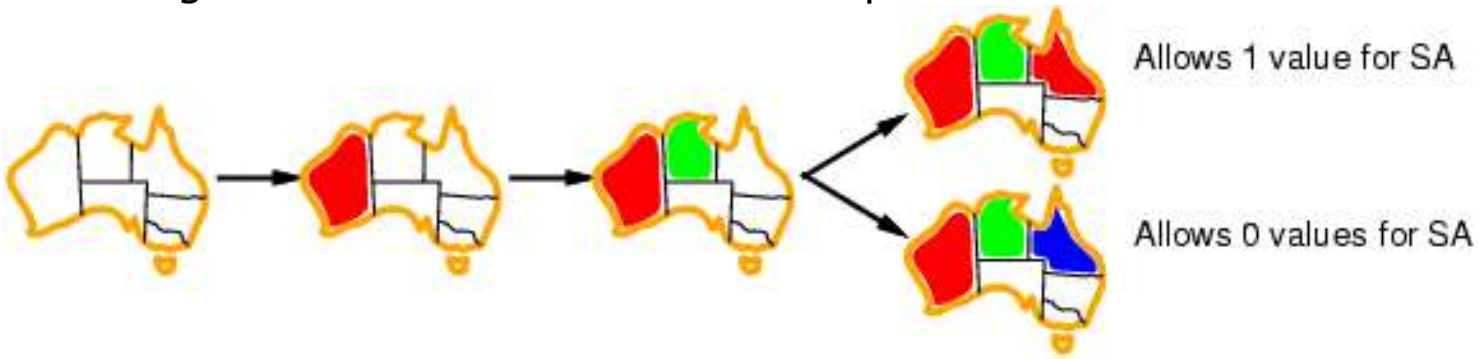
- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables



Least constraining value

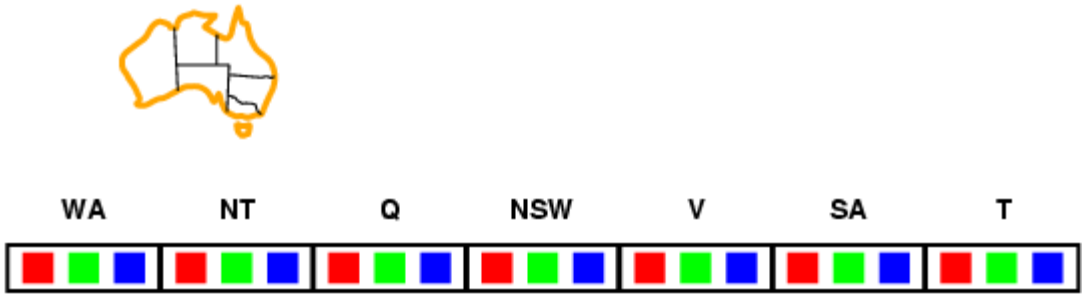
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

- Combining these heuristics makes 1000 queens feasible



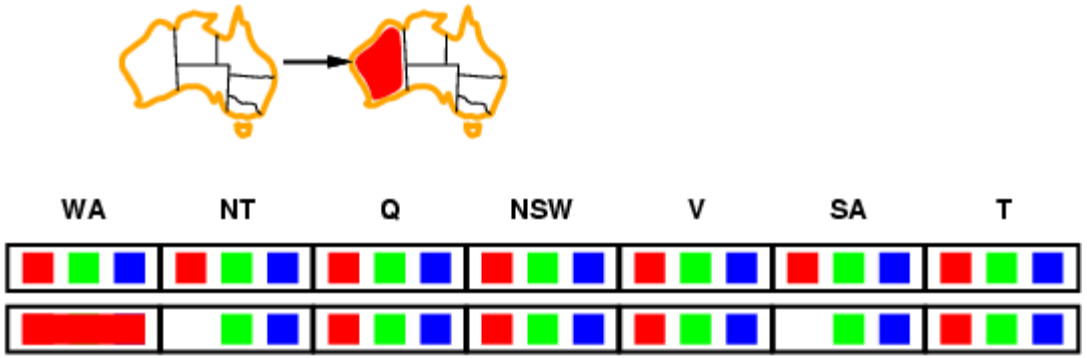
Forward checking ("edge consistency" is a variant)

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



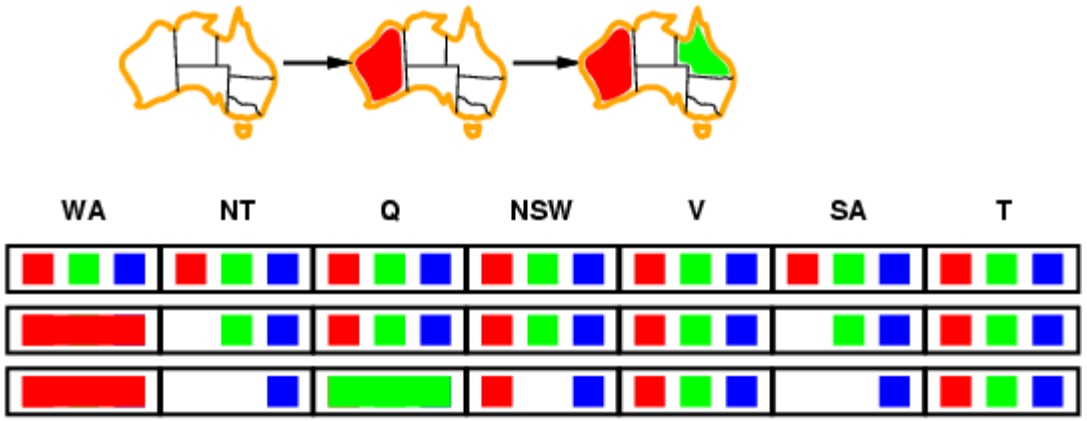
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



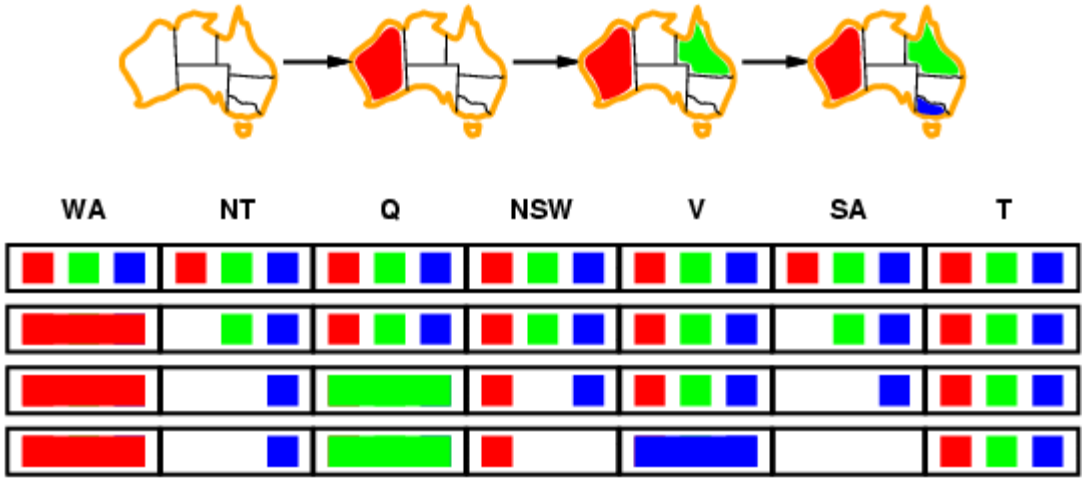
Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



Forward checking

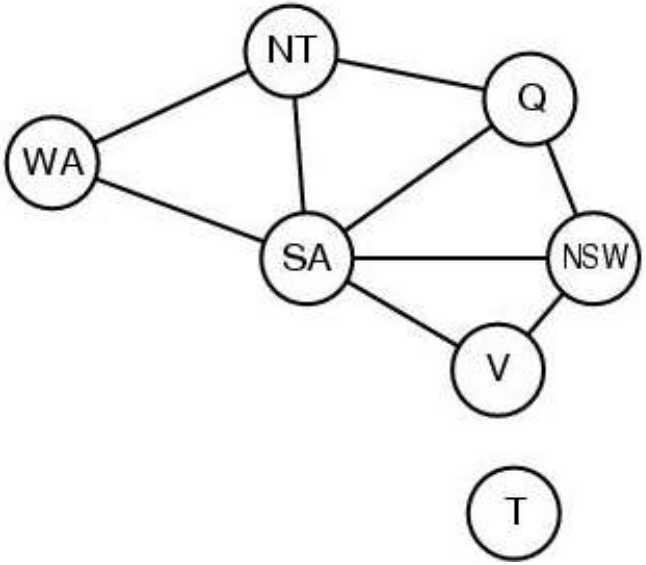
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



- A Step toward AC-3: The most efficient algorithm

Constraint graph

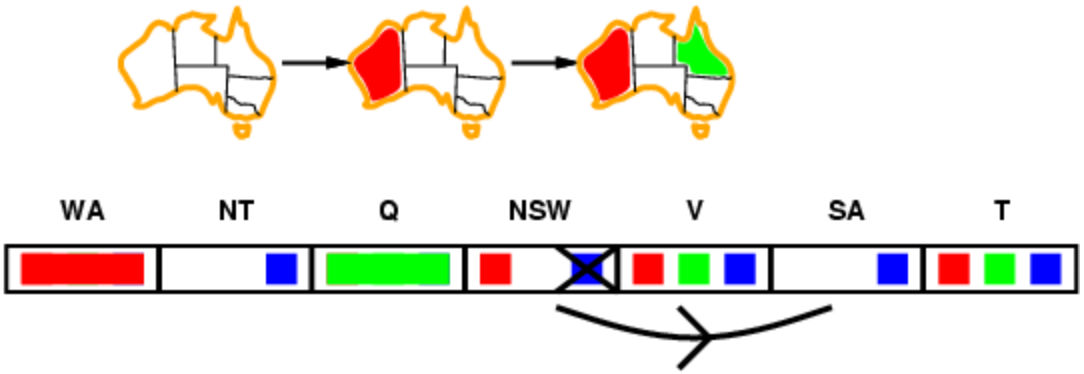
- *Binary CSP:* each constraint relates two variables
- *Constraint graph:*
  - *nodes* are variables
  - *arcs* are constraints
- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).
- Graph can be used to simplify search.
  - **e.g. Tasmania is an independent subproblem**.



Arc Consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
-

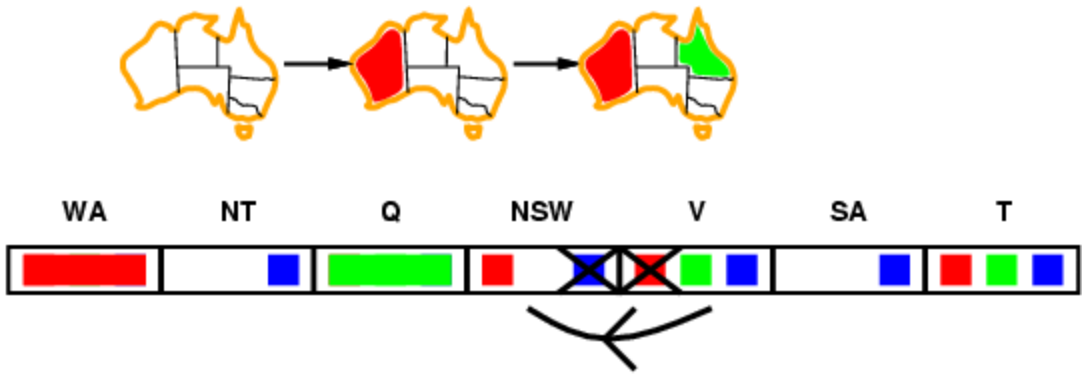
  for every value x of X there is some allowed y for Y



Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
-

  for every value x of X there is some allowed y

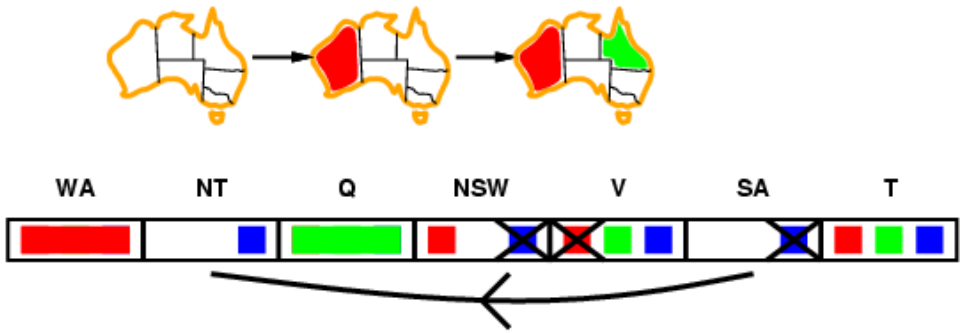


- If X loses a value, neighbors of X need to be rechecked
-

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
-

  for every value x of X there is some allowed y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
-
- Can be run as a preprocessor or after each assignment
-

Types of Consistency

- Would like to achieve consistency in our variable assignments
  - Will look at two types of consistency …

- NODE CONSISTENCY (1-consistency) –

  - a node is consistent if and only if all values in its domain satisfy all unary constraints on the corresponding variable. (**note change here)**
  - Unary constraint contains only one variable, e.g., $x_1 \neq R$

- ARC CONSISTENCY (2-consistency) –

  - an arc, or edge, $(x_i \rightarrow x_j)$ in the constraint graph is consistent if and only if for every value 'a' in the domain of x_i, there is some value 'b' in the domain of x_j such that the assignment $\{x_i, x_j\} = \{a, b\}$ is permitted by the constraint between x_i and x_j.
  - Example on next slide.

Consistency cont'd

- NODE CONSISTENCY is simple to achieve; simply scan values for variables and remove those that are not valid. (**note change here**)

- ARC CONSISTENCY needs to examine edges and delete values from domains to make arcs consistent.

  - In previous slide, can remove B from domain of X_5 since (X_5, X_3) is not arc consistent with B in the domain of X_5.

  - When a value is removed from a domain, arcs *to that variable* (**change**) need to be examined again
    - e.g., if B removed from domain of X_5 to make (X_5, X_3) arc consistent, will need to recheck (X_3, X_5) for arc consistency.

- The main point is that maintaining NODE and ARC consistency further reduces the potential DOMAINS of variables, thereby reducing the amount of searching.

AC-3 Algorithm

- The most popular algorithm to check arc consistency is known as AC-3 (short for arc consistency algorithm #3).
- Keeps a queue of arcs (X_i, X_j) to be verified for consistency.
- If a value(s) need be removed from the domain of X_i, every arc (X_k, X_i) pointing to X_i must be checked (reinserted in the queue).
- It is substantially more expensive than forward checking, but usually the cost is worthwhile.
  - Consistent assignment found faster than with forward checking.

Pseudo-code for AC-3

```
1.  AC_3( csp ) {
2.      queue = find_arcs( csp );
3.      while( !empty(queue) ) {
4.          arc = pop_front( queue ); // remove first arc (Xi, Xj)
5.          if( inconsistent( arc ) == TRUE ) {
6.              // add neighbors to queue, N = # neighbors
7.              for( n = 1; n <= N; n++ )
8.                  push( queue, n );
9.  }   }   }

10. bool inconsistent( arc ) {
11.     removed = FALSE;
12.     for( all xi in the domain of Xi )
13.         if( no xj exists such that (xi, xj) is legal )
14.             delete xi; removed = TRUE;
15.     return removed;
16. }
```

Final Comment on Constraint Propagation

- As constraint propagation techniques get more involved (in order to more effectively prune variable domains), CPU time increases.
  - Involves consistencies at a higher level than arc-consistency
    - E.g., 3-consistent ("path consistency"), k-consistent.
    - if a n-node CSP can be shown to be n-consistent, a solution can be guaranteed **with no backtracking**!
- There is a fundamental tradeoff between pruning and searching.
  - If pruning takes longer than simple searching, it is not worth it even though it reduces the amount of search.

Conflict-Directed Backjumping (CBJ)

- In its simple form, BACKTRACKING search backtracks to the first point where a variable can be assigned a new value.
  - It backs up ONE level in the search tree at a time.
  - Example of a CHRONOLOGICAL BACKTRACK.
- When we hit a dead end due to an inconsistency, we can try and deduce the reason for the problem.
  - Rather than backing up one level in the search tree, we can try to go directly to one of the variables that caused the problem.
  - Might need to skip levels in the tree.

.

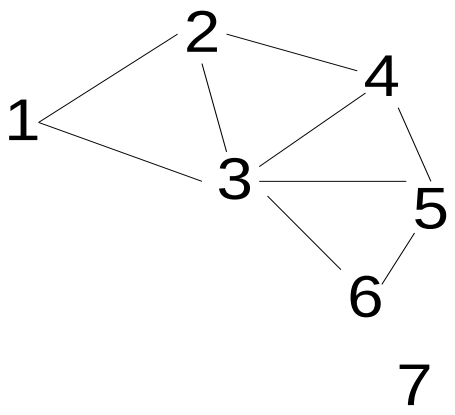
Conflict-Directed Backjumping (CBJ)

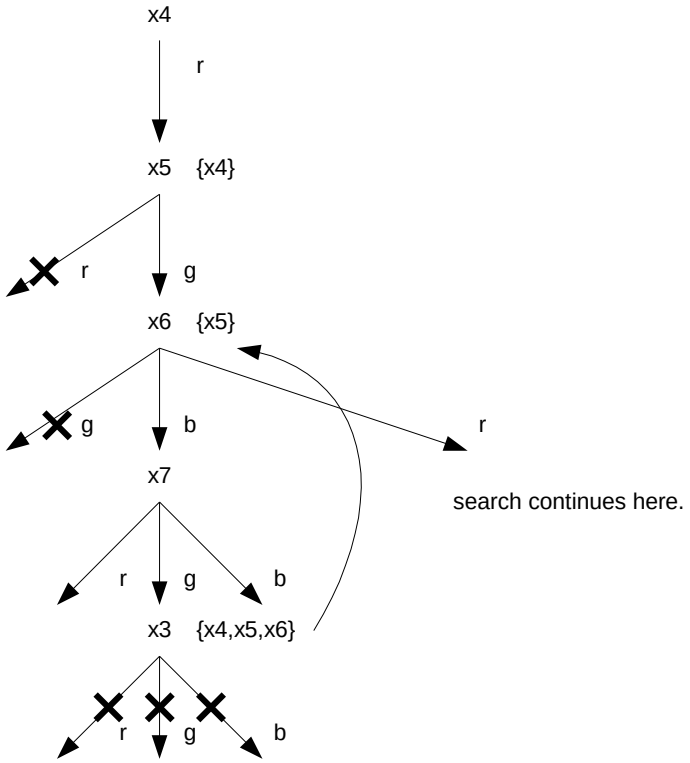
- Idea: Maintain a CONFLICT SET for every variable as it gets set.
- Assume we are presently setting X_i. The CONFLICT SET for X_i is the set of PREVIOUSLY ASSIGNED VARIABLES connected to X_i by a constraint.
- When we are done with the current variable X_i (no solution found), we can BACKJUMP to the deepest X_k in the conflict set for X_i.
- We can also update the conflict set of X_k to indicate that we have learned something in exploring the subtree below X_i.

  $CONFLICT\_SET(X_k) = CONFLICT\_SET(X_k) \mathbf{U} CONFLICT\_SET(X_i) - X_k$

Example of Conflict-Directed Backjumping

- Consider our graph coloring problem:





- X_7 is not in the conflict set for X_3, so back up to the closest that is (X_6).

- There are other methods besides CBJ, but we will NOT consider them.

Boolean Satisfiability (SAT)

- An important (special) sub-class of CSPs.
  - All variables are binary and have domain {0,1}.
  - Constraints are expressed in Conjunctive Normal Form (CNF)
    - Individual literals are combined using disjunction (OR-operation) to make constraints or "clauses." Clauses combined using conjunction (AND-operation).

$$C_1 = (x_1 \vee \overline{x_2} \vee x_9)$$ ← Single constraint

$$C = \wedge C_i, \forall i = 1, \cdots, m$$ ← All constraints

- We seek an variable assignment such that $C = 1$.
- For SAT problems, COMPLETE algorithms based on DFS search are commonly referred to as Davis-Putnam (DP) or Davis-Putnam Logemann Loveland (DPLL) algorithms.
  - They are also a tree search (BINARY TREE).

Issues

- Some problems with CSPs:
  - Variable selection and value ordering.
  - Constraint propagation and pruning.
  - Backtracking.

Constraint Propagation

- Consider selecting and setting a binary variable.
  - If we examine constraints involving this variable, we can reduce domains of some other variables.

- Situations for a constraint:
  - Some literal is true – constraint **satisified** (due to disjunction).
  - All literals are false, except for one unknown literal – **unit** constraint/clause.
  - All literals are zero – **conflict** constraint/clause.

- Some example constraints:

c1	!x1 or x2
c2	!x1 or x3 or x9
c3	!x2 or !x3 or x4
c4	!x4 or x5 or x10
c5	!x4 or x6 or x11
c6	!x5 or x6
c7	x1 or x7 or !x12
c8	x1 or x8
c9	!x13 or x12
c10	x11 or !x10

Constraint Propagation

- If a constraint has a CONFLICT, we backtrack. If a constraint is UNIT, then we reduce the domain of the unset variable such that the constraint is SATISFIED.

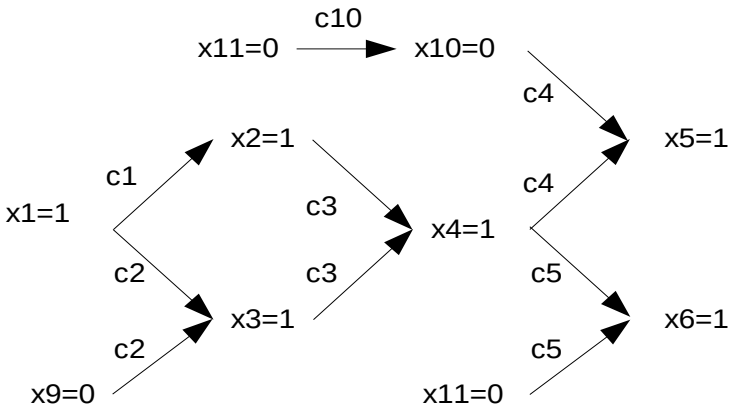
- Since we only have one choice, its value is IMPLIED.
  - E.g., recall constraint C_1 on previous slide: $C_1 = !X_1$ or X_2.
    - if $X_2 = 0$, constraint C_1 is unit (one unresolved variable).
    - In order to have $C_1 = 1$, the domain of X_1 is reduced to 0.

Unit Propagation and Implication Graphs

- Consider certain decisions: x9=0@1, x13=1@2, x11=0@3 and x1=1@6
  - '@' refers to the level in the search that the decision was made.

- Consider the effect of these decision using an "implication graph"

c1	!x1 or x2
c2	!x1 or x3 or x9
c3	!x2 or !x3 or x4
c4	!x4 or x5 or x10
c5	!x4 or x6 or x11
c6	!x5 or x6
c7	x1 or x7 or !x12
c8	x1 or x8
c9	!x13 or x12
c10	x11 or !x10

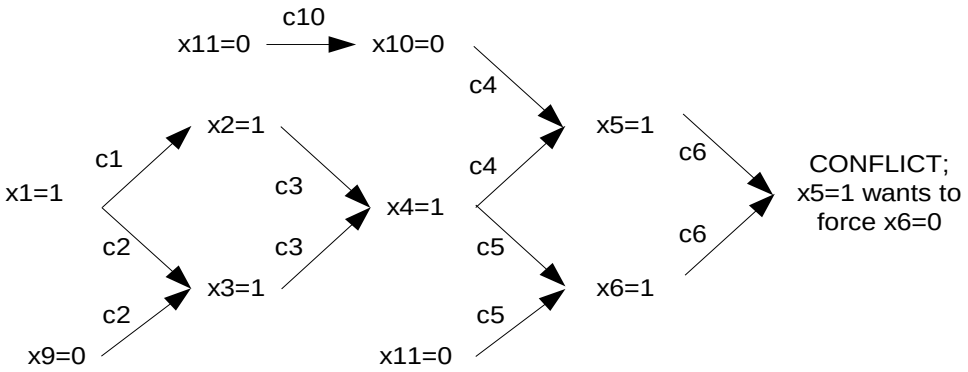


- Some values get implied due to these DECISIONS.

Unit Propagations and Conflicting Clauses

- Consider the following set of constraints and decisions: x9=0@1, x13=1@2, x11=0@3 and x1=1@6

c1	!x1 or x2
c2	!x1 or x3 or x9
c3	!x2 or !x3 or x4
c4	!x4 or x5 or x10
c5	!x4 or x6 or x11
c6	!x5 or !x6
c7	x1 or x7 or !x12
c8	x1 or x8
c9	!x13 or x12
c10	x11 or !x10



- **Note:** After setting x1=1@6, we get a conflict on the value for x6 (it can't be 0 and 1 to satisfy c6 and c5, respectively).

Algorithm for Unit Propagation

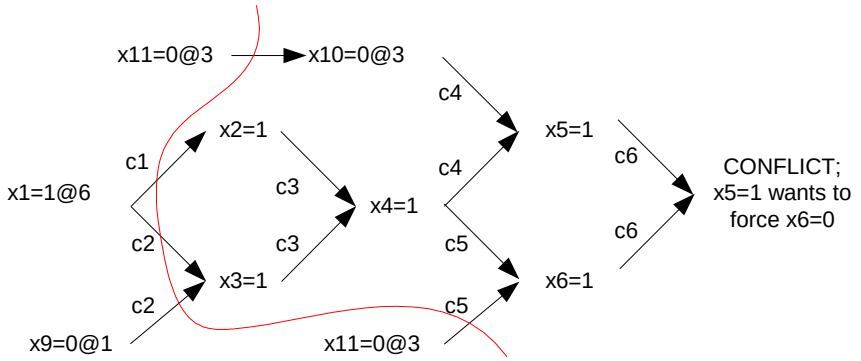
```
1.  UP(decision_var,decision_val) {
2.        queue.push(decision_var,decision_val,NULL);
3.        while (queue.size() != 0) {
4.                  (var,val) = queue.pop();
5.                  set_var(var,val);
6.                  for each constraint c_i {
7.                            if (is_unit_constraint(c_i)) {
8.                                      queue.push(new_var,new_val,c_i);
9.                            }
10.                           else if (is_empty_constraint(c_i)) {
11.                                     return NOT_CONSISTENT;
12.                           }
13.                 }
14.       }
15.       return CONSISTENT;
16. }
```

- State of the art SAT solvers use almost exclusively Unit Propagation (GRASP, zChaff, etc).
- There are more complex methods for deriving implications.

Conflict-Directed Backtracking In SAT

- Assume decisions are x9=0@1, x13=1@2, x11=0@3,x1=1@6.

c1	!x1 or x2
c2	!x1 or x3 or x9
c3	!x2 or !x3 or x4
c4	!x4 or x5 or x10
c5	!x4 or x6 or x11
c6	!x5 or !x6
c7	x1 or x7 or !x12
c8	x1 or x8
c9	!x13 or x12
c10	x11 or !x10



- Our decisions have lead to a conflict and we can deduce a reason for the conflict.
  - Backtrack to the most recent decision that could have produced the conflict (x1=1@6)

Learning From Conflicts

- Conflict because of the decisions x9=0@1, x13=1@2, x11=0@3,x1=1@6 .
- Therefore, it must be true that we do NOT have these variable settings. I.e., that we pick these variables such that:

  NOT(x9=0, x11=0,x1=1)
- Using rules from Boolean logic, it must be true in any solution that the NEW CONSTRAINT is true:

  x9 OR x11 OR !x1

Learning From Conflicts

- The reason for the conflict is a clause that must also be sastisfied, but does not exist in our database of clauses.
- We can add this constraint into our constraint database, and we have LEARNED something.
  - We will now and forever avoid the "bad" variable settings.
- NOTE: We can backtrack from level 6 to level 3, where x11 was decided:
  - Our new constraint will IMPLY x1 to 0, preventing a conflicting setting and force us down a different path in the search tree.

Algorithm for SAT

```
1.  SAT {
2.        while (1) {
3.                if (decide() != NO_VARIABLE) {
4.                    while (up() = CONFLICT) {
5.                          backtrack_level = analyze_conflicts();
6.                          if (backtrack_level < 0) {
7.                                    return NO_SOLUTION;
8.                          }
9.                          else {
10.                                   backtrack(backtrack_level);
11.                         }
12.                   }
13.               else {
14.                         return SOLUTION;
15.               }
16.       }
17. }
```