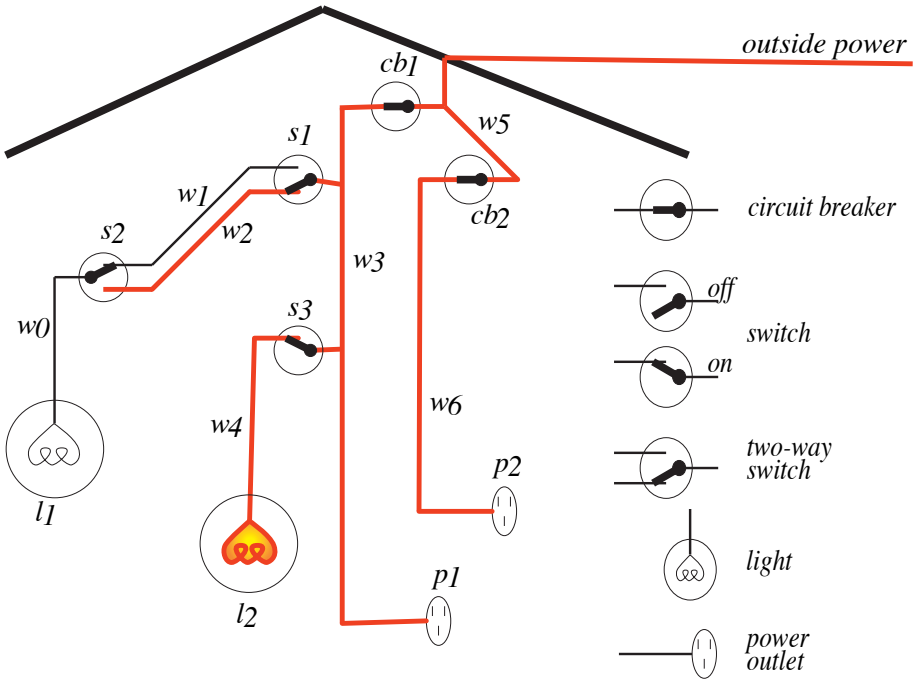
# Features Describing Problems

- Instead of reasoning explicitly in terms of states, it is often better to describe states in terms of *features* and then to reason in terms of these features.
- Often these features are not independent and there are constraints that specify legal combinations of assignments of values to variables corresponding to features.

# Features Describing Problems

- For any practical problem, an agent cannot reason in terms of states; there are simply too many of them.
- Most problems do not come with an explicit list of states; the states are typically and more naturally described implicitly in terms of features.
- States can be defined in terms of features: features can be primitive and a state corresponds to an assignment of a value to each feature.
- Features can be defined in terms of states: the states can be primitive and a feature is a function of the states. Given a state, the function returns the value of the feature on that state.
- Each feature has a domain that is the set of values that it can take on.
- The domain of the feature is the range of the function on the states.

# Example




- a feature for each switch (up or down);
- a feature for each light (lit or not);
- a feature for each component (working properly or broken).

- A state consists of the position of every switch, the status of every device, and so on.
- A state may be described as switch 1 is up, switch 2 is down, fuse 1 is okay, wire 3 is broken, and so on.
- If the states are primitive, a function may be, for example, the position of switch 1. The position is a function of the state, and it may be up in some states and down in other states.

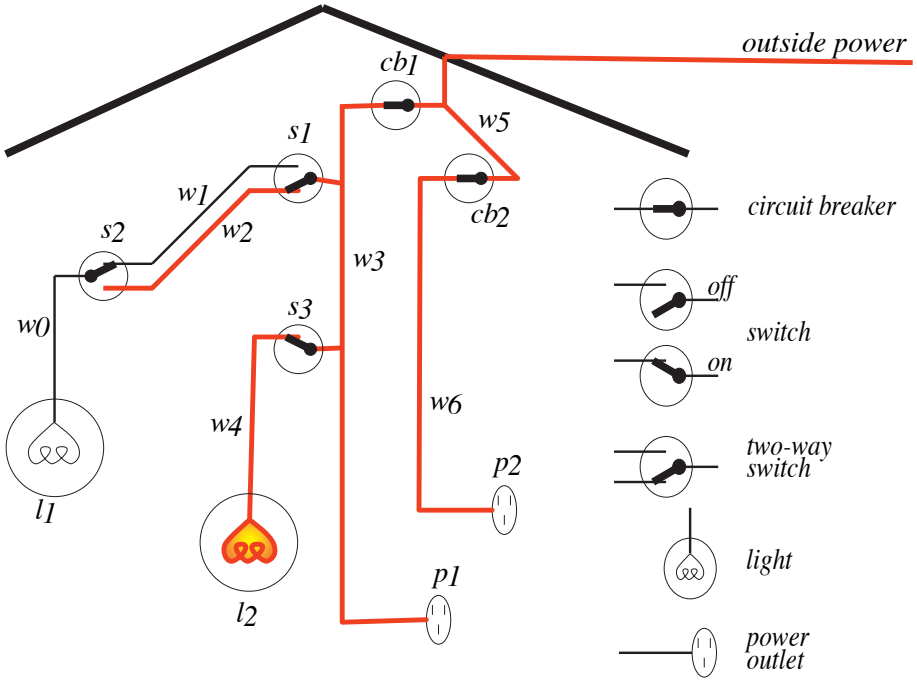
# Features vs States

- One main advantage of reasoning in terms of features is the computational savings:
  - 10 binary features can describe 210=1,024 states.
  - 20 binary features can describe 220=1,048,576 states.
  - 30 binary features can describe 230=1,073,741,824 states.
  - 100 binary features can describe 2100=1,267,650,600,228,229,401,496,703,205,376 states.
- Reasoning in terms of thirty features may be easier than reasoning in terms of more than a billion states.
- Many problems have thousands if not millions of features.
- Typically the features are not independent, in that there may be constraints on the values of different features. One problem is to determine what states are possible given the features and the constraints.

# Variables and Possible Worlds

- A direct one-to-one correspondence between features and variables, and between states and possible worlds.
- Possible worlds are described by (algebraic) variables, which are symbols used to denote features of possible worlds.
- Algebraic variables will be written starting with an upper-case letter.
- Each algebraic variable $V$ has an associated domain, $\mathbf{D}_V$, which is the set of values the variable can take on.
- A discrete variable is one whose domain is finite or countably infinite.
- One particular case of a discrete variable is a Boolean variable, which is a variable with domain $\{\text{true}, \text{false}\}$.
- A non-discrete variable whose domain corresponds to a subset of the real line is, for example, a continuous variable.

# Example




- S1-pos - a binary variable with domain {up, down}, where S1-pos=up means switch s1 is up, and S1-pos=down means switch s1 is down

- S1-st - a variable with domain {ok, upside-down, short, intermittent, broken}
- Number-of-broken-switches - an integer-valued variable denoting the number of switches that are broken.
- Current-w1 - a real-valued variable denoting the current, in amps, flowing through wire w1.
- We also allow inequalities between variables and constants as Boolean features; for example, Current-w1 $\geq$ 1.3 is true when there are at least 1.3 amps flowing through wire w1.

# Variables and Possible Worlds

- Possible worlds can be defined in terms of variables or variables can be defined in terms of possible worlds:
  - Variables can be primitive and a possible world corresponds to a total assignment of a value to each variable.
  - Worlds can be primitive and a variable is a function from possible worlds into the domain of the variable; given a possible world, the function returns the value of that variable in that possible world.

# Constraints

- In many domains, not all possible assignments of values to variables are permissible.
- A hard constraint, or simply constraint, specifies legal combinations of assignments of values to the variables.
- A scope or scheme is a set of variables.
- A tuple on scope $S$ is an assignment of a value to each variable in $S$.
- A constraint $c$ on a scope $S$ is a set of tuples on $S$.
- A constraint is said to involve each of the variables in its scope.
- If $S'$ is a set of variables such that $S \subseteq S'$, and $t$ is a tuple on $S'$, constraint $c$ is said to **satisfy** $t$ if $t$, restricted to $S$, is in $c$.

# Representing Constraints

- Constraints are also often defined intensionally, in terms of predicates (Boolean functions), to recognize legal assignments rather than extensionally by representing each assignment explicitly in a table.
- Extensional definitions can be implemented either by representing the legal assignments or by representing the illegal assignments.

# Models

- A possible world $w$ satisfies a set of constraints if, for every constraint, the values assigned in $w$ to the variables in the scope of the constraint satisfy the constraint.
- In this case, we say that the possible world is a model of the constraints.
- That is, a model is a possible world that satisfies all of the constraints.

# Example: Scheduling a Delivery Robot

- Suppose the delivery robot must carry out a number of delivery activities: a, b, c, d, and e.
- Suppose that each activity happens at any of times 1, 2, 3, or 4.
- Let A be the variable representing the time that activity a will occur, and similarly for the other activities.
- The variable domains, which represent possible times for each of the deliveries, are
  $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1,2,3,4\}$.
- Suppose the following constraints must be satisfied:
  $\{(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D), (E < A), (E < B), (E < C), (E < D), (B \neq D)\}$
- The aim is to find a model, an assignment of a value to each variable, such that all the constraints are satisfied.

# Constraint Satisfaction Problems

- Given a set of variables, each with a set of possible values (a domain), assign a value to each variable that either
  - satisfies some set of constraints: satisfiability problems — "hard constraints"
  - minimizes some cost function, where each assignment of values to variables has some cost: optimization problems — "soft constraints"
- Many problems are a mix of hard and soft constraints.

# Relationship to Search

- The path to a goal isn't important, only the solution is.
- Many algorithms exploit the multi-dimensional nature of the problems.
- There are no predefined starting nodes.
- Often these problems are huge, with thousands of variables, so systematically searching the space is infeasible.
- For optimization problems, there are no well-defined goal nodes.

# Posing a Constraint Satisfaction Problem

A (finite) CSP is characterized by

- A (finite) set of variables $V_1, V_2, \ldots, V_n$.
- Each variable $V_i$ has an associated (finite) domain $\mathbf{D}_{V_i}$ of possible values.
- For satisfiability problems, there are constraints on various subsets of the variables which specify legal combinations of values for these variables.
- A solution to the CSP is an $n$-tuple of values for the variables that satisfies all the constraints.

# Example: Scheduling a Delivery Robot

- Variables: $A$, $B$, $C$, $D$, $E$ that represent the starting times of various activities.
- Domains: $\mathbf{D}_A = \{1, 2, 3, 4\}$, $\mathbf{D}_B = \{1, 2, 3, 4\}$, $\mathbf{D}_C = \{1, 2, 3, 4\}$, $\mathbf{D}_D = \{1, 2, 3, 4\}$, $\mathbf{D}_E = \{1, 2, 3, 4\}$
- Constraints:

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$$
$$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge$$
$$(E < C) \wedge (E < D) \wedge (B \neq D).$$

# Tasks for a CSP

- Given a CSP, there are a number of tasks that can be performed:
  - Determine whether or not there is a model.
  - Find a model.
  - Find all of the models.
  - Count the number of models.
  - Find the best model, given a measure of how good models are.
  - Determine whether some statement holds in all models.
- We mostly considers the problem of finding a model.
- Some of the methods can also determine if there is no solution.

# Complexity

- CSPs are very common, so it is worth trying to find relatively efficient ways to solve them.
- Determining whether there is a model for a CSP with finite domains is NP-hard and no known algorithms exist to solve such problems that do not use exponential time in the worst case.
- However, just because a problem is NP-hard does not mean that all instances are difficult to solve. Many instances have structure that can be exploited.

# Generate-and-Test Algorithm

- Generate the assignment space $\mathbf{D} = \mathbf{D}_{V_1} \times \mathbf{D}_{V_2} \times \ldots \times \mathbf{D}_{V_n}$. Test each assignment with the constraints.
- Example:

$$\begin{aligned}
\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\
&\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, ..., \langle 4,4,4,4,4\rangle\}.
\end{aligned}$$

- Generate-and-test is always exponential in the number of variables.

# Backtracking Algorithms

- Systematically explore **D** by instantiating the variables one at a time
- evaluate each constraint predicate as soon as all its variables are bound
- any partial assignment that doesn't satisfy the constraint can be pruned.
- Example:
  - In the previous delivery scheduling problem, assignment $A = 1 \wedge B = 1$ is inconsistent with constraint $A \neq B$ regardless of the value of the other variables.
  - If the variables $A$ and $B$ are assigned values first, this inconsistency can be discovered before any values are assigned to $C$, $D$, or $E$, thus saving a large amount of work.

# CSP as Graph Searching

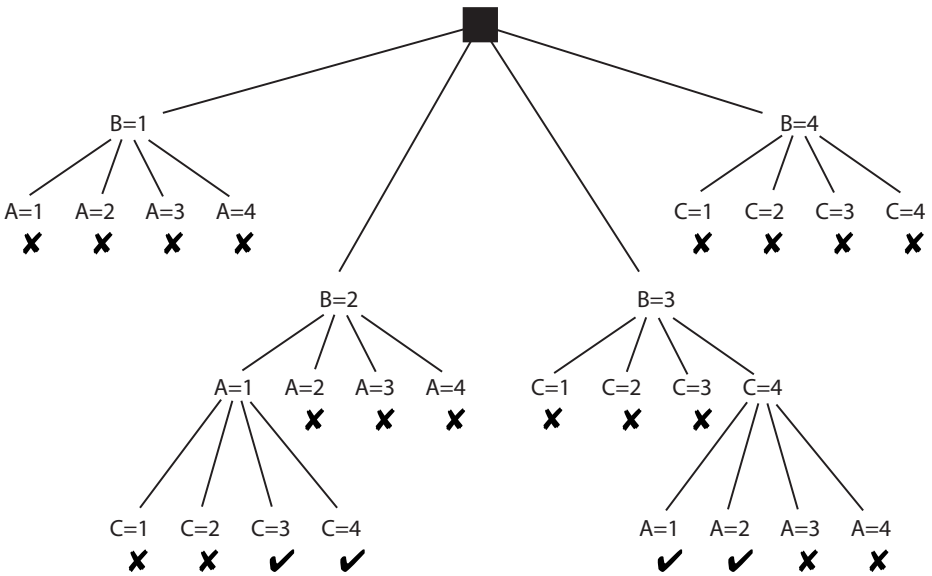
A CSP can be represented as a graph-searching algorithm:

- A node is an assignment values to some of the variables.
- Suppose node $N$ is the assignment $X_1 = v_1, \ldots, X_k = v_k$. Select a variable $Y$ that isn't assigned in $N$.
  For each value $y_i \in dom(Y)$ there is a neighbour
  $X_1 = v_1, \ldots, X_k = v_k, Y = y_i$ if this assignment is consistent with the constraints on these variables.
- The start node is the empty assignment.
- A goal node is a total assignment that satisfies the constraints.

- Generate and test is equivalent to not checking constraints until reaching the leaves.
- Checking constraints higher in the tree can prune large subtrees that do not have to be searched.

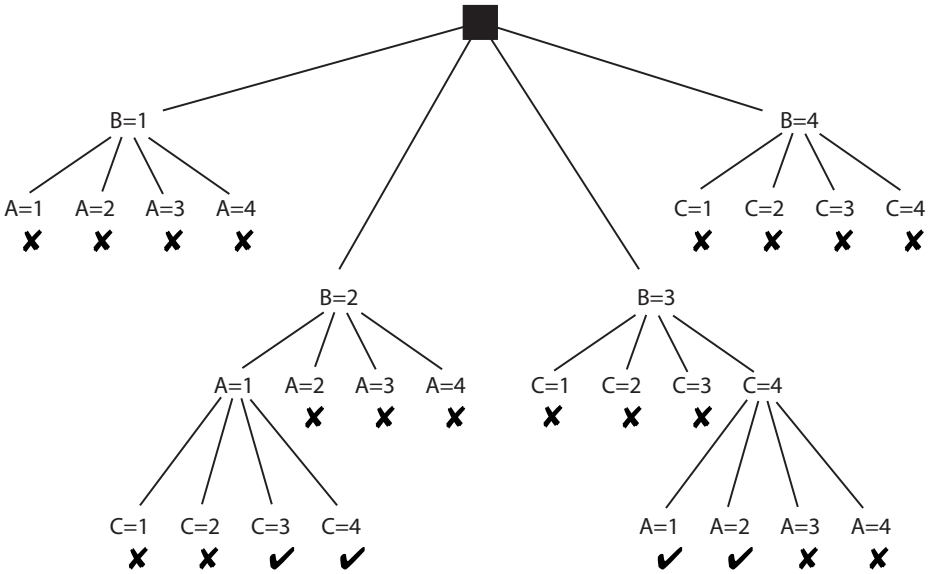
# Example

- Suppose you have a CSP with the variables A, B, and C, each with domain 1,2,3,4.
- Suppose the constraints are A<B and B<C.
- A possible search tree is:




- A node corresponds to all of the assignments from the root to that node.
- The potential nodes that are pruned because they violate constraints are labeled with $\mathcal{X}$.
- The leftmost $\mathcal{X}$ corresponds to the assignment A=1, B=1. This violates the A<B constraint, and so it is pruned.
- This CSP has four solutions. The leftmost one is A=1, B=2, C=3.
- There would be $4^3 = 64$ assignments tested in a generate-and-test algorithm. For the search method, there are 22 assignments generated.

# Example (cont'd)




- In the previous example, the variables A and B are related by the constraint A<B.
- The assignment A=4 is inconsistent with each of the possible assignments to B because $\mathbf{D}_B = \{1,2,3,4\}$.
- In the course of the backtrack search, this fact is rediscovered for different assignments to B and C.
- This inefficiency can be avoided by the simple expedient of deleting 4 from $\mathbf{D}_A$, once and for all.
- This idea is the basis for the consistency algorithms.

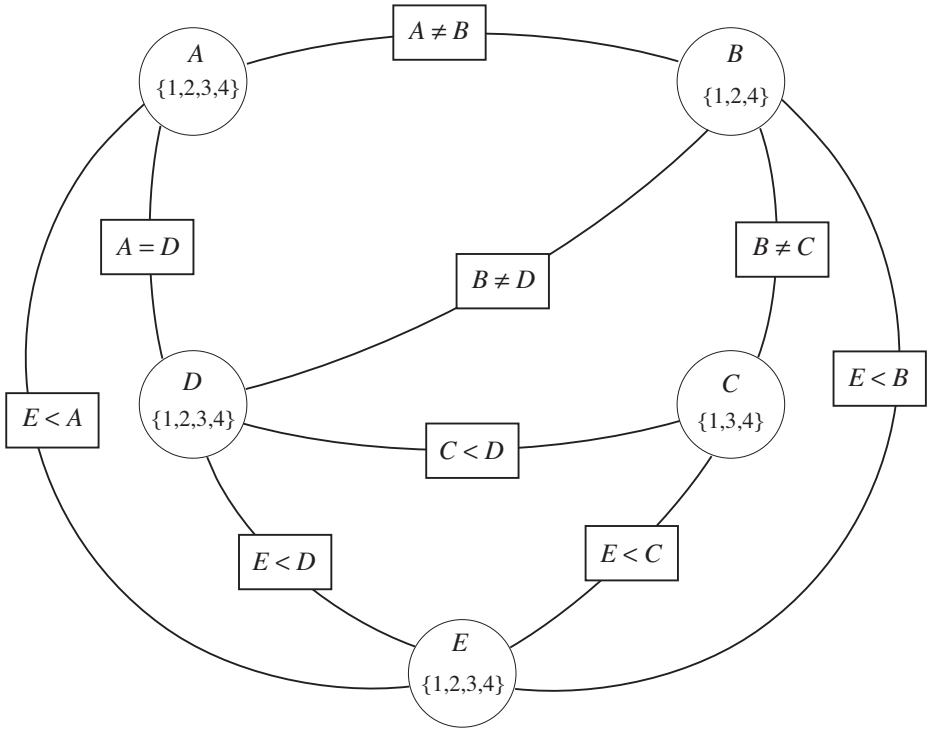
# Consistency Algorithms

- Idea: prune the domains as much as possible before selecting values from them.
- A variable is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints.
- Example: $\mathbf{D}_B = \{1, 2, 3, 4\}$ isn't domain consistent as $B = 3$ violates the constraint $B \neq 3$.

# Constraint Network

- There is a oval-shaped node for each variable.
- There is a rectangular node for each constraint.
- There is a domain of values associated with each variable node.
- There is an arc from variable $X$ to each constraint that involves $X$.

# Example Constraint Network

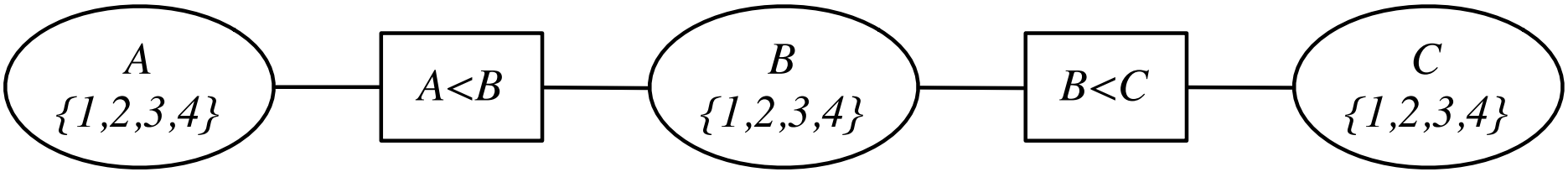



- Variables: $A$, $B$, $C$, $D$, $E$ that represent the starting times of various activities.
- Domains: $\mathbf{D}_A = \{1, 2, 3, 4\}$, $\mathbf{D}_B = \{1, 2, 3, 4\}$, $\mathbf{D}_C = \{1, 2, 3, 4\}$, $\mathbf{D}_D = \{1, 2, 3, 4\}$, $\mathbf{D}_E = \{1, 2, 3, 4\}$
- Constraints:

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D).$$

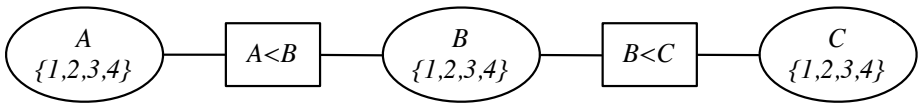
# Another example Constraint Network

- Suppose you have a CSP with the variables A, B, and C, each with domain 1,2,3,4.
- Suppose the constraints are A<B and B<C.
- The corresponding constraint network is:



# Arc Consistency

- An arc $\langle X, r(X, \overline{Y}) \rangle$ is arc consistent if, for each value $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is satisfied.
- A network is arc consistent if all its arcs are arc consistent.




- None of the arcs are arc consistent.
- The first arc is not arc consistent because for A=4 there is no corresponding value for B for which A<B.
- If 4 were removed from the domain of A, then it would be arc consistent.
- The second arc is not arc consistent because there is no corresponding value for A when B=1.

# Arc Consistency

- If an arc $\langle X, r(X, \overline{Y}) \rangle$ is *not* arc consistent, all values of $X$ in $dom(X)$ for which there is no corresponding value in $dom(\overline{Y})$ may be deleted from $dom(X)$ to make the arc $\langle X, r(X, \overline{Y}) \rangle$ consistent.

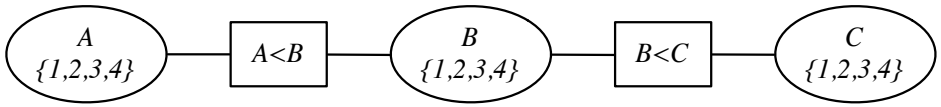
# Arc Consistency Algorithm

- The arcs can be considered in turn making each arc consistent.
- An arc $\langle X, r(X, \overline{Y}) \rangle$ needs to be revisited if the domain of one of the $Y$'s is reduced.
- Three possible outcomes (when all arcs are arc consistent):
  - One domain is empty $\Longrightarrow$ no solution
  - Each domain has a single value $\Longrightarrow$ unique solution
  - Some domains have more than one value $\Longrightarrow$ there may or may not be a solution

# Arc Consistent Network without Solutions

- Suppose there are three variables, A, B and C, each with the domain $\{1,2,3\}$.
- Consider the constraints A=B, B=C, and A$\neq$C.
- This is arc consistent: no domain can be pruned using any single constraint.
- However, there are no solutions.
- There is no assignment to the three variables that satisfies the constraints.

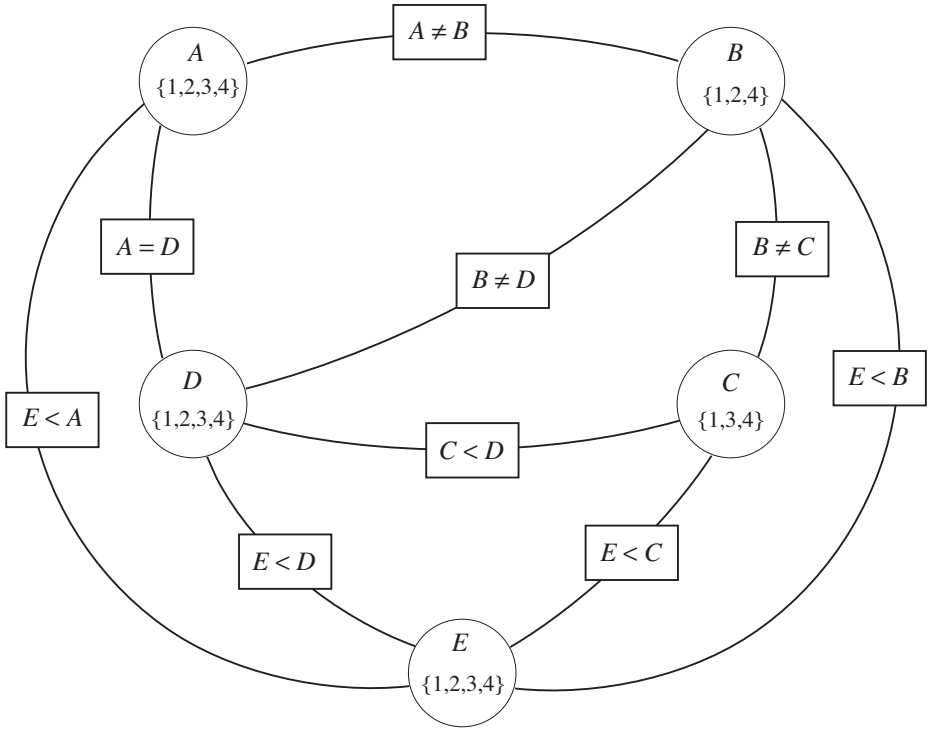
# Example Arc Consistency (1)




- Initially, all of the arcs are in the TDA ("To-Do Arcs") set.
- Suppose the algorithm selects the arc $\langle A, A < B \rangle$.
- For A=4, there is no value of B that satisfies the constraint. Thus, 4 is pruned from the domain of A. Nothing is added to TDA because there is no other arc currently outside TDA.
- Suppose that $\langle B, A < B \rangle$ is selected next.
- The value 1 can be pruned from the domain of B. No element is added to TDA.
- Suppose that $\langle B, B < C \rangle$ is selected next. The value 4 can be removed from the domain of B. Because the domain of B has been reduced, the arc $\langle A, A < B \rangle$ must be added back into the TDA set because the domain of A could potentially be reduced further now that the domain of B is smaller.
- If the arc $\langle A, A < B \rangle$ is selected, the value A=3 is pruned from the domain of A.
- The remaining arc on TDA is $\langle C, B < C \rangle$. The values 1 and 2 can be removed from the domain of C. No arcs are added to TDA and TDA becomes empty.
- The algorithm terminates with $dom(A) = \{1, 2\}$, $dom(B) = \{2, 3\}$, $dom(C) = \{3, 4\}$.

# Example Arc Consistency (2)

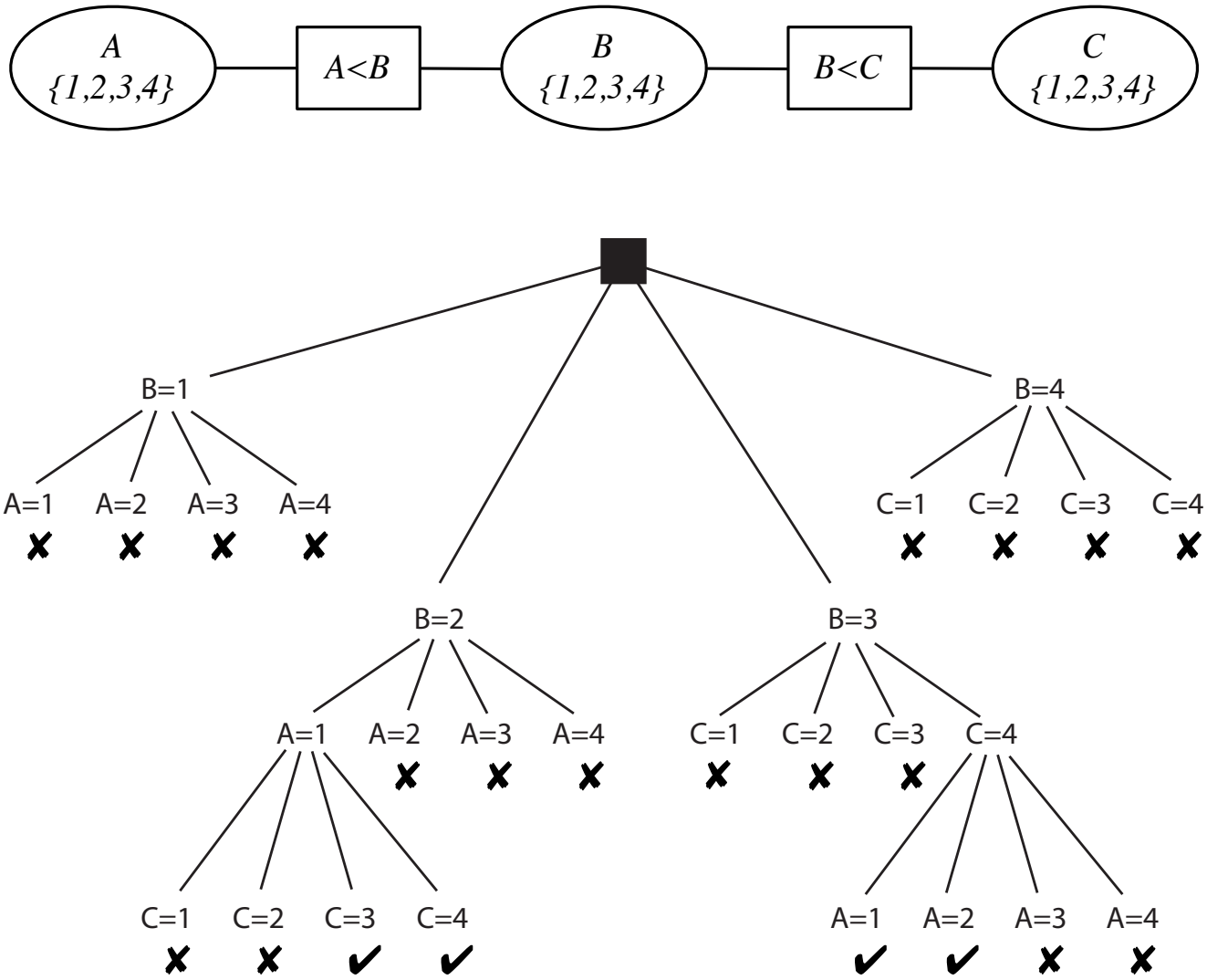
- The network has already been made domain consistent (the value 3 has been removed from the domain of B and 2 has been removed from the domain of C).
- Suppose arc $\langle D, C < D\rangle$ is considered first. The arc is not arc consistent because D=1 is not consistent with any value in dom(C), so 1 is deleted from dom(D). dom(D) becomes 2,3,4 and arcs $\langle A, A = D\rangle$,$\langle B, B \neq D\rangle$, and $\langle E, E < D\rangle$ could be added to TDA but they are on it already.
- Suppose arc $\langle C, E < C\rangle$ is considered next; then dom(C) is reduced to 3,4 and arc $\langle D, C < D\rangle$ goes back into the TDA set to be reconsidered.
- Suppose arc $\langle D, C < D\rangle$ is next; then dom(D) is further reduced to the singleton 4.
- Processing arc $\langle C, C < D\rangle$ prunes dom(C) to 3.
- Making arc $\langle A, A = D\rangle$ consistent reduces dom(A) to 4.
- Processing $\langle B, B \neq D\rangle$ reduces dom(B) to 1,2.
- Then arc $\langle B, E < B\rangle$ reduces dom(B) to 2.
- Finally, arc $\langle E, E < B\rangle$ reduces dom(E) to 1.
- All arcs remaining in the queue are consistent, and so the algorithm terminates with the TDA set empty.
- The set of reduced variable domains is returned. In this case, the domains all have size 1 and there is a unique solution: A=4, B=2, C=3, D=4, E=1.



# Finding solutions when AC finishes: Splitting

- If some domains have more than one element $\Longrightarrow$ search
- Split a domain, then recursively solve each part.
- Recursively solving the cases using domain splitting, recognizing when there is no solution based on the assignments, is equivalent to the search algorithm seen before.
- We can be more efficient by interleaving arc consistency with the search: use arc consistency to simplify the network before each step of domain splitting.
- That is, to solve a problem:
  - simplify the problem using arc consistency; and,
  - if the problem is not solved, select a variable whose domain has more than one element, split it, and recursively solve each case.
- We only need to revisit arcs affected by the split.
- It is often best to split a domain in half.

# Example


A
{1,2,3,4}
A<B
B
{1,2,3,4}
B<C
C
{1,2,3,4}
B=1
B=4
A=1
A=2
A=3
A=4
C=1
C=2
C=3
C=4
B=2
B=3
A=1
A=2
A=3
A=4
C=1
C=2
C=3
C=4
C=1
C=2
C=3
C=4
A=1
A=2
A=3
A=4

# The Propositional Satisfiability Problem

- Boolean variables: a variable with domain $\{true, false\}$.
- Given a Boolean variable *Happy*, the literal *happy* means *Happy*=*true*, and the literal $\neg happy$ means *Happy*=*false*.
- Clausal constraints: a clause is an expression of the form $l_1 \vee l_2 \vee \cdots \vee l_k$, where each $l_i$ is a literal. A clause is satisfied in a possible world if and only if at least one of the literals that makes up the clause is true in that possible world.
- A clause is a constraint on Boolean variables removing one assignment - the assignment that makes all literals false.
- Since there are only the two values $\{true, false\}$, pruning a value from the domain is equivalent to assigning the opposite value.
- Arc consistency can be used to prune values and constraints.
- Pruning the domains and constraints, domain splitting, assigning pure symbols, and efficiently managing the constraints make a very efficient algorithm for propositional satisfiability: the DPLL algorithm.

# DPLL

- Arc consistency can be used to prune the set of values and the set of constraints. Assigning a value to a Boolean variable can simplify the set of constraints:
  - If X is assigned true, all of the clauses with X=true become redundant; they are automatically satisfied. These clauses can be removed.
  - If X is assigned true, any clause with X=false can be simplified by removing X=false from the clause.
  - Similarly, if X is assigned the value of false, then X=true can be removed from any clause it appears in. This step is called **unit resolution**.
  - Following some steps of pruning the clauses, clauses may exist that contain just one assignment, Y=v. In this case, the other value can be removed from the domain of Y.

# Variable Elimination

- Idea: eliminate the variables one-by-one passing their constraints to their neighbours

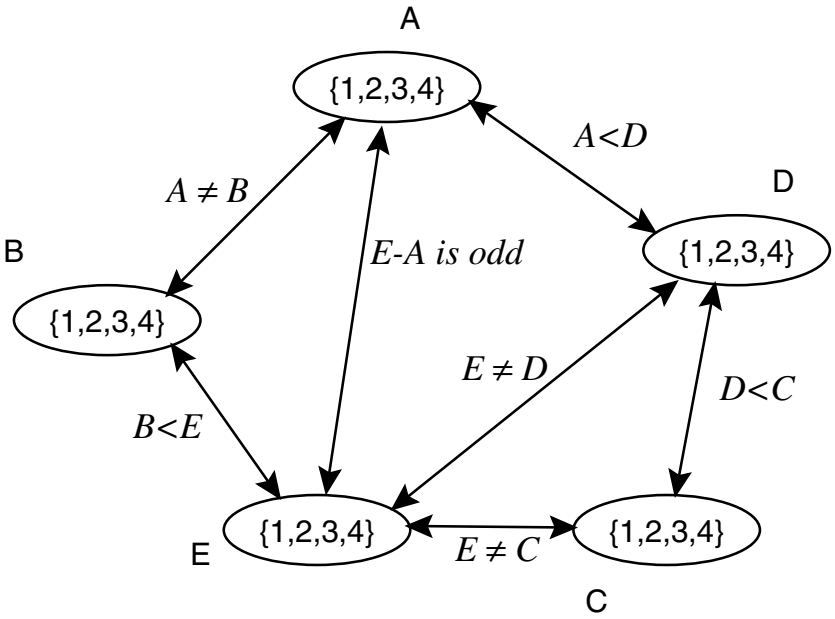
Variable Elimination Algorithm:

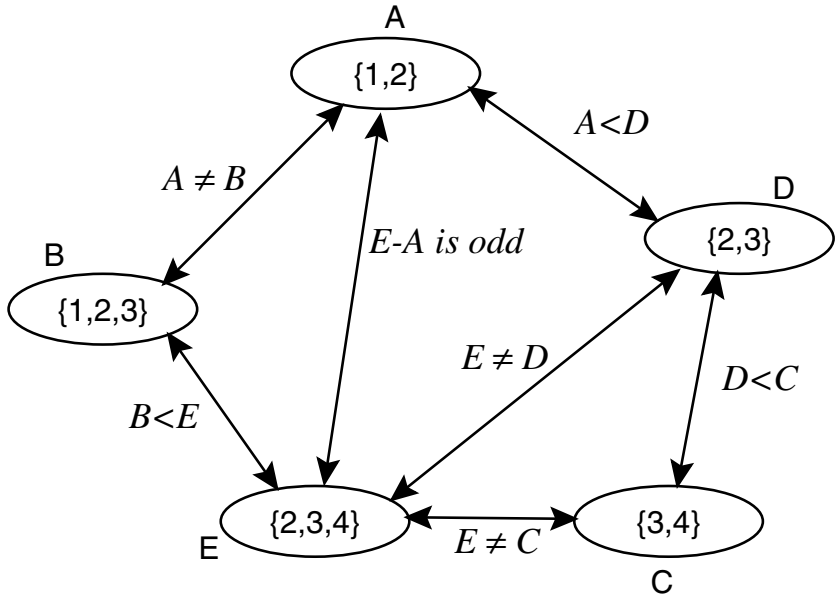
- If there is only one variable, return the intersection of the (unary) constraints that contain it
- Select a variable $X$
- Join the constraints in which $X$ appears, forming constraint $R_1$
- Project $R_1$ onto its variables other than $X$, forming $R_2$
- Replace all of the constraints in which $X$ appears by $R_2$
- Recursively solve the simplified problem, forming $R_3$
- Once it has a solution for the reduced CSP, it can extend that solution to a solution for the original CSP by joining the solution $R_3$ with $R_1$.

# Variable elimination (cont.)

- When there is a single variable remaining, if it has no values, the network was inconsistent.
- The variables are eliminated according to some elimination ordering
- Different elimination orderings result in different size intermediate constraints.

# Example network: eliminate $C$



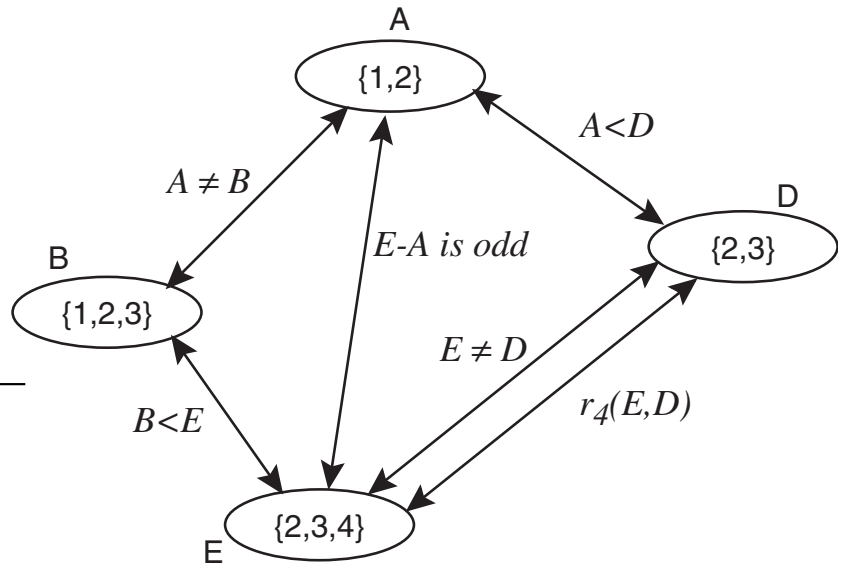

| $r_1 : C \neq E$ | $C$ | $E$ |
|---|---|---|
| | 3 | 2 |
| | 3 | 4 |
| | 4 | 2 |
| | 4 | 3 |

| $r_2 : C > D$ | $C$ | $D$ |
|---|---|---|
| | 3 | 2 |
| | 4 | 2 |
| | 4 | 3 |

| $r_3 : r_1 \bowtie r2$ | $C$ | $D$ | $E$ |
|---|---|---|---|
| | 3 | 2 | 2 |
| | 3 | 2 | 4 |
| | 4 | 2 | 2 |
| | 4 | 2 | 3 |
| | 4 | 3 | 2 |
| | 4 | 3 | 3 |

| $r_4 : \pi_{\{D,E\}} r_3$ | $D$ | $E$ |
|---|---|---|
| | 2 | 2 |
| | 2 | 3 |
| | 2 | 4 |
| | 3 | 2 |
| | 3 | 3 |

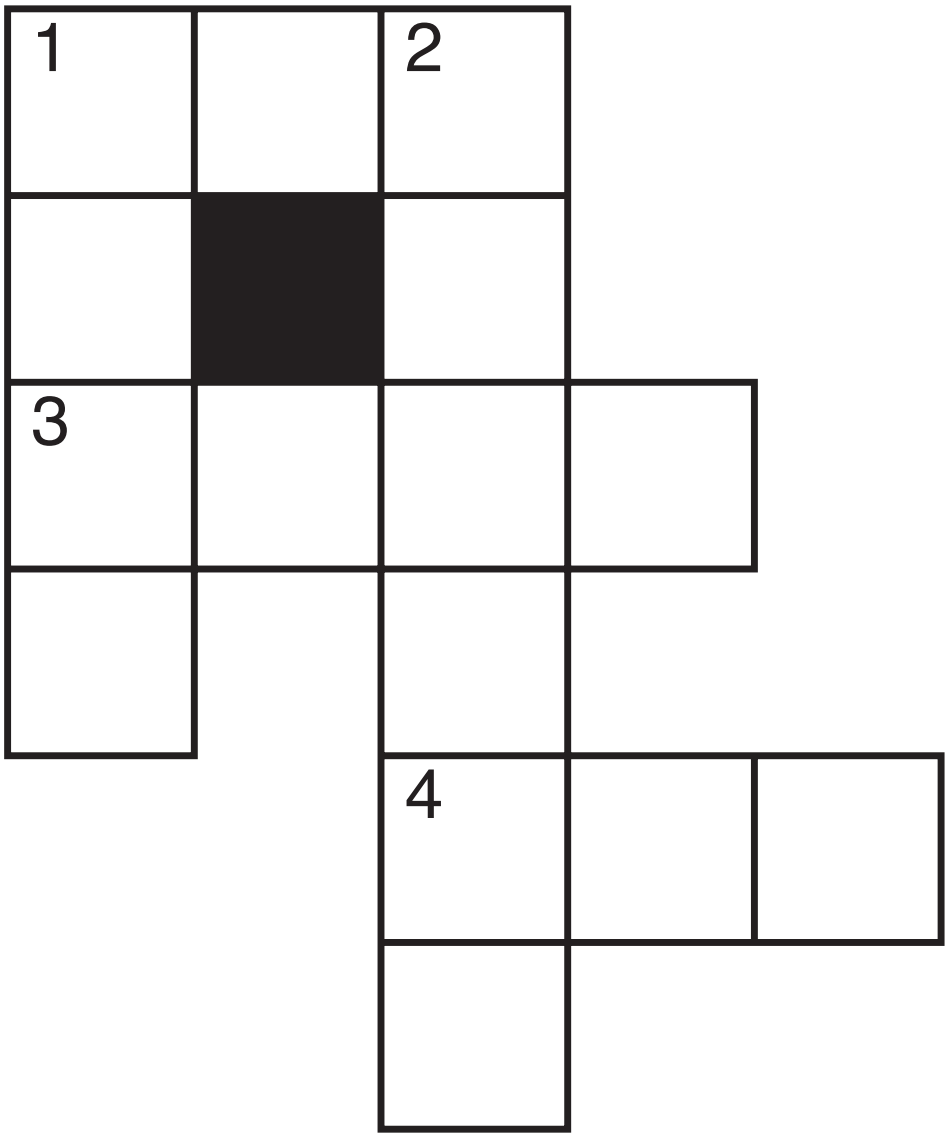
↪ new constraint



# Variable Elimination ordering

- Finding an elimination ordering leading to optimal efficiency (i.e., smallest *treewidth*) is NP-hard.
- Good heuristics exist:
  - **min-factor**: at each stage, select the variable that results in the smallest relation.
  - **minimum deficiency** or **minimum fill**: at each stage, select the variable that adds the smallest number of arcs to the remaining constraint network. The deficiency of a variable X is the number of pairs of variables that are in a relationship with X that are not in a relationship with each other. The intuition is that it is okay to remove a variable that results in a large relation as long as it does not make the network more complicated.
- The minimum deficiency has usually been found empirically to give a smaller treewidth than min-factor, but it is more difficult to compute.
- VE can also be combined with arc consistency; whenever VE removes a variable, arc consistency can be used to further simplify the problem. This approach can result in smaller intermediate tables.

# Example: Crossword Puzzle




Words:

ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax