

CS 4700:
Foundations of Artificial Intelligence

Bart Selman
selman@cs.cornell.edu

Module:
Constraint Satisfaction

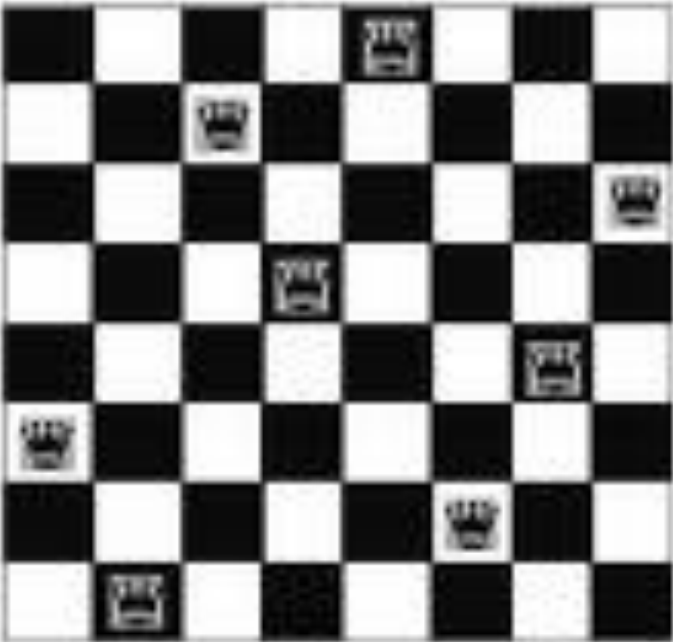
Chapter 6, R&N
(Completes part II – Problem Solving)

Constraint Satisfaction Problems (CSP) Backtracking search for CSPs

Key issue: So far, we have treated nodes in search trees as “black boxes,” only looked inside to check whether the node is a goal state.

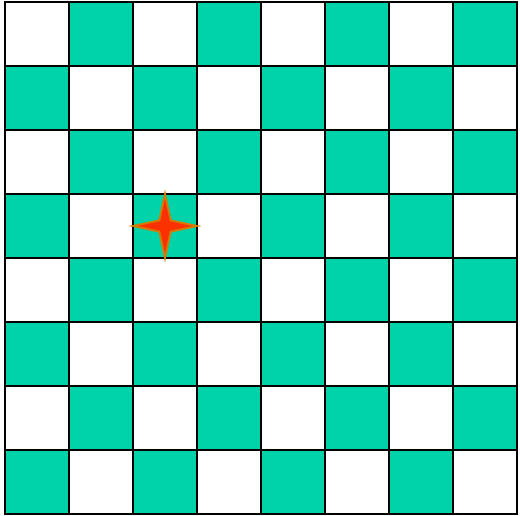
In CSPs, we want to “look inside the nodes” and exploit problem structure during the search. Sometimes, reasoning or inference (“propagation techniques”) will led us find solutions without any search!

Motivational Example: 8-Queens



Usual goal:
place 8 non-attacking
queens.

Already shown
effectiveness of local
search.



Generic (DFS/BFS) search

Is this how you would program this problem?

Node is a (partial) state of the board
Action: "place a queen on board"
Goal test: are there 8 non-attacking queens?
How many placements of 8 queens will be considered?

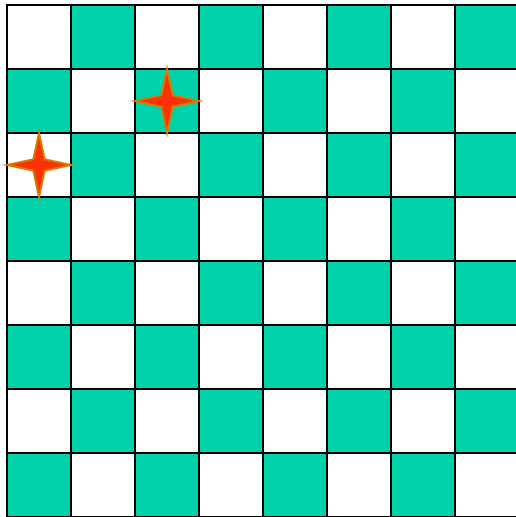
The DFS/BFS tree will enumerate up to **64⁸** combinations (assume limit depth to 8).

$$64^8 = 2^{48} = 2.8 \times 10^{14}$$

Note redundancy: Q1 in (1,3), Q2 in (2,7) ...
vs. Q1 in (2,7), Q2 in (1,3) ...

Alternative: Use “factored representation.”

Factoring refers to “splitting things into smaller parts.”



State has internal structure.

1) Here, a set of 8 *variables*.

x_1, x_2, \dots, x_8 .

2) Each with domain $\{1, \dots, 8\}$, the possible variable *values*.

3) A set of *constraints*:

e.g. no two vars can be assigned the same value. (Why?)

Each variable gives the position of a queen in a row.

Number of board states to explore:

“only” $8^8 = 16.7 \times 10^6$ combinations.

Set of *vars*, set of possible *values* for each vars
& set of *constraints* defines a *CSP*.

Set of **vars**, set of possible **values** for each vars & set of **constraints** defines a **CSP**.

A **solution** to the CSP is an assignment of values to the variables so that all constraints are satisfied (no “violated constraints.”)

A CSP is **inconsistent** if no such solution exists.

Eg try to place 9 non-attacking queens on an 8x8 board.

Hmm. Can a search program figure out that you can't place 101 queens on 100x100 board?

Not so easy! Most search approaches can't. *Need much more clever reasoning, instead of just search. (Need to use Pigeon Hole principle.)*

Aside: Factored representation does not even allow one to ask the question. Knowledge is build in.)

Alternative question: With N queens is there a solution with queen in bottom right hand corner on a N x N board?

How do we search for a solution?

Start with empty variable assignment (no vars assigned). Then, build up partial assignments until all vars assigned.

Action: “assign a variable a value.”

Goal test: “all vars assigned and no constraint violation.”

What is the search space? (n vars, each with d possible values)

Top level branching: $n \cdot d$

Next branching: $(n-1) \cdot d$

Next branching: $(n-2) \cdot d$

...

Bottom level: d

(one var remains to be set)

Hmm. But “only” n^d distinct value assignments! Different var ordering can lead to the same assignment! Wasteful...

Just “fix” a variable ordering:

Backtrack search.

Check only n^d full var-value assignments.

So, tree with $n! d^n$ leaves.

Backtrack search

Aside: “legal” and “feasible”

Already assumes a bit of “reasoning.” (Next.)

There are many improvements on intuitive idea...



Intuitive:

- 1) Fix some ordering on the variables. Eg $x_1, x_2, x_3 \dots$
- 2) Fix some ordering on possible values. Eg 1, 2, 3, ...
- 3) Assign first variable, first (legal) value.
- 4) Assign next variable, its first (legal) value.
- 5) Etc.
- 6) Until no remaining (feasible) value exist for variable x_i ,
backtrack to previous var setting of $x_{(i-1)}$, try next possible setting. If none exists, move back another level. Etc.

Visually, very intuitive on the N-Queens board (“the obvious strategy”).

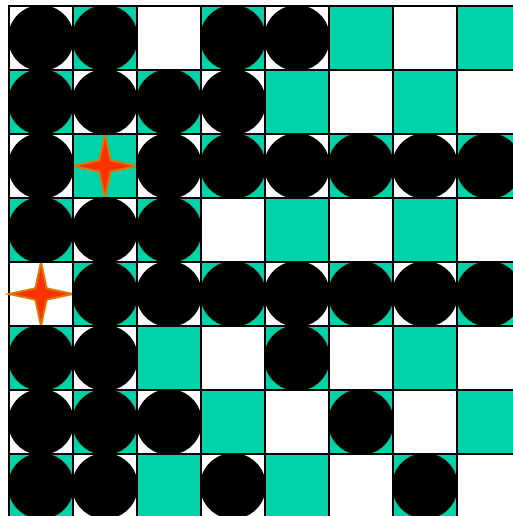
See figure 6.5 book. Recursive implementation. Practice: Iterative with stack.

Reasoning, inference or “propagation.”

Message:

CSP propagation techniques can dramatically reduce search.

Sometimes to no search at all! Eg. Sudoku puzzles.



After placing the first queen, what would you do for the 2nd?

General Search vs. Constraint satisfaction problems (CSPs)

Standard search problem:

- **state** is a "black box" – can be accessed only in limited way: successor function; heuristic function; and goal test.

What is needed for CSP:

Not just a successor function and goal test. Also a means of propagating the constraints (e.g. imposed by one queen on the others and an early failure test).

→ **Explicit representation of constraints and constraint manipulation algorithms**

→ **Constraint Satisfaction Problems (CSP)**

Constraint satisfaction problems (CSPs)

States and goal test have a *standard* representation.

- **state** is defined by **variables** X_i with **values** from **domain** D_i
- **goal test** is a set of **constraints** specifying allowable combinations of values for subsets of variables

Interesting tradeoff:

Example of a (restricted) formal representation language.

Allows useful **general-purpose algorithms more powerful than standard search algorithms that have to resort to problem specific heuristics to enable solution of large problems.**

Constraint Satisfaction Problem

Set of variables $\{X_1, X_2, \dots, X_n\}$

Each variable X_i has a **domain** D_i of possible values

Usually D_i is discrete and finite

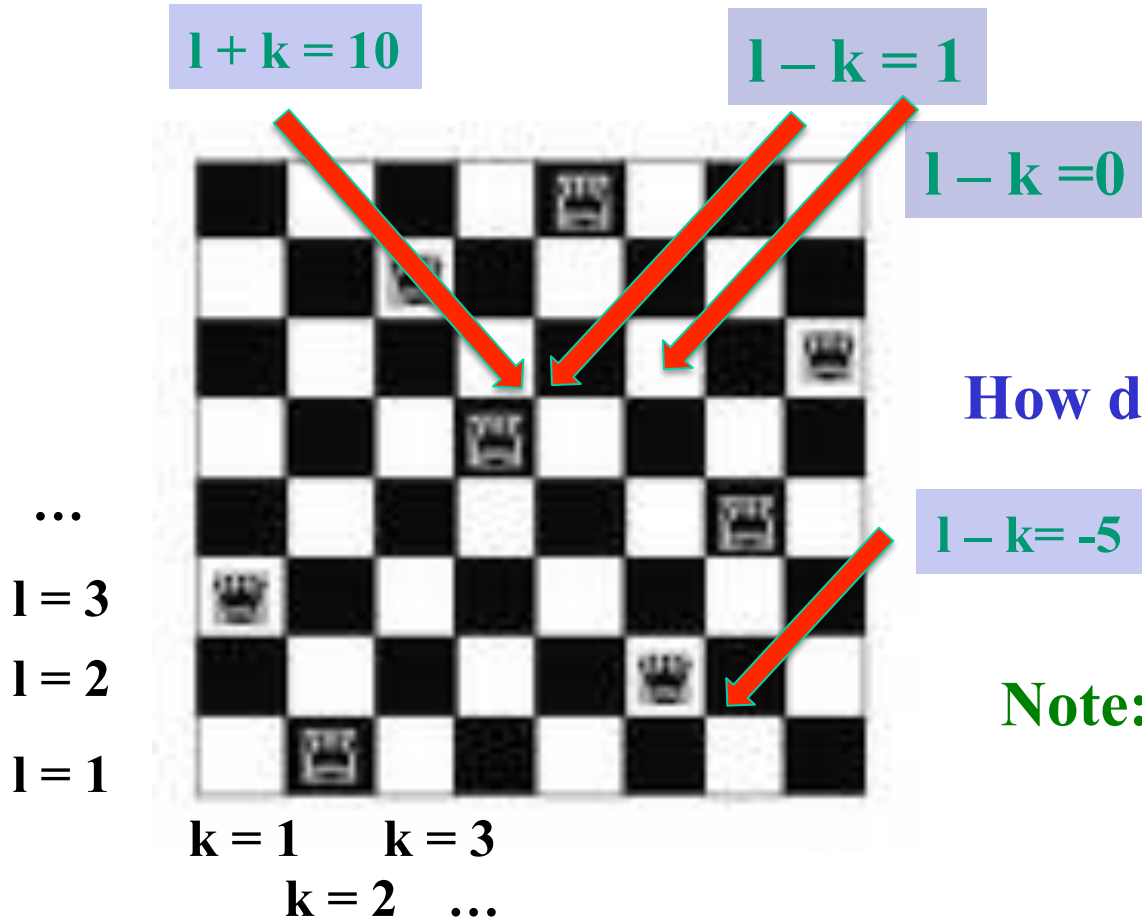
Set of **constraints** $\{C_1, C_2, \dots, C_p\}$

Each constraint C_k involves a subset of variables and specifies the allowable combinations of values of these variables

Goal:

Assign a value to every variable such that all constraints are satisfied

Motivational Example: 8-Queens



How do we represent 8-Queens
as a CSP:
Variables?
Constraints?

Note: More than one option.

Example: 8-Queens Problem

X_i – column for queen in row i

8 variables X_i , $i = 1$ to 8 (one per row)

Domain for each variable $\{1,2,\dots,8\}$

Constraints are of the form:

– $X_i \neq X_j$ when $j \neq i$ (i.e. no two in the same column)

– No queens in same diagonal:

1) $X_i - X_j \neq i - j$

2) $X_i - X_j \neq j - i$

(check that this works!)

Alternative?

Boolean vars

64 variables X_{ij} , $i = 1$ to 8 , $j = 1$ to 8

Domain for each variable $\{0,1\}$ (or $\{\text{False}, \text{True}\}$)

Constraints are of the form:

$X_{ij} = 1$ iff “there is a queen on location (i,j) .”

Row and columns

- If $(X_{ij} = 1)$ then $(X_{ik} = 0)$ for all $k = 1$ to 8 , $k \neq j$ (logical constraint)
- $X_{ij} = 1 \rightarrow X_{kj} = 0$ for all $k = 1$ to 8 , $k \neq i$

Diagonals

- $X_{ij} = 1 \rightarrow X_{i+l,j+l} = 0$ $l = 1$ to 7 , $i+l \leq 8$; $j+l \leq 8$ (right and up)
- $X_{ij} = 1 \rightarrow X_{i-l,j+l} = 0$ $l = 1$ to 7 , $i-l \geq 1$; $j+l \leq 8$ (right and down)
- $X_{ij} = 1 \rightarrow X_{i-l,j-l} = 0$ $l = 1$ to 7 , $i-l \geq 1$; $j-l \geq 1$ (left and down)
- $X_{ij} = 1 \rightarrow X_{i+l,j-l} = 0$ $l = 1$ to 7 , $i+l \leq 8$; $j-l \geq 1$ (left and up)

What’s missing?

Need $N (= 8)$ queens on board!

3 options:

1) Maximize sum X_{ij} (optimization formulation)

2) Sum $X_{ij} = N$ (CSP; bit cumbersome in Boolean logic)

3) For each row i : $(X_{i1} \text{ OR } X_{i2} \text{ OR } X_{i3} \dots X_{iN})$

Logical equivalence

Two sentences **p** and **q** are **logically equivalent** (\equiv or \Leftrightarrow) iff $p \leftrightarrow q$ is a tautology
(and therefore p and q have the same truth value for all truth assignments)

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$

$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$

$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$

$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$

$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$

$$(\alpha \rightarrow \beta) \equiv (\neg\beta \rightarrow \neg\alpha) \quad \text{contraposition}$$

$$(\alpha \rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$

$$(\alpha \leftrightarrow \beta) \equiv ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)) \quad \text{biconditional elimination}$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{de Morgan}$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{de Morgan}$$

$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$

$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

Propositional Satisfiability problem

Satisfiability (SAT): Given a formula in propositional calculus, is there a model (i.e., a satisfying interpretation, an assignment to its variables) making it true?

We consider clausal form, e.g.:

$$(a \vee \neg b \vee \neg c) \text{ AND } (b \vee \neg c) \text{ AND } (a \vee c)$$

2^n possible assignments

SAT: prototypical hard combinatorial search and reasoning problem. Problem is NP-Complete. (Cook 1971)

Surprising “power” of SAT for encoding computational problems.

Significant progress in Satisfiability Methods

Software and hardware verification – complete methods are critical - e.g. for verifying the correctness of chip design, using SAT encodings

Going from 50 variable, 200 constraints to 1,000,000 variables and 5,000,000 constraints in the last 10 years

Current methods can verify automatically the correctness of large portions of a chip

**Many Applications:
Hardware and
Software Verification
Planning,
Protocol Design,
Scheduling, Materials
Discovery etc.**



Turing Award: Model Checking

2008 Turing Award Winners Announced

Posted by [ScuttleMonkey](#) on Monday February 04, @05:30PM
from the [nobel-of-computing-awards](#) dept.

The Association for Computing Machinery has announced the [2008 Turing Award Winners](#). Edmund M. Clarke, Allen Emerson, and Joseph Sifakis received the award for their work on an automated method for finding design errors in computer hardware and software.

"Model Checking is a type of "formal verification" that analyzes the logic underlying a design, much as a mathematician uses a proof to determine that a theorem is correct. Far from hit or miss, Model Checking considers every possible state of a hardware or software design and determines if it is consistent with the designer's specifications. Clarke and Emerson originated the idea of Model Checking at Harvard in 1981. They developed a theoretical technique for determining whether an abstract model of a hardware or software design satisfies a formal specification, given as a formula in Temporal Logic, a notation for describing possible sequences of events. Moreover, when the system fails the specification, it could identify a counterexample to show the source of the problem. Numerous model checking systems have been implemented, such as Spin at Bell Labs."



A “real world” example

From “SATLIB”:

<http://www.satlib.org/benchm.html>

SAT-encoded bounded model checking instances
(contributed by Ofer Shtrichman)

In Bounded Model Checking (BMC) [BCCZ99], a rather newly introduced problem in formal methods, the task is to check whether a given model M (typically a hardware design) satisfies a temporal property P in all paths with length less or equal to some bound k. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and in fact if the property is in the form of an invariant (Invariants are the most common type of properties, and many other temporal properties can be reduced to their form. It has the form of 'it is always true that ... '), it has a structure which is similar to many AI planning problems.

Bounded Model Checking instance:

The instance `bmc-ibm-6.cnf`, IBM LSU 1997:

`p cnf 51639 368352`

`-1 7 0`

`-1 6 0`

`-1 5 0`

`-1 -4 0`

`-1 3 0`

`-1 2 0`

`-1 -8 0`

`-9 15 0`

`-9 14 0`

`-9 13 0`

`-9 -12 0`

`-9 11 0`

`-9 10 0`

`-9 -16 0`

`-17 23 0`

`-17 22 0`

*i.e. $((\text{not } x_1) \text{ or } x_7)$
and $((\text{not } x_1) \text{ or } x_6)$
and ... etc.*

Dimacs Format for CNF

File format

The benchmark file format will be in a simplified version of the DIMACS format: c

c start with comments

c

c

p cnf 5 3

1 -5 4 0

-1 5 3 4 0

-3 -4 0

The file can start with comments, that is lines beginning with the character c.

Right after the comments, there is the line p cnf nbvar nbclauses indicating that the instance is in CNF format; nbvar is the exact number of variables appearing in the file; nbclauses is the exact number of clauses contained in the file.

Then the clauses follow. Each clause is a sequence of distinct non-null numbers between -nbvar and nbvar ending with 0 on the same line; it cannot contain the opposite literals i and $-i$ simultaneously. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

Example of Sat Solver

SAT Solver : **Lingeling**

<http://fmv.jku.at/lingeling/>

Nqueen4-v1.cnf

Nqueens4-v2.cnf

p cnf 16 84

1 2 3 4 0

-1 -2 0

-1 -3 0

-1 -4 0

-2 -3 0

-2 -4 0

-3 -4 0

5 6 7 8 0

-5 -6 0

-5 -7 0

-5 -8 0

-6 -7 0

-6 -8 0

-7 -8 0

9 10 11 12 0

-9 -10 0

-9 -11 0

-9 -12 0

-10 -11 0

-10 -12 0

-11 -12 0

13 14 15 16 0

-13 -14 0

-13 -15 0

-13 -16 0

-14 -15 0

-14 -16 0

-15 -16 0

1 5 9 13 0

-1 -5 0

-1 -9 0

-1 -13 0

-5 -9 0

-5 -13 0

-9 -13 0

2 6 10 14 0

-2 -6 0

-2 -10 0

-2 -14 0

-6 -10 0

-6 -14 0

-10 -14 0

3 7 11 15 0

-3 -7 0

-3 -11 0

-3 -15 0

-7 -11 0

-7 -15 0

-11 -15 0

4 8 12 16 0

-4 -8 0

-4 -12 0

-4 -16 0

-8 -12 0

-8 -16 0

-12 -16 0

-1 -6 0

-1 -11 0

-1 -16 0

-6 -11 0

-6 -16 0

-11 -16 0

-2 -7 0

-2 -12 0

-7 -12 0

-3 -8 0

-5 -10 0

-5 -15 0

-10 -15 0

-9 -14 0

-4 -7 0

-4 -10 0

-4 -13 0

-7 -10 0

-7 -13 0

-10 -13 0

-3 -6 0

-3 -9 0

-6 -9 0

-2 -5 0

-8 -11 0

-8 -14 0

-11 -14 0

-12 -15 0

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

How Large are the Problems?

A bounded model checking problem:

From "SATLIB":

<http://www.satlib.org/benchm.html>

SAT-encoded bounded model checking instances
(contributed by Ofer Shtrichman)

Source: IBM

In Bounded Model Checking (BMC) [BCCZ99], a rather newly introduced problem in formal methods, the task is to check whether a given model M (typically a hardware design) satisfies a temporal property P in all paths with length less or equal to some bound k . The BMC problem can be efficiently reduced to a propositional satisfiability problem, and in fact if the property is in the form of an invariant (Invariants are the most common type of properties, and many other temporal properties can be reduced to their form. It has the form of 'it is always true that ... '), it has a structure which is similar to many AI planning problems.

SAT Encoding

(automatically generated from problem specification)

The instance `bmc-ibm-6.cnf`, IBM LSU 1997:

p cnf 51639 368352

-1 7 0

-1 6 0

-1 5 0

-1 -4 0

-1 3 0

-1 2 0

-1 -8 0

-9 15 0

-9 14 0

-9 13 0

-9 -12 0

-9 11 0

-9 10 0

-9 -16 0

-17 23 0

-17 22 0

i.e., $((\text{not } x_1) \text{ or } x_7)$

$((\text{not } x_1) \text{ or } x_6)$

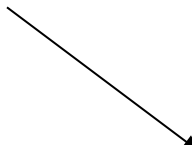
etc.

x_1, x_2, x_3 , etc. are our Boolean variables
(to be set to True or False)

Should x_1 be set to False??

10 Pages Later:

185 -9 0
185 -1 0
177 169 161 153 145 137 129 121 113 105 97
89 81 73 65 57 49 41
33 25 17 9 1 -185 0
186 -187 0
186 -188 0
...



i.e., (x_{177} or x_{169} or x_{161} or x_{153} ...
 x_{33} or x_{25} or x_{17} or x_9 or x_1 or (not x_{185}))

clauses / constraints are getting more interesting...

Note x_1 ...

4,000 Pages Later:

10236 -10050 0
10236 -10051 0
10236 -10235 0
10008 10009 10010 10011 10012 10013 10014
10015 10016 10017 10018 10019 10020 10021
10022 10023 10024 10025 10026 10027 10028
10029 10030 10031 10032 10033 10034 10035
10036 10037 10086 10087 10088 10089 10090
10091 10092 10093 10094 10095 10096 10097
10098 10099 10100 10101 10102 10103 10104
10105 10106 10107 10108 -55 -54 53 -52 -51 50
10047 10048 10049 10050 10051 10235 -10236 0
10237 -10008 0
10237 -10009 0
10237 -10010 0

...

Finally, 15,000 Pages Later:

```
-7 260 0
7 -260 0
1072 1070 0
-15 -14 -13 -12 -11 -10 0
-15 -14 -13 -12 -11 10 0
-15 -14 -13 -12 11 -10 0
-15 -14 -13 -12 11 10 0
-7 -6 -5 -4 -3 -2 0
-7 -6 -5 -4 -3 2 0
-7 -6 -5 -4 3 -2 0
-7 -6 -5 -4 3 2 0
185 0
```

Search space of truth assignments: $2^{50000} \approx 3.160699437 \cdot 10^{15051}$

Current SAT solvers solve this instance in under 10 seconds!

Example of a Boolean Satisfiability (SAT) encoding.

Very “simple” but effective representation.

Example of a logical knowledge representation language.

For propositional logic, see R&N 7.4.1 & 7.4.2.

Which encoding is better? Allows for faster solutions?

One would think, fewer variables is better...

Search spaces:

$$8^8 = 1.6 \times 10^6 \quad \text{vs} \quad 2^{64} = 1.8 \times 10^{19}$$









However, in practice SAT encodings can be surprisingly effective, even with millions of Boolean variables. Often, few true local minima in search space.

Demo of backtrack search and local search on Boolean encoding of N-queens.

N-Queens

The standard N by N Queen's problem asks how to place N queens on an ordinary chess board so that they don't attack each other









N=8

7								
6								
5								
4								
3								
2								
1								
0								
	0	1	2	3	4	5	6	7

Is this problem NP-Complete?

N-Queens

N=8 (another solution)

7								
6								
5								
4								
3								
2								
1								
0								
	0	1	2	3	4	5	6	7

Linear congruence equations

$$N = 6 \alpha;$$

$$N = 6 \alpha + 1;$$

$$N = 6 \alpha \pm 2; \text{ (not } N=4\text{;)}$$

$$N = 6 \alpha + 3; \text{ (not } N=9\text{;)}$$

$$N = 6 \alpha - 2; \text{ (inc. } N=4\text{)}$$

$$N = 12 \alpha - 3; \text{ (inc. } N=9\text{)}$$

N-Queens

N=8

$$N = 6\alpha \pm 2, \alpha=1$$

$$S(8) = \{3\}.$$

For all $c \in S(8)$, the linear congruence equations
Let's consider $c=3$.

$$6x + y \equiv 3 \pmod{8},$$









where $x = 0, 1, 2, 3$.

$$x = 0 \ y=3; x=1 \ y = 5; x=2 \ y= 7; x=3 \ y = 1;$$

$$6x + y \equiv 6 \pmod{8},$$

where $x = 4, 5, 6, 7, 8$

$$X=4, y = 6; x=5 \ y = 0; x=6 \ y = 2; x=7 \ y=4;$$

0								
1								
2								
3								
4								
5								
6								
7								
	0	1	2	3	4	5	6	7

0									
1					♔				
2								♔	
3									
4									
5	♔								
6									
7					♔				
	0	1	2	3	4	5	6	7	

Partially Filled Nqueens

So the N-queens problem is easy when we start with an empty board.

What about if we pre-assign some queens and ask for a completion?

Open question

Conjecture: completing a partially filled N-queens board is NP-complete.

Example: Crypt-arithmetic Puzzle

S E N D
+ M O R E

M O N E Y

Variables: S, E, N, D, M, O, R, Y

Domains:

[0..9] for S, M, E, N, D, O, R, Y

Search space: 1,814,400

Aside: could have [1..9] for S and M

Soln.:

9567

1085

====

10652

Option 1:

$$\begin{aligned}
\text{C1a) } & 1000 S + 100 E + 10 N + D + \\
& 1000 M + 100 O + 10 R + E \\
& = 10000 M + 1000 O + 100 N + 10 E + Y
\end{aligned}$$

Or use 5 equality constraints, using auxiliary “carry” variables $C1, \dots, C4 \in [0\dots9]$

SEND
+MORE

MONEY

Option 2: C1b)

$$\begin{aligned}
D + E &= 10 C1 + Y \\
C1 + N + R &= 10 C2 + E \\
C2 + E + O &= 10 C3 + N \\
C3 + S + M &= 10 C4 + O \\
C4 &= M
\end{aligned}$$

Which constraint set better for solving? C1a or C1b? Why?

C1b, more “factored”. Smaller pieces. Gives more propagation!

Need two more sets of constraints:

$$\text{C2) } S \neq 0, M \neq 0$$

$$\text{C3) } S \neq M, S \neq O, \dots E \neq Y \quad (28 \text{ not equal constraints})$$

Note: need to assert everything!

Alt. “All_diff(S,M,O,...Y)” for C3.

Some Reflection: Reasoning/Inference vs. Search

How do human solve this?

What is the first step?

$$\begin{array}{r} \\ \\ + \\ \hline = M O N E Y \end{array}$$

1) $M = 1$, because $M \neq 0$ and ...

the carry over of the addition of two digits (plus previous carry) is at most 1.

Actually, a somewhat subtle piece of *mathematical background knowledge*.

Also, what made us focus on M?

Experience / intuition ...

$$\begin{array}{r}
 \\
 \\
 \\
 \hline
 + \\
 = M O N E Y
 \end{array}$$

1) $M = 1$, because $M \neq 0$ and ...

the carry over of the addition of two digits (plus previous carry) is at most 1.

2) $O = 0$. Because $M=1$ and we have to have a carry to the next column. $S + 1 + C3$ is either 10 or 11. So, O equals 0 or 1. 1 is taken. So, $O = 0$.

3) $S = 9$. There cannot be a carry to the 4th column (if there were, N would also have to be 0 or 1. Already taken.). So, $S = 9$.

A collection of “small pieces” of local reasoning, using basic constraints from the rules of arithmetic. A logic (SAT) based encoding will likely “get these steps.”

And further it goes...

4. If there were no carry in column 3 then $E = N$, which is impossible. Therefore there is a carry and $N = E + 1$.
5. If there were no carry in column 2, then $(N + R) \bmod 10 = E$, and $N = E + 1$, so $(E + 1 + R) \bmod 10 = E$ which means $(1 + R) \bmod 10 = 0$, so $R = 9$. But $S = 9$, so there must be a carry in column 2 so $R = 8$.
6. To produce a carry in column 2, we must have $D + E = 10 + Y$.
7. Y is at least 2 so $D + E$ is at least 12.
8. The only two pairs of available numbers that sum to at least 12 are (5,7) and (6,7) so either $E = 7$ or $D = 7$.
9. Since $N = E + 1$, E can't be 7 because then $N = 8 = R$ so $D = 7$.
10. E can't be 6 because then $N = 7 = D$ so $E = 5$ and $N = 6$.
11. $D + E = 12$ so $Y = 2$.

Largely, a clever chain of reasoning / inference / propagation steps (no search) except for...
exploring 2 remaining options (i.e., search) to find complete solution.

Human problem solving nicely captures key idea behind how to solve CSPs: replace as much of search by propagation (inference/reasoning).

One difficulty: Humans often use subtle background knowledge.

Can search be completely avoided?

**A.: Most likely NO. General cryptarithmic is NP-complete
(Eppstein 1987)**

Example: Map-Coloring



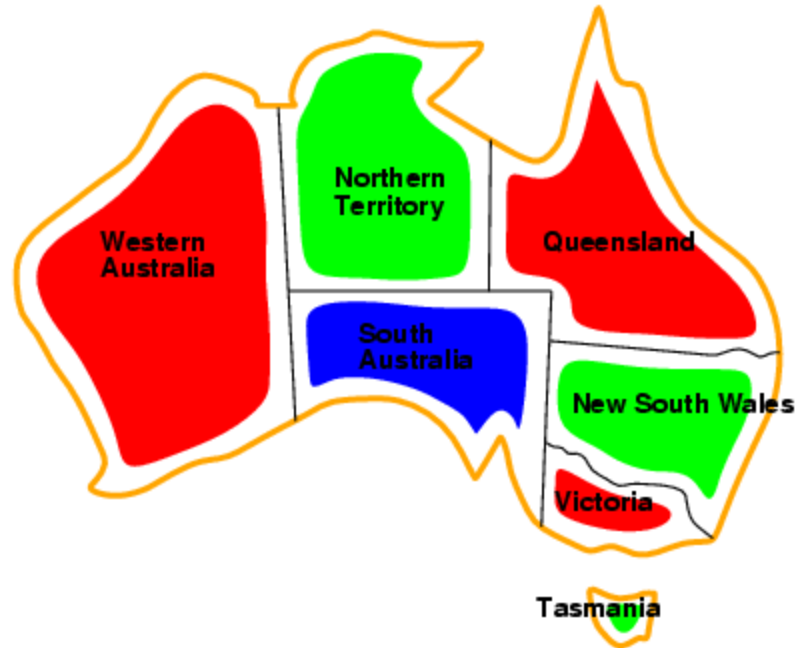
Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$, or (WA, NT) in $\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$

Example: Map-Coloring



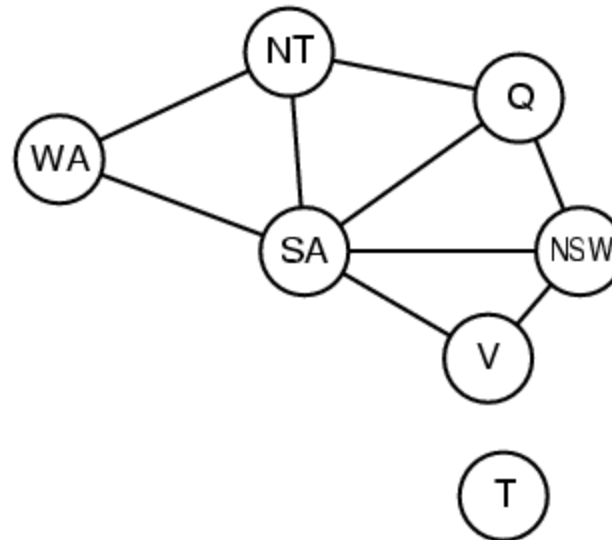
Solutions are **complete** and **consistent** assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

(Aside: Four colors suffice. (Appel and Haken 1977))

Constraint graph: Graph Coloring

Binary CSP: each constraint relates two variables

Constraint graph: nodes are variables, arcs are constraints

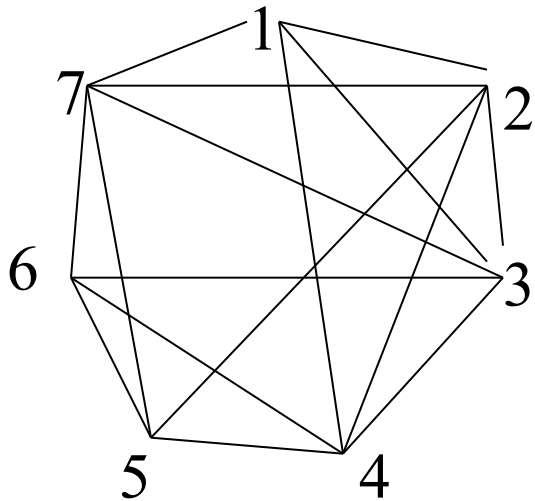


Two variables are adjacent or neighbors if they are connected by an edge or an arc

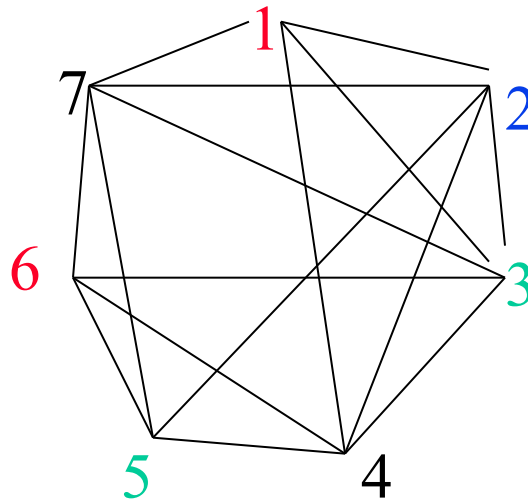
Application of Graph Coloring

Lots of applications involving scheduling and assignments.

Scheduling of final exams – nodes represent finals, edges between finals denote that both finals have common students (and therefore they have to have different colors, or different periods).



Graph of finals for 7 courses



Time Period \rightarrow courses

I (red) \rightarrow 1,6

II (blue) \rightarrow 2

III (green) \rightarrow 3,5

IV (black) \rightarrow

Varieties of CSPs

Discrete variables

- **finite domains:** our focus
 - n variables, domain size $d \rightarrow O(d^n)$ complete assignments
(includes Boolean satisfiability 1st problem to be shown NP-complete.)
- **infinite domains:**
 - integers, strings, etc.
 - e.g., job scheduling, variables are start/end days for each job
 - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

Continuous variables

- e.g., start/end times for Hubble Space Telescope observations
- linear constraints solvable in polynomial time by linear programming

Varieties of constraints

Unary constraints involve a single variable,

- e.g., $SA \neq \text{green}$

Binary constraints involve pairs of variables,

- e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

- e.g., cryptarithmic column constraints

CSP as a Search Problem

Initial state: empty assignment

Successor function: a value is assigned to any unassigned variable, which does not conflict with the currently assigned variables

Goal test: the assignment is complete

Path cost: irrelevant

Remark

Finite CSP include 3SAT as a special case (under logical reasoning).
3SAT is known to be NP-complete.

So, in the worst-case, we cannot expect to solve a finite CSP in less than exponential time.

Solving CSP by search : Backtrack Search

BFS vs. DFS

– BFS → not a good idea.

- A tree with $n!d^n$ leaves : $(nd)*((n-1)d)*((n-2)d)*...*(1d) = n!d^n$
- Reduction by commutativity of CSP
 - A solution is not in the permutations but in combinations.
 - A tree with d^n leaves

– DFS

- Used popularly
 - Every solution must be a complete assignment and therefore appears at depth n if there are n variables
 - The search tree extends only to depth n .
- A variant of DFS: **Backtrack search**
 - Chooses values for one variable at a time
 - **Backtracks when failed *even before reaching a leaf.***
- ***Better than BFS due to backtracking but still need more “cleverness” (reasoning/propagation).***

Backtrack search

Variable assignments are **commutative**, i.e.,
[WA = red then NT = green] same as [NT = green then WA = red]

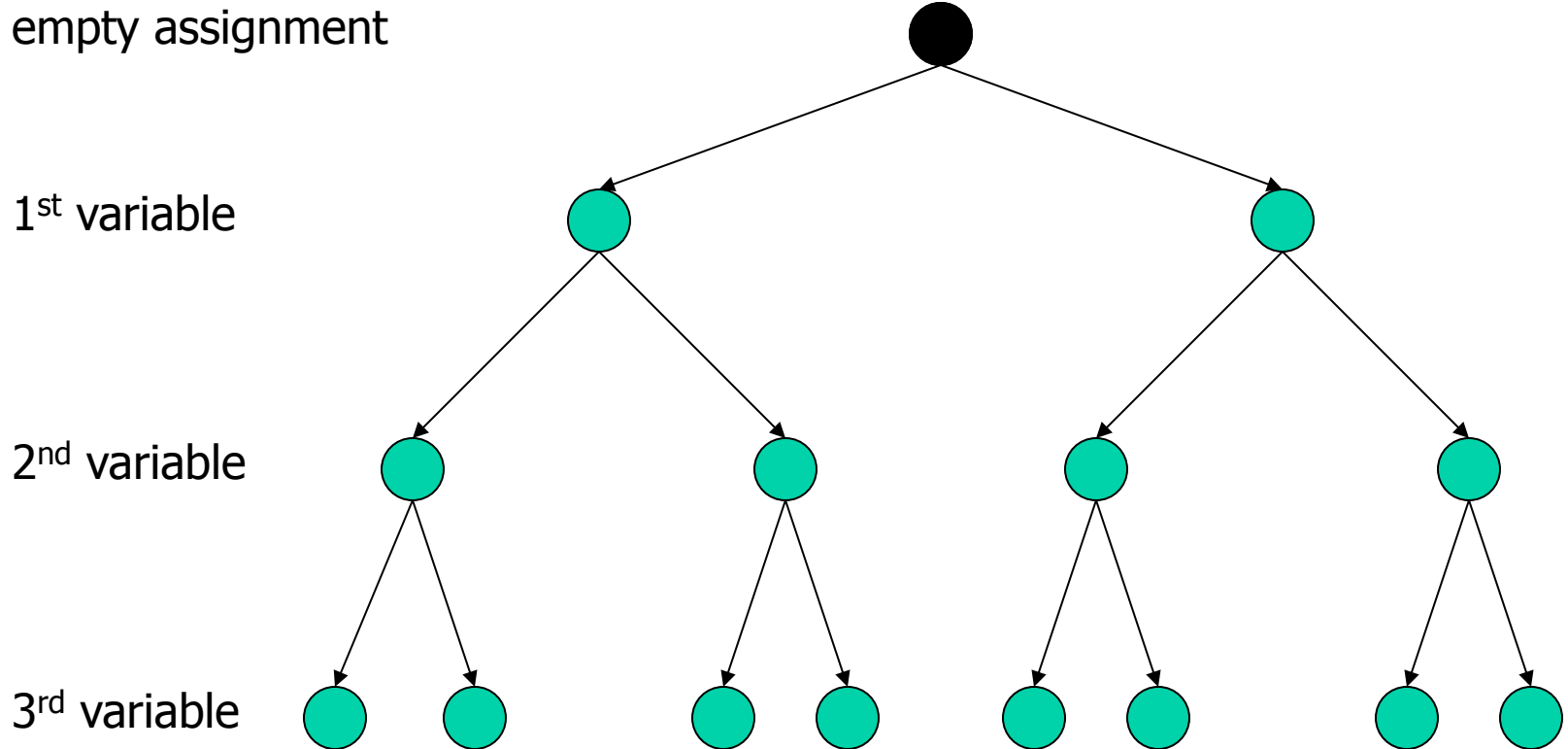
Only need to consider assignments to a single variable at each node
→ $b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called
backtrack search

Backtracking search is the basic uninformed algorithm for CSPs

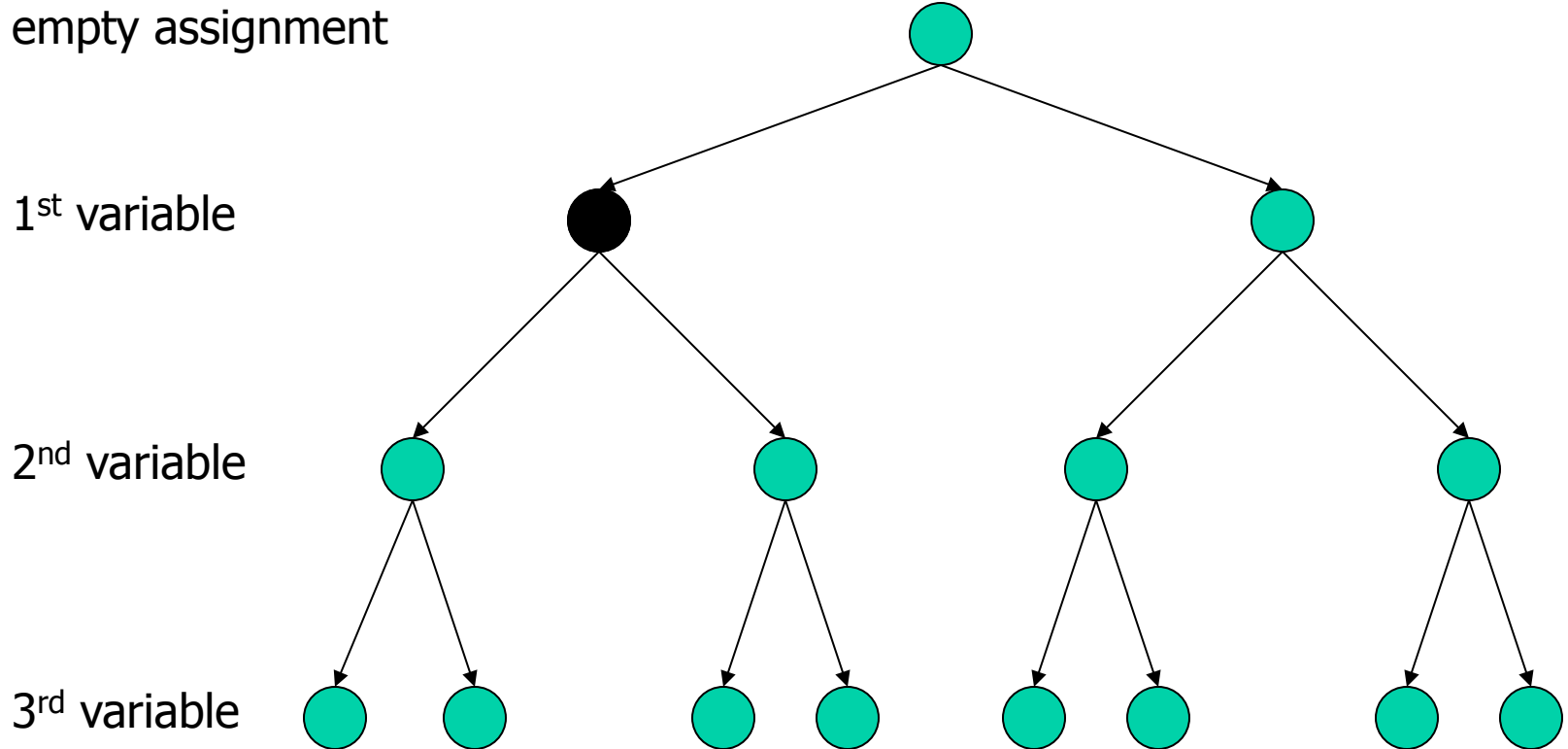
Can solve n -queens for $n \approx 30$

→ Backtrack Search



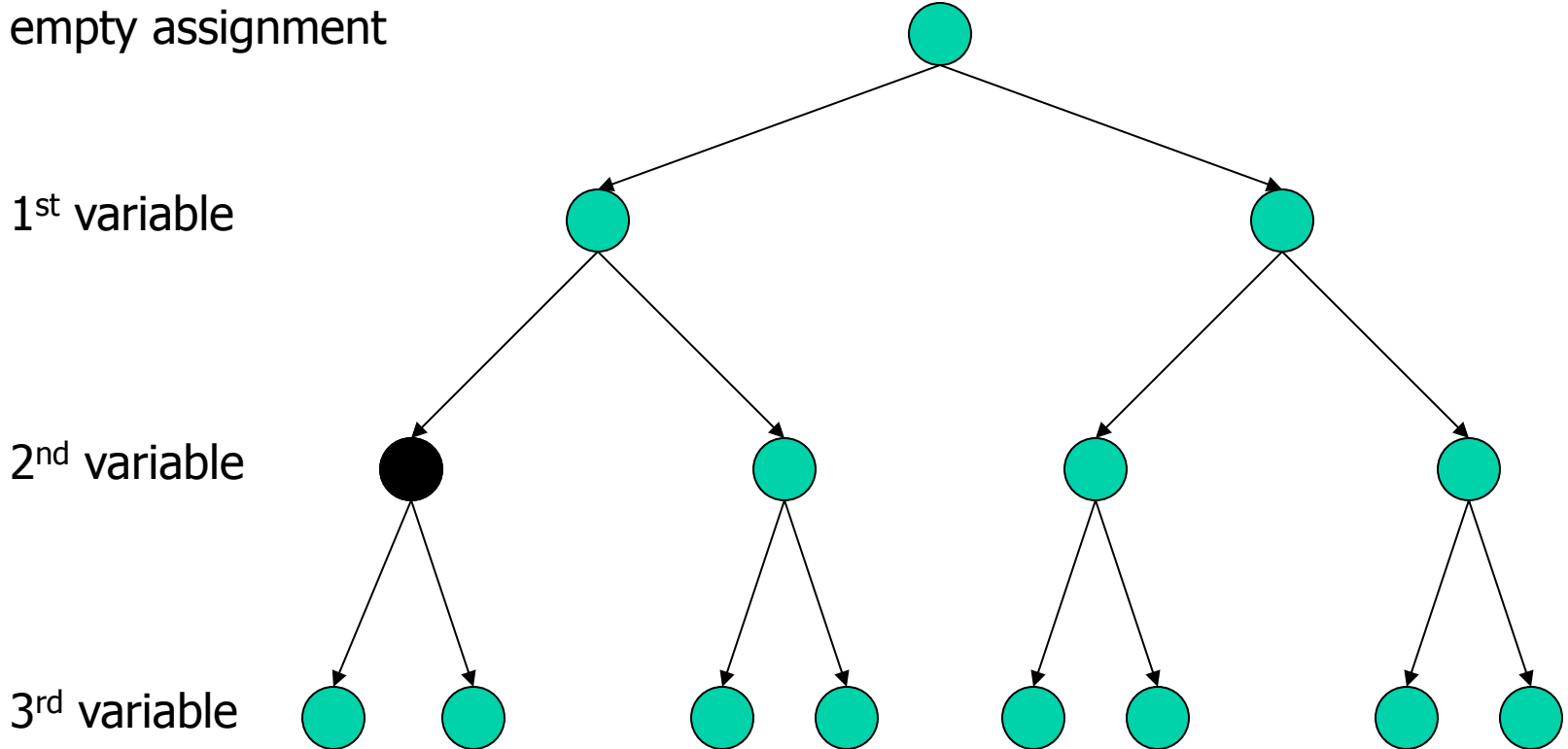
Assignment = { }

→ Backtrack Search



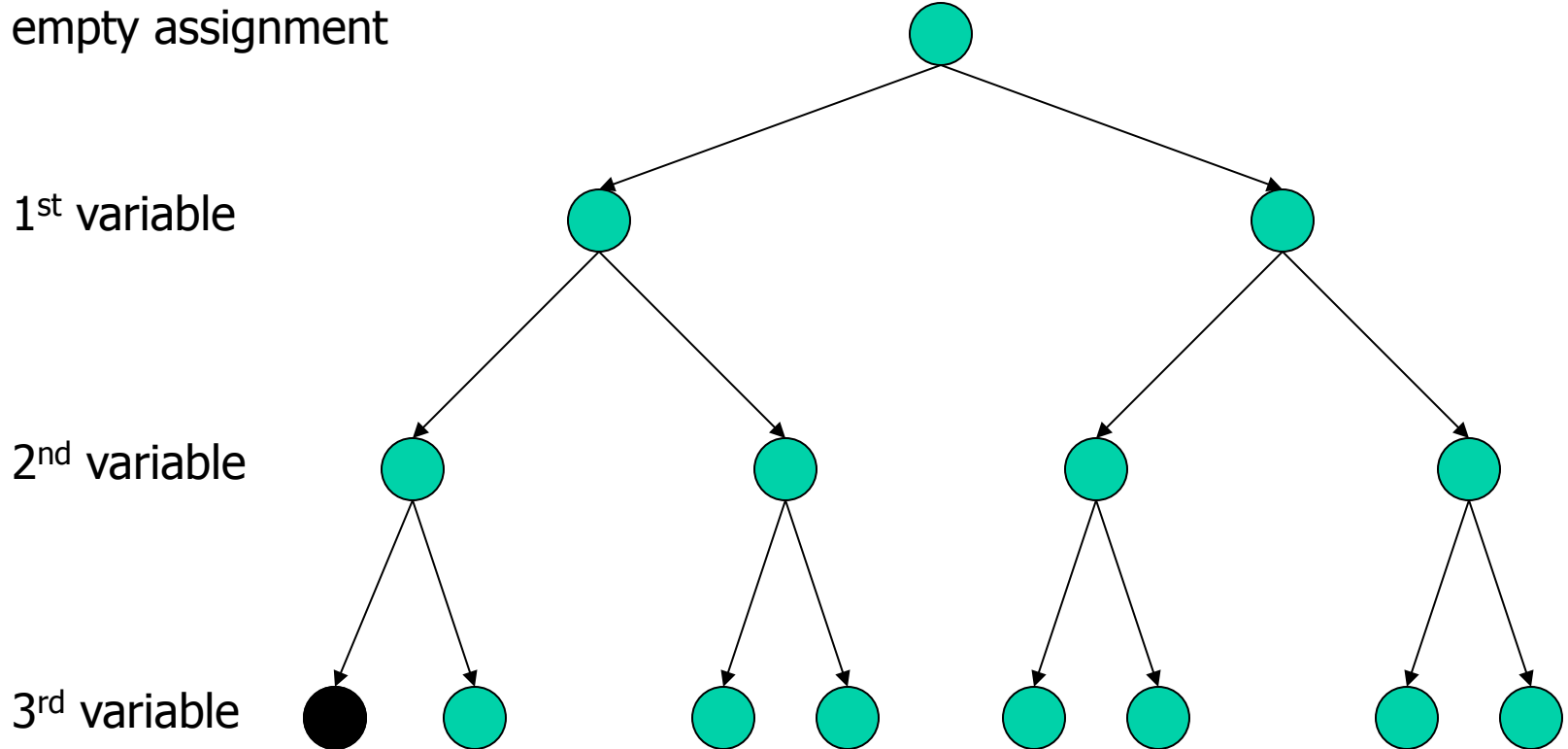
Assignment = {(var1=v11)}

→ Backtrack Search



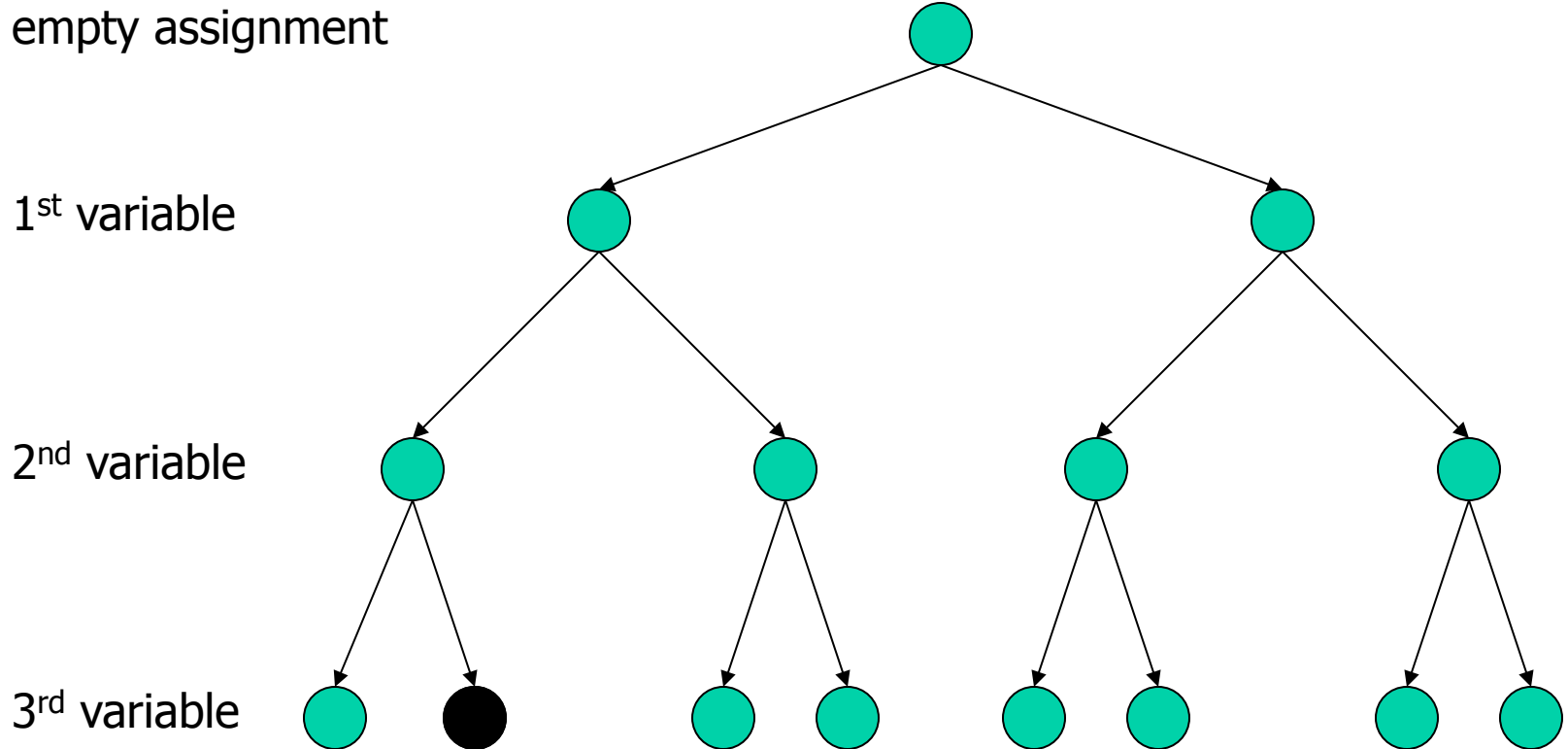
Assignment = {(var1=v11),(var2=v21)}

→ Backtrack Search



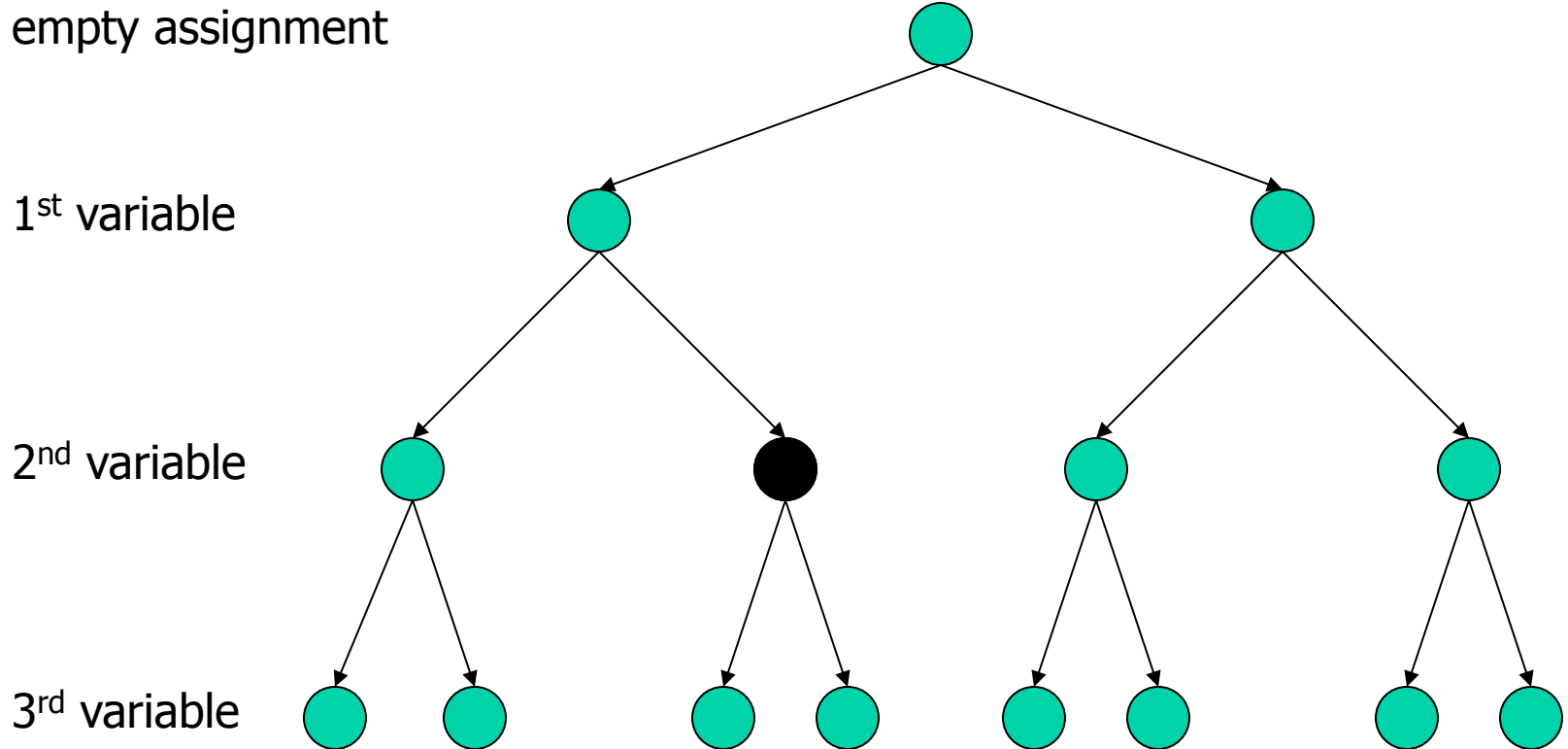
Assignment = $\{(var1=v11),(var2=v21),(var3=v31)\}$

→ Backtrack Search



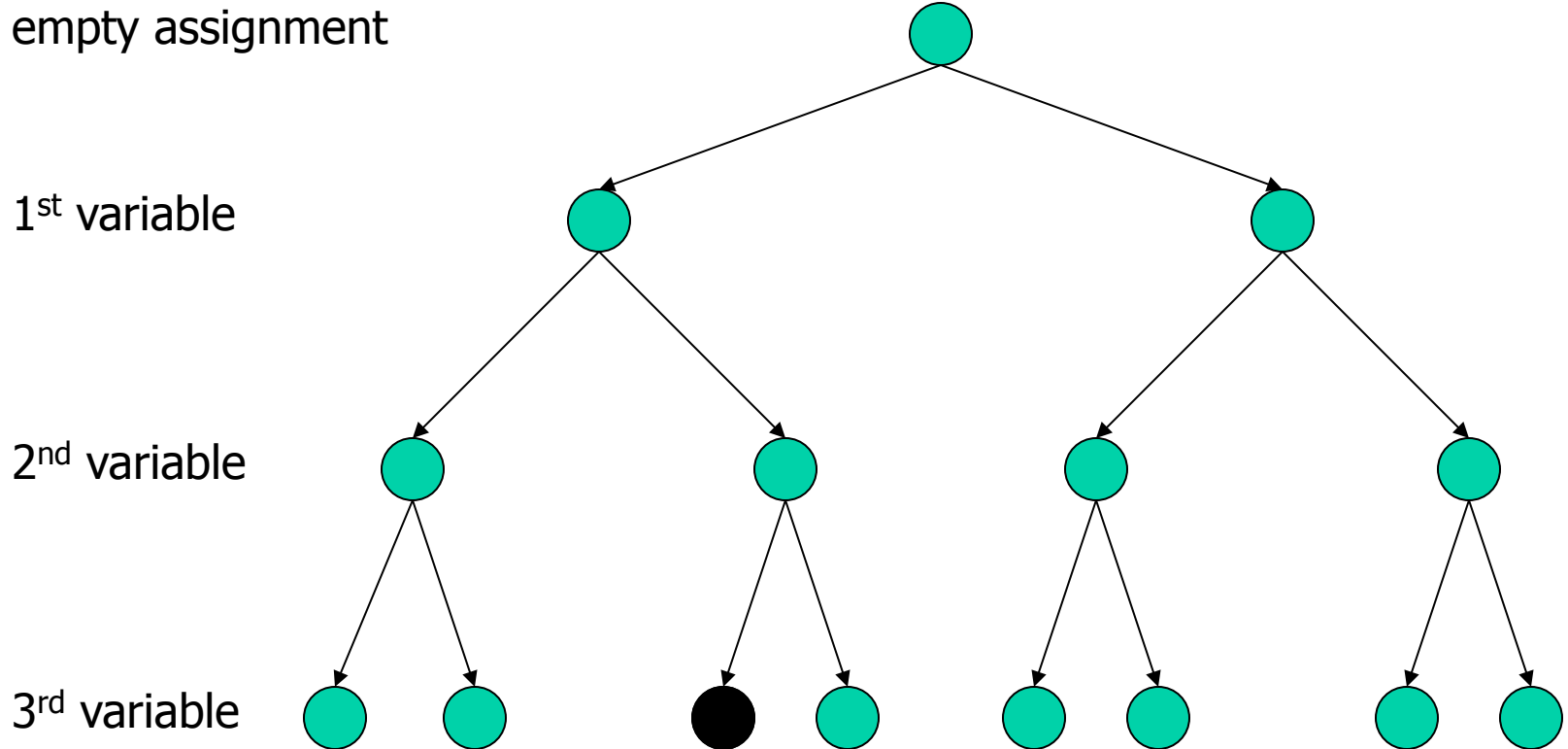
Assignment = $\{(var1=v11),(var2=v21),(var3=v32)\}$

→ Backtrack Search



Assignment = {(var1=v11),(var2=v22)}


→ Backtrack Search



Assignment = {(var1=v11),(var2=v22),(var3=v31)}

Solving CSP by search : Backtrack Search

function BACKTRACKING-SEARCH (*csp*) **returns** a solution, or failure
 return RECURSIVE-BACKTRACKING($\{\}$, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution, or failure
 if *assignment* is complete **then return** *assignment*
 var \leftarrow SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*)
 for each *value* in ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
 if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
 add {*var*=*value*} to *assignment*
 result \leftarrow RECURSIVE-BACKTRACKING(*assignment*, *csp*)
 if *result* \neq failure **then return** *result*
 remove {*var* = *value*} from *assignment*  **BACKTRACKING OCCURS HERE!!**
 return failure

♪

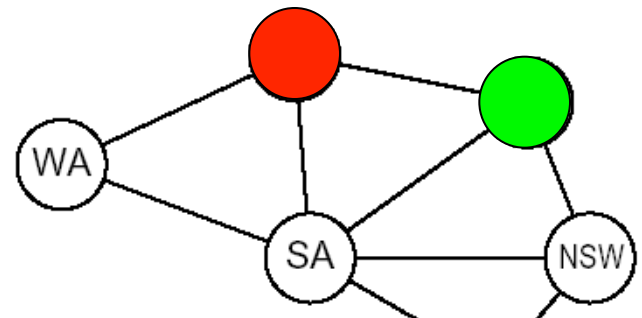
Improving Backtracking Efficiency

- Which variable should be assigned next?
 - **Minimum Remaining Values** heuristic
- In what order should its values be tried?
 - **Least Constraining Values** heuristic
- Can we detect inevitable failure early?
 - **Forward checking**

- **Constraint propagation (Arc Consistency)**
- When a path fails, can the search avoid repeating this failure?
 - **Backjumping**

- Can we take advantage of problem structure?
 - **Tree-structured CSP**

Variable & value ordering to increase the likelihood to success



Early failure-detection to decrease the likelihood to fail

Restructuring to reduce the problem's complexity

General purpose techniques

Improving backtracking efficiency

function BACKTRACKING-SEARCH (*csp*) **returns** a solution, or failure
return RECURSIVE-BACKTRACKING($\{\}$, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution, or failure
if *assignment* is complete **then return** *assignment*

var \leftarrow SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*)

for each *value* in ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**

if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**

add {*var*=*value*} to *assignment*

result \leftarrow RECURSIVE-BACKTRACKING(*assignment*, *csp*)

if *result* \neq failure **then return** *result*

remove {*var* = *value*} from *assignment*

return failure



Choice of Variable

#1: Minimum Remaining Values (aka Most-constrained-variable heuristic):

Select a variable with the fewest remaining values

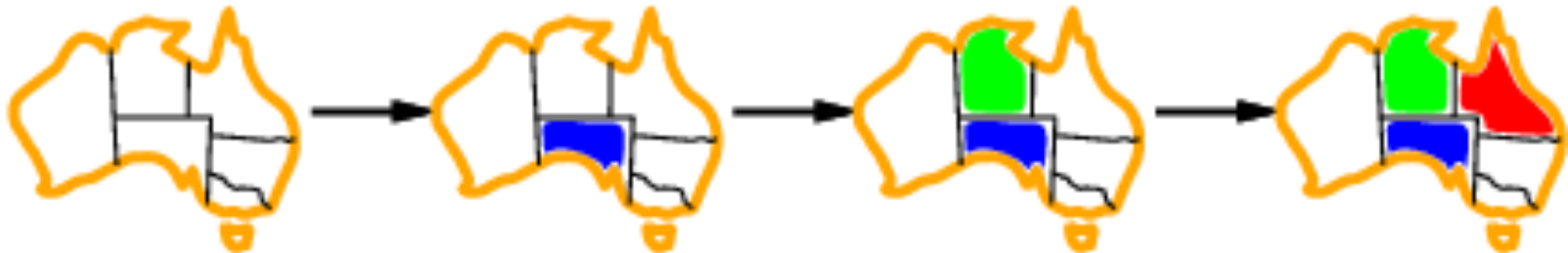


Choice of Variable, cont.

Tie-breaker among most constrained variables

#2 Most constraining variable:

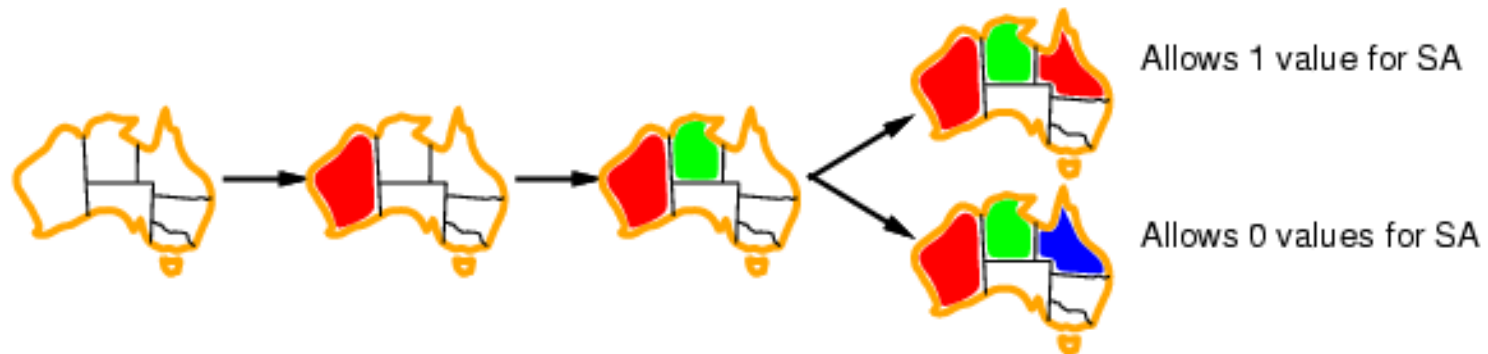
- choose the variable with the most constraints on remaining variables



Choice of Value: Least constraining value

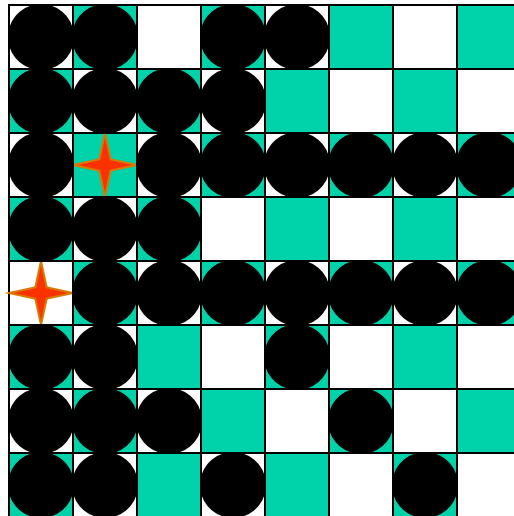
#3 Given a variable, choose the least constraining value:

- the one that rules out the fewest values in the remaining variables



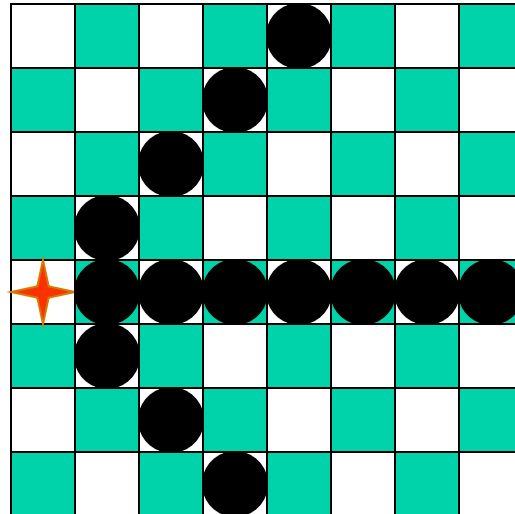
Constraint Propagation

The process of determining how the possible values of one variable affect the possible values of other variables



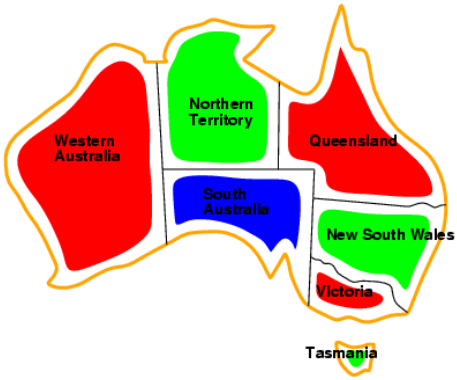
Forward Checking

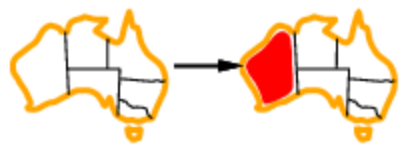
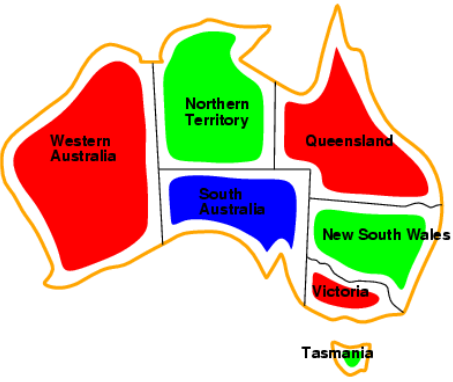
After a variable **X** is assigned a value **v**, look at each unassigned variable **Y** that is connected to **X** by a constraint and deletes from **Y**'s domain any value that is inconsistent with **v**



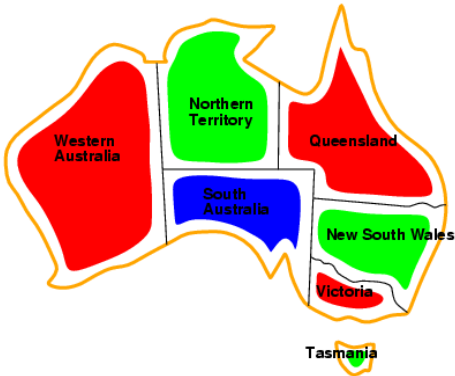
Terminate branch when any variable has no legal values & backtrack.

Forward checking



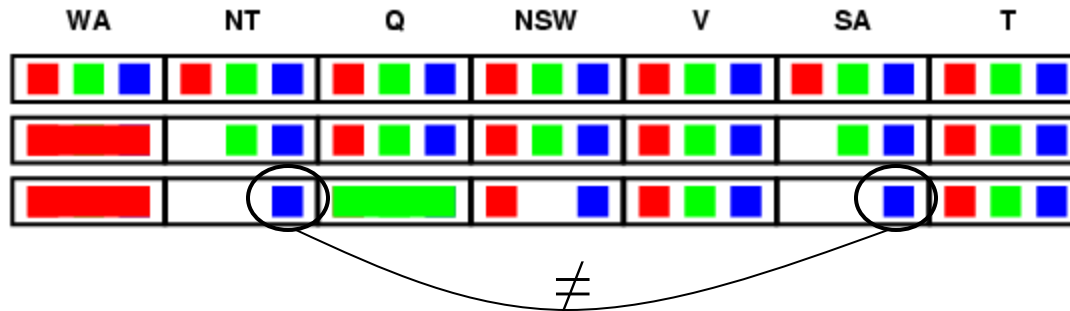


Constraint propagation



Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

What's the problem here?



NT and SA cannot both be blue!

Use: constraint propagation repeatedly to enforce constraints locally.

Definition (Arc consistency)

A constraint C_{xy} is said to be arc consistent w.r.t. x iff
for each value v of x there is an allowed value of y .

Similarly, we define that C_{xy} is arc consistent w.r.t. y .

A binary CSP is arc consistent iff every constraint C_{xy} is arc consistent wrt x as well as wrt y .

When a CSP is not arc consistent, we can make it arc consistent.

This is also called “enforcing arc consistency”.

Example

Let domains be

$$D_x = \{1, 2, 3\}, D_y = \{3, 4, 5, 6\}$$

One constraint

$$C_{xy} = \{(1,3), (1,5), (3,3), (3,6)\} \quad \text{[“allowed value pairs”]}$$

C_{xy} is not arc consistent w.r.t. x , neither w.r.t. y . Why?

To enforce arc consistency, we filter the domains, removing inconsistent values.

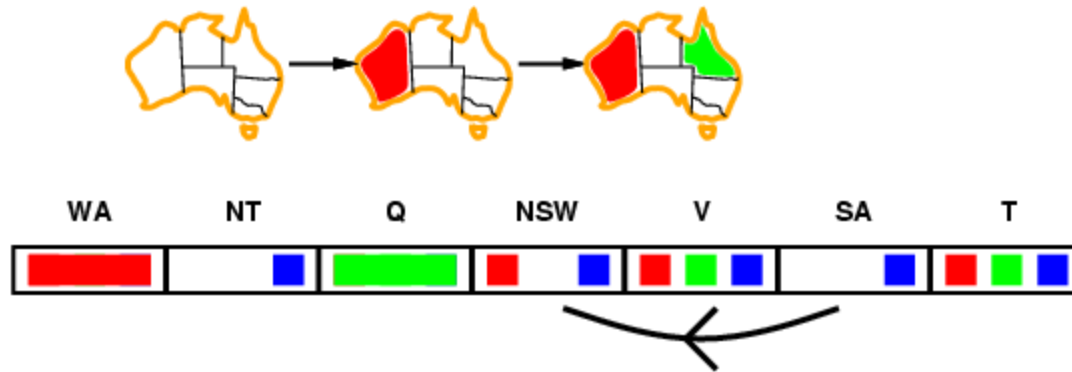
$$D'_x = \{1, 3\}, D'_y = \{3, 5, 6\}$$

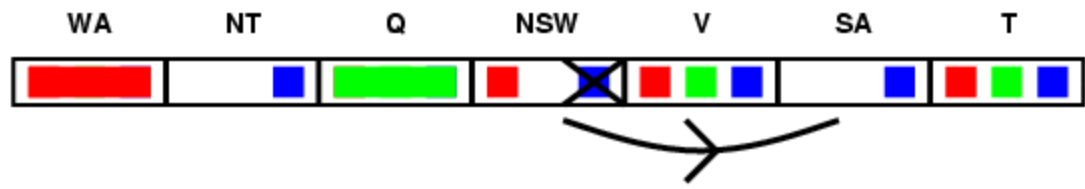
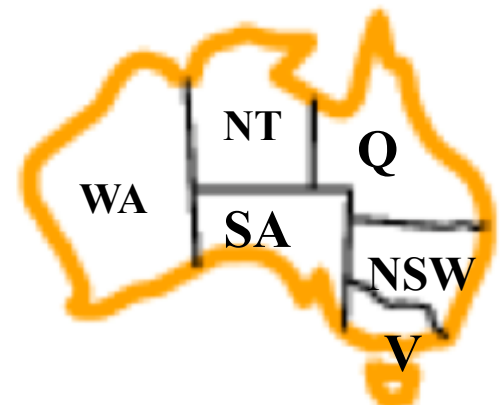
Arc consistency

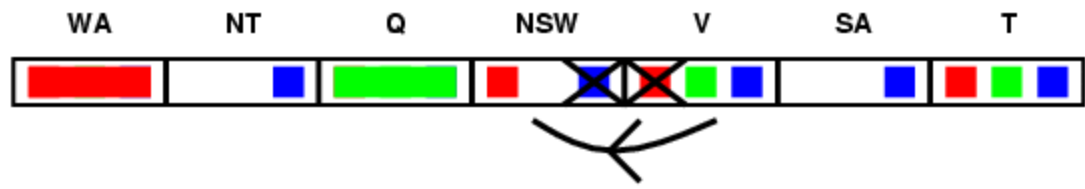
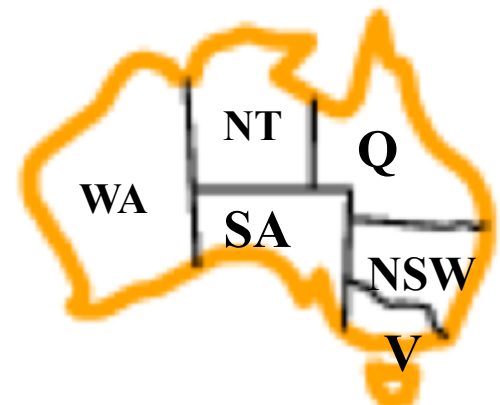
Simplest form of propagation makes each arc **consistent**.

I.e., $X \rightarrow Y$ is consistent iff

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

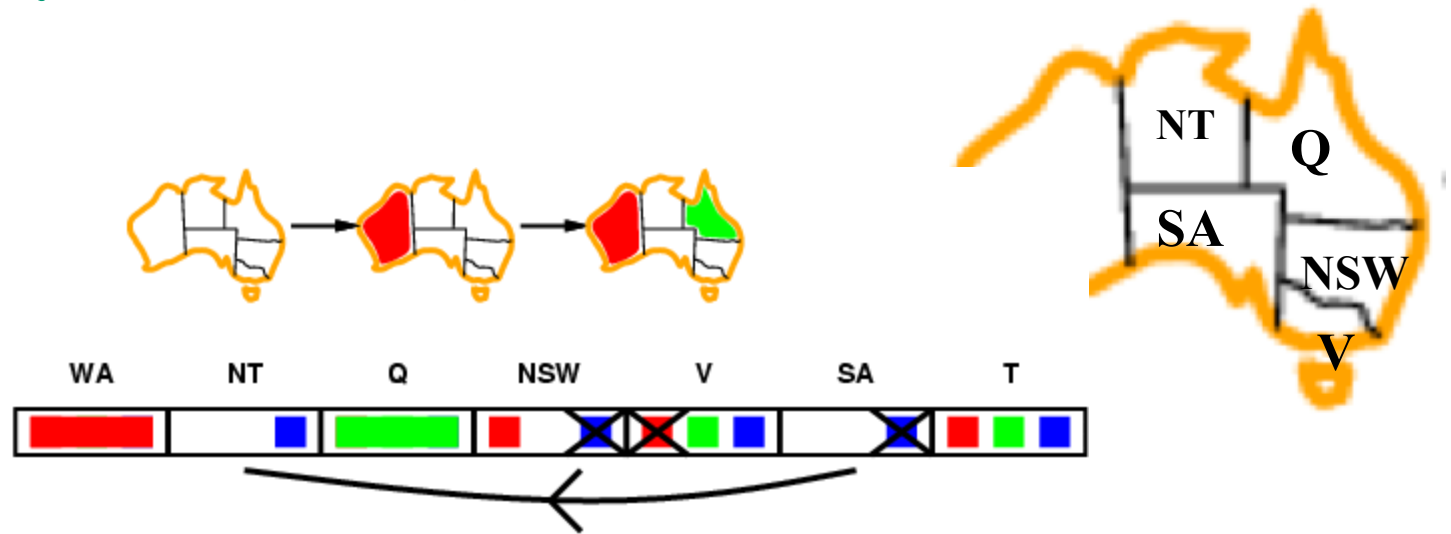






If X loses a value, neighbors of X need to be rechecked.
Arc consistency detects failure earlier than forward checking.

Can be run as a preprocessor or after each assignment.
(takes polytime each time)



Empty domain detected! Backtrack early.

Arc consistency algorithm AC-3

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \dots, X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp* If X_i 's domain

while *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$

if RM-INCONSISTENT-VALUES(X_i, X_j) **then**

for each X_k **in** NEIGHBORS[X_i] **do**

add (X_k, X_i) to *queue*

is filtered all the constraints associated with it and other variables are added to the queue

function RM-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff remove a value

removed \leftarrow false

for each x **in** DOMAIN[X_i] **do**

if no value y in DOMAIN[X_j] allows (x, y) to satisfy constraint(X_i, X_j)

then delete x from DOMAIN[X_i]; *removed* \leftarrow true

return *removed*

Binary
constraint
 X_i, X_j

n^2 = number of constraints (edges; n is the # of variables)

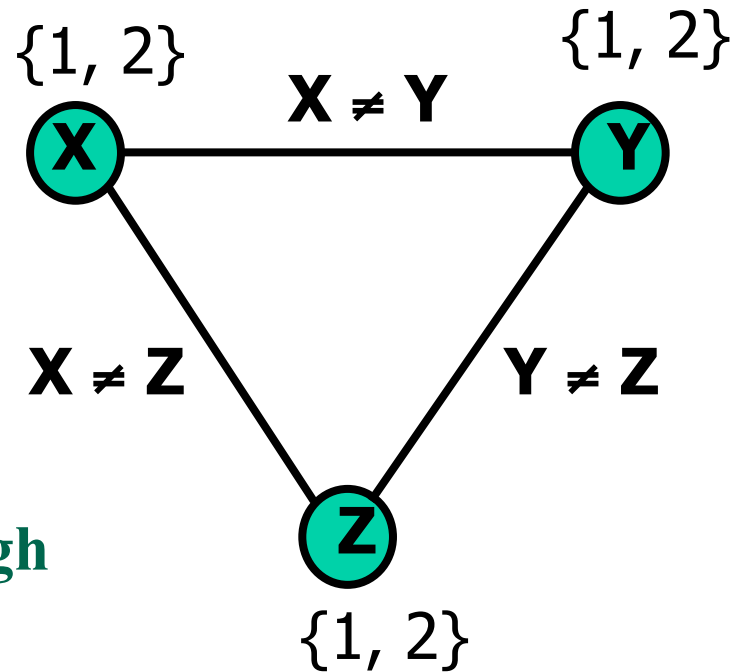
d = number of values per variable

Time complexity: REMOVE-ARC-INCONSISTENCY takes $O(d^2)$ time

Each variable is inserted in Queue up to d times, since at most d values can be deleted

\rightarrow AC3 takes $O(n^2d^3)$ time to run

Beyond Arc Consistency



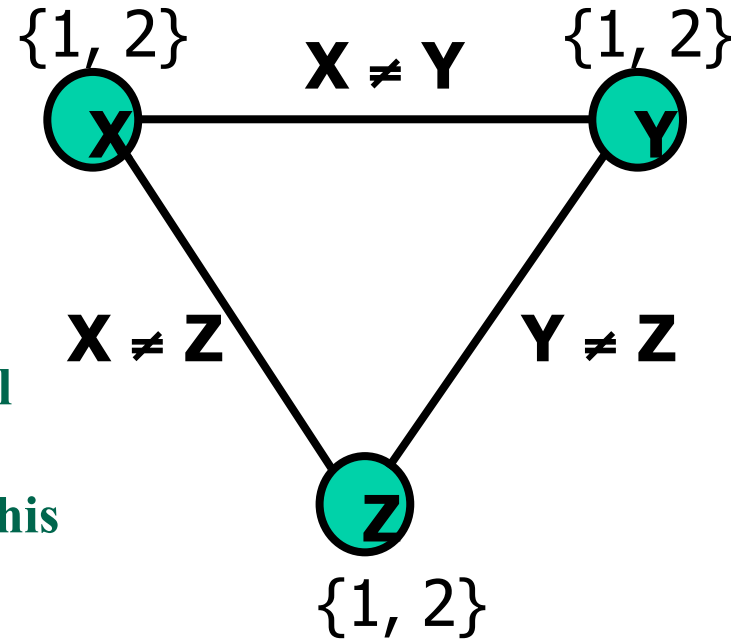
Is this network arc consistent?

What is the solution?

Clearly arc consistency is not enough to guarantee global consistency. There are other forms of consistency, such as k-consistency.

But when $k = n$ (num vars), we are looking at the original problem!

k - Consistency



A graph is **K-consistent** iff the following is true:

Choose values of any $K-1$ variables that satisfy all the constraints among these variables and choose any K th variable. Then, there exists a value for this K th variable that satisfies all the constraints among these K variables.

A graph is **strongly K-consistent** if it is **J-consistent** for all $J \leq K$.

What type of consistency would we need here to solve any constraint problem without search?

$$K = N$$

Consistency

Node consistency = strong 1- consistency

Arc consistency = strong 2- consistency

**(note: arc-consistency is usually
assumed to include node-consistency as well).**

See R&N sect. 6.2.3 for “path-consistency” = 3-consistency for binary CSPs.

**Algorithms exist for making a constraint graph strongly K -consistent for $K > 2$
but in practice they are rarely used because of efficiency issues.**

**Other consistency notions involve “global constraints,” spanning many
variables. E.g. AllDiff constraint can handle Pigeon Hole principle.**

Summary: Solving a CSP

Search:

- can find solutions, but may examine many non-solutions along the way

Constraint Propagation:

- can rule out non-solutions, but but may not lead to full solution.

Interweave **constraint propagation** and **search**

- Perform constraint propagation at each search step.
- *Goal: Find the right balance between search (backtracking) and propagation (reasoning).*

Surprising efficiency (last 10 yrs):

100K + up to one million variable CSP problems are now solvable!

See also local search. R&N 6.4