# Constraint Satisfaction

**Reading: Russell & Norvig Chapter 5,**
**Kumar: "Algorithms for constraint satisfaction problems: A survey"**



## Overview

- Constraint Satisfaction Problems (CSP) share some common features and have specialized methods
  - View a problem as a **set of variables** to which we have to assign **values** that satisfy a number of **problem-specific constraints**.
  - Constraint solvers, constraint logic programming…
- Algorithms for CSP
  - Backtracking (systematic search)
  - Constraint propagation (k-consistency)
  - Variable ordering heuristics
  - Backjumping and dependency-directed backtracking



## Informal Definition of CSP

- CSP = Constraint Satisfaction Problem
- Given
  - (1) a finite set of variables
  - (2) each with a domain of possible values (often finite)
  - (3) a set of constraints that limit the values the variables can take on
- A **solution** is an assignment of a value to each variable such that all the constraints are satisfied.
- Tasks might be to decide if a solution exists, to find a solution, to find all solutions, or to find the "best solution" according to some metric.



## SEND + MORE = MONEY

Assign distinct digits to the letters
S, E, N, D, M, O, R, Y
such that

```
       S E N D
  +    M O R E
  -----------
     = M O N E Y
```

holds.



## SEND + MORE = MONEY

Assign distinct digits to the letters
S, E, N, D, M, O, R, Y
such that

```
       S E N D
  +    M O R E
  -----------
     = M O N E Y
```

holds.

Solution

```
      9 5 6 7
  +   1 0 8 5
  -----------
  = 1 0 6 5 2
```



## Modeling

Formalize the problem as a CSP:

- number of variables: $n$
- constraints: $c_1, \ldots, c_m \subseteq Z^n$
- problem: Find $a = (v_1, \ldots, v_n) \in Z^n$ such that $a \in c_i$, for all $1 \leq i \leq m$

# A Model for MONEY

- number of variables: 8
- constraints:

$c_1 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid 0 \le S,\dots,Y \le 9\}$

$$c_2 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid \begin{array}{r} 1000*S + 100*E + 10*N + D \\ +\ 1000*M + 100*O + 10*R + E \\ =\ 10000*M + 1000*O + 100*N + 10*E + Y\} \end{array}$$



# A Model for MONEY (continued)

- more constraints

$c_3 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid S \neq 0\}$
$c_4 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid M \neq 0\}$
$c_5 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid$ S…Y all different$\}$



# Solution for MONEY

$c_1 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid \mathbf{0 \le S,\dots,Y \le 9}\}$

$$c_2 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid \begin{array}{r} \mathbf{1000*S + 100*E + 10*N + D} \\ \mathbf{+\ 1000*M + 100*O + 10*R + E} \\ \mathbf{=\ 10000*M + 1000*O + 100*N + 10*E + Y}\} \end{array}$$

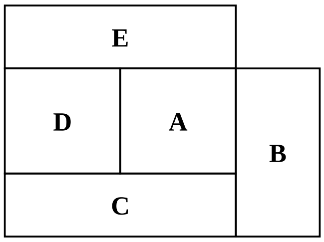
$c_3 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid \mathbf{S \neq 0}\}$
$c_4 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid \mathbf{M \neq 0}\}$
$c_5 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid$ **S…Y all different**$\}$

**Solution:** $(9,5,6,7,1,0,8,2) \in Z^8$



# Example: Map Coloring

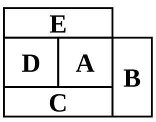
- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.



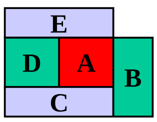



# Example: Map Coloring

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: A≠B, A≠C, A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
- One solution: A=red, B=green, C=blue, D=green, E=blue



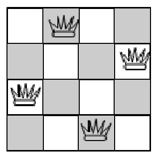

=>






# N-queens Example (4 in our case)

- Standard test case in CSP research
- Variables are the rows: r1, r2, r3, r4
- Values are the columns: {1, 2, 3, 4}
- So, the constraints include:
  - $C_{r1,r2}$ = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
  - $C_{r1,r3}$ = {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
  - Etc.
  - What do these constraints mean?



## Example: SATisfiability

- Given a set of propositions containing variables, find an assignment of the variables to {false,true} that satisfies them.
- Example, the clauses:
  - $A \lor B \lor \sim C$, $\sim A \lor D$
  - (equivalent to $C \rightarrow A \lor B$, $A \rightarrow D$)
- Are satisfied by

  A = false

  B = true

  C = false

  D = false



## Real-world problems

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design



## Formal definition of a CSP

A constraint satisfaction problem (CSP) consists of

- a set of variables $X = \{x_1, x_2, \ldots x_n\}$
  - each with an associated domain of values $\{d_1, d_2, \ldots d_n\}$.
  - The domains are typically finite
- a set of constraints $\{c_1, c_2 \ldots c_m\}$ where
  - each constraint defines a predicate which is a relation over a particular subset of X.
  - E.g., $C_i$ involves variables $\{X_{i1}, X_{i2}, \ldots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \ldots D_{ik}$
- **Unary** constraint: only involves one variable
- **Binary** constraint: only involves two variables



## Formal definition of a CSP

- Instantiations
  - An **instantiation** of a subset of variables S is an assignment of a legal domain value to each variable in in S
  - An instantiation is **legal** iff it does not violate any (relevant) constraints.
- A **solution** is an instantiation of all of the variables in the network.



## Typical Tasks for CSP

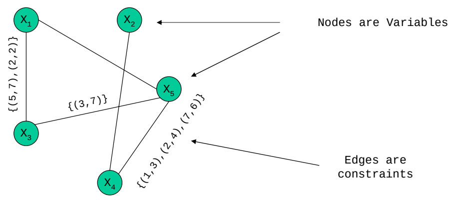
- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CSP into an equivalent CSP that is easier to solve.



## Binary CSP

- A **binary CSP** is a CSP in which all of the constraints are binary or unary.
- Any non-binary CSP can be converted into a binary CSP by introducing additional variables.
- A binary CSP can be represented as a **constraint graph**, which has a node for each variable and an arc between two nodes if and only there is a constraint involving the two variables.
  - Unary constraint appears as self-referential arc

# Binary Constraint Graph

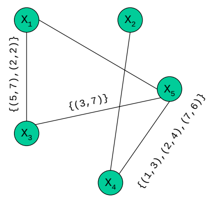





# Matrix Representation for Binary Constraints

| C | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | | | X | | | | |
| 2 | | | | X | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | X | |

Matrix for the constraint between $X_4$ and $X_5$ in the graph






# Constraint Solving is Hard

Constraint solving is not possible for general constraints.

Example:

C: $n > 2$

C': $a^n + b^n = c^n$

Constraint programming separates constraints into

- basic constraints: complete constraint solving
- non-basic constraints: propagation (incomplete); search needed



# CSP as a Search Problem

States are defined by the values assigned so far

- Initial state: the empty assignment { }
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment
  → fail if no legal assignments
- Goal test: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth *n* with *n* variables
   → use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. Local search methods are useful.



# Systematic search: Backtracking

**(a.k.a. depth-first search)**

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we've reached a dead end and need to backtrack to the previous variable
- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values.



# Backtracking search

- Variable assignments are commutative, i.e.,
  [ A = red then B = green ] same as [ B = green then A = 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 backtracking search
- Backtracking search is the basic algorithm for CSPs
- Can solve *n*-queens for $n \approx 100$

## Backtracking search

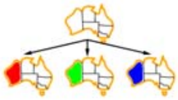
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, 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 ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failue then return result
      remove { var = value } from assignment
  return failure
```



## Backtracking example

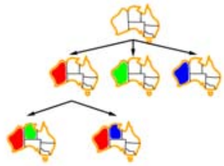


## Backtracking example



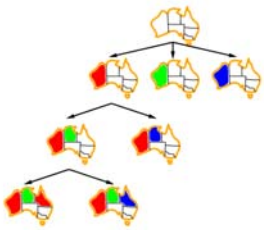


## Backtracking example



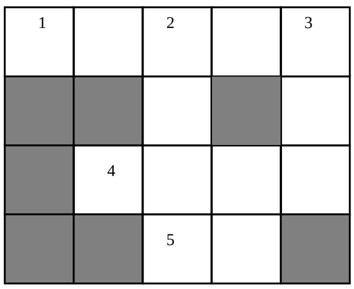


## Backtracking example

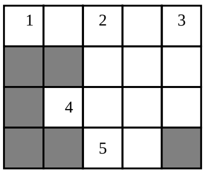




## Example: Crossword Puzzle



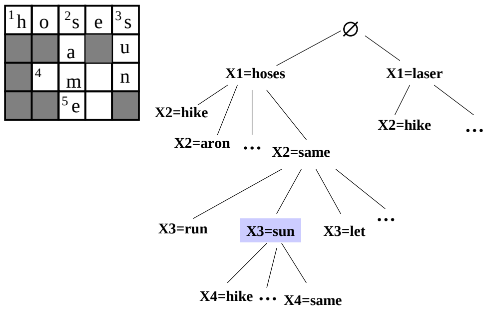
## Example: XWORD Puzzle




- Variables and their domains
  - X1 is 1 across D1 is 5-letter words
  - X2 is 2 down D2 is 4-letter words
  - X3 is 3 down D3 is 3-letter words
  - X4 is 4 across D4 is 4-letter words
  - X5 is 5 across D5 is 2-letter words
- Constraints (implicit/intensional)
  - C12 is “the 3rd letter of X1 must equal the 1st letter of X2”
  - C13 is “the 5th letter of X1 must equal the 1st letter of X3”.
  - C24 is …
  - C25 is …
  - C34 is ...



## Backtracking: XWORD






S ∈ Z
E ∈ Z
N ∈ Z
D ∈ Z
M ∈ Z
O ∈ Z
R ∈ Z
Y ∈ Z

```
  S E N D
+ M O R E
---------
= M O N E Y
```

0≤S,…,Y≤9
S ≠ 0
M ≠ 0
S…Y all different

1000*S + 100*E + 10*N + D
\+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y



Propagate

S ∈ {0..9}
E ∈ {0..9}
N ∈ {0..9}
D ∈ {0..9}
M ∈ {0..9}
O ∈ {0..9}
R ∈ {0..9}
Y ∈ {0..9}

```
  S E N D
+ M O R E
---------
= M O N E Y
```

0≤S,…,Y≤9
S ≠ 0
M ≠ 0
S…Y all different

1000*S + 100*E + 10*N + D
\+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y



Propagate

S ∈ {1..9}
E ∈ {0..9}
N ∈ {0..9}
D ∈ {0..9}
M ∈ {1..9}
O ∈ {0..9}
R ∈ {0..9}
Y ∈ {0..9}

```
  S E N D
+ M O R E
---------
= M O N E Y
```

0≤S,…,Y≤9
S ≠ 0
M ≠ 0
S…Y all different

1000*S + 100*E + 10*N + D
\+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y



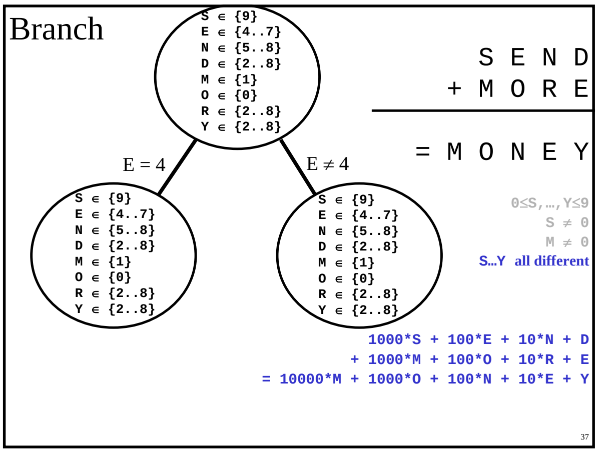
Propagate

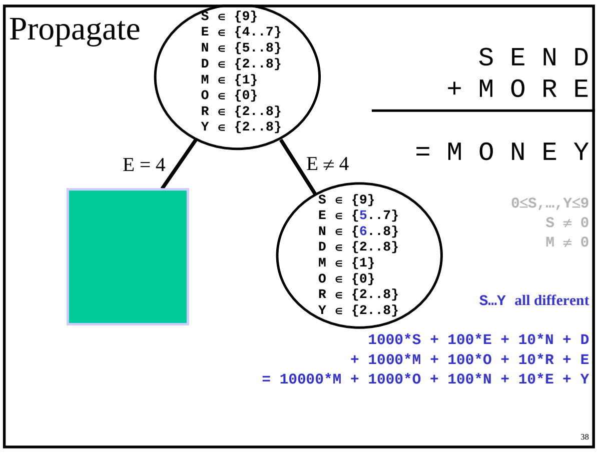
S ∈ {9}
E ∈ {4..7}
N ∈ {5..8}
D ∈ {2..8}
M ∈ {1}
O ∈ {0}
R ∈ {2..8}
Y ∈ {2..8}

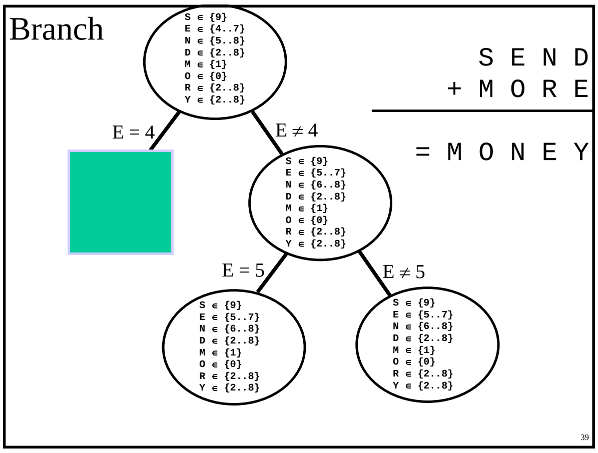
```
  S E N D
+ M O R E
---------
= M O N E Y
```

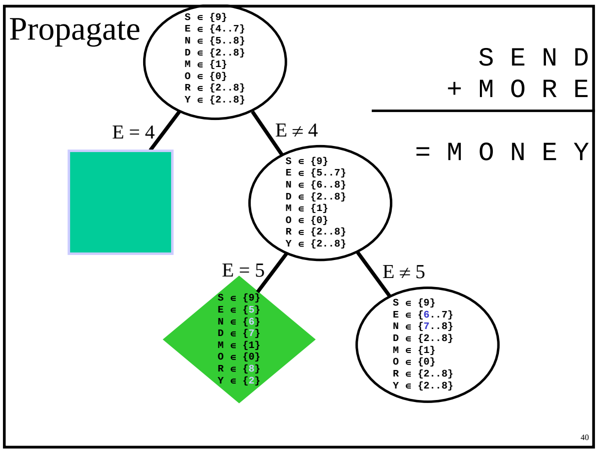
0≤S,…,Y≤9
S ≠ 0
M ≠ 0
S…Y all different

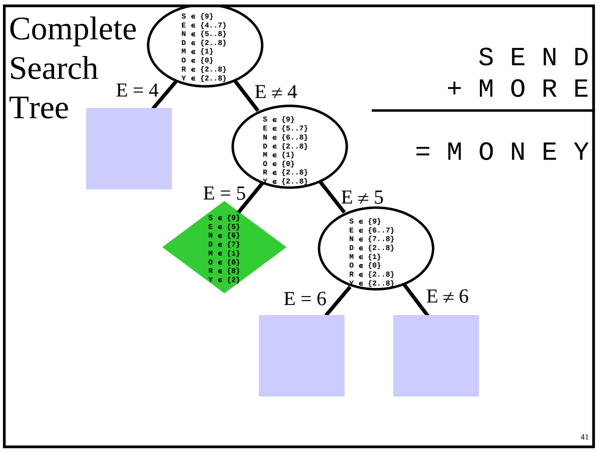
1000*S + 100*E + 10*N + D
\+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y




## Problems with backtracking

- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can explore areas of the search space that aren't likely to succeed
  - Variable ordering can help

## 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.k.a. minimum remaining values (MRV) heuristic



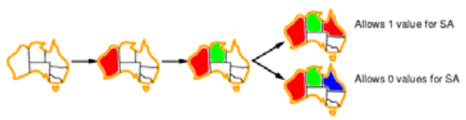
## Most constraining variable

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



## Least constraining value

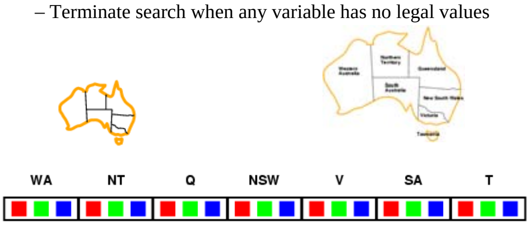
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables






## Forward checking

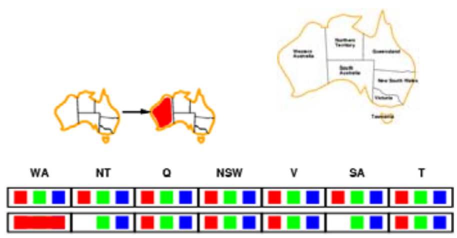
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values





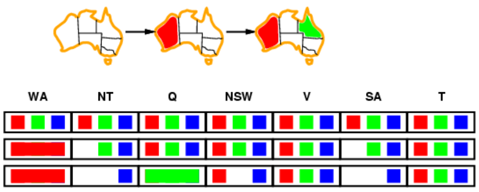

## Forward checking

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



## Forward checking

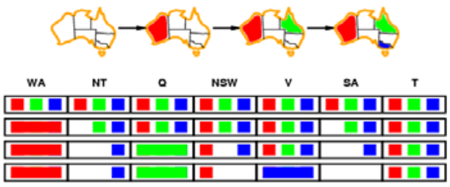
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values






## Forward checking

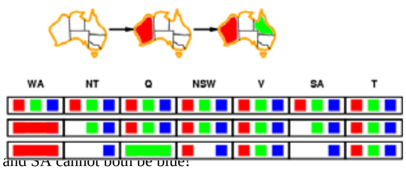
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values






## Constraint propagation

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




- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally



## Issues in Propagation

- Expressivity: What kind of information can be expressed as propagators?
- Completeness: What behavior can be expected from propagation?
- Efficiency: How much computational resources does propagation consume?



## Completeness of Propagation

- Given: Basic constraint C and propagator P.
- Propagation is complete, if for every variable `x` and every value `v` in the domain of `x`, there is an assignment in which `x=v` that satisfies C and P.
- Complete propagation is also called *domain-consistency* or *arc-consistency*.



## Example: Complete All Different

- C: `w` $\in$ `{1,2,3,4}`
  `x` $\in$ `{ 2,3,4}`
  `y` $\in$ `{ 2,3,4}`
  `z` $\in$ `{ 2,3,4}`
- P: `all_different(w,x,y,z)`

## Example: Complete All Different

- C: `w ∈ {1,2,3,4}`
  `x ∈ {  2,3,4}`
  `y ∈ {  2,3,4}`
  `z ∈ {  2,3,4}`
- P: `all_different(w,x,y,z)`
- Most efficient known algorithm: $O(|X|^2 \, d_{max}{}^2)$



## Basic Constraints vs. Propagators

- Basic constraints
  - are conjunctions of constraints of the form
    $X \in S$, where $S$ is a finite set of integers
  - enjoy complete constraint solving
- Propagators
  - can be arbitrarily expressive (arithmetic, symbolic)
  - implementation typically fast but incomplete



## Symmetry Breaking

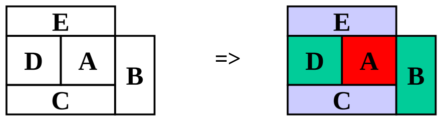
Often, the most efficient model admits many different solutions that are essentially the same (“symmetric” to each other).

Symmetry breaking tries to improve the performance of search by eliminating such symmetries.



## Example: Map Coloring

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: A≠B, A≠C,A ≠ E, A ≠ D, B ≠ C, C ≠ D, D ≠ E
- One solution: A=red, B=green, C=blue, D=green, E=blue






## Performance of Symmetry Breaking

- All solution search: Symmetry breaking usually improves performance; often dramatically
- One solution search: Symmetry breaking may or may not improve performance



## Optimization

- Modeling: define optimization function
- Propagation algorithms: identify propagation algorithms for optimization function
- Branching algorithms: identify branching algorithms that lead to good solutions early
- Exploration algorithms: extend existing exploration algorithms to achieve optimization

# Optimization: Example

**SEND + MOST = MONEY**



# SEND + MOST = MONEY

Assign distinct digits to the letters
S, E, N, D, M, O, T, Y
such that

```
      S E N D
+     M O S T
  ___________
= M O N E Y
```

holds and

M O N E Y is maximal.



# Branch and Bound

Identify a branching algorithm that finds good solutions early.

Example: TSP: Traveling Salesman Problem

Idea: Use the earlier found value as a bound for the rest branches.



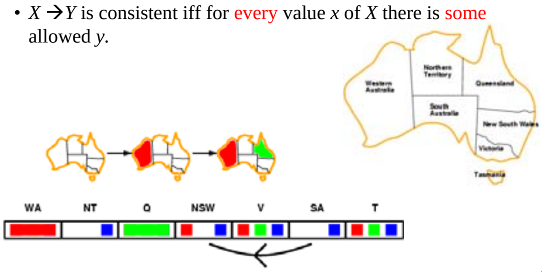
# Consistency

- Node consistency
  - A node X is **node-consistent** if every value in the domain of X is consistent with X's unary constraints
  - A graph is node-consistent if all nodes are node-consistent
- Arc consistency
  - An arc (X, Y) is **arc-consistent** if, for every value x of X, there is a value y for Y that satisfies the constraint represented by the arc.
  - A graph is arc-consistent if all arcs are arc-consistent
- To create arc consistency, we perform **constraint propagation**: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs



# Arc consistency

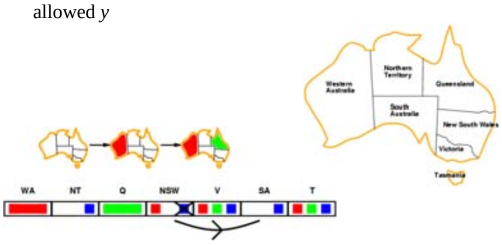
- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff for every value $x$ of $X$ there is some allowed $y$.





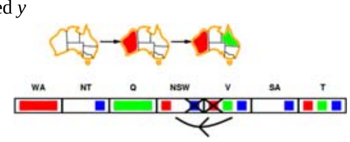

# Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff for every value $x$ of $X$ there is some allowed $y$



# Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff for every value *x* of *X* there is some allowed *y*

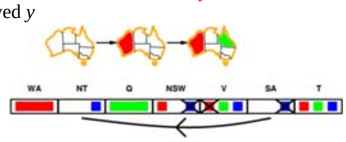



- If *X* loses a value, neighbors of *X* need to be rechecked



# Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *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

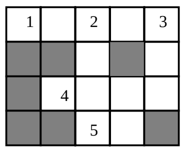


# Arc consistency algorithm AC-3

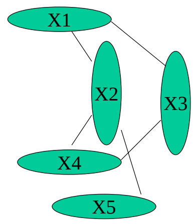
```
function AC-3(csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) <- REMOVE-FIRST(queue)
      if RM-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue

function RM-INCONSISTENT-VALUES(Xi, Xj) returns true iff remove a value
   removed <- false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
         then delete x from DOMAIN[Xi]; removed <- true
   return removed
```

- Time complexity: $O(n^2d^3)$






**Variables:**
X1
X2
X3
X4
X5

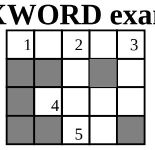


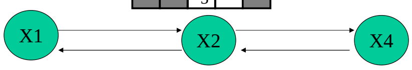

**Domains:**
D1 = {hoses, laser, sheet, snail,steer}
D2 = {hike, aron, keet, earn, same}
D3 = {run, sun, let, yes, eat, ten}
D4 = {hike, aron, keet, earn, same}
D5 = {no, be, us, it}

**Constraints (explicit/extensional):**
C12 = {(hoses,same),
(laser,same),
(sheet,earn),
(steer,earn)}
C13 = ...



# Constraint propagation: XWORD example



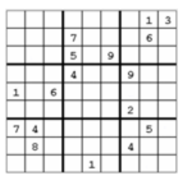




| X1 | X2 | X4 |
|---|---|---|
| ~~hoses~~ | ~~hike~~ | ~~hike~~ |
| ~~laser~~ | ~~aron~~ | aron |
| sheet | ~~keet~~ | ~~keet~~ |
| ~~snail~~ | earn | ~~earn~~ |
| steer | ~~same~~ | ~~same~~ |



# The Sudoku Puzzle

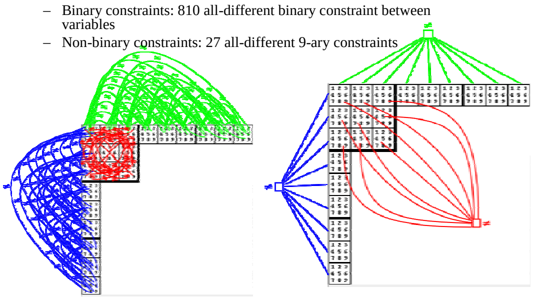
- Number place
- Main properties
  - NP-complete [Yato 03]
  - Well-formed Sudoku: has 1 solution [Simonis 05]
  - Minimal Sudoku
    - In a 9x9 Sudoku: smallest known number of givens is 17 [Royle]
  - Symmetrical puzzles
    - Many axes of symmetry
    - Position of the givens, not their values
    - Often makes the puzzle non-minimal
  - Level of difficulties
    - Varied ranking systems exist
    - Mimimality and difficulty not related



## Sudoku as a CSP

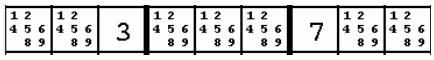
**(9x9 puzzles)**

- Variables are the cells
- Domains are sets {1,…,9}
- Two models
  - Binary constraints: 810 all-different binary constraint between variables
  - Non-binary constraints: 27 all-different 9-ary constraints



## Solving Sudoku as a CSP

- Search
  - Builds solutions by enumerating consistent combinations
- Constraint propagation
  - Removes values that do not appear in a solution
  - Example, arc-consistency:

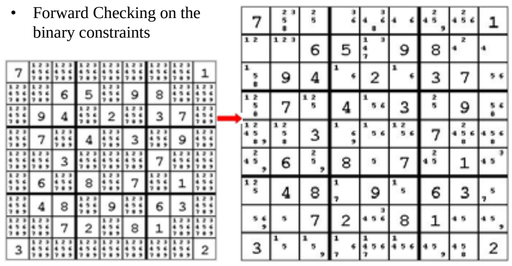


## Search

- Backtrack search
  - Systematically enumerate solutions by instantiating one variable after another
  - Backtracks when a constraint is broken
  - Is sound and complete (guaranteed to find solution if one exists)
- Propagation
  - Cleans up domain of ‘future’ variables, during search, given current instantiations
    - Forward Checking (FC): weak form of arc-consistency
    - Maintaining Arc-Consistency (MAC): arc-consistency

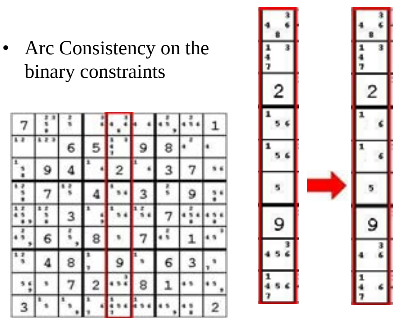
## Forward Checking (FC)

- Forward Checking on the binary constraints



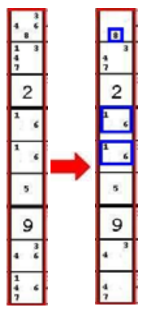
## Maintaining Arc-Consistency (MAC)

- Arc Consistency on the binary constraints



## Generalized Arc-Consistency (GAC)

- Operates on the non-binary constraints
- Subsumes AC
- Can be done in polynomial time for all-different constraints [Régin 94]



## The way people play

- 'Cross-hash,' sweep through lines, columns, and blocks
- Pencil in possible positions of values
- Generally, look for patterns, some intricate, and give them funny names:
  - Naked single, locked pairs, swordfish, medusa, etc.
- 'Identified' dozens of strategies
  - Many fall under a unique constraint propagation technique
- But humans do not seem to be able to carry **simple** inference (i.e., propagation) in a **systematic** way for more than a few steps

## K-consistency

- K- consistency generalizes the notion of arc consistency to sets of more than two variables.
  - A graph is K-consistent if, for legal values of any K-1 variables in the graph, and for any Kth variable $V_k$, there is a legal value for $V_k$
- Strong K-consistency = J-consistency for all J<=K
- Node consistency = strong 1-consistency
- Arc consistency = strong 2-consistency
- Path consistency = strong 3-consistency



## Why do we care?

1. If we have a CSP with N variables that is known to be **strongly N-consistent**, we can solve it **without backtracking**
2. For any CSP that is **strongly K-consistent**, if we find an **appropriate variable ordering** (one with "small enough" branching factor), we can solve the CSP **without backtracking**



## Improving Backtracking

- Use other search techniques: uniform cost, A*, …
- Variable ordering can help improve backtracking.
- Typical heuristics:
  - Prefer variables which maximally constrain the rest of the search space
  - When picking a value, choose the least constraining value



## The Future

- Constraint programming will become a standard technique in OR for solving combinatorial problems, along with local search and integer programming.
- Constraint programming techniques will be tightly integrated with integer programming and local search.



## ACC 1997/98: A Success Story of Constraint Programming

- Integer programming + enumeration, 24 hours

  Nemhauser, Trick: Scheduling a Major College Basketball Conference, Operations Research, 1998, 46(1)
- Constraint programming, less than 1 minute.

  Henz: Scheduling a Major College Basketball Conference - Revisited, Operations Research, to appear

## Round Robin Tournament Planning Problems

- n teams, each playing a fixed number of times r against every other team
- r = 1: single, r = 2: double round robin.
- Each match is home match for one and away match for the other
- Dense round robin:
  - At each date, each team plays at most once.
  - The number of dates is minimal.



## The ACC 1997/98 Problem

- 9 teams participate in tournament
- Dense double round robin:
  - there are 2 * 9 dates
  - at each date, each team plays either home, away or has a "bye"
- Alternating weekday and weekend matches



## The ACC 1997/98 Problem

- No team can play away on both last dates.
- No team may have more than two away matches in a row.
- No team may have more than two home matches in a row.
- No team may have more than three away matches or byes in a row.
- No team may have more than four home matches or byes in a row.



## The ACC 1997/98 Problem

- Of the weekends, each team plays four at home, four away, and one bye.
- Each team must have home matches or byes at least on two of the first five weekends.
- Every team except FSU has a traditional rival. The rival pairs are Clem-GT, Duke-UNC, UMD-UVA and NCSt-Wake. In the last date, every team except FSU plays against its rival, unless it plays against FSU or has a bye.



## The ACC 1997/98 Problem

- The following pairings must occur at least once in dates 11 to 18: Duke-GT, Duke-Wake, GT-UNC, UNC-Wake.
- No team plays in two consecutive dates away against Duke and UNC. No team plays in three consecutive dates against Duke UNC and Wake.
- UNC plays Duke in last date and date 11.
- UNC plays Clem in the second date.
- Duke has bye in the first date 16.



## The ACC 1997/98 Problem

- Wake does not play home in date 17.
- Wake has a bye in the first date.
- Clem, Duke, UMD and Wake do not play away in the last date.
- Clem, FSU, GT and Wake do not play away in the fist date.
- Neither FSU nor NCSt have a bye in the last date.
- UNC does not have a bye in the first date.