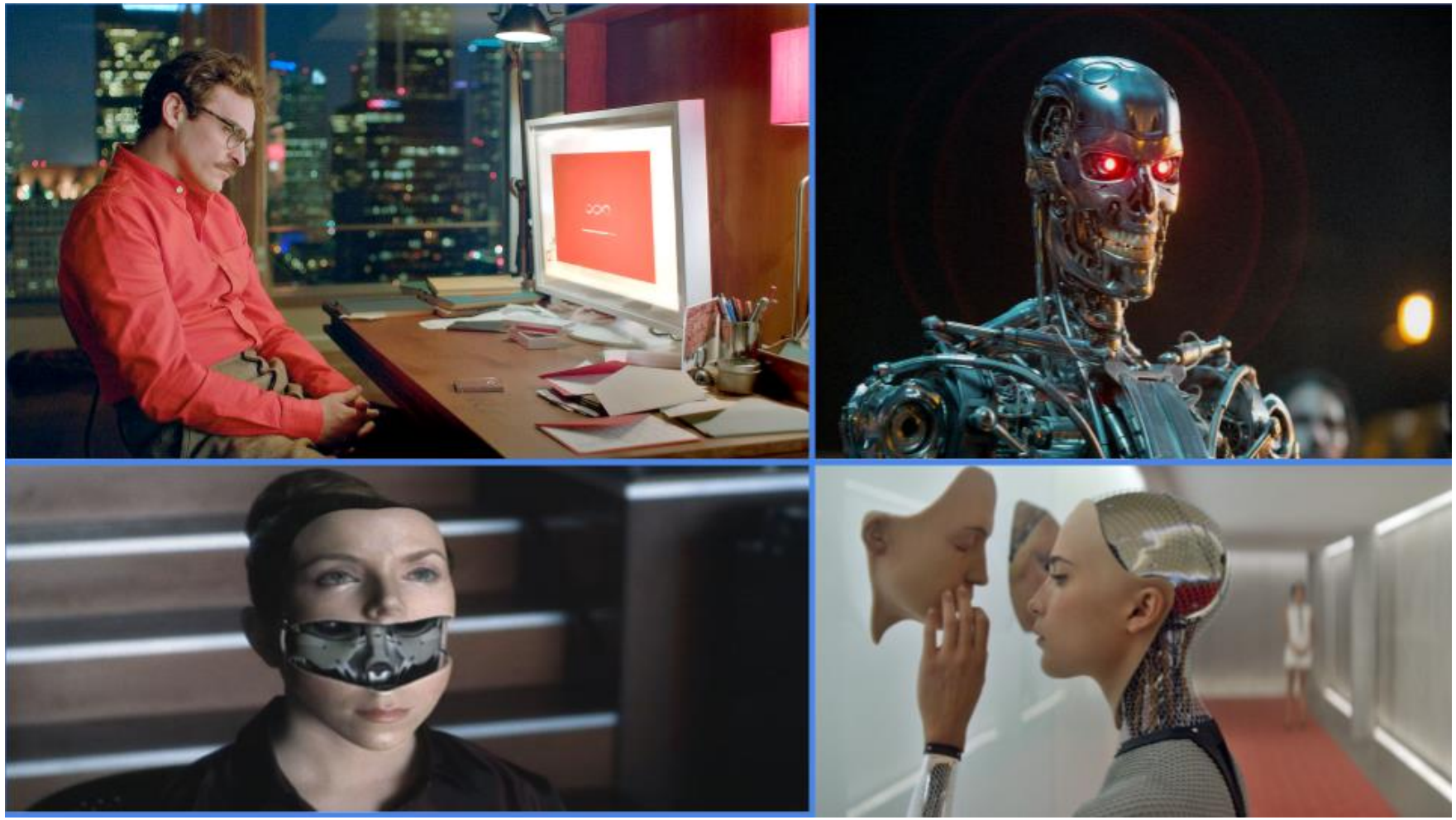
AI lecture 3-1

intro AI

solving by searching

Hanzehogeschool Groningen
University of Applied Sciences



Contents

- **Constraint Satisfaction Problems**
- backtracking search
- variable ordering
- arc-consistency
- domain splitting
- Sudoku
- a performance trade-off

Variable-based models

- in state-based models, a solutions is a sequence of steps/actions how to go from A to B
- however, in many applications, the order in which things are done is not important; only the goal is important, not the path
- solution = an assignment of values to variables that matches all constraints
- Constraint Satisfaction Problems: hard constraints (Sudoku, Zebra puzzle, crossword puzzles)
- Bayesian networks: soft constraints, variables and their conditional dependencies (e.g. diseases and symptoms)

Some CSP applications

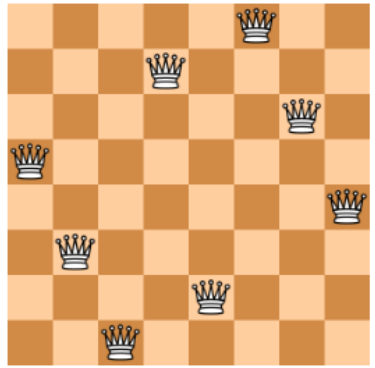
- all kinds of scheduling problems
  - school scheduling
  - transportation scheduling
  - factory scheduling
- circuit layout (assigning components to locations)
- hardware configuration
- fault diagnosis
- ...and many more

- many practical CSP-problems are NP-complete

Some CSP puzzles

HOUSE ORDER	1	2	3	4	5
COLOR	YELLOW	BLUE	IVORY	GREEN	RED
NATIONALITY	NORWEGIAN	UKRANIAN			ENGLISHMAN
PET		HORSE			SNAILS
DRINK		TEA	MILK	COFFEE	
CIGARETTES	KOOLS				OLD GOLD

- Zebra puzzle (Einstein's Puzzle)
  - There are five houses.
  - The Englishman lives in the red house...etc.
- Sudoku
- crossword puzzle
- Mastermind
- N-queens puzzle
  - placing N queens so that no two queens threaten each other

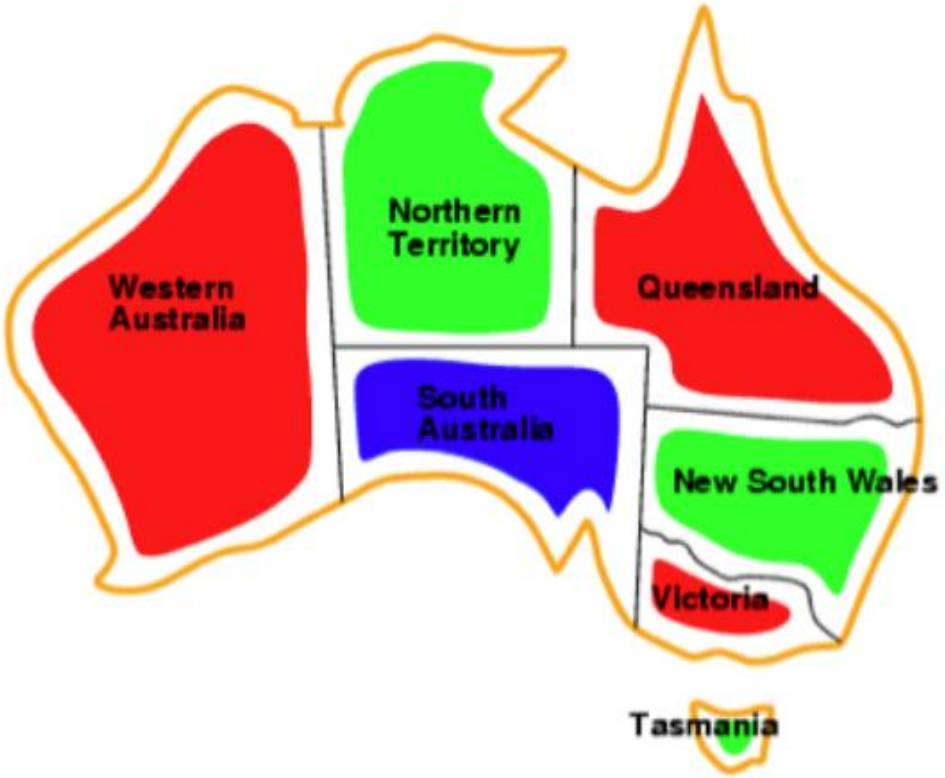


CSP's

- a set of variables
- a domain for each variable (= possible values)
- a set of constraints
  - unary (trivial, e.g. not empty)
  - binary (example: X <> Y, easy to check)
  - N-ary (expensive to check, but can be broken down to binaries)
- a goal: find any solution, find all solutions
- solution = an assignment of values to variables that matches all constraints

Classic example: map coloring

- can we color each of the 7 states (territories) using red, green and blue so that **no two neighboring** provinces have the same color?



Combination and permutation

- when the order doesn't matter, it is called a combination
  - a fruit salad is a combination of apples, grapes and bananas
- when the order does matter it is called a permutation
  - the key to the safe is 4721
  - let x, y and z be variables with domain {0, 1}, then there are 2^3=8 permutations: 000, 001, 010, 011, 100, 101, 110 and 111
- cartesian product: a cartesian product of two sets A and B is the set of all ordered pairs (a, b) where a is in A and b is in B
  - in Python: ((x,y) for x in A for y in B)

Python itertools

```
>>> import itertools
>>> L = ['a', 'b', 'c']
>>> tuple(itertools.permutations(L, 3))
(('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('c', 'a', 'b')
, ('c', 'b', 'a'))
>>> M = ['a', 'b', 'c', 'd']
>>> tuple(itertools.combinations(M, 2))
(('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd'))
>>> N = [0, 1]
>>> tuple(itertools.product(N, repeat=3))
((0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1,
1))
```

Python itertools

```
>>> import itertools
>>> domain = [1,2,3,4,5]
>>> for A,B,C,D in itertools.permutations(domain,4):
        A,B,C,D


(1, 2, 3, 4)
(1, 2, 3, 5)
(1, 2, 4, 3)
(1, 2, 4, 5)
(1, 2, 5, 3)
(1, 2, 5, 4)
(1, 3, 2, 4)
(1, 3, 2, 5)
(1, 3, 4, 2)
(1, 3, 4, 5)
(1, 3, 5, 2)
```

```
(5, 3, 2, 4)
(5, 3, 4, 1)
(5, 3, 4, 2)
(5, 4, 1, 2)
(5, 4, 1, 3)
(5, 4, 2, 1)
(5, 4, 2, 3)
(5, 4, 3, 1)
(5, 4, 3, 2)
>>>
```

How many?

- permutations
  - let S = {a, b, c, d, e}
  - how many orderings are possible?
  - 5 * 4 * 3 * 2 * 1 = 5!

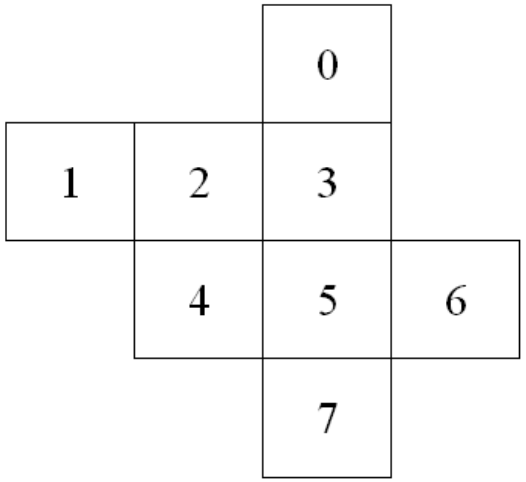
```
>>> from math import factorial
>>> factorial(5)
120
>>>
```

- combinations
  - let S = {a, b, c, d, e}
  - how many selections of 2 items are possible (order of selection doesn't matter)?
    - in other words: how many 2-combinations?
  - 5 choose 2 = 5! / 2! * (5-2)!
  - EN : n choose k or nCk
  - NL: n boven k

```
>>> from math import comb
>>> comb(5, 2)
10
```

A card puzzle

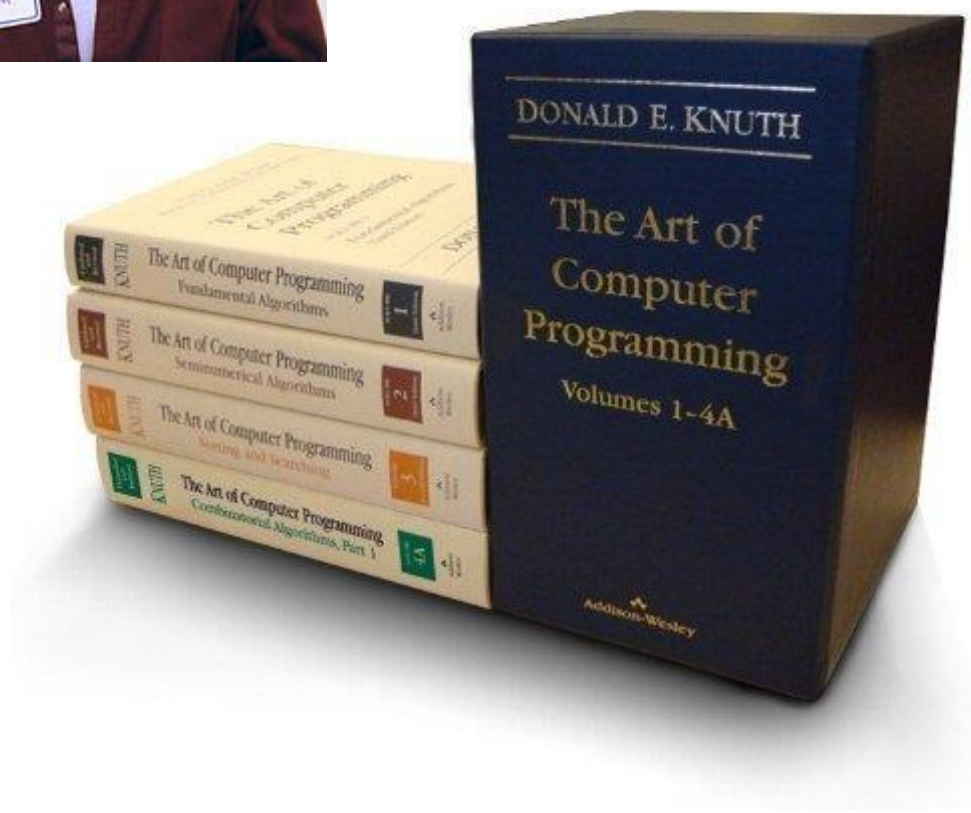
- eight cards are placed in the diagram shown below, one in each numbered cell
- place 2 Aces, 2 Kings, 2 Queens and 2 Jacks in the spaces below so that:
  - every Ace borders a King
  - every King borders a Queen
  - every Queen borders a Jack
  - no Ace borders a Queen
  - no two of the same cards border each other
  - border means: horizontal or vertical



A card puzzle

- what are the variables?
- what are the domains?
- how many assignment possible?

- test all assignments (permutations): is there a solution that matches all constraints?



Volume 4A – Combinatorial Algorithms, Part 1 [edit]

- Chapter 7 – Combinatorial Searching
  - 7.1. Zeros and Ones
    - 7.1.1. Boolean Basics
    - 7.1.2. Boolean Evaluation
    - 7.1.3. Bitwise Tricks and Techniques
    - 7.1.4. Binary Decision Diagrams
  - 7.2. Generating All Possibilities
    - 7.2.1. Generating Basic Combinatorial Patterns
      - 7.2.1.1. Generating all n-tuples
      - 7.2.1.2. Generating all permutations
      - 7.2.1.3. Generating all combinations
      - 7.2.1.4. Generating all partitions
      - 7.2.1.5. Generating all set partitions
      - 7.2.1.6. Generating all trees
      - 7.2.1.7. History and further references

Contents

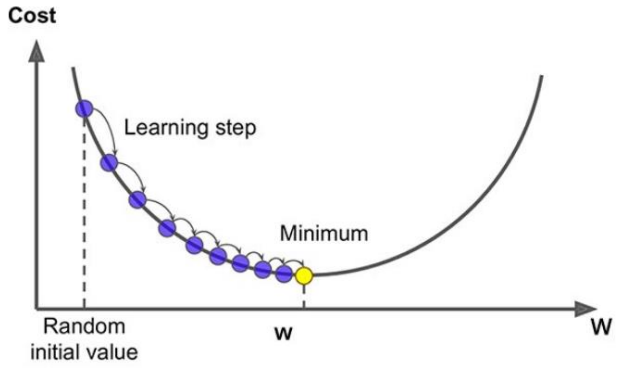
- Constraint Satisfaction Problems
- **backtracking search**
- variable ordering
- arc-consistency
- domain splitting
- Sudoku
- a performance trade-off

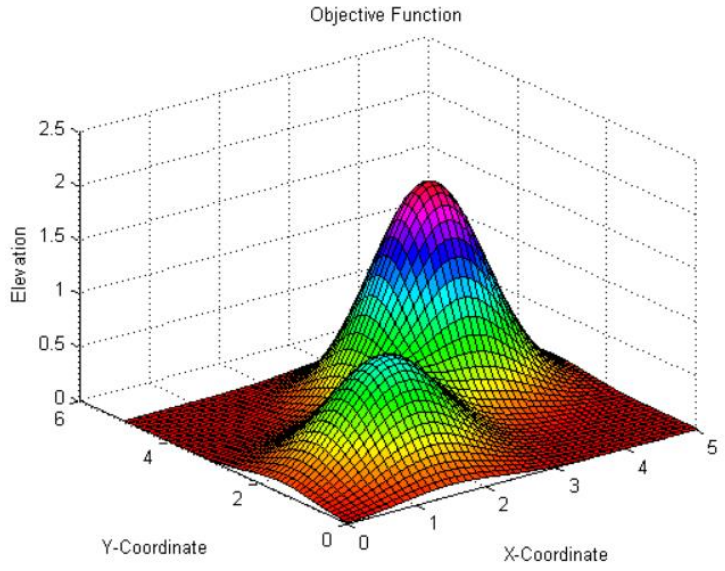
Possible strategies

- generate-and-test (aka brute force)
- backtracking search (DFS), each variable is seen as a layer in a tree
- improving backtracking by ordering of variables
- improving backtracking with consistency checking:
  - domain consistency: looking at a *single* variable, cross off values inconsistent with the constraints
  - arc-consistency (aka constraint propagation): looking at *pairs* of variables, cross off values inconsistent with the constraints

Possible strategies

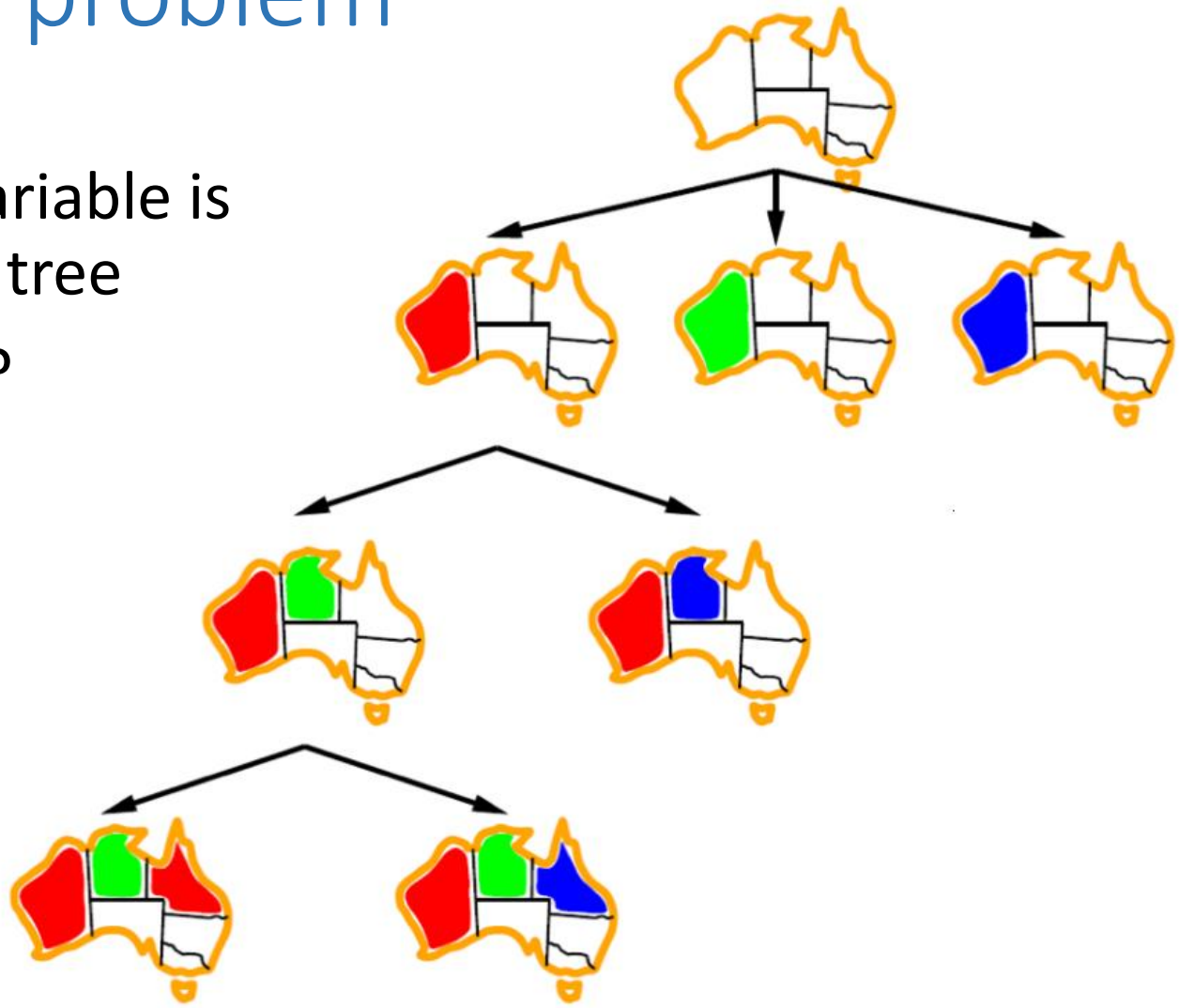
- local search
  - improve a single option until you can’t make it better
  - fast, but incomplete and sub-optimal
- examples:
  - iterative best improvement, aka hill-climbing
  - stochastic local search, such as simulated annealing
- local search will not be covered here...





Map coloring problem

- 6 variables, each variable is seen as a layer in a tree
- when to backtrack?



Backtracking search

- generate-and-test will reach and try all leaf-nodes
- **solutions** are leaf nodes: all variables have a value
- sometimes a partial assignment can be tested: if test fails, we **backtrack** (= prune the search tree)
- note: root node does not assign a value to any variables

Backtracking search

- order variables, say X0, X1, .. , Xn
- repeat:
  - select next variable, start with X0
  - assign a value to the variable (a ‘candidate’)
  - test the (partial) assignment
  - if assignment doesn't meet constraints then we backtrack

Backtracking search

```
# dict represents a tree: a dictionary variable:value
def solve(dict, untested_keys):
   # if all constraints satisfied
   if all tests pass:
      if all keys have a value
         # found a solution!
         print solution
         return True
      # select a var that has not yet been assigned
      key = untested_keys[0]
      # try all values
      for all values v in domain:
         # assign value to variable
         dict[key] = v
         # recursive call
         if solve(dict, untested_keys[1:]):
            return True
    # didn't find a solution, go back up, undo assignment
    dict[key] = empty_value
    return False
```

Contents

- Constraint Satisfaction Problems
- backtracking search
- **variable ordering**
- arc-consistency
- domain splitting
- Sudoku
- a performance trade-off

Variable ordering

- always choose first the variable with the smallest domain
  - a general test strategy: try to fail fast
  - meaning: prune the branch as early as possible
- note: apart from ordering variables, we also have to order the values in the domains (= implementation)

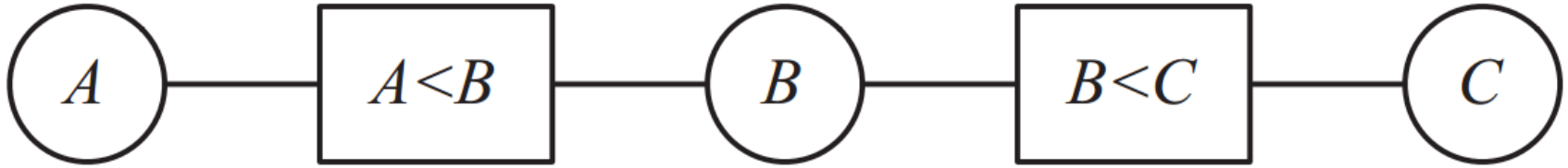
Contents

- Constraint Satisfaction Problems
- backtracking search
- variable ordering
- **arc-consistency**
- domain splitting
- Sudoku
- a performance trade-off

A simple CSP

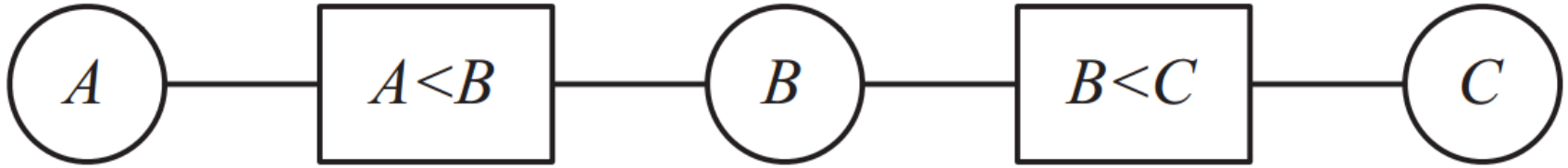
- variables A, B and C each with a domain {1, 2, 3, 4}
- arc consistency always evaluates two variables
  - example: if A=2 but then there is no value C possible, then A=2 cannot be part of a solution
- two constraints:
  - A < B
  - B < C
- how many solutions?

Arc-consistency



- variables are circles
- constraints in rectangles
- draw an arc for every constraint and every variable:
- 4 arcs:
  - `A < B: arc(A, A < B) + arc(B, A < B)`
  - `B < C: arc(B, B < C) + arc(C, B < C)`

Example arc-consistency

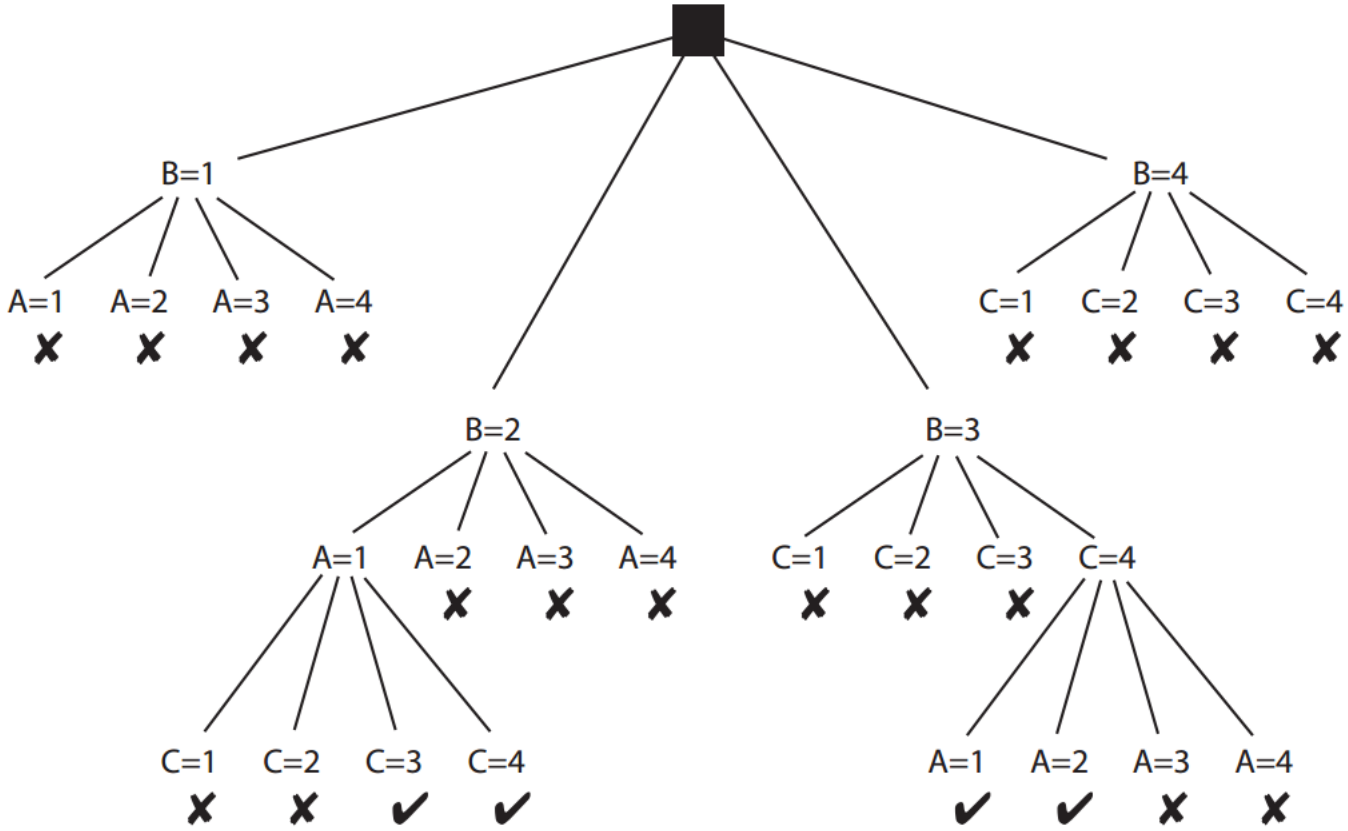


- give A a value and consider pairs (A, B)
- next consider pairs (B,C)

A	B	C
1	1	1
2	2	2
3	3	3
4	4	4

Assignments represented as a tree

- give B a value and consider pairs (B,A)
- next consider pairs (A,C)



from Poole & Mackworth ch 4

Arc-consistency

- a recursive process: if a value is removed from a domain, the consequence can be that values from other domains have to be removed as well
- sometimes only applying arc-consistency will result in a unique solution (= all domains have a single value left)

Arc-consistency

- arc-consistency can be applied in the backtracking search
  - assign a value x to a first variable X (a candidate)
  - assign a value y to a second variable Y
  - apply arc-consistency to Y, meaning: remove values from domain of Y that are not possible
  - if domain of Y becomes empty, the search fails, backtrack and try another assignment for X
  - if we find a pair (x,y) that is ok, then we continue with third variable Z
- this is used in solving Sudoku

Backtracking with ac

```
# dict represents a tree: a dictionary variable:value
def solve(dict, untested_keys):
   # if all constraints satisfied
   if all tests pass:
      if all keys have a value
         # found a solution!
         print solution
      # select a var that has not yet been assigned
      key = untested_keys[0]
      # try all values
      for all values v in domain:
         # assign value to variable
         dict[key] = v
         if make_arc_concistent(dict):
               # recursive call
               solve(dict, untested_keys[1:])
    # didn't find a solution, go back up
    # undo assignment
    dict[key] = empty_value
    return
```

Arc-consistency

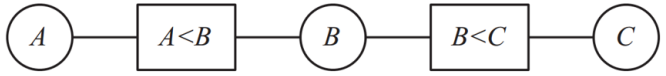
```
function ac(X, Y):
    changed = False
    for x in X.domain:
        if no y in Y.domain satisfies constraint (X, Y):
            delete x from X.domain
            changed = True
    return changed

function make_arc_consistent(dict):
    queue = all arcs or binary constraints
    while queue non-empty:
        # pop next arc from queue
        (X, Y) = DEQUEUE(queue)
        if ac(X, Y): # if values were removed from X.domain
            if size of X.domain == 0:
                return false
            # removing value from X.domain may introduce
            # new constraints (X, Z)
            for each Z in X.neighbors - {Y}:
                ENQUEUE(queue, (Z, X))
    return True
```

Contents

- Constraint Satisfaction Problems
- backtracking search
- variable ordering
- arc-consistency
- **domain splitting**
- Sudoku
- a performance trade-off

Domain splitting

- split a domain into disjoint sets
- solve the sub-problems (may use arc-consistency)
- combine the solutions



- example
  - after applying arc-consistency $D_A = \{1,2\}$, $D_B = \{2,3\}$, $D_C = \{3, 4\}$
  - suppose we start by splitting domain B into {2} and {3}
  - B=2: A!=2, further splitting on C gives 2 solutions
  - B=3: C!=3, further splitting on A gives 2 solutions

A	B	C
1	1	1
2	2	2
3	3	3
4	4	4

Contents

- Constraint Satisfaction Problems
- backtracking search
- variable ordering
- arc-consistency
- domain splitting
- **Sudoku**
- a performance trade-off

Sudoku

								8
2		1						
			9			6	7	3
				5				
9		7			3		4	1
				7				
	5					2	8	9
					5			
3	6			4				

7	9	6	5	3	4	1	2	8
2	3	1	7	8	6	9	5	4
5	4	8	9	1	2	6	7	3
6	2	3	4	5	1	8	9	7
9	8	7	6	2	3	5	4	1
4	1	5	8	7	9	3	6	2
1	5	4	3	6	7	2	8	9
8	7	2	1	9	5	4	3	6
3	6	9	2	4	8	7	1	5

Sudoku

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- terminology:
  - cells
  - rows, columns and boxes
  - peers: the cell's that are in the same row, column or box (green)
  - domain: all possible values a cell can have
  - clue: a given number
  - candidate: a possible number which we could place in a cell
- four constraints:
  - every cell has exact one value
  - in every box: a value may only appear once
  - in every row: a value may only appear once
  - in every column: a value may only appear once

Sudoku & arc-consistency

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- what does arc-consistency mean?
- if a cell has a *single* value (say D4 = 7)
  - it's a clue, or
  - it's a candidate-value during backtracking search
- the value (say 7) can be removed from the domains of its peers
- this is a recursive process

Repeating arc-consistency (2 x 2)

{1,2,4}	{2,4}
{2,4}	3

domain with least values > 1, let's try 2 from {2,4}

{1,4}	2
4	3

domain with least values > 1, let's try 4 from {1,4}

1	2
4	3

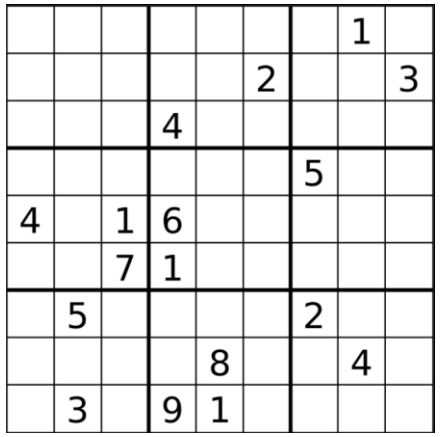
found a solution!

```
# grid is a dictionary cell:values
# all cells have values 1..9, except cells with clues
def solve(grid):
   if all cells have exactly one value:
      # found a solution!
      print solution
      return True
   select a cell c with minimum number of values > 1
   # try them all
   for all possible values v in grid[c]:
      # if assignment is possible
      if no_conflicts(grid, c, v):
         new_grid = grid.copy()
         # assign value to variable
         new_grid[c] = v
         if make_arc_concistent(new_grid, c, v)
            if solve(new_grid): # recursive call
               # found a solution!
               return True
    # didn't find a solution, go back up
    return False
```

```
def make_arc_concistent(grid, c, v):
   for all peers of c:
      if value v in peer:
         if peer has <= 1 value:
            # conflict, no solution on this ‘path’
            return False
         else:
            remove v from peer
   if grid changed:
      for all peers of c2 !=c with exactly one value v2
         # removing a value v may introduce new constraints
         if not make_arc_consistent(grid, c2, v2):
            return False
   return True
```

Math and Sudoku

- for a 4x4 Sudoku there are 288 possible grids
- for a 9x9 this is roughly 6.7×10^{21}
- for a 4x4 to have a single solution, the minimum number of clues is 4, of which there are only 13 non-equivalent puzzles
- for a 9x9 the minimum number of clues for a single solution appears to be 17 (McGuire, Tugemann, Civario)
- see
  - Wiki "Mathematics of Sudoku"
  - [math-cornell-mahmood]



Contents

- Constraint Satisfaction Problems
- backtracking search
- variable ordering
- arc-consistency
- domain splitting
- Sudoku
- **a performance trade-off**

A performance trade-off

- grade-off between quality of consistency checking vs. trying many paths that fail
  - consistency checking (and propagating constraints) takes time
  - trying (all) permutations takes time
- this is a similar trade-off as with A* (quality of heuristic function vs. trying many paths that fail)