Spezielle Themen der Künstlichen Intelligenz

2. Termin: Constraint Satisfaction

Dr. Stefan Kopp
Center of Excellence „Cognitive Interaction Technology“
AG Sociable Agents

Recall: Best-first search

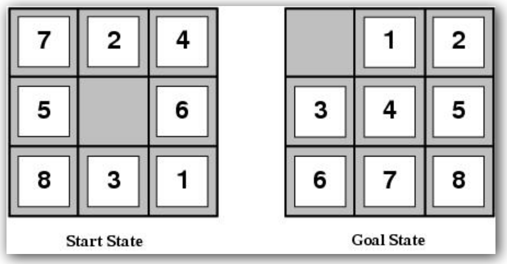
- Best-first search = graph-search with node expansion in order of cost heuristic h(n)
- *Greedy* best-first search = expand node with minimal h(n)
  - not optimal but often efficient
- A* search = expand node with minimal f = g+h
  - complete & optimal: admissible (tree-search) or consistent (graph-search) h
- SMA* (Simplified Memory-bounded A*)
  - drop worst leaf node when memory is full, backs up f-value to its parent for later re-expansion
- RBFS (Recursive Best-First Search) ~ recursive DF search with...
  - keep track of f-values of alternative paths, backtrack if f > alternative f
  - upon backtracking, change f-value of node to best f-value of its children, to decide later whether to re-expand

Performance depends crucially on the quality of the heuristics!

Heuristic functions

Example: 8-puzzle

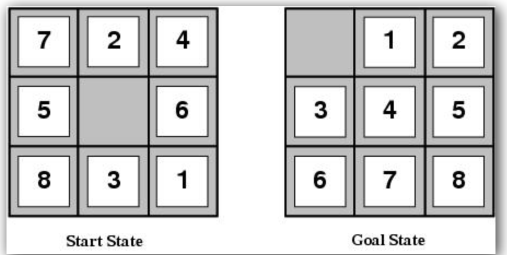
- avgerage solution cost is about 22 steps (branching factor ~3)
- exhaustive search to depth 22: 322 ~ 3.1 x 10^{10} states
- a good heuristic function is needed to reduce the search process



Heuristic functions

Two commonly used heuristics

- h1 = number of misplaced tiles ➔ h1(start)=8
- h2 = manhattan distance = sum of distances of tiles from their goal positions ➔ h2(start)=3+1+2+2+2+3+3+2=18



True solution cost = 26

Heuristic quality and dominance

Example: 1200 random 8-puzzle problems with solution lengths from 2 to 24

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

If h2(n)>=h1(n) for all n (and both admissible), then h2 is said to *dominate* h1 and is better for search!

How good is a heuristic?

Effective branching factor b*

- N = #nodes generated by A* in total, d solution depth
- b* = branching factor that a *uniform* tree of depth d would have in order to contain N+1 nodes

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- measure is fairly constant for sufficiently hard problems
- measurement of b* on small problems can provide a good guide to the heuristic's overall usefulness (a good value is 1)

Inventing admissible heuristics

from an exact solution of a relaxed version of the problem

- *Example*: relaxed 8-puzzle for h1: a tile can move anywhere
- never greater than the optimal solution cost of the real problem
  - „ABSolver“ automatically found heuristic for the rubic cube

from the solution cost of a subproblem of the problem

- lower bound on the cost of the real problem

from a database of exact solutions for possible subproblem instances

- construct complete heuristic from the patterns in the DB
- can use disjoint databases for different subproblems, when solutions don't interfere (works only for some problems)

Constraint satisfaction problems

Dr. Stefan Kopp
Center of Excellence „Cognitive Interaction Technology“
AG Sociable Agents

Constraint satisfaction problems

Standard search problem:

- state is a "black box" – any data structure that supports successor function, heuristic function, and goal test

Constraint satisfaction problem (CSP):

- structured state = variables X_i, values from domain D_i
- goal test = set of constraints specifying allowable combinations of values for subsets of variables
- solution = complete assignment of values to (all) variables that passes the goal test (consistent or legal)

Enables useful standard algorithms (for all CSPs) with effective, generic heuristics, without domain expertise

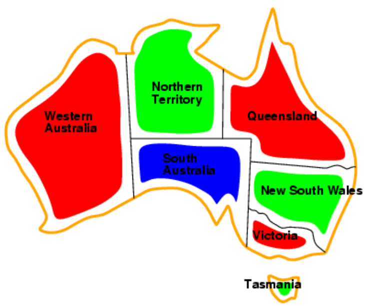
Example: map coloring

„Dreifarbenproblem“
np-complete!

Variables: WA, NT, Q, NSW, V, SA, T
Domains: Di = {red,green,blue}
Constraints: adjacent regions must have different colors

e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

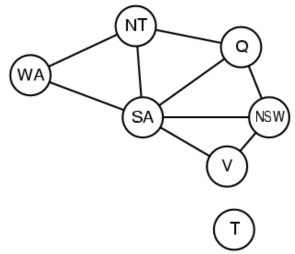
Example: map coloring



Solutions are complete and consistent assignments

- Example: WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

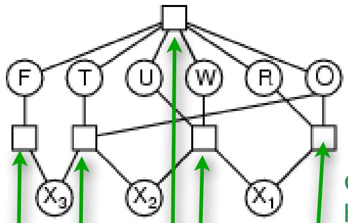
Constraint graph



Binary constraints: each constraint relates two variables
Constraint graph: nodes are variables, arcs are constraints

Higher-order constraints

```
   T W O
+  T W O
 F O U R
```



Constraint hypergraph

Variables: F T U W R O X_1 X_2 X_3
Domains: {0,1,2,3,4,5,6,7,8,9}
Constraints: *Alldiff* (F,T,U,W,R,O)

$O + O = R + 10 \cdot X_1$
$X_1 + W + W = U + 10 \cdot X_2$
$X_2 + T + T = O + 10 \cdot X_3$
X3 = F, T ≠ 0, F ≠ 0

Variables in CSPs

Discrete variables

- finite domains: *n* variables, domain size *d* ➔ $O(d^n)$ assignments
  - e.g. Boolean CSPs (3SAT): exponential time, NP-complete
- infinite domains: integers, strings, etc.
  - e.g., job scheduling, variables: 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
- must obey a variety of constraints
  - linear constraints (forming a convex region) solvable in polynomial time by linear programming methods

Solving CSPs

Standard search algorithm can be applied directly:

- States: defined by the values assigned so far
- Initial state: the empty assignment { }
- Successor function: assign value to variable without conflict
  - fail, if no legal assignments possible
- Goal test: the current assignment is complete
- Path cost: constant step cost

Every solution with *n* variables (domain size *d*) appears at depth *n*

- branching factor $b = (n\text{-}i)d$ at depth i,
- ➔ tree with $n! \cdot d^n$ leaves even though only d^n assignments!!
  - commutativity is ignored: same combinations are explored multiple times along different paths (in different order)

Backtracking search

Variable assignments are commutative!

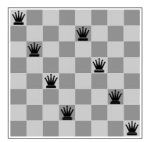
- [WA = red *then* NT = green] ~ [NT = green *then* WA = red]

Only need to consider assignments to a single variable at each node

- b = *d* and there are d^n leaves

Backtracking search = depth-first search for CSPs with single-variable assignments

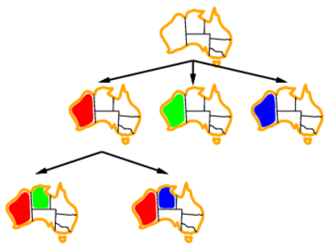
- basic uninformed algorithm for CSPs
- example: can solve „n-queens" for up to $n \approx 25$

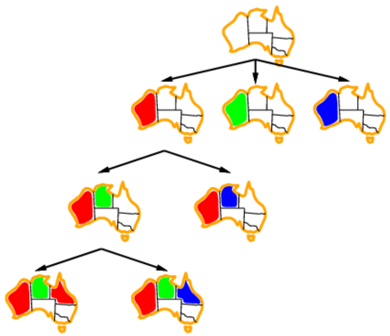


Backtracking search

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

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** 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-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var =value}* from *assignment*
    return *failure*





Each level explores different assignements to a *single variable*

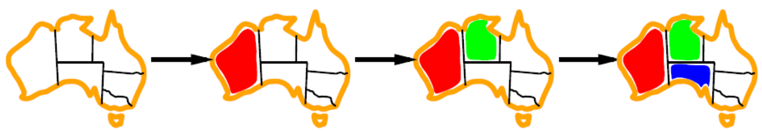
Improving backtracking

Standard search improved by incorporating domain-specific knowledge (heuristics)

CSPs can be improved by using general-purpose methods to address the questions:

- Which variable should be assigned next?
- In what order should its values be tried?
- What implications (i.e. restrictions) has an assignment for other possible variable assignments?
- Can we detect inevitable failure (inconsistent assignments) early?
- Can we avoid repeating a failing path?

Minimum remaining values heuristic (MRV)

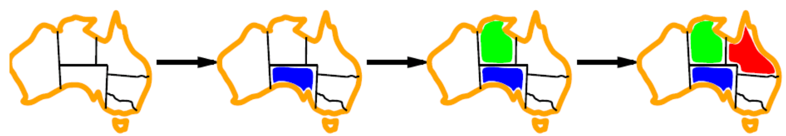


var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

Also known as most constrained variable heuristic

- *Rule*: choose variable with the fewest legal moves left
- Which variable shall we try first?

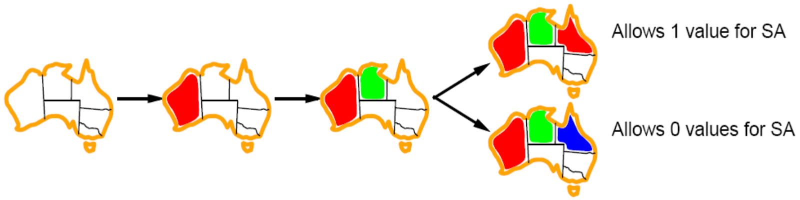
Degree heuristic



Degree heuristic

- *Rule*: select variable that is involved in the largest number of constraints on other unassigned variables
- attempts to reduce future branching factors, very useful as a „tie breaker"
- In what order should its values be tried?

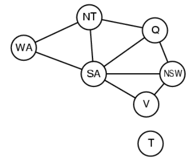
Least constraining value heuristic



Least constraining value heuristic

- *Rule:* given a variable choose the least constraining value i.e. the one that leaves the maximum flexibility for subsequent variable assignments.

Forward checking

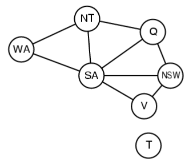


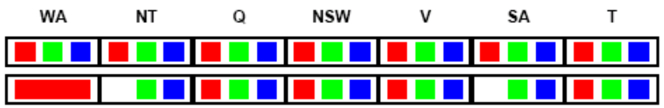
Can we detect inevitable failure early (to reduce search space), and avoid it later?

Forward checking:

- on assigning X, check every connected variable Y
- remove all values from domain of Y inconsistent with X
- terminate search when any variable has no legal moves left

Forward checking



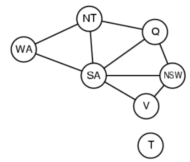


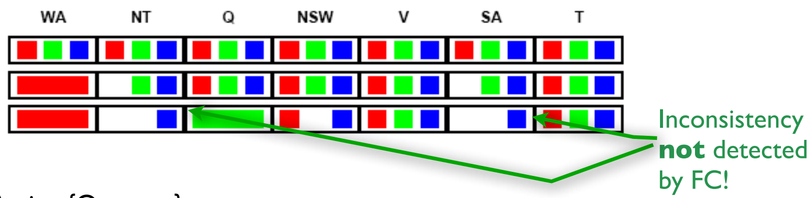
Assign {WA=red}

Effects variables connected by constraints with WA

- **NT can no longer be red**
- **SA can no longer be red**

Forward checking





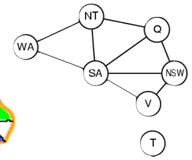
Assign {Q=green}

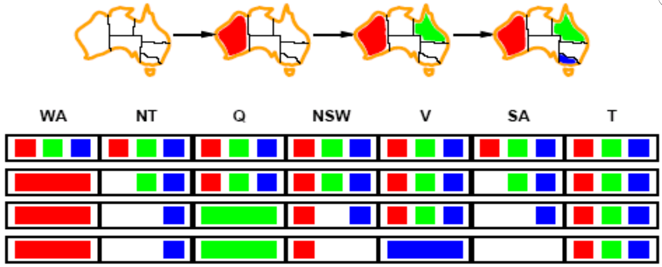
Effects variables connected by constraints with Q

- **NT can no longer be green**
- **NSW can no longer be green**
- **SA can no longer be green**

MRV heuristic will automatically select NT and SA next, why?

Forward checking





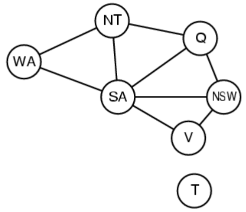
If V is assigned blue

Effects variables connected by constraints with V

- **NSW can no longer be blue**
- **SA is empty**

Now, FC has detected that the partial assignment is inconsistent with the constraints, and backtracking will occur

Constraint propagation



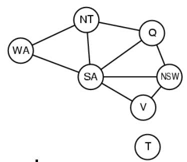
Better approach: forward checking combined with heuristics

- more efficient and less error-prone than either approach alone
- forward checking does not provide (early enough) detection of all failures

\+ constraint propagation

- implications of a constraint on one variable must be *repeatedly* propagated onto other connected variables

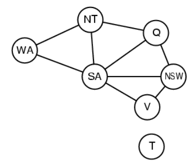
Arc consistency

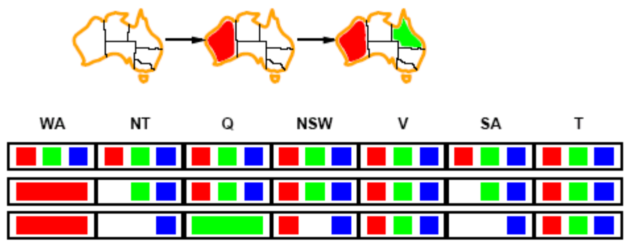


Fast method for constraint propagation:

- constraints considered as directed arcs in constraint graph
- $X \rightarrow Y$ is (arc) consistent iff for <u>every</u> value x of X there is some allowed value y of Y
- if y changes, keep $X \rightarrow Y$ consistent by setting domain of X
- iterative procedure to continuously re-check all constraints on neighbouring variables in the constraint graph
- backtrack, if any variable‘s domain empty (arc inconsistent)
- Possible outcomes (when all arcs consitent)
  - one domain is empty - no solution
  - each domains has single value - unique solution
  - some domains have more than on value - may or may not be a solution ➔ search and run again

Arc consistency

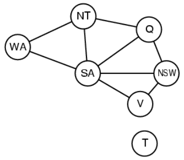


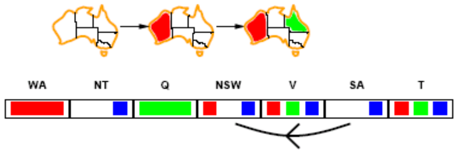


Example: After setting $Q=green$ and forward checking (NT, SA, NSW)

- having produced inconsistency between NT and SA

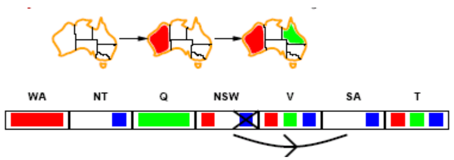
Arc consistency





$SA \rightarrow NSW$ is consistent iff

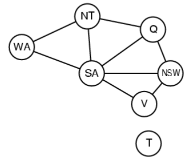
- SA=*blue* and NSW=*red*

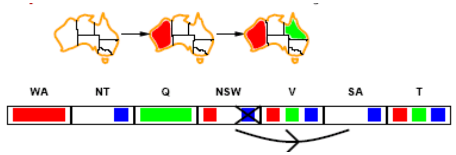


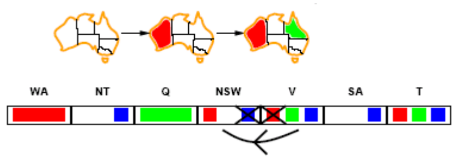
$NSW \rightarrow SA$ is consistent iff

- NSW=*red* and SA=*blue*
- NSW=*blue* and SA=*???*
  ➔ arc inconsistent
- Remove *blue* from domain of NSW

Arc consistency



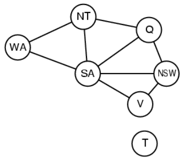


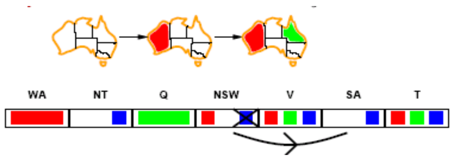


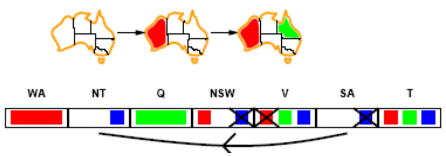
$V \rightarrow NSW$ is consistent iff

- V=*blue* and NSW=*red*
- V=*green* and NSW=*red*
- V=*red* and NSW=*red ???*
  ➔ arc inconsistent
- Remove *red* from domain of V

Arc consistency







$SA \rightarrow NT$ is consistent iff

- SA=*blue* and NT=*blue???*
  ➔ arc inconsistent
- Remove *blue* from domain of SA
- empty domain
  ➔ backtrack

➔ arc consistency detects failure earlier than Forward Checking, can be run as a preprocessor or after each assignment

- must run repeatedly until no inconsistency remains

AC-3: Arc consistency algorithm

function AC-3(*csp*) **return** the CSP, possibly with reduced domains
    inputs: *csp*, a binary csp with variables $\{X_1, X_2, \ldots, X_n\}$
    local variables: *queue*, a queue of arcs to check, initially all arcs in *csp*
    while queue is not empty **do**
        $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
        if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
            for each X_k **in** NEIGHBORS[X_i] **do**
                add (X_i, X_j) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **return** *true* iff we 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 the constraints between X_i and X_j
        then delete x from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
    return *removed*

K-consistency

Arc consistency (AC-3)

- runs in $O(n^2d^3)$: at most $O(n^2)$ arcs (=binary constraints), each arc inserted only d times, consistency check of an arc in $O(d^3)$
- but does *not* detect all inconsistencies
  - *{WA=red, NSW=red}* inconsistent but not found

stronger forms of propagation can be defined using the notion of k-consistency

- a CSP is k-consistent if for any consistent assignment to any subset of *k-1* variables, a consistent value can *always* be assigned to any *k-th* variable
- Examples:
  - 1-consistency or node-consistency
  - 2-consistency or arc-consistency
  - 3-consistency or path-consistency

Further improvements

Checking special constraints

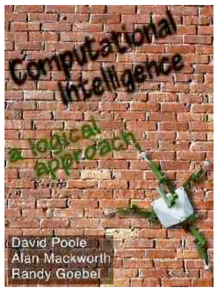
- e.g. *Alldiff*(…) constraint
- e.g. *Atmost*(…) constraint (resource constraint)
- Bounds propagation useful in larger value domains

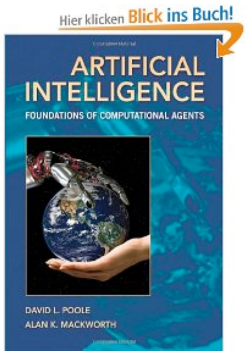
Intelligent backtracking

- standard form is chronological backtracking, i.e. try different value for most recent preceding variable
- more intelligent: backtrack to conflict set for variable *X*
  - set of variables that caused the failure, or set of previously assigned variables that are connected to *X* by constraints
  - „backjumping“: to most recent element of the conflict set
  - forward checking can be used to determine conflict set

Examples

www.aispace.org





(Cambridge Univ. Press 2010)
fully available online!

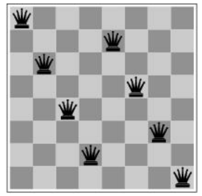
Local search

Previously: systematic exploration of search space

- often, path to goal is solution to the problem

Yet, for CSPs the path is irrelevant

- E.g 8-queens



Different algorithms can be used

- Local search

Local search and optimization

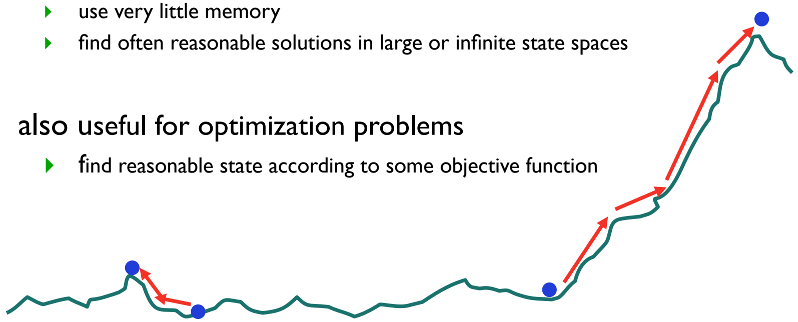
Local search = use only single current state (local knowledge) and move to neighboring states

advantages:

- use very little memory
- find often reasonable solutions in large or infinite state spaces

also useful for optimization problems

- find reasonable state according to some objective function



Important local search techniques

Random walk: choose fully randomly from among neighbors

Hill-climbing aka. gradient descent/ascent aka. greedy local search

- stochastic: choose randomly from among uphill moves
- 1st choice: create successors randomly until better found
- random restart: reset variables randomly at regular intervals

Simulated Annealing

- allow random guesses (even when bad moves), with decreasing size & frequency

Local-Beam Search

- *k* parallel search threads that pass information about the local milieu among them

Genetic algorithms

Local search for CSP

use complete-state representation

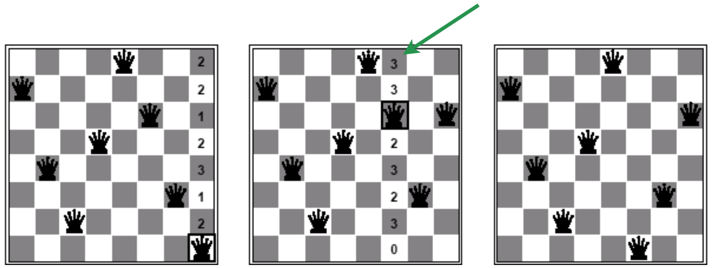
- initial state assigns a value to every variable
- allow states with unsatisfied constraints
- operators *reassign* variables

questions during CSP search: which variable to change how?

- randomly select any conflicted variable
- select a new value that results in a minimum number of conflicts with the other variables („min-conflicts heuristic“)

Min-conflicts example

number of conflicts



Two-step solution for 8-queens problem (with reasonable initial state)

- variable selection: at each stage a queen is chosen for reassignment in its column
- value selection: the algorithm moves the queen to the min-conflict square, breaking ties randomly

Comparision of CSP algorithms

Problem	Back-tracking	BT-MRV	FC	FC+ MRV	Min-conflicts
USA coloring	>1.000K	>1.000K	2K	60	64
n-Queens (2-50)	>40.000K	13.500K	>40.000K	817K	4K
Zebra puzzle	3.859K	1K	35K	0.5K	2K

Bottom line: local search suprisingly good, can even be used online!

- n-queens.: roughly independent of problem size, solves million-queens in ~50 steps (because solutions densely distributed)
- Hubble: schedules a week in ~10 min., instead of 3 weeks

Examples

www.aispace.org