Knowledge Representation and Reasoning: Constraint Satisfaction and Local Search

Chapter 6

Constraint Satisfaction Problems

♢ Binary constraint network $\gamma = \langle V, D, C \rangle$
- — V a finite set of variables $v_1, \ldots, v_n$
- — D a set of [finite] sets $D_{v_1}, \ldots, D_{v_n}$
- — C a set of binary relations $\{C_{u,v} \mid u, v \in V, u \neq v\}$
  $C_{u,v} \subseteq D_u \times D_v$

Outline of the lecture

- ♢ Recall [optimal] constraint solving
- ♢ Local Search in general / different style of search
- ♢ Large Neighbourhood Search in particular
- ♢ Constraint modelling
- ♢ Summary

Recall

- ♢ CSP may be solved for satisfiability (any solution will do)
- ♢ May instead require optimality (best solution)
- ♢ Objective function (i.e. cost) measures badness of solutions
- ♢ FD solvers commonly use Depth-First Branch and Bound (DFBB)
  - — Use a lower bound L on the objective and an upper bound B
  - — Backtrack whenever $L \geq B$
  - — Revise B whenever a solution is found
- ♢ DFBB fits well with backtracking CSP solution methods
- ♢ Gives a sequence of monotonically improving solutions

maximize ? / minimize ? — depends on the problem

// cost of violating "optional constraints"

can be Depth-Limited

There is a problem

- ♢ Absolute optimality is often too hard to compute
- ♢ Need methods that scale up better and yield (probably) good solutions
- ♢ Enter **Local Search**
  (actually a generic name for many search techniques)
- ♢ Widely used in industrial applications

Local Search

① Randomly Assign Values
② Attempts to Improve by Changing Some of the Values
[Randomly choose some variables that violate the constraints & change the values]

- The general idea: search in the space of total assignments
- An alternative to backtracking: not so rigidly defined
  — Usually involves randomness at some point
- Example (for satisfiability):
  Start with a random assignment of values to all variables
  (Of course, this doesn't satisfy all constraints)
  Repeatedly:
  Choose a variable (random choice is good)
  Revise its value to minimise its constraint violations
  Stop when all constraints are satisfied
- Optimisation version might remember the best assignment so far, and stop when the objective function hasn't improved for a while.

Loop

① ② it's not a complete & systematic search

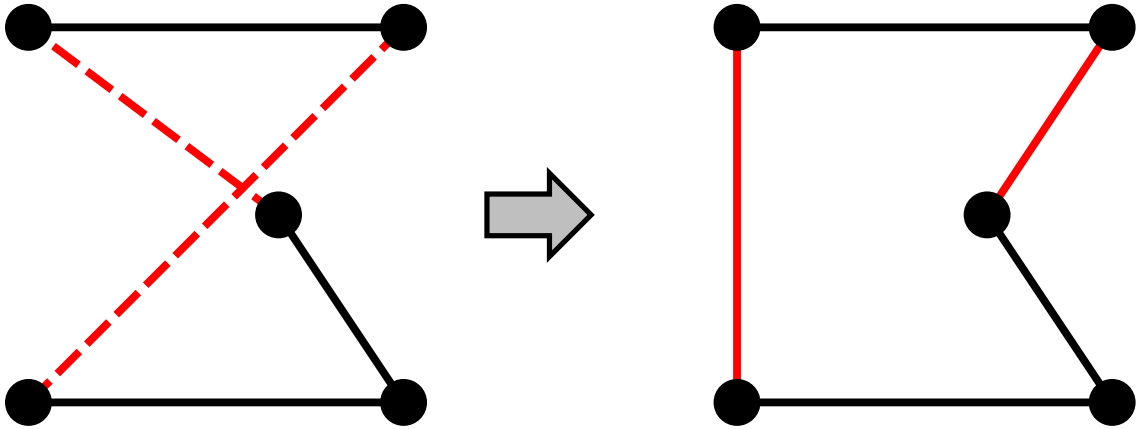
guess that's near optimal.

Iterative improvement algorithms

- ♢ Local search of this kind iteratively improves total assignments
- ♢ General idea: keep a single "current" state, try to improve it
  - — perform local moves in the neighbourhood of the current state
  - — no guarantee of completeness (may fail to find any solution)
  - — no guarantee of optimality
  - — no possibility of showing unsatisfiability
- ♢ However, method scales up better than complete search in many domains
- ♢ Small memory requirements, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges to reduce tour duration

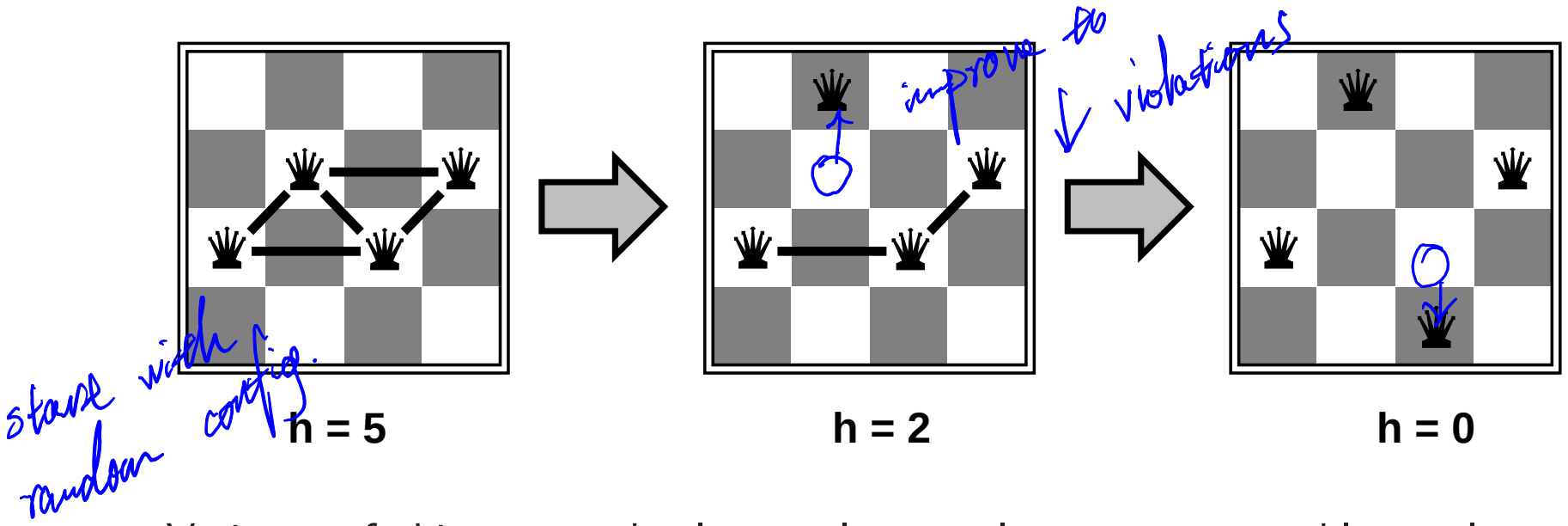


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen in its column to reduce number of conflicts



Variants of this approach almost always solve n-queens problems almost instantaneously for very large n, e.g., $n = 1$ **million**

Hill-climbing (or gradient ascent/descent)

Moves to the best neighbour

Climbing a mountain in the fog without a map: just go up!

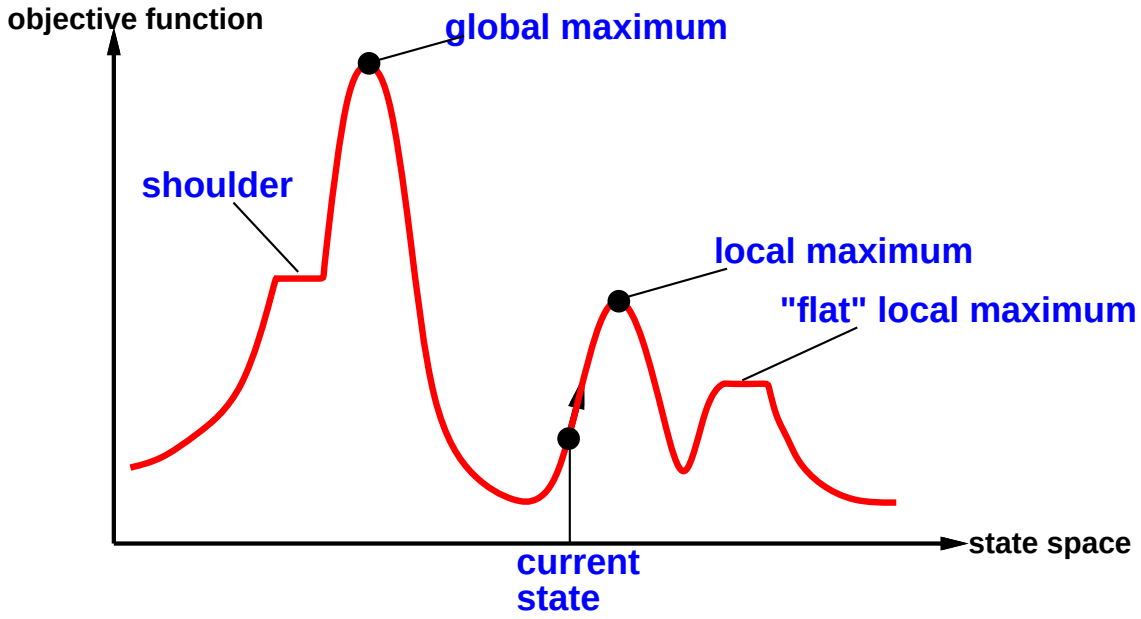
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbour, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbour ← a highest-valued successor of current
        if VALUE[neighbour] ≤ VALUE[current] then return STATE[current]
        current ← neighbour
    end
```

"we are at the peak"

Hill-climbing contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves ☺ escape from shoulders ☹ loop on flat maxima

8 queens: simple HC has 14% success rate, HC + sideway moves 94%

3 million queens: HC + random restart + sideway moves needs < 1 mn

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their badness and frequency

until converges.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    neighbour, a node
    T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    neighbour ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*neighbour*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *neighbour*
    else *current* ← *neighbour* only with probability $e^{\Delta E/T}$

Population-based methods

♢ Local beam search: maintain a population of k states; add some neighbours at each step; delete the worst ones to keep the population stable

— As usual, many variants exist

~~♢~~ Genetic (evolutionary) algorithms: define "crossover" between pairs of states in the population (compare genetics); offspring have some features of each parent; cull according to a "fitness" measure

— Much research over several decades

— Again, many variations on the general method

objective function essentially [good ones reproduce more, bad ones deleted]

// half of A & half of B, glue them together

Population-based search and simulated annealing will not be covered in this course: COMP4660 / COMP8420 covers evolutionary algorithms in some detail

A hybrid: Large Neighbourhood Search

- Search in the space of solutions rather than states
- Given a current solution:
  - Destroy part of it by forgetting the values of some variables
  - See the problem of assigning values to those variables as a CSP
  - Solve [optimally] using a complete search method such as DFBB
- Alternates destroy and repair phases
  - — Repair phase searches a neighbourhood of the current solution
  - — Neighbourhood size remains tolerable, even if the problem is huge
- Performance is sensitive to the choice of what to destroy
- The old solution is still in the neighbourhood, so there is always a next solution available, given enough search
- May search for the optimal solution in the neighbourhood, or just for any solution in the neighbourhood.

that's why hybrid.

① Random Decay
② "More Systematic Ways" / make the data structured

LNS: What to destroy?

Imagine a complex production scheduling problem, allocating jobs to machines, employees to tasks and start times to processes. A set of orders must be filled within time and capacity constraints, minimising cost...

A solution could be partially destroyed by randomly selecting decision variables and forgetting their values—this resembles random decay.

We are more likely to do the partial destruction systematically, to make intuitive sense, say by forgetting the values of all variables that mention two of the machines, or the schedules of a random selection of the employees.

The destruction parameters may be changed as the search progresses: make neighbourhoods larger or smaller, sometimes remove aset of tasks rather than a machine, etc.

Only experiments will determine what is a good choice of destruction algorithm: what to destroy, and how much (one machine? two? three? ...)

LNS: Notes

- The initial solution must come from somewhere.
- We may choose to abstract from the current solution
  - use only some decision variables, for a partial description
  - designed so the rest can be recovered by easy search
  - destroy part of the abstract solution
  - gives the complete search freedom to optimise minor aspects
- Local optima are still a problem, as with all local search
  - random restart is commonly used to escape
- There is always a tradeoff between neighbourhood size and speed
  - Large neighbourhoods increase the chance of improvement
  - but they may create hard problems for the complete search

have to experiment

Constraint Modelling

- ♢ Before any constraint solving can happen, the CSP must be defined
- ♢ Model must define V, D and C
- ♢ Explicit definition of C is painful, so use high-level description
- ♢ Hence we want to write a logical models
  - — Write constraints as formulae of first order logic
  - — Describe what would count as a solution to the problem
  - — Compiler will turn this into a low-level constraint network
  - — **Logical model is purely declarative: no algorithm!**
- ♢ Old style constraint programming: logic is implicit in the program

MiniZinc

We shall use the constraint modelling language MiniZinc [under language "Zinc"]

```
-----------------------------------------------
% N Queens Problem: place N queens on an NxN
% chessboard so that no queen attacks another

int: N;
array[1..N] of var 1..N: q;
constraint forall (x,y in 1..N where x < y) (
           q[x] != q[y] /\
           abs(q[x]-q[y]) != y-x);
solve satisfy;
-----------------------------------------------
```

MiniZinc is essentially first order logic with some syntactic sugar and basic support for arithmetic etc.

Minizinc

- ♢ MiniZinc model is completely solver-independent
- ♢ Also good to separate the "conceptual model" of the problem from data defining a specific problem instance
  — e.g. for the N queens, the data file could specify `N = 8;`
- ♢ MiniZinc gets transformed into a simple fragment "Flat Zinc" [black box here]
- ♢ Flat Zinc can be turned into input code for many solvers
  — Finite domain (FD) solvers
  — Mixed integer programming (MIP) solvers
  — SAT solvers
  — Local search solvers
- ♢ Mapping into low-level data structures is internal to the solvers
- ♢ Logical model + default mapping: the Holy Grail