Today

See Russell and Norvig, chapters 4 & 5

• Local search and optimisation
• Constraint satisfaction problems (CSPs)
• CSP examples
• Backtracking search for CSPs

Iterative improvement algorithms

In many optimization problems, path is irrelevant; the goal state itself is the solution.

Then state space = set of "complete" configurations;
find optimal configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable.

In such cases, can use iterative improvement algorithms;
keep a single "current" state, try to improve it.

Typically these algorithms run in constant space, and are suitable for online as well as offline search.

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges:

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 to reduce number of conflicts.
Hill-climbing (or gradient ascent/descent)

“Like climbing Everest in thick fog with amnesia”

function $Hill\text{-}Climbing(problem)$ returns a state that is a local maximum

inputs: problem, a problem

local variables: current, a node
neighbour, a node

$\text{current} \leftarrow \text{Make\text{-}Node}(\text{Initial\text{-}State[problem]})$

loop do
    neighbour $\leftarrow$ a highest-valued successor of current
    if $\text{Value[neighbour]} < \text{Value[current]}$ then return $\text{State[current]}$
    current $\leftarrow$ neighbour
end

Hill-climbing contd.

Problem: depending on initial state, can get stuck on local maxima.

In continuous spaces, problems with choosing step size, slow convergence.

Simulated annealing

Idea: escape local maxima by allowing some “bad” moves but gradually decrease their size and frequency.

The name comes from the process used to harden metals and glass by heating them to a high temperature, and then letting them cool slowly, to reach a low energy crystalline state.

function $Simulated\text{-}Annealing(problem, schedule)$ returns a solution state

inputs: problem, a problem

schedule, a mapping from time to “temperature”

local variables: current, a node
next, a node
$T$, a “temperature” controlling prob. of downward steps

$\text{current} \leftarrow \text{Make\text{-}Node}(\text{Initial\text{-}State[problem]})$

for $t \leftarrow 1$ to $\infty$ do
    $T \leftarrow \text{schedule}[t]$
    if $T = 0$ then return current
    next $\leftarrow$ a randomly selected successor of current
    $\Delta E \leftarrow \text{Value[next]} - \text{Value[current]}$
    if $\Delta E > 0$ then current $\leftarrow$ next
    else current $\leftarrow$ next only with probability $e^{\Delta E / T}$
Properties of simulated annealing

In the inner loop, this picks a **Random** move:
– if it improves the state, it is accepted;
– if not, it is accepted with decreasing probability,
  depending on how much worse the state is, and time elapsed.
It can be shown that, if $T$ decreased slowly enough, then always reach best state.

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling;
now widely used in VLSI layout, airline scheduling, etc.

Constraint satisfaction problems (CSPs)

Standard search problem:
 state is a “black box”—any old data structure
  that supports goal test, eval, successor

CSP:
 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
This is a simple example of a **formal representation language**.
Allows useful **general-purpose** algorithms with more power
  than standard search algorithms

Example: Map-Colouring

Colour the map with **three** colours so that no two adjacent states have the same colour.

<table>
<thead>
<tr>
<th>Variables</th>
<th>WA, NT, Q, NSW, V, SA, T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains</td>
<td>$D_i = {\text{red}, \text{green}, \text{blue}}$</td>
</tr>
<tr>
<td>Constraints</td>
<td>$WA \neq NT, WA \neq SA, \ldots$ (if the language allows this)</td>
</tr>
<tr>
<td></td>
<td>or ${(WA, NT) \in {(\text{red, green}), (\text{red, blue}), (\text{green, red})}, \ldots}$</td>
</tr>
<tr>
<td></td>
<td>${(WA, Q) \in {(\text{red, green}), (\text{red, blue}), (\text{green, red})}, \ldots}$</td>
</tr>
</tbody>
</table>
Example: Map-Coloring contd.

Solutions satisfy all constraints, e.g. \{WA = \text{red}, NT = \text{green}, SA = \text{blue}, \ldots\}

Constraint graph

Binary CSP: each constraint relates at most two variables
Constraint graph: nodes are variables, arcs show constraints

General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Varieties of CSPs

Discrete variables
- finite domains; size $d \Rightarrow O(d^n)$ complete assignments
  - e.g., Boolean CSPs, incl. Boolean satisfiability
- infinite domains (integers, strings, etc.)
  - e.g., job scheduling, variables are start/end days for each job
  - need a constraint language, e.g., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$
  - linear constraints solvable, nonlinear undecidable

Continuous variables
- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in poly time by LP methods

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., cryptarithmetic column constraints

Preferences (soft constraints), e.g., \text{red} is better than \text{green}
- often representable by a cost for each variable assignment
  $\Rightarrow$ constrained optimization problems
Example: Cryptarithmetic

\[
\begin{array}{c}
T \ W \ O \\
+ \ T \ W \ O \\
\hline
F \ U \ R
\end{array}
\]

<table>
<thead>
<tr>
<th>Variables</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains</td>
<td>?</td>
</tr>
<tr>
<td>Constraints</td>
<td>?</td>
</tr>
</tbody>
</table>

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
\[ \text{alldiff}(F,T,U,W,R), \ O + O = R + 10 \cdot X_1, \ldots \]

Real-world CSPs

Assignment problems  
e.g., who teaches what class
Timetabling problems  
e.g., which class is offered when and where?
Hardware configuration
Spreadsheets
Transportation scheduling
Factory scheduling
Floorplanning

Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

Let’s start with the straightforward, dumb approach, then fix it
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.  
  \( \Rightarrow \) fail if no legal assignments (not fixable!)
- **Goal test**: the current assignment is complete

- This is the same for all CSPs!
- Every solution appears at depth \( n \) with \( n \) variables:  
  \( \Rightarrow \) use depth-first search
- Path is irrelevant, so can also use complete-state formulation
- \( b = (n - \ell)d \) at depth \( \ell \), hence \( n!d^n \) leaves!!!!
Backtracking search

Variable assignments are commutative, i.e.,
\[ W = red \text{ then } N = green \] same as \[ N = green \text{ then } W = red \]

Only need to consider assignments to a single variable at each node
\[ \Rightarrow b = d \text{ and there are } d^n \text{ leaves} \]

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

Backtracking search is the basic uninformed algorithm for CSPs.

Can solve \( n \)-queens for \( n \approx 25 \)

---

Backtracking search

function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING([], csp)

function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
    if assigned is complete then return assigned
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
        if value is consistent with assigned according to CONSTRAINTS[csp] then
            result ← RECURSIVE-BACKTRACKING([var = value|assigned], csp)
            if result ≠ failure then return result
        end
    return failure
Summary

Local search:
- iterative improvement algorithms – hill climbing, simulated annealing

CSPs are a special kind of problem:
  states defined by values of a fixed set of variables
  goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node