Non-classical search algorithms
by Stuart Russell
modified by Jacek Malec for LTH lectures
January 19th, 2018

Chapter 4 of AIMA

Outline

♦ Hill-climbing
♦ Simulated annealing (briefly)
♦ Genetic algorithms (briefly)
♦ Local search in continuous spaces (briefly)
♦ Searching with nondeterministic actions (briefly)
♦ Searching with partial observations (briefly)
♦ Online search and unknown environments (briefly)

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

Constant space, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges

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 to reduce number of conflicts.

Almost always solves \( n \)-queens problems almost instantaneously for very large \( n \), e.g., \( n = 1 \text{million} \).

Hill-climbing (or gradient ascent/descent)

"Like climbing Everest in thick fog with amnesia"

```
function Hill-Climbing(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node
  current ← Make-Node(Initial-State[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if Value[neighbor] ≤ Value[current] then return State[current]
    current ← neighbor
  end
```

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.

Simulated annealing

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

```
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
                  next, a node
                  T, a “temperature” controlling prob. of downward steps
  current ← Make-Node(Initial-State[problem])
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← Value[next] − Value[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability \( e^{ΔE/T} \)
```

Properties of simulated annealing

At fixed “temperature” $T$, state occupation probability reaches Boltzmann distribution

$$p(x) = \alpha e^{-\frac{E(x)}{kT}}$$

$T$ decreased slowly enough $\Rightarrow$ always reach best state $x^*$

because $e^{\frac{E(x)}{kT}}/e^{\frac{E(x^*)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small $T$

Is this necessarily an interesting guarantee?!!

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.

---

Local beam search

Idea: keep $k$ states instead of 1; choose top $k$ of all their successors

Not the same as $k$ searches run in parallel!

Searches that find good states recruit other searches to join them

Problem: quite often, all $k$ states end up on same local hill

Idea: choose $k$ successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic algorithms

= stochastic local beam search + generate successors from pairs of states

GAs require states encoded as strings (GPs use programs)

Crossover helps iff substrings are meaningful components

GAs $\neq$ evolution: e.g., real genes encode replication machinery!
Continuous state spaces

Suppose we want to site three robot battery loading stations in the hospital:
- 6-D state space defined by \((x_1, y_1), (x_2, y_2), (x_3, y_3)\)
- objective function \(f(x_1, y_2, x_2, y_2, x_3, y_3) = \) sum of squared distances from each location to nearest loading station.

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers \(\pm \delta\) change in each coordinate.

Gradient methods compute
\[
\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)
\]
to increase/reduce \(f\), e.g., by \(x \leftarrow x + \alpha \nabla f(x)\)

Sometimes can solve for \(\nabla f(x) = 0\) exactly (e.g., with one location).
Newton–Raphson (1664, 1690) iterates \(x \leftarrow x - H^{-1}(x) \nabla f(x)\) to solve \(\nabla f(x) = 0\), where \(H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}\)

Searching with nondeterministic actions

And-or search trees

For the erratic case:

Erratic vacuum world: modified \textbf{Suck};
Slippery vacuum world: modified \textbf{Right} and \textbf{Left}.

And-or search trees

For the slippery case:
Searching with nondeterministic actions

function **AND-OR-GRAph-Search**\((problem)\) returns a cond. plan, or failure

Or-Search\((problem,\text{Initial-State},\text{problem},[])\)

function **Or-Search**\((state,problem,path)\) returns a conditional plan or failure:

if problem.Goal-Test\((state)\) then return the empty plan

if state is on path then return failure

for each action in problem.Actions\((state)\) do

plan ← And-Search\((\text{Results}(state,action),problem,[state \mid path])\)

if plan ≠ failure then return [action | plan]
return failure

function **And-Search**\((states,problem,path)\) returns a conditional plan or failure:

for each \(s\) in \(states\) do

plan\(_i\) ← Or-Search\((s,problem,path)\)
if plan\(_i\) = failure then return failure
return [if \(s_1\) then plan\(_1\) else if \(s_2\) then plan\(_2\) else if \(\ldots\) plan\(_{n-1}\) else plan\(_n\) ]

Searching with partial observations

Diamond no-information case:
- sensorless problem, or
- conformant problem

Diamond state-space search is made in belief space

Diamond Problem solving: and-or search!

Searching with partial observations

Deterministic case:

Local sensing, deterministic and slippery cases:

\[
\begin{align*}
(1,3,5,7) & \quad L \quad R \\
(1,2,3,4,6,7,8) & \quad R \\
(5,7) & \quad L \quad (3,5,7) \\
(6,8) & \quad L \\
(8) & \quad R \\
(2,4,6,8) & \quad R \\
(4,8) & \quad L \\
(4,5,7,8) & \quad L \\
(1,3) & \quad R \\
(2,4) & \quad \text{Right} \\
(2) & \quad \text{[B, Dirty]} \quad \text{[B, Clean]} \\
(1,3) & \quad \text{Right} \\
(2) & \quad \text{[B, Dirty]} \\
(1,3) & \quad \text{Right} \\
(2) & \quad \text{[B, Dirty]} \quad \text{[B, Clean]} \\
(1,3) & \quad \text{Right} \\
(2) & \quad \text{[B, Dirty]} \quad \text{[B, Clean]} \\
(1,3) & \quad \text{Right} \\
(2) & \quad \text{[B, Dirty]} \quad \text{[B, Clean]} \\
\end{align*}
\]
Searching with partial observations

Planning for the local sensing case:

Online search and unknown environments

Interleaving computations and actions:

- act
- observe the results
- find out (compute) next action

Useful in dynamic domains.

Online search usually exploits locality of depth-first-like methods.

- random walk
- modified hill-climbing
- Learning Real-Time A* (LRTA*)
  - optimism under uncertainty
    - (unexplored areas assumed to lead to goal with least possible cost)