Local and Online search algorithms

Chapter 4
Outline

♦ Local search algorithms
  ◇ Hill-climbing
  ◇ Simulated annealing
  ◇ Genetic algorithms
♦ Searching with non-deterministic actions
♦ Searching with partially/no observation
♦ Online search
Local search algorithms

The search algorithms that we have seen so far are designed to explore search spaces systematically: The path is important and must be included in the solution.

In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all: Local search.

Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbours of that node: Advantages:
(1) they use very little memory usually a constant amount.
(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
Local search algorithms

In addition to finding goals, local search algorithms are useful for solving **optimization problems**, in which the aim is to find the best state according to an **objective function**.
**Hill-climbing (or gradient ascent/descent)**

“Like climbing Everest in thick fog with amnesia”

```plaintext
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 (Example)

Local search algorithms typically use a **complete-state formulation**.

The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors).

The heuristic cost function $h$ is the number of pairs of queens that are attacking each other, either directly or indirectly.

The global minimum of this function is zero, which occurs only at perfect solutions.

Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.
Hill-climbing (Example)

Figure 4.3  (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of $h$ for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.
Hill-climbing (Disadvantages)

Unfortunately, hill climbing often gets stuck for the following reasons:
♦ **Local maxima**: a local maximum is a peak that is higher than each of its neighbouring states but lower than the global maximum.
Hill-climbing (Disadvantages)

◊ **Ridges**: Because hill climbers only adjust one element in the vector at a time, each step will move in an axis-aligned direction. If the target function creates a narrow ridge that ascends in a non-axis-aligned direction, then the hill climber can only ascend the ridge by zig-zagging. If the sides of the ridge (or alley) are very steep, then the hill climber may be forced to take very tiny steps as it zig-zags toward a better position. Thus, it may take an unreasonable length of time for it to ascend the ridge (or descend the alley).
Hill-climbing (Disadvantages)

◊ **Plateaux**: a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau.
Hill-climbing (Variants)

**Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

**First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

**Random-restart hill climbing** conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.
Simulated annealing

A hill-climbing algorithm that never makes downhill moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.

In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.

It seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness: Simulated annealing is such an algorithm.

Idea of Simulated annealing: escape local maxima by allowing some “bad” moves but gradually decrease their frequency.
Simulated annealing

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 $e^{ΔE/T}$
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

stochastic beam search: choose $k$ successors randomly, biased towards good ones
A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.

Each state, or individual, is represented as a string over a finite alphabet. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares (ranges from 1 to 8).

Each state is rated by the objective function, or (in GA terminology) the fitness function. Example for 8-queen problem: the number of nonattacking pairs of queens.
Genetic algorithms

Fitness  Selection  Pairs  Cross-Over  Mutation
Genetic algorithms contd.

GAs require states encoded as strings (GPs use programs)

Crossover helps iff substrings are meaningful components
**Genetic algorithms contd.**

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals
        FITNESS-FN, a function that measures the fitness of an individual

repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
    population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
inputs: x, y, parent individuals

n ← LENGTH(x); c ← random number from 1 to n
return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```
More complex environments

Up to this point, we assumed that the environment is **fully observable** and **deterministic** and that the agent knows what the effects of each action are.

When the environment is either partially observable or nondeterministic (or both), percepts become useful.

→ In a **partially observable environment**, every percept helps narrow down the set of possible states the agent might be in.

→ When the environment is **nondeterministic**, percepts tell the agent which of the possible outcomes of its actions has actually occurred.
The erratic vacuum world:

1 2
3 4
5 6
7 8

the Suck action works as follows:
→ When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
→ When applied to a clean square the action sometimes deposits dirt on the carpet.
Searching with non-deterministic actions

Instead of defining the transition model by a \( \text{R RESULT} \) function that returns a single state, we use a \( \text{R RESULTS} \) function that returns a set of possible outcome states.

For example, in the erratic vacuum world, the Suck action in state 1 leads to a state in the set 5, 7.

Solutions for non-deterministic problems can contain nested \text{ifthenelse} statements; this means that they are trees rather than sequences.

For example, \([\text{Suck, if State = 5 then [Right, Suck] else [...]}]\).
AND-OR search trees

An extension of a search tree introduced in deterministic environments:

One branching type is introduced by the agents own choices in each state: OR nodes. One branching type is also introduced by the environments choice of outcome for each action: And nodes.

A solution for an AND-OR search problem is a subtree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes.
AND-OR search trees

Example of AND-OR tree in the erratic vacuum world with solution in bold lines.
AND-OR search algorithm

function AND-OR-GRAH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(problem.INITIAL-STATE, 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(RESULTS(state, action), problem, [state | 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_i in states do
        plan_i ← OR-SEARCH(s_i, 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 s_n-1 then plan_{n-1} else plan_n]
Search in partial observable environments

♦ Searching with no observation

♦ Searching with partial observation
Searching with no observation

When the agents percepts provide no information at all, we have what is called a sensor-less problem or sometimes a conformant problem.

Benefits of using sensor-less agents:
→ They don’t rely on sensors working properly.
→ They avoid the high cost of sensing.

To solve sensor-less problems, we search in the space of belief states rather than physical states.
Defining a sensor-less problem

The underlying problem $P$: $\text{ACTIONS}_P$, $\text{RESULT}_P$, $\text{GOAL-TEST}_P$, $\text{STEP} - \text{COST}_P$.

◇ **Belief states:** The entire belief-state space contains every possible set of physical states. If $P$ has $N$ states, then the sensor-less problem has up to $2^N$ states, although many may be unreachable from the initial state.

◇ **Initial state:** Typically the set of all states in $P$.

◇ **Actions:**
  → If we assume that illegal actions have no effect on the environment, then it is safe to take the union of all the actions in any of the physical states in the current belief state $b$.
  → If an illegal action might be the end of the world, it is safer to allow only the intersection, that is, the set of actions legal in all the states.
Defining a sensor-less problem

◊ **Transition model**: The process of generating the new belief state after the action is called the *prediction* step

→ For deterministic actions:

\[ b' = RESULT(b, a) = \{ s' : s' = RESULT_P(s, a) \text{ and } s \in b \} \]

→ For non-deterministic actions:

\[ b' = RESULT(b, a) = \bigcup_{s \in b} RESULT_S P(s, a) \]

◊ **Goal test**: A belief state satisfies the goal only if all the physical states in it satisfy \( GOAL - TEST_P \).

◊ **Path cost**: We assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.
Example

Reachable belief-state space for the deterministic, sensor-less vacuum world.
Difficulties of sensor-less problem-solvers

- The vastness of the belief-state space which is exponentially larger than the underlying physical state space.

- The size of each belief state. For example, the initial belief state for the 10 x 10 vacuum world contains $100 \times 2^{100}$ physical states.
Now we need a new function called \( \text{PERCEPT}(s) \) that returns the percept received in a given state. Fully observable problems are a special case in which \( \text{PERCEPT}(s) = s \) for every state \( s \), while sensor-less problems are a special case in which \( \text{PERCEPT}(s) = \text{null} \).

When observations are partial, it will usually be the case that several states could have produced any given percept. For example, the percept \([A, \text{Dirty}]\):
Defining a partially observable problem

The ACTIONS, STEP−COST, and GOAL−TEST are constructed from the underlying physical problem just as for sensor-less problems, but the transition model is a bit more complicated.

◊ The prediction stage is the same as for sensor-less problems: given the action a in belief state b, the predicted belief state is
\[ \hat{b} = \textit{PREDICT}(b, a). \]

◊ The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state:
\[ \textit{PERCEPTS}_{\text{possible}}(\hat{b}) = \{ o : o = \textit{PERCEPT}(s) \text{ and } s \in \hat{b} \}. \]

◊ The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state \( b_o \) is just the set of states in b that could have produced the percept:
\[ b_o = \textit{UPDATE}(\hat{b}, o) = \{ s : o = \textit{PERCEPT}(s) \text{ and } s \in \hat{b} \}. \]
Figure 1: (a) Deterministic world, (b) The non-deterministic slippery world (change location may or may not work)
Solving a partially observable problem

Applying And-Or search algorithm on the constructed belief state space.
Online search

So far we have concentrated on agents that use offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution.

An online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.

Online search is a good idea in
→ dynamic or semi-dynamic domains.
→ non-deterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won’t.
→ unknown environments, where the agent does not know what states exist or what its actions do.
Online search problem

We assume a deterministic and fully observable environment. However the agent knows only the following: → ACTIONS(s), which returns a list of actions allowed in state s
→ The step-cost function $c(s, a, s)$
→ GOAL-TEST(s).

the agent cannot determine RESULT(s, a) except by actually being in s and doing a.

We assume that the state space is safely explorable: that is, some goal state is reachable from every reachable state.
After each action, an online agent receives a percept telling it what state it has reached.

This **interleaving of planning and action** means that online search algorithms are quite different from the offline search algorithms we have seen previously.

Offline algorithms can expand a node in one part of the space and then immediately expand a node in another part. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies: **Depth-first** search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.
function ONLINE-DFS-AGENT(s′) returns an action

inputs: s′, a percept that identifies the current state
persistent: result, a table indexed by state and action, initially empty
untried, a table that lists, for each state, the actions not yet tried
unbacktracked, a table that lists, for each state, the backtracks not yet tried
s, a, the previous state and action, initially null

if GOAL-TEST(s′) then return stop
if s′ is a new state (not in untried) then untried[s′] ← ACTIONS(s′)
if s is not null then
  result[s, a] ← s′
  add s to the front of unbacktracked[s′]
if untried[s′] is empty then
  if unbacktracked[s′] is empty then return stop
  else a ← an action b such that result[s′, b] = POP(unbacktracked[s′])
else a ← POP(untried[s′])
s ← s′
return a
Online local search

Hill-climbing search has the property of locality in its node expansions.

In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm!

Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go.

Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Solutions:
→ Hill-climbing with random walk.
→ Augmenting hill climbing with memory: LRTA*.
The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited.

(a) 

(b) 

(c) 

(d) 

(e)
**LRTA* algorithm**

function LRTA*-AGENT($s'$) returns an action

inputs: $s'$, a percept that identifies the current state

persistent: result, a table, indexed by state and action, initially empty

$H$, a table of cost estimates indexed by state, initially empty

$s$, $a$, the previous state and action, initially null

if GOAL-TEST($s'$) then return stop

if $s'$ is a new state (not in $H$) then $H[s'] \leftarrow h(s')$

if $s$ is not null

\[
\text{result}[s, a] \leftarrow s'
\]

\[
H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, \text{result}[s, b], H)
\]

$a \leftarrow$ an action $b$ in $\text{ACTIONS}(s')$ that minimizes $\text{LRTA*-COST}(s', b, \text{result}[s', b], H)$

$s \leftarrow s'$

return $a$

function LRTA*-COST($s$, $a$, $s'$, $H$) returns a cost estimate

if $s'$ is undefined then return $h(s)$

else return $c(s, a, s') + H[s']$
Local search methods such as hill climbing operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including simulated annealing, which returns optimal solutions when given an appropriate cooling schedule.

A genetic algorithm is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover, which combines pairs of states from the population.

In nondeterministic environments, agents can apply AND-OR search to generate contingent plans that reach the goal regardless of which outcomes occur during execution.
Summary

♦ When the environment is partially observable, the belief state represents the set of possible states that the agent might be in.

♦ Standard search algorithms can be applied directly to belief-state space to solve sensor-less problems, and belief-state AND-OR search can solve general partially observable problems.

♦ Exploration problems arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, online search agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.