Problem-solving agents

Restricted form of general agent:

```python
function SIMPLE-PROBLEM-SOLVING-AGENT( percept ) returns an action
    static: seq, an action sequence, initially empty
             state, some description of the current world state
             goal, a goal, initially null
             problem, a problem formulation
    state ← Update-State( state, percept )
    if seq is empty then
        goal ← Formulate-Goal( state )
        problem ← Formulate-Problem( state, goal )
        seq ← Search( problem )
        action ← Recommendation( seq, state )
        seq ← Remainder( seq, state )
    return action
```

Note: this is offline problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.
Example: Blocket

Service robot Odin, delivering drugs to divisions. Currently in the Pharmacy. There is a drug order from Intensive Care Unit.

Formulate goal:
be in Intensive Care Unit

Formulate problem:
states: various locations
actions: drive between locations

Find solution:
sequence of locations, e.g., Pharmacy, Elevator A, Surgery, ICU

Problem types

Deterministic, fully observable \(\implies\) single-state problem
Agent knows exactly which state it will be in; solution is a sequence

Non-observable \(\implies\) conformant problem
Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable \(\implies\) contingency problem
percepts provide new information about current state
solution is a contingent plan or a policy
often interleave search, execution

Unknown state space \(\implies\) exploration problem (“online”)
Example: vacuum world

Single-state, start in #5. Solution??
[Right, Suck]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}. Solution??

Example: vacuum world

Single-state, start in #5. Solution??
[Right, Suck]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}. Solution??
[Right, Suck, Left, Suck]

Contingency, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??

Example: vacuum world

Single-state problem formulation

A problem is defined by four items:
initial state e.g., “at Pharmacy”

successor function $S(x) =$ set of action–state pairs
e.g., $S(Pharmacy) = \{ (Pharmacy \rightarrow Storage, Storage), \ldots \}$

goal test, can be
explicit, e.g., $x =$ “at ICU”
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of actions executed, etc.
$c(x, a, y)$ is the step cost, assumed to be $\geq 0$

A solution is a sequence of actions
leading from the initial state to a goal state
Selecting a state space

Real world is absurdly complex
⇒ state space must be abstracted for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions
  e.g., “Pharmacy → Storage” represents a complex set
  of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state “in Pharmacy”
  must get to some real state “in Storage”

(Abstract) solution =
  set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!
Example: vacuum world state space graph

- **States??**: integer dirt and robot locations (ignore dirt amounts etc.)
- **Actions??**: Left, Right, Suck, NoOp
- **Goal test??**: no dirt
- **Path cost??**

Example: The 8-puzzle

- **States??**: integer locations of tiles (ignore intermediate positions)
- **Actions??**
- **Goal test??**
- **Path cost??**
Example: The 8-puzzle

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\quad \quad \quad
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Start State \quad \quad Goal State

\textit{states}?: integer locations of tiles (ignore intermediate positions)
\textit{actions}?: move blank left, right, up, down (ignore unjamming etc.)
\textit{goal test}?: = goal state (given)
\textit{path cost}?: 1 per move

[Note: optimal solution of \(n\)-Puzzle family is NP-hard]

---

Example: The 8-puzzle

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\quad \quad \quad
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Start State \quad \quad Goal State

\textit{states}?: integer locations of tiles (ignore intermediate positions)
\textit{actions}?: move blank left, right, up, down (ignore unjamming etc.)
\textit{goal test}?: = goal state (given)
\textit{path cost}?:

---

Example: robotic assembly

\[
\begin{array}{ccc}
\end{array}
\]

\textit{states}?: real-valued coordinates of robot joint angles
\textit{actions}?: continuous motions of robot joints
\textit{goal test}?: complete assembly with no robot included!
\textit{path cost}?: time to execute
Tree search algorithms

Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

\[
\text{function Tree-Search}(\text{problem, strategy}) \text{ returns a solution, or failure}
\]
\[
\text{initialize the search tree using the initial state of problem}
\]
\[
\text{loop do}
\]
\[
\text{if there are no candidates for expansion then return failure}
\]
\[
\text{choose a leaf node for expansion according to strategy}
\]
\[
\text{if the node contains a goal state then return the corresponding solution}
\]
\[
\text{else expand the node and add the resulting nodes to the search tree}
\]
\[
\text{end}
\]

Tree search example

Pharmacy
Laryngology
A
Storage

© Stuart Russell  Chapter 3 of AIMA  23
### Implementation: states vs. nodes

A **state** is a (representation of) a physical configuration.

A **node** is a data structure constituting part of a search tree, includes parent, children, depth, path cost $g(x)$.

States do not have parents, children, depth, or path cost!

The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

### Implementation: general tree search

```plaintext
function Tree-Search(problem, fringe) returns a solution, or failure
    fringe ← Insert(Make-Node(Initial-State(problem)), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State(node)) then return node
        fringe ← InsertAll(Expand(node, problem), fringe)
    end loop

function Expand(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn(problem, State[node]) do
        s ← a new Node
        Parent-Node[s] ← node,
        Action[s] ← action,
        State[s] ← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(State[node], action, result)
        Depth[s] ← Depth[node] + 1
        add s to successors
    end loop
    return successors
```

### Search strategies

A strategy is defined by picking the **order of node expansion**.

Strategies are evaluated along the following dimensions:

- **completeness**—does it always find a solution if one exists?
- **time complexity**—number of nodes generated/expanded
- **space complexity**—maximum number of nodes in memory
- **optimality**—does it always find a least-cost solution?

Time and space complexity are measured in terms of:

- $b$—maximum branching factor of the search tree
- $d$—depth of the least-cost solution
- $m$—maximum depth of the state space (may be $\infty$)

### Uninformed search strategies

**Uninformed** strategies use only the information available in the problem definition.

- Sometimes called **blind** search strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

\[ A \]

\[ B \]

\[ C \]

\[ D \]

\[ E \]

\[ F \]

\[ G \]
Properties of breadth-first search

Complete? Yes (if \( b \) is finite)

Time? \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \), i.e., exp. in \( d \)

Space? \( O(b^{d+1}) \) (keeps every node in memory)

Optimal??
Properties of breadth-first search

Complete?? Yes (if $b$ is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
Space?? $O(b^{d+1})$ (keeps every node in memory)
Optimal?? Yes (if cost = 1 per step); not optimal in general
Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:
fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front

Depth-first search

Expand deepest unexpanded node

Implementation:
fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation:

\[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]
Depth-first search

Expand deepest unexpanded node

Implementation:

 fringe = LIFO queue, i.e., put successors at front
**Depth-first search**

Expand deepest unexpanded node

**Implementation:**

\( fringe = \text{LIFO queue}, \text{i.e., put successors at front} \)

---

**Properties of depth-first search**

Complete??

---

**Depth-first search**

Expand deepest unexpanded node

**Implementation:**

\( fringe = \text{LIFO queue}, \text{i.e., put successors at front} \)

---

**Properties of depth-first search**

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

\( \Rightarrow \) complete in finite spaces

Time??
Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
   Modify to avoid repeated states along path
   ⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$
   but if solutions are dense, may be much faster than breadth-first

Space??

Optimal?? No

Depth-limited search

= depth-first search with depth limit $l$,
  i.e., nodes at depth $l$ have no successors

Recursive implementation:

```python
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if Goal-Test(problem, State[node]) then return node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```
Iterative deepening search

function Iterative-Deepening-Search(problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← Depth-Limited-Search(problem, depth)
        if result ≠ cutoff then return result
    end

© Stuart Russell
Chapter 3 of AIMA 61

Iterative deepening search \( l = 0 \)

Limit = 0

© Stuart Russell
Chapter 3 of AIMA 62

Iterative deepening search \( l = 1 \)

Limit = 1

© Stuart Russell
Chapter 3 of AIMA 63

Iterative deepening search \( l = 2 \)

Limit = 2

© Stuart Russell
Chapter 3 of AIMA 64
Properties of iterative deepening search

**Complete**? Yes

**Time**? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$

**Space**?
Properties of iterative deepening search

**Complete??** Yes

**Time??** \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

**Space??** \(O(bd)\)

**Optimal??** Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Numerical comparison for \(b = 10\) and \(d = 5\), solution at far right leaf:

\[N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450\]

\[N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100\]

IDS does better because other nodes at depth \(d\) are not expanded

BFS can be modified to apply goal test when a node is generated

---

**Summary of algorithms**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-Limited</th>
<th>Depth-First</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes(^a)</td>
<td>Yes(^a)</td>
<td>No</td>
<td>Yes, if (l \geq d)</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>(b^{l+1})</td>
<td>(b^{\lceil C^* / \epsilon \rceil})</td>
<td>(b^l)</td>
<td>(b^l)</td>
<td>(b^d)</td>
</tr>
<tr>
<td>Space</td>
<td>(b^{l+1})</td>
<td>(b^{\lceil C^* / \epsilon \rceil})</td>
<td>(bd)</td>
<td>(bd)</td>
<td>(bd)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes(^a)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes(^a)</td>
</tr>
</tbody>
</table>

---

**Repeated states**

Failure to detect repeated states can turn a linear problem into an exponential one!
**Graph search**

```plaintext
function Graph-Search(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← Insert(Make-Node(Initial-State(problem)), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then return node
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← InsertAll(Expand(node, problem), fringe)
        end
    end
```

**Partial summary**

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

Variety of uninformed search strategies

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Graph search can be exponentially more efficient than tree search.

**Informed Search Algorithms**

- Best-first search
- A* search
- Heuristics

**Best-first search**

**Idea:** use an evaluation function for each node — estimate of “desirability”

⇒ Expand most desirable unexpanded node

**Implementation:**

- `fringe` is a queue sorted in decreasing order of desirability

**Special cases:**
- greedy search
- A* search
**Greedy search**

Evaluation function $h(n)$ (heuristic)

- estimate of cost from $n$ to the closest goal

E.g., $h_{SLD}(n) =$ straight-line distance from $n$ to ICU

Greedy search expands the node that **appears** to be closest to goal

---

**Greedy search example**

- Pharmacy
- Laryngology
- A
- Storage
- ER
- Surgery
- Nephrology
- Radiology
- Orthopedic
- Dermatology
- Ophthalmology
- Surgery
- ICU

Direct 3D distances to ICU:
- Radiology: 200
- Orthopedic: 285
- Nephrology: 250
- Dermatology: 90
- Ophthalmology: 110
- A: 230
- Surgery: 100
- C: 77
- ER: 102
- Geriatrics: 280
- Laryngology: 312
- Pharmacy: 280
- Storage: 235
- Laundry: 220
- ICU: 0
**Greedy search example**

Properties of greedy search

*Complete??*

*Time??*
Properties of greedy search

**Complete??** No—can get stuck in loops, e.g.,
Radiology → Orthopedy → Radiology → Orthopedy →
Complete in finite space with repeated-state checking

**Time??** $O(b^m)$, but a good heuristic can give dramatic improvement

**Space??**

---

A* search

**Idea:** avoid expanding paths that are already expensive

**Evaluation function** $f(n) = g(n) + h(n)$

- $g(n) =$ cost so far to reach $n$
- $h(n) =$ estimated cost to goal from $n$
- $f(n) =$ estimated total cost of path through $n$ to goal

A* search uses an *admissible* heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from $n$. (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal $G$.)

E.g., $h_{SLD}(n)$ never overestimates the actual road distance

**Theorem:** A* search is optimal
Optimality of $A^*$ (standard proof)

Suppose some suboptimal goal $G_2$ has been generated and is in the queue. Let $n$ be an unexpanded node on a shortest path to an optimal goal $G_1$.

\[
\begin{align*}
    f(G_2) &= g(G_2) \quad \text{since } h(G_2) = 0 \\
    &> g(G_1) \quad \text{since } G_2 \text{ is suboptimal} \\
    &\geq f(n) \quad \text{since } h \text{ is admissible}
\end{align*}
\]

Since $f(G_2) > f(n)$, $A^*$ will never select $G_2$ for expansion.

Properties of $A^*$

Complete? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time? Exponential in \(h \times \text{length of solution}\)

Space?
Properties of A∗

- **Complete??** Yes, unless there are infinitely many nodes with \( f \leq f(G) \)
- **Time??** Exponential in [relative error in \( h \times \text{length of soln.} \)]
- **Space??** Keeps all nodes in memory
- **Optimal??** Yes—cannot expand \( f_{i+1} \) until \( f_i \) is finished

A∗ expands all nodes with \( f(n) < C^* \)
A∗ expands some nodes with \( f(n) = C^* \)
A∗ expands no nodes with \( f(n) > C^* \)

Admissible heuristics

E.g., for the 8-puzzle:

- \( h_1(n) = \) number of misplaced tiles
- \( h_2(n) = \) total Manhattan distance
  (i.e., no. of squares from desired location of each tile)

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array} & \begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

Start State  Goal State

\[
\begin{align*}
h_1(S) &= ?? \\
h_2(S) &= ??
\end{align*}
\]
**Dominance**

If \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible),
then \( h_2 \) dominates \( h_1 \) and is better for search.

Typical search costs:

- \( d = 14 \), IDS = 3,473,941 nodes
  - \( A^*(h_1) = 539 \) nodes
  - \( A^*(h_2) = 113 \) nodes
- \( d = 24 \), IDS \( \approx \) 54,000,000,000 nodes
  - \( A^*(h_1) = 39,135 \) nodes
  - \( A^*(h_2) = 1,641 \) nodes

Given any admissible heuristics \( h_a, h_b \),

\[ h(n) = \max(h_a(n), h_b(n)) \]

is also admissible and dominates \( h_a, h_b \).

**Relaxed problems**

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere,
then \( h_1(n) \) gives the shortest solution.

If the rules are relaxed so that a tile can move to any adjacent square,
then \( h_2(n) \) gives the shortest solution.

Key point: the optimal solution cost of a relaxed problem
is no greater than the optimal solution cost of the real problem.

**Summary**

Heuristic functions estimate costs of shortest paths.

Good heuristics can dramatically reduce search cost.

Greedy best-first search expands lowest \( h \)
  - incomplete and not always optimal.

\( A^* \) search expands lowest \( g + h \)
  - complete and optimal.
  - also optimally efficient (up to tie-breaks, for forward search).

Admissible heuristics can be derived from exact solution of relaxed problems.