Solving problems by searching

Chapter 3

Some slide credits to Hwee Tou Ng (Singapore)
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
- Heuristics
Intelligent agent solves problems by?
**Problem-solving agents**

```plaintext
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 do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
action ← FIRST(seq)
seq ← REST(seq)
return action
```
Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

**Formulate goal:**
- be in Bucharest

**Formulate problem:**
- **states**: various cities
- **actions**: drive between cities

**Find solution:**
- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Romania: problem type?
Romania: Problem type

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

- Non-observable $\rightarrow$ sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence

- Nondeterministic and/or partially observable $\rightarrow$ contingency problem
  - percepts provide new information about current state
  - often we interle} search, execution

- Unknown state space $\rightarrow$ exploration problem
A problem is defined by four items:

1. **initial state** e.g., "at Arad"
2. **actions or successor function** $S(x) =$ set of action-state pairs
   - e.g., $S(Arad) = \{<Arad \rightarrow Zerind, Zerind>, \ldots \}$
3. **goal test**, can be
   - explicit, e.g., $x =$ "at Bucharest"
   - implicit, e.g., $Checkmate(x)$
4. **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
Tree search algorithms

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

```plaintext
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```
Tree search example
Tree search example
Tree search example

![Tree search example diagram]
Implementation: general tree search

```
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 SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

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(node, action, s)
        Depth[s] ← Depth[node] + 1
        add s to successors
    return successors
```
Uninformed search strategies

Uninformed search strategies use only the information available in the problem definition

- 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
Breadth-first search

- Expand shallowest unexpanded node

**Implementation:**
- *fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

- Expand shallowest unexpanded node

**Implementation:**
- *fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

- Expand shallowest unexpanded node

**Implementation:**

- *fringe* is a FIFO queue, i.e., new successors go at end
Example: Romania (Q)
Search strategies

- A search 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
  - **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 $\infty$)
Properties of breadth-first search

- **Complete?** Yes (if \( b \) is finite)
- **Time?** \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \)
- **Space?** \( O(b^{d+1}) \) (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step)

- Space is the bigger problem (more than time)
Uniform-cost search

Video
Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation:**
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- **Complete?** Yes, if step cost $\geq \varepsilon$
- **Time?** # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C*/\varepsilon)})$ where $C^*$ is the cost of the optimal solution
- **Space?** # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C*/\varepsilon)})$
- **Optimal?** Yes – nodes expanded in increasing order of $g(n)$
Comparison of Searches

So far:

• Breadth-first search
• Uniform-cost (cheapest) search
• New: Depth-first search

Optimal?
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:
  - *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:
  - 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:**
  - *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:
  - 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:
  - *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
Why depth-first?
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 $m >> d$
  - but if solutions are dense, may be much faster than breadth-first

- **Space?** $O(bm)$, i.e., linear space!

- **Optimal?** No
# Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*/\epsilon})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*/\epsilon})$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Limitations

Video
Best-first search

Idea: use an evaluation function $f(n)$ for each node estimate of "desirability"
   → Expand most desirable unexpanded node

Implementation:
   Order the nodes in fringe in decreasing order of desirability

Special cases:
   greedy best-first search
   A* search
Romania with step costs in km

Straight-line distance to Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobreta</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>176</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamț</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>10</td>
</tr>
<tr>
<td>Rimnița Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timișoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Greedy best-first search

Evaluation function $f(n) = h(n)$ (heuristic)

$= \text{estimate of cost from } n \text{ to goal}$

e.g., $h_{SLD}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

Greedy best-first search expands the node that appears to be closest to goal
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Properties of greedy best-first search

- **Complete?** No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- **Time?** $O(b^m)$, but a good heuristic can give dramatic improvement
- **Space?** $O(b^m)$ -- keeps all nodes in memory
- **Optimal?** No
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 from $n$ to goal

$f(n) =$ estimated total cost of path through $n$ to goal
A* search example
A* search example
A* search example
A* search example
A* search example
A* search example
State Spaces
Example: vacuum world

- Single-state, start in #5. Solution?
Example: vacuum world

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

- **Sensorless**, start in \{1,2,3,4,5,6,7,8\} e.g., Right goes to \{2,4,6,8\}. Solution?
Example: vacuum world

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

- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: \[L, \text{Clean}\], i.e., start in #5 or #7
    
    **Solution?**
Example: vacuum world

- **Sensorless**, 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**
  - Nondeterministic: Suck may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: [L, Clean], i.e., start in #5 or #7
    
    **Solution?** [Right, if dirt then Suck]
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., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem
Vacuum world state space graph

- **states:**
- **actions?**
- **goal test?**
- **path cost?**
Vacuum world state space graph

- **states?** integer dirt and robot location
- **actions?** *Left, Right, Suck*
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Modified vacuum world?
Example: The 8-puzzle

- states?
- actions?
- goal test?
- path cost?
Example: The 8-puzzle

- **states?** locations of tiles
- **actions?** move blank left, right, up, down
- **goal test?** = goal state (given)
- **path cost?** 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard]
Example: robotic assembly

- **states?**: real-valued coordinates of robot joint angles parts of the object to be assembled
- **actions?**: continuous motions of robot joints
- **goal test?**: complete assembly
- **path cost?**: time to execute
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 **state**, **parent node**, **action**, **path cost** $g(x)$, **depth**.

- The **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.
Depth-limited search

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

- Recursive implementation:

```plaintext
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    return 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 SOLUTION(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 \textsc{Iterative-Deepening-Search}(\textit{problem}) \textbf{returns} a solution, or failure

\hspace*{1em} \textbf{inputs:} \textit{problem}, a problem

\hspace*{1em} \textbf{for} depth $\leftarrow$ 0 \textbf{to} $\infty$ \textbf{do}

\hspace*{2em} result $\leftarrow$ \textsc{Depth-Limited-Search}(\textit{problem}, depth)

\hspace*{2em} \textbf{if} result $\neq$ cutoff \textbf{then return} result
Iterative deepening search
\( l = 0 \)
Iterative deepening search

\[ l = 1 \]
Iterative deepening search

\( l = 2 \)
Iterative deepening search
\( l = 3 \)
Iterative deepening search

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
  \[ N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d \]

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  \[ N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d \]

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead = \( \frac{(123,456 - 111,111)}{111,111} = 11\% \)
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** \((d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)\)
- **Space?** \(O(bd)\)
- **Optimal?** Yes, if step cost = 1
Admissible heuristics

A heuristic \( h(n) \) is **admissible** if for every node \( n \),

\[
    h(n) \leq h^*(n), \quad \text{where } h^*(n) \text{ is the true cost to reach the goal state from } n.
\]

An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**

Example: \( h_{SLD}(n) \) (never overestimates the actual road distance)

- **Theorem**: If \( h(n) \) is admissible, \( A^* \) using \( \text{TREE-SEARCH} \) is optimal
Optimality of A* (proof)

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

\[
f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0
\]
\[
g(G_2) > g(G) \quad \text{since } G_2 \text{ is suboptimal}
\]
\[
f(G) = g(G) \quad \text{since } h(G) = 0
\]
\[
f(G_2) > f(G) \quad \text{from above}
\]
Consistent heuristics

A heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,

$$h(n) \leq c(n,a,n') + h(n')$$

If $h$ is consistent, we have

\[
\begin{align*}
    f(n') &= g(n') + h(n') \\
          &= g(n) + c(n,a,n') + h(n') \\
          &\geq g(n) + h(n) \\
          &= f(n)
\end{align*}
\]

i.e., $f(n)$ is non-decreasing along any path.

- **Theorem**: If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal.
Optimality of A*

A* expands nodes in order of increasing $f$ value

Gradually adds "$f$-contours" of nodes

Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Properties of A$^*$

- **Complete?** Yes (unless there are infinitely many nodes with $f \leq f(G)$ )
- **Time?** Exponential
- **Space?** Keeps all nodes in memory
- **Optimal?** Yes
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)

\[ \begin{align*}
\text{Start State} & \quad \begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array} \\
\text{Goal State} & \quad \begin{array}{ccc}
1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
\end{array}
\end{align*} \]

- \( h_1(S) = ? \)
- \( h_2(S) = ? \)
Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n) = \text{number of misplaced tiles}$
- $h_2(n) = \text{total Manhattan distance}$

(i.e., no. of squares from desired location of each tile)

- $h_1(S) = \_\_\_ = 8$
- $h_2(S) = \_\_\_ = 3+1+2+2+2+3+3+2 = 18$
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

![Diagram showing repeated states]

![Diagram showing exponential growth due to repeated states]

2013 CS 325 - Ch3 Search
function \textsc{Graph-Search}(\textit{problem}, fringe) \textbf{returns} a solution, or failure

\begin{itemize}
\item \textit{closed} $\leftarrow$ an empty set
\item \textit{fringe} $\leftarrow$ \textsc{Insert}(\textsc{Make-Node}([\textit{Initial-State}[\textit{problem}]abyrin, fringe)
\end{itemize}

loop do

\begin{itemize}
\item \textbf{if} \textit{fringe} is empty \textbf{then} return failure
\item \textit{node} $\leftarrow$ \textsc{Remove-Front}(fringe)
\item \textbf{if} \textsc{Goal-Test}([\textit{problem}][\textit{State}[\textit{node}])] \textbf{then} return \textsc{Solution}(\textit{node})
\item \textbf{if} \textit{State}[\textit{node}] is not in \textit{closed} \textbf{then}
\begin{itemize}
\item add \textit{State}[\textit{node}] to \textit{closed}
\item \textit{fringe} $\leftarrow$ \textsc{InsertAll}(\textsc{Expand}(\textit{node}, \textit{problem}), fringe)
\end{itemize}
\end{itemize}
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