Artificial Intelligence

Introduction to Search
(Ch. 3.1-4)
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
Problem types

- 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
  - Solution is a contingent plan or a policy
  - Often interleave search, execution

- Unknown state and action space \(\rightarrow\) **exploration problem**
Example: vacuum world

- **Single-state**, start in #5. Solution?

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

- **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?
A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function \( S(x) = \text{set of action–state pairs} \)
   - e.g., \( S(\text{Arad}) = \{<\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}>, \ldots \} \)
3. goal test, can be
   - explicit, e.g., \( x = \text{"at Bucharest"} \)
   - implicit, e.g., \( \text{Checkmate}(x) \)
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
Selecting a state space

- (Abstract) Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- 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?
Example: The 8-puzzle

- states?
- actions?
- goal test?
- path cost?

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

- **states?**
- **actions?**
- **goal test?**
- **path cost?**
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
Implementation: general tree search

\[
\text{function TREE-SEARCH( problem, fringe) returns a solution, or failure}
\]
\[
fringe \leftarrow \text{INSERT(MAKE-NODE(\text{INITIAL-STATE}[\text{problem}]), fringe)}
\]
\[
\text{loop do}
\]
\[
\quad \text{if fringe is empty then return failure}
\]
\[
\quad node \leftarrow \text{REMOVE-FRONT(fringe)}
\]
\[
\quad \text{if GOAL-TEST[problem](\text{STATE}[node]) then return SOLUTION(node)}
\]
\[
\quad fringe \leftarrow \text{INSERT-ALL(\text{EXPAND}(node, problem), fringe)}
\]

\[
\text{function EXPAND( node, problem) returns a set of nodes}
\]
\[
\quad \text{successors} \leftarrow \text{the empty set}
\]
\[
\text{for each action, result in SUCCESSOR-FN[problem](\text{STATE}[node]) do}
\]
\[
\quad s \leftarrow \text{a new NODE}
\]
\[
\quad \text{PARENT-NODE}[s] \leftarrow node; \ \text{ACTION}[s] \leftarrow action; \ \text{STATE}[s] \leftarrow result
\]
\[
\quad \text{PATH-COST}[s] \leftarrow \text{PATH-COST}[node] + \text{STEP-COST}(node, action, s)
\]
\[
\quad \text{DEPTH}[s] \leftarrow \text{DEPTH}[node] + 1
\]
\[
\quad \text{add } s \text{ to } \text{successors}
\]
\[
\text{return } \text{successors}
\]
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 **Successor-Fn** of the problem to create the corresponding states.
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?
  – optimality: does it always find a least-cost solution?
  – time complexity: number of nodes generated
  – space complexity: maximum number of nodes in memory

• 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** 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
Properties of breadth-first search

- Complete?
- Optimal?
- Time?
- Space?
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?**
- **Optimal?**
- **Time?**
- **Space?**
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:**
  - \textit{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
Properties of depth-first search

- **Complete?**
- **Optimal?**
- **Time?**
- **Space?**
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 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 Iterative-Deepening-Search( problem) returns a solution, or failure

inputs: problem, a problem

for depth ← 0 to ∞ do
    result ← Depth-Limited-Search( problem, depth)
    if result ≠ cutoff 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 $b$:
  $N_{DLS} =$

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  $N_{IDS} =$

- For $b = 10$, $d = 5$,
  - $N_{DLS} =$
  - $N_{IDS} =$

- Overhead =
Properties of iterative deepening search

- Complete?
- Optimal?
- Time?
- Space?
### 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>$O(b^{[C^*]}/\varepsilon)$</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>$O(b^{[C^*]}/\varepsilon)$</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>
Repeated states

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

- The only difference is detecting repeated states

```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```
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