Artificial Intelligence

For HEDSPI Project

Lecturer 3 - Search

Lecturers:
- Le Thanh Huong
  Dept of Information Systems
  School of Information and Communication Technology - HUST

Outline
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - breadth-first search
  - depth-first search
  - depth-limited search
  - iterative deepening depth-first search

Problem-solving agents

```java
function SIMPLE-PROBLEM-SOLVING-AGENT(state, problem)
    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, problem)
    if seq is empty then do
        goal = FORMULATE-COAL(state, problem)
        problem = FORMULATE-PROBLEM(state, goal)
        seq = Succeed(problem)
    action = FIRST(seq)
    seq = REST(seq)
    return action
```

Example 1: Route Planning

- Performance: Get from Arad to Bucharest as quickly as possible
- Environment: The map, with cities, roads, and guaranteed travel times
- Actions: Travel a road between adjacent cities
Example 2: Finding letters

- Replace letters by numbers from 0 to 9 such as no different letter is replaced by the same number and satisfying the following constraint:

<table>
<thead>
<tr>
<th>SEND</th>
<th>CROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ MORE</td>
<td>+ ROADS</td>
</tr>
<tr>
<td>MONEY</td>
<td>DANGER</td>
</tr>
</tbody>
</table>

Example 3: Pouring water

- Given 2 containers A(m litres), B(n litres). Finding a method to measure k litres (k ≤ max(m, n)) by 2 containers A, B and a container C
- Actions (how):
  - C → A; C → B; A → B; A → C; B → A; B → C
- Conditions: no overflow, pouring all water
- Eg: m = 5, n = 6, k = 2 (what)
- Mathematical model:
  - (x, y) → (x', y')
  - A B A B

Example 4: The 8-puzzle

- In a table size n*n, each cell contains a number in the range of 1 to n^2 - 1 so that no two cells having the same value. There is only one empty cell. Starting from a certain state of these numbers in the table, move the empty cell to the right, left, up, down to get the following table:

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 5: Hanoi tower

- Given 3 poles 1,2,3. At the beginning, there are n discs on the first pole, with their sizes increasing from top to bottom. Find a way to move these disks to the pole 3 such as:
  - Only one disk is moved each time.
  - On each pole, there is no large disk laying above a small disk.
Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave → search, execution
- Unknown state space → exploration problem

Search Problem Definition

A problem is defined by four items:

1. initial state: e.g., Arad
2. actions or successor function \( S(x) \) = set of action-state pairs
   - e.g., \( S(\text{Arad}) = \{ \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \}, ... \} \)
3. goal test, can be
   - explicit, e.g., \( x = \text{Bucharest} \)
   - 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

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

Search tree

- Search trees:
  - Represent the branching paths through a state graph.
  - Usually much larger than the state graph.
  - Can a finite state graph give an infinite search tree?
Search space of the game Tic-Tac-Toe

Tree and graph
- B is parent of C
- C is child of B
- A is ancestor of C
- C is descendant of A

Convert from search graph to search tree

- We can turn graph search problems into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)

Tree search algorithms

- Basic idea:
  - simulated exploration of state space by generating successors of already-explored states

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
Implementation: general tree search

```plaintext
function Tree-Search(problem, fringe) returns a solution or failure
  fringe ← Enqueue(Make-Node(INITIAL-STATE(problem)), fringe)
  loop do
    if fringe is empty then return failure
    node ← Remove-Front(fringe)
    if Goal-Test(problem(Solve(node))) then return Solution(node)
    fringe ← InheritAll(Expand(node, problem), fringe)

function Expand(node, problem) returns a set of nodes
  successors ← the empty set
  for each action result in SuccessorFn(problem)(problem[node]) do
    s ← a new Node
    Parent[s] ← node, Action[s] ← action, State[s] ← result
    Path-Cost[s] ← Path-Cost[node] + 1
    Depth[s] ← Depth[node] + 1
    add s to successors
  return 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 `SuccessorFn` 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?
  - **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 \( \infty \))
Uninformed search strategies

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

- Breadth-first search
  - Expand shallowest unexpanded node
  - fringe = queue (FIFO)

- Depth-first search
  - Expand deepest unexpanded node
  - fringe = stack (LIFO)

- Depth-limited search: depth-first search with depth limit
- Iterative deepening search

Breadth-first search

- Expand shallowest unexpanded node

Breadth-first search (con’t)

- Complete? Yes (if \( b \) is finite)
- Time? \( 1+b+b^2+b^3+\ldots+b^{d-1} = O(b^{d-1}) \)
- Space? \( O(b^{d-1}) \) (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
Depth-first search (con’t)

- **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?** \( O(bm) \), i.e., linear space!
- **Optimal?** No

Depth-limited search

- Depth-first search can get stuck on infinite path when a different choice would lead to a solution
  ⇒ Depth-limited search = depth-first search with depth limit \( l \), i.e., nodes at depth \( l \) have no successors

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns solution/fail/cutoff
    Recursive-DLS(Make-Node(INITIAL-STATE(problem)), problem, level)
```

```
function Recursive-DLS(node, problem, limit) returns solution/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

- Problem with depth-limited search: if the shallowest goal is beyond the depth limit, no solution is found.
  ⇒ Iterative deepening search:
  1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
  2. If “1” failed, do a DFS which only searches paths of length 2 or less.
  3. If “2” failed, do a DFS which only searches paths of length 3 or less.
  4. .... and so on.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail
    inputs: problem, a problem
    for depth ← 0 to \infty do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result # cutoff then return result
```
Iterative deepening search (con’t)

- 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 \( b \):
  \[ N_{IDS} = (d+1)b^0 + db^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d \]

- 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 = \((123,456 - 111,111)/111,111 = 11\%\)

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
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First Limitized</th>
<th>Depth-First</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>$O(b^{d+1})$</td>
<td>$O(b^a)$</td>
<td>$O(b^a)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>