Problem solving and search
Outline

♦ Problem-solving agents
♦ Problem types
♦ Problem formulation
♦ Search algorithms
♦ Summary
Problem-solving agents

Restricted form of general agent:

```
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.
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")
A problem is defined by four items:

**initial state**    e.g., “at Arad”

**successor function** \( S(x) = \) set of action–state pairs
- e.g., \( S(Arad) = \{⟨Arad \rightarrow Zerind, Zerind⟩, \ldots⟩ \)

**goal test**, can be
- **explicit**, e.g., \( x = “at Bucharest” \)
- **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
Tree search algorithms

Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding 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
**end**
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 $\infty$)
Search strategies

Uninformed strategies

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterativedeepening search
## 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>Yes, if $l \geq d$</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^{\lceil C^*/\epsilon \rceil}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{\lceil C^*/\epsilon \rceil}$</td>
<td>$bm$</td>
<td>$bl$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
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
Summary of uninformed searches

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.
Summary of informed searches

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
Exercises

♦ Define in your own words the following terms: state, state space, search tree, search node, goal, action, successor action, and branching factor.

♦ The heuristic path algorithm is a best-first search in which the objective function is \( f(n) = (2 - w)g(n) + wh(n) \). For what values of \( w \) is this algorithm guaranteed to be optimal? (You may assume that \( h \) is admissible.) What kind of search does this perform when \( w = 0 \)? when \( w = 1 \)? when \( w = 2 \)?
1. A **state** is a situation that an agent can find itself in. We distinguish two types of states: world states (the actual concrete situations in the real world) and representational states (the abstract descriptions of the real world that are used by the agent in deliberating about what to do). A **state space** is a graph whose nodes are the set of all states, and whose links are actions that transform one state into another.

A **search tree** is a tree (a graph with no undirected loops) in which the root node is the start state and the set of children for each node consists of the states reachable by taking any action.

A **search node** is a node in the search tree.

A **goal** is a state that the agent is trying to reach. An action is something that the agent can choose to do.

A **successor function** described the agents options: given a state, it returns a set of (action, state) pairs, where each state is the state reachable by taking
the action.

The **branching factor** in a search tree is the number of actions available to the agent.

2. \( w = 0 \) gives \( f(n) = 2g(n) \). This behaves exactly like uniform-cost search, the factor of two makes no difference in the ordering of the nodes. \( w = 1 \) gives A* search. \( w = 2 \) gives \( f(n) = 2h(n) \), i.e., greedy best-first search. We also have

\[
    f(n) = (2 - w)[g(n) + \frac{w}{2-w}h(n)]
\]

which behaves exactly like A* search with a heuristic \( \frac{w}{2-w}h(n) \). For \( w \leq 1 \), this is always less than \( h(n) \) and hence admissible, provided \( h(n) \) is itself admissible.