Basic Search

Philipp Koehn

22 September 2015
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
problem-solving agents
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.
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
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?*
**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**, 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?
[Right, if dirt then Suck]
problem formulation
Single-State Problem Formulation

- A **problem** is defined by four items:
  - **initial state** e.g., “at Arad”
  - **successor function** $S(x) =$ set of action–state pairs
    e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \ldots \}$
  - **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
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!
Example: Vacuum World State Space Graph

- states?
- actions?
- goal test?
- path cost?
Example: Vacuum World State Space Graph

states?: integer dirt and robot locations (ignore dirt amounts etc.)
actions?
goal test?
path cost?
Example: Vacuum World State Space Graph

states?: integer dirt and robot locations (ignore dirt amounts etc.)
actions?: Left, Right, Suck, NoOp
goal test?
path cost?
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: 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?: 1 per action (0 for NoOp)
Example: The 8-Puzzle

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

**states?**: integer locations of tiles (ignore intermediate positions)

**actions?**

**goal test?**

**path cost?**
Example: The 8-Puzzle

**states?**: integer locations of tiles (ignore intermediate positions)

**actions?**: move blank left, right, up, down (ignore unjamming etc.)

**goal test?**

**path cost?**
Example: The 8-Puzzle

states?: integer locations of tiles (ignore intermediate positions)
actions?: move blank left, right, up, down (ignore unjamming etc.)
goal test?: = goal state (given)
path cost?
Example: The 8-Puzzle

- **states?**: integer locations of tiles (ignore intermediate positions)
- **actions?**: move blank left, right, up, down (ignore unjamming etc.)
- **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 **with no robot included!**

**path cost?**: time to execute
tree search
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
end
```
Tree Search Example
Tree Search example
Tree Search Example
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)
```

```plaintext
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
return successors
```

**Philipp Koehn Artificial Intelligence: Basic Search 22 September 2015**
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 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
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?** 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 → 24hrs = 8640GB.
uniform cost search
Uniform-Cost Search

• Expand least-cost unexpanded node

• **Implementation**: 
  \(\text{fringe} = \text{queue ordered by path cost, lowest first}\)

• Equivalent to breadth-first if step costs all equal

• Properties
  
  – **Complete?** Yes, if step cost \(\geq \epsilon\)
  
  – **Time?** \# of nodes with \(g \leq \text{cost of optimal solution}\), \(O(b^{\lceil C^*/\epsilon \rceil})\)
    where \(C^*\) is the cost of the optimal solution

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

  – **Optimal?** Yes—nodes expanded in increasing order of \(g(n)\)
depth first search
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 Diagram]
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:**
  
  \( \text{fringe} = \text{LIFO queue, i.e., put successors at front} \)
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
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? $O(bm)$, i.e., linear space!

- Optimal? No
iterative deepening
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
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
Iterative Deepening Search $l = 0$

Limit = 0
Iterative Deepening Search $l = 1$
Iterative Deepening Search \( l = 2 \)
Iterative Deepening Search $l = 3$
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:
  \[
  N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
  N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 999,990 = 1,111,100
  \]

- 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
<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^d)</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>
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one.
Graph Search

function \textsc{Graph-Search}(\textit{problem}, \textit{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}(\textsc{Initial-State}[\textit{problem}]), \textit{fringe})
\end{itemize}

\textbf{loop do}

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

\textbf{end}
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