Search I

Tuomas Sandholm
Carnegie Mellon University
Computer Science Department

[Read Russell & Norvig Chapter 3]
Search I

Goal-based agent (problem solving agent)

Goal formulation (from preferences). Romania example, (Arad → Bucharest)

Problem formulation: deciding what actions & state to consider. E.g. not “move leg 2 degrees right.”

No map vs. Map
physical search  deliberative search

Figure 3.3  A simplified road map of Romania.
“Formulate, Search, Execute” (sometimes interleave search & execution)

For now we assume
  full observability = known state
  known effects of actions

Data type *problem*
  Initial state (perhaps an abstract characterization) \(\leftrightarrow\) partial observability (set)
  Operators
  Goal-test (maybe many goals)
  Path-cost-function

Knowledge representation
  Mutilated chess board
Search I

Example problems demonstrated in terms of the problem definition.

I. 8-puzzle (general class is NP-complete)

How to model operators? (moving tiles vs. blank)
Path cost = 1
Search I

II. 8-queens (general class has efficient solution)  path cost = 0

Incremental formulation:
(constructive search)
**States:** any arrangement of 0 to 8 queens on board  
**Ops:** add a queen to any square  
# sequences = 64^8

Complete State formulation:
(iterative improvement)
**States:** arrangement of 8 queens, 1 in each column  
**Ops:** move any attacked queen to another square in the same column

Improved incremental formulation:
**States:** any arrangement of 0 to 8 queens on board *with none attacked*  
**Ops:** place a queen in the left-most empty column s.t. it is not attacked by any other queen  
# sequences = 2057

Almost a solution to the 8-queen problem:
Search I

III. Rubik’ Cube $\sim 10^{19}$ states

IV. Crypt arithmetic
   
   \[
   \begin{array}{ccc}
   & \text{FORTY} & 29786 \\
   + & \text{TEN} & + & 850 \\
   + & \text{TEN} & + & 850 \\
   \hline
   & \text{SIXTY} & 31486 \\
   \end{array}
   \]

V. Real world problems
   1. Routing (robots, vehicles, salesman)
   2. Scheduling & sequencing
   3. Layout (VLSI, Advertisement, Mobile phone link stations)
   4. Winner determination in combinatorial auctions
   …
Data type *node*

- State
- Parent-node
- Operator
- Depth
- Path-cost

Fringe = frontier = open (as queue)
Partial search tree for route finding from Arad to Bucharest.
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

nodes — MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))

loop do
    if nodes is empty then return failure
    node — REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes — QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end

The general search algorithm. (Note that QUEUING-FN is a variable whose value will be a function.)
Goodness of a search strategy

- Completeness
- Time complexity
- Space complexity
- Optimality of the solution found
  (path cost = domain cost)
- Total cost = domain cost + search cost
Uninformed vs. informed search

Can only distinguish goal states from non-goal state
Breadth-First Search

function BREADTH-FIRST-SEARCH (problem) returns a solution or failure

return GENERAL-SEARCH (problem, ENQUEUE-AT-END)

Breadth-first search tree after 0,1,2 and 3 node expansions
Breadth-First Search ...

Max $1 + b + b^2 + \ldots + b^d$ nodes ($d$ is the depth of the shallowest goal)
- Complete
- Exponential time & memory $O(b^d)$
- Finds optimum if path-cost is a non-decreasing function of the depth of the node.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 seconds</td>
<td>11 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 minutes</td>
<td>111 megabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11,111 terabytes</td>
</tr>
</tbody>
</table>

Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.
Uniform-Cost Search

Insert nodes onto open list in ascending order of $g(h)$.

Finds optimum if the cost of a path never decreases as we go along the path.

$g(\text{SUCCESSORS}(n)) \geq g(n)$

$\leq$ Operator costs $\geq 0$

If this does not hold, nothing but an exhaustive search will find the optimal solution.
**Depth-First Search**

**function** DEPTH-FIRST-SEARCH *(problem)* **returns** a solution or failure

GENERAL-SEARCH *(problem, ENQUEUE-AT-FRONT)*

- Time $O(b^m)$ (m is the max depth in the space)
- Space $O(bm)$
- Not complete (m may be $\infty$)
  - E.g. grid search in one direction
- Not optimal

Alternatively can use a recursive implementation.
Depth-Limited Search

- Depth limit in the algorithm, or
- Operators that incorporate a depth limit

$L = \text{depth limit}$

Complete if $L \geq d$ (d is the depth of the shallowest goal)

Not optimal (even if one continues the search after the first solution has been found, because an optimal solution may not be within the depth limit $L$)

$O(b^L)$ time

$O(bL)$ space

Diameter of a search space?
Iterative Deepening Search

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure

Breadth first search:
1 + b + b^2 + ... + b^{d-1} + b^d
E.g. b=10, d=5: 1+10+100+1,000+10,000+100,000 = 111,111

Iterative deepening search:
(d+1)*1 + (d)*b + (d-1)*b^2 + ... + 2b^{d-1} + 1b^d
E.g. 6+50+400+3000+20,000+100,000 = 123,456
Complete, Optimal, O(b^d) time, O(bd) space
Preferred when search space is large & depth of (optimal) solution is unknown
Iterative Deepening Search...

Four iterations of iterative deepening search on a binary tree.
Iterative Deepening Search…

If branching factor is large, most of the work is done at the deepest level of search, so iterative deepening does not cost much relatively speaking.
Bi-Directional Search

A schematic view of a bidirectional breadth-first search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Time $O(b^{d/2})$
Bi-Directional Search ...

Need to have operators that calculate predecessors.
What if there are multiple goals?
• If there is an explicit list of goal states, then we can apply a predecessor function to the state set just as we apply the successors function in multiple-state forward search.
• If there is only a description of the goal set, it MAY be possible to figure out the possible descriptions of “sets of states that would generate the goal set”.

Efficient way to check when searches meet: hash table
- 1-2 step issue if only one side stored in the table
Decide what kind of search (e.g. breadth-first) to use in each half.

Optimal, complete, $O(b^{d/2})$ time. $O(b^{d/2})$ space (even with iterative deepening) because the nodes of at least one of the searches have to be stored to check matches
# Time, Space, Optimal, Complete?

<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>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Space Optimal? Complete?</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
<td>$b^{d/2}$</td>
</tr>
<tr>
<td></td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$bm$</td>
<td>$bl$</td>
<td>$bd$</td>
<td>$b^{d/2}$</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Evaluation of search strategies. $b$ is the branching factor; $d$ is the depth of solution; $m$ is the maximum depth of the search tree; $l$ is the depth limit.

$b = \text{branching factor}$
$d = \text{depth of shallowest goal state}$
$m = \text{depth of the search space}$
$l = \text{depth limit of the algorithm}$
Avoiding repeated states

A state space that generates an exponentially larger search tree. The left-hand side shows the state space, in which there are two possible actions leading from A to B, two from B to C, and so on. The right-hand side shows the corresponding search tree.

- Do not return to the state you just came from. Have the expand function (or the operator set) refuse to generate any successor that is the same state as the node’s parent.
- Do not create paths with cycles in them. Have the expand function (or the operator set) refuse to generate any successor of a node that is the same as any of the node’s ancestors.
- Do not generate any state that was ever generated before. This requires every state that is generated to be kept in memory, resulting in a space complexity of $O(b^d)$, potentially. It is better to think of this as $O(s)$, where $s$ is the number of states in the entire state space.

To implement this last option, search algorithms often make use of a hash table that stores all the nodes that are generated. This makes checking for repeated states reasonably efficient. The trade-off between the cost of storing and checking and the cost of extra search depends on the problem: the “loopier” the state space, the more likely it is that checking will pay off.

With loops, the search tree may even become infinite