Search

Search is finding a sequence of actions/operators that achieve a goal from an initial state.

Idea: An agent that can predict the results of its actions can return better actions.

Assumptions: Operators are deterministic. Relevant features of initial state and goal can be perceived.

Search Agent

function SEARCH-AGENT()
    static: goal, actions
    action-sequence ← null
    loop
        percept ← perceive environment
        state ← DETERMINE-STATE(percept)
        exit if state satisfies goal
        if action-sequence does not go from state to goal
            then find action-sequence from state to goal
                fail if no action-sequence is found
            action ← remove first action from action-sequence
            perform action on environment
Search Problems

A search problem specifies:

*Initial state.* The relevant features at the start.

*Operators, successor function, search graph.*
  Operators: possible actions.
  Successor function: all states within one op.
  Search graph: states map to vertices/nodes.

*State space.* States reachable from initial state.

*Goal test.* Determines if a state is a goal state.

*Path cost.* The cost of performing the actions.

A solution is a path from the initial state to a goal state.

Example Search Problems

- 8-puzzle, 15-puzzle
- \( n \)-queens
- Cryptarithmetic
  \( \text{TWO} + \text{TWO} = \text{FOUR} \)
- Missionaries and cannibals
- Rubik’s cube
- Blocks-world
- Tower of Hanoi
- Monkey and bananas
- Airline booking
- Traveling salesman
- Device assembly
- Calculus, algebra problems
Characteristics of Search Algorithms

A search algorithm needs the following inputs:

*Initial state.*

*Successors/Expand Function.* Return the states that can be reached from a given state by applying an operator.

*Goal test function.* Determine if a given state is a goal state.

Optional: *Path/edge cost function.* Determine the cost of a given path/edge.

Optional: *Resource limit.* When to give up.

A search algorithm needs to keep track of:

*Fringe, frontier.* States that have been generated, but not yet processed.

Optional: *Closed states.* Processed states.

Search algorithms can be characterized by how the fringe is managed.
Breadth-First Search

function BFS(initial, Expand, Goal)
    q ← New-Queue()
    Enqueue(initial, q)
    while q is not empty
        do current ← Dequeue(q)
            if Goal(current) then return solution
            for each next in Expand(current)
                do Enqueue(next, q)
    return failure

Depth-First Search

function DFS(current, Expand, Goal, bound)
    if Goal(current) then return solution
    if bound = 0 then return failure
    for each next in Expand(current)
        do DFS(next, Expand, Goal, bound − 1)
            if DFS succeeds then return solution
    return failure
Iterative Deepening

function IDS(initial, EXPAND, GOAL)
  for depth ← 0 to depth of search graph
    do DFS(initial, EXPAND, GOAL, depth)
      if DFS succeeds then return solution
  return failure

More Search Algorithms

Uniform Cost (Dijkstra’s Algorithm):
Select least costly state from fringe.
Implement: modify BFS with priority queue.

Bidirectional Search:
Search from both initial state and goal state.
Implement: dual BFS with efficient equality.
Assumes single (or few) goal state(s).
Assumes operators can be inverted.
Search: Performance

Assume the search space is *tree-structured* with:

\( b = \text{branching factor}, \)

\( d = \text{depth of closest solution}, \)

\( m = \text{maximum depth of search graph}, \)

\( l = \text{depth bound} \)

<table>
<thead>
<tr>
<th>Search Method</th>
<th>States Visited</th>
<th>States Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First</td>
<td>( O(b^d) )</td>
<td>( O(b^d) )</td>
</tr>
<tr>
<td>Depth-First</td>
<td>( O(b^m) )</td>
<td>( O(bm) )</td>
</tr>
<tr>
<td>DFS (bounded)</td>
<td>( O(b^l) )</td>
<td>( O(bl) )</td>
</tr>
<tr>
<td>Iterative Deep.</td>
<td>( O(b^d) )</td>
<td>( O(bd) )</td>
</tr>
</tbody>
</table>

BFS and IDS return optimal (shortest) solution.

Ratio of states visited is

\[
\frac{\text{IDS}}{\text{BFS}} \approx \frac{b}{b - 1}
\]
Other Issues

Repeated states:
Do not want to search the same state twice.
In increasing effectiveness and overhead:
1. Avoid going back to a state’s parent, or
2. Avoid circular paths, or
3. Avoid any state previously generated.

Constraint satisfaction search:
Assign values to variables satisfying constraints.
Choose a variable and branch on possible values.