function GENERAL-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

Essence of search: which node to expand first?

→ search strategy

A strategy is defined by picking the order of node expansion
Types of Search

Uninformed: use only information available in problem definition

Heuristic: exploits some knowledge of the domain

Uninformed search strategies

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening depth-first search
6. Bidirectional search

Search strategies

Criteria for evaluating search:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated/expanded
3. Space complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Time/space complexity measured in terms of:

- $b$: maximum branching factor of the search tree
- $d$: depth of the least-cost solution
- $m$: maximum depth of the search space (may be $\infty$)
Breadth-first search (1)

→ Expand root node
→ Expand all children of root
→ Expand each child of root
→ Expand successors of each child of root, etc.

→ Expands nodes at depth $d$ before nodes at depth $d + 1$
→ Systematically considers all paths length 1, then length 2, etc.
→ Implement: put successors at end of queue... FIFO

Breadth-first search (2)
Breadth-first search (3)

→ One solution?
→ Many solutions? Finds shallowest goal first

1. Complete? Yes, if $b$ is finite

2. Optimal? provided cost increases monotonically with depth, not in general

3. Time? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$

   $O(b^{d+1}) \begin{cases} \text{branching factor } b \\ \text{depth } d \end{cases}$

4. Space? same, $O(b^{d+1})$, keeps every node in memory, big problem
can easily generate nodes at $10\text{MB/sec}$ so $24\text{hrs} = 860\text{GB}$

Uniform-cost search (I)

→ Breadth-first does not consider path cost $g(x)$
→ Uniform-cost expands first lowest-cost node on the fringe
→ Implement: sort queue in decreasing cost order

When $g(x) = \text{Depth}(x) \rightarrow$ Breadth-first $\equiv$ Uniform-cost
**Uniform-cost search (2)**

1. Complete?
   Yes, if cost $\geq \epsilon$

2. Optimal?
   If the cost is a monotonically increasing function
   When cost is added up along path, an operator's cost .......?

3. Time?
   # of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\epsilon]})$
   where $C^*$ is the cost of the optimal solution

4. Space?
   # of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\epsilon]})$

---

**Depth-first search (I)**

→ Expands nodes at deepest level in tree
→ When dead-end, goes back to shallower levels
→ Implement: put successors at front of queue.. LIFO

→ Little memory: path and unexpanded nodes
For $b$: branching factor, $m$: maximum depth, space .......?
Depth-first search (2)

Depth-first search (3)

Time complexity:

We may need to expand all paths, \(O(b^m)\)
When there are many solutions, DFS may be quicker than BFS
When \(m\) is big, much larger than \(d\), \(\infty\) (deep, loops), .. troubles
→ Major drawback of DFS: going deep where there is no solution..

Properties:

1. Complete? No in infinite-spaces, complete in finite spaces
2. Optimal?
3. Time? \(O(b^m)\) \hspace{1cm} Woow..
   \hspace{1cm} terrible if \(m\) is much larger than \(d\), but if solutions are dense, may be much faster than breadth-first
4. Space? \(O(bm)\), linear! \hspace{1cm} Woow..
**Depth-limited search (I)**

→ DFS is going too deep, put a threshold on depth!
   For instance, 20 cities on map for Romania, any node deeper than 19 is cycling. Don’t expand deeper!
→ Implement: nodes at depth $l$ have no successor

**Properties:**

1. Complete?
2. Optimal?
3. Time? (given $l$ depth limit)
4. Space? (given $l$ depth limit)

**Problem:** how to choose $l$?

---

**Iterative-deepening search (I)**

→ DLS with depth = 0
→ DLS with depth = 1
→ DLS with depth = 2
→ DLS with depth = 3...

→ Combines benefits of DFS and BFS
Iterative-deepening search (2)

Limit = 0  \[
\bullet
\]

Limit = 1  \[
\bullet
\]

Limit = 2  \[
\bullet
\]

Limit = 3  \[
\bullet
\]

→ combines benefits of DFS and BFS

Properties:

1. Time? \((d + 1).b^0 + (d).b + (d - 1).b^2 + \ldots + 1.b^d = O(b^d)\)
2. Space? \(O(bd)\), like DFS
3. Complete? like BFS
4. Optimal? like BFS (if step cost = 1)
Iterative-deepening search (4)

→ Some nodes are expanded several times, wasteful?
\[ N(\text{BFS}) = b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - d) \]
\[ N(\text{IDS}) = (d)b + (d - 1)b^2 + \ldots + (1)b^d \]

Numerical comparison for \( b = 10 \) and \( d = 5 \):
\[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]
\[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \]
→ IDS is preferred when search space is large and depth unknown

Bidirectional search (I)

→ Given initial state and the goal state, start search from both ends and meet in the middle

→ Assume same \( b \) branching factor, \( \exists \) solution at depth \( d \), time:
\[ O(2b^{d/2}) = O(b^{d/2}) \]
\( b = 10, d = 6 \), DFS= 1,111,111 nodes, BDS=2,222 nodes!
**Bidirectional search (2)**

**In practice:**

- Need to define predecessor operators to search backwards
  If operator are invertible, no problem
- What if \( \exists \) many goals (set state)?
  do as for multiple-state search
- need to check the 2 fringes to see how they match
  need to check whether any node in one space appears in the other space (use hashing)
  need to keep all nodes in a half in memory \( O(b^{d/2}) \)
- What kind of search in each half space?

---

**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^{[C^* / \epsilon]} )</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^{[C^* / \epsilon]} )</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>

\( b \) branching factor
\( d \) solution depth
\( m \) maximum depth of tree
\( l \) depth limit
**Loops:** Avoid repeated states (I)

Avoid expanding states that have already been visited

Valid for both infinite and finite trees

\[
\begin{aligned}
& \begin{cases}
  m & \text{maximum depth} \\
  m + 1 & \text{states} \\
  2^m & \text{possible branches (paths)}
\end{cases}
\end{aligned}
\]

**Loops:** (2)

Keep nodes in two lists:

\[
\begin{aligned}
\text{Open list: Fringe} & \\
\text{Closed list: Leaf and expanded nodes}
\end{aligned}
\]

Discard a current node that matches a node in the closed list

Tree-Search $\rightarrow$ Graph-Search

**Issues:**

1. Implementation: hash table, access is constant time
   
   Trade-off cost of storing + checking vs. cost of searching

2. Losing optimality
   
   When new path is cheaper/shorter of the one stored

3. BFS and IDS now require exponential storage
Summary

Path: sequence of actions leading from one state to another

Partial solution: a path from an initial state to another state

Search: develop a sets of partial solutions
  - Search tree & its components (node, root, leaves, fringe)
  - Data structure for a search node
  - Search space vs. state space
  - Node expansion, queue order
  - Search types: uninformed vs. heuristic
  - 6 uninformed search strategies
  - 4 criteria for evaluating & comparing search strategies

Searching with partial information (I)

So far, we assumed:
  - Environment fully observable
  - Environment deterministic
  - Agent knows effects of actions

Thus, agent
  - always knows where it is
  - can compute state where it will be after a sequence of actions

What happens when knowledge about states and actions is incomplete?
Searching with partial information (2)

Incompleteness yields 3 types of problems:

- Sensorless (conformant) problems
- Contingency problems
- Exploration problems

Sensorless problems (conformant)

- Environment not observable, no percepts
- Agent does not know in which exact state it is
  - agent may be in one of more possible initial states
  - an action may lead to one or more possible successor states
Contingency problems

- environment partially observable or actions are uncertain
- agent’s percepts provide new input after each action, a contingency to plan for
- Adversarial problems: uncertainty caused by action of other agents

Exploration problems

- States and actions of the environment are unknown
- Agent must act to discover them
- Extreme case of contingency problem
Sensorless problems (1)

Vacuum cleaner: no sensors, but agent knows effects of actions

Agent may be in any state \( \{1, 2, 3, 4, 5, 6, 7, 8\} \)
- \([\text{Right}]\) always ends in \( \{2, 4, 6, 8\} \)
- \([\text{Right}, \text{Suck}]\) always ends in \( \{4, 8\} \)
- \([\text{Right}, \text{Suck}, \text{Left}, \text{Suck}]\) always works, coerces the world into 7

Sensorless problems (2)

Environment not (fully) observable:
- Agent must think about sets of states,
- Agent has a belief state (set of possible states)

Environment fully observable: 1 belief state has 1 state

Solving sensorless problems: search in space of beliefs
- initial state is a belief state (all possible states)
- actions map 1 belief state into another
- belief state is union of applying action to each state in initial belief state
- goal is reached when all states in belief state are goal states
Sensorless problems (2)

vacuum cleaner: 12 belief states

In general:
8 states, $2^8$ possible belief states
$S$ states, $2^S$ possible belief states

Sensorless problems (3)

So far assumed deterministic environment
Approach/results hold for nondeterministic environment

Example: Murphy’s law, Suck sometimes deposits dirt on carpet but only if there is no dirt there already

- [Suck] applied to State 4 leads to \{2, 4\}
- [Suck] applied to \{1, 2, 3, 4, 5, 6, 7, 8\} leads to ...
- Problem is unsolvable (Exercise 3.18)!!
  Agent cannot tell whether state is dirty and cannot predict whether Suck is going to make it dirty or clean
Contingency problems (I)

Environment partially observable or actions are uncertain

When agent can get some information:

- about environment
- from sensors
- after acting

Solution to a contingency problem is not a path, but a tree

—→ branches are selected depending on percepts

Contingency problems (2)

Example: vacuum cleaner

- has ‘local dirt’ sensor, no ‘remote dirt’ sensor
- has location sensor
- Murphy’s law

Now,

- Agent perceives \([L, Dirty]\), thinks in state \(\{1, 3\}\)
- Action \([Suck]\) leads to \(\{5, 7\}\)
- Action \([Suck, Right]\) leads to \(\{6, 8\}\)
- Action \([Suck, Right, Suck]\) leads to \(\{8, 6\}\)

Plan can succeed \(8\), or fail \(6\)

Thus, action \([Suck, Right, if[R, Dirty]\ then Suck]\) leads to \(\{8, 6\}\)

Solution is a tree
Contingency problems (3)

Example: vacuum cleaner
- has ‘local dirt’ sensor and ‘remote dirt’ sensor
- has location sensor (fully observable)
- Murphy’s law

Solution is a sequence of actions
Agent can proceed...

Contingency problems (4)

In general, agent
- acts before having a guaranteed plan (solution is a tree)
- needs to consider every possibility that might arise
  \[\rightarrow\text{ may be an overkill}\]

It is (sometimes) necessary to start acting, and deal with contingencies as they arise
- \(\rightarrow\text{ Interleave Search and Execution}\)
- \(\rightarrow\text{ Useful for game playing and exploration problems}\)