Search Problems

Professor Chris Callison-Burch
Example Search Problems

AIMA 3.1-3.3
Reflex Agents

A simple reflex agent is one that selects an action based only on the **current percept**. It **ignores** the rest of the **percept history**.
Problem-Solving Agent

A problem-solving agent must **plan ahead**.

The computational process that it undertakes is called **search**.

It will consider a **sequence of actions** that form a **path** to a **goal state**.

Such a sequence is called a **solution**.
Impact of Task Environments

The properties of the task environments change the types of solutions that we need.

If an environment is:

- **Fully observable**
- **Deterministic**
- **Known environment**

The solution to any problem in such an environment is a fixed sequence of actions.

In environments that are

- **Partially observable** or
- **Nondeterministic**

The solution must recommend different future actions depending on the what percepts it receives. This could be in the form of a *branching strategy*. 
Example search problem: Holiday in Romania

You are here

You need to be here
Example search problem: Holiday in Romania
Holiday in Romania

On holiday in Romania; currently in Arad
  • Flight leaves tomorrow from Bucharest

Formulate \textit{goal}
  • Be in Bucharest

Formulate \textit{search problem}
  • States: various cities
  • Actions: drive between cities
  • Performance measure: minimize travel time / distance

Find \textit{solution}
  • Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, ...
Example search problem: 8-puzzle

Formulate goal
- Pieces to end up in order as shown...

Formulate search problem
- **States**: configurations of the puzzle (9! configurations)
- **Actions**: Move one of the movable pieces (≤4 possible)
- **Performance measure**: minimize total moves

Find solution
- Sequence of pieces moved: 3,1,6,3,1,...
Defining Search Problems
Formal Definition

1. **States**: a set $S$
2. An *initial state* $s_i \in S$
3. **Actions**: a set $A$
   
   $\forall s \text{ Actions}(s) = \text{the set of actions that can be executed in } s, \text{ that are applicable in } s.$
4. **Transition Model**: $\forall s \forall a \in \text{Actions}(s) \text{ Result}(s, a) \rightarrow s_r$
   
   $s_r$ is called a *successor* of $s$

   $\{s_i\} \cup \text{Successors}(s_i)^* = \text{state space}$
5. **Path cost** (*Performance Measure*): Must be additive, e.g. sum of distances, number of actions executed, ...
   
   $c(x,a,y)$ is the step cost, assumed $\geq 0$
   
   • (where action $a$ goes from state $x$ to state $y$)
6. **Goal test**: $\text{Goal}(s)$
   
   Can be implicit, e.g. $\text{checkmate}(s)$
   
   $s$ is a goal state if $\text{Goal}(s)$ is true
Vacuum World

**States:** A state of the world says which objects are in which cells.

In a simple two cell version,
- the agent can be in one cell at a time
- each cell can have dirt or not

2 positions for agent * $2^2$ possibilities for dirt = 8 states.

With $n$ cells, there are $n*2^n$ states.
**Vacuum World**

**States:** A state of the world says which objects are in which cells.

In a simple two cell version,
- the agent can be in one cell at a time
- each cell can have dirt or not

2 positions for agent $\times 2^2$ possibilities for dirt $= 8$ states.

With $n$ cells, there are $n \times 2^n$ states.
Vacuum World

**States:** A state of the world says which objects are in which cells.

In a simple two cell version,

- the agent can be in one cell at a time
- each cell can have dirt or not

2 positions for agent * $2^2$ possibilities for dirt = 8 states.

With $n$ cells, there are $n*2^n$ states.

**Goal states:** States where everything is clean.

One state is designated as the **initial state**
Vacuum World

Actions:
- **Suck**
- **Move Left**
- **Move Right**
- (Move Up)
- (Move Down)

Transition:
Suck – removes dirt
Move – moves in that direction, unless agent hits a wall, in which case it stays put.
Vacuum World

Suck

Right
Left

Right
Left
Suck

Suck

Right
Left
Suck

Action cost:
Uniform (all actions are equal cost)
Vacuum World

Path cost:
Sum of all action costs along a path
Vacuum World

Initial state

Solution:
A path from the initial state to a goal state
Art: Formulating a Search Problem

Decide:

Which properties matter & how to represent
- *Initial State, Goal State, Possible Intermediate States*

Which actions are possible & how to represent
- *Operator Set: Actions and Transition Model*

Which action is next
- *Path Cost Function*

Formulation greatly affects combinatorics of search space and therefore speed of search
Hard subtask: Selecting a state space

Real world is absurdly complex
State space must be abstracted for problem solving

(abstract) **State** = set (equivalence class) of real-world states

(abstract) **Action** = equivalence class of combinations of real-world actions
  - e.g. *Arad → Zerind* represents a complex set of possible routes, detours, rest stops, etc
  - The abstraction is valid if the path between two states is reflected in the real world

Each abstract action should be “easier” than the real problem
Useful Concepts

**State space:** the set of all states reachable from the initial state by *any* sequence of actions

- *When several operators can apply to each state, this gets large very quickly*
- * Might be a proper subset of the set of configurations*

**Path:** a sequence of actions leading from one state $s_j$ to another state $s_k$

**Solution:** a path from the initial state $s_i$ to a state $s_j$ that satisfies the goal test

**Search tree:** a way of representing the paths that a search algorithm has explored. The root is the initial state, leaves of the tree are successor states.

**Frontier:** those states that are available for *expanding* (for applying legal actions to)
Solutions and *Optimal* Solutions

A *solution* is a sequence of *actions* from the *initial state* to a *goal state*.

*Optimal Solution:* A solution is *optimal* if no solution has a lower *path cost*. 
Basic Search Algorithms
Basic search algorithms: *Tree Search*

Generalized algorithm to solve search problems

**Enumerate in some order all possible paths from the initial state**

- Here: search through *explicit tree generation*
  - ROOT= initial state.
  - Nodes in search tree generated through *transition model*
  - Tree search treats different paths to the same node as distinct
Generalized tree search

function TREE-SEARCH(problem, strategy) return a solution or failure

Initialize frontier to the initial state of the problem

do

if the frontier is empty then return failure
choose leaf node for expansion according to strategy & remove from frontier
if node contains goal state then return solution
else expand the node and add resulting nodes to the frontier
States Versus Nodes

A **state** is a (representation of a) **physical configuration**
A **node** is a data structure constituting **part of a search tree**

- Also includes *parent, children, depth, path cost g(x)*
- Here *node* = <*state, parent-node, children, action, path-cost, depth>*

States do not have parents, children, depth or path cost!

The **EXPAND** function
- uses the Actions and Transition Model to create the corresponding states
  - creates new nodes,
  - fills in the various fields
**8-Puzzle Search Tree**

(Nodes show state, parent, children - leaving Action, Cost, Depth Implicit)

(Suppressing useless “backwards” moves)
Problem: Repeated states

Failure to detect *repeated states* can turn a linear problem into an *exponential* one!
Solution: Graph Search!

Graph search

- Simple Mod from tree search: *Check to see if a node has been visited before adding to search queue*
  - must keep track of all possible states (can use a lot of memory)
  - e.g., 8-puzzle problem, we have $9!/2 \approx 182K$ states
**Graph Search vs Tree Search**

**function** `TREE-SEARCH(problem)` **returns** a solution, or failure

initialize the frontier using the initial state of `problem`

**loop do**

  **if** the frontier is empty **then return** failure

  choose a leaf node and remove it from the frontier

  **if** the node contains a goal state **then return** the corresponding solution

  expand the chosen node, adding the resulting nodes to the frontier

**function** `GRAPH-SEARCH(problem)` **returns** a solution, or failure

initialize the frontier using the initial state of `problem`

initialize the explored set to be empty

**loop do**

  **if** the frontier is empty **then return** failure

  choose a leaf node and remove it from the frontier

  **if** the node contains a goal state **then return** the corresponding solution

  **add node to the explored set**

  expand the chosen node, adding the resulting nodes to the frontier **only if not in the frontier of explored set**
Uninformed Search Strategies

AIMA 3.3-3.4
Uninformed search strategies:

AKA “Blind search”
Uses only information available in problem definition

Informally:

Uninformed search: All non-goal nodes in frontier look equally good
Informed search: Some non-goal nodes can be ranked above others.
Search Strategies

Review: **Strategy** = order of tree expansion
  - Implemented by different queue structures (LIFO, FIFO, priority)

Dimensions for evaluation
  - **Completeness**: always find the solution?
  - **Optimality**: finds a least cost solution (lowest path cost) first?
  - **Time complexity**: # of nodes generated *(worst case)*
  - **Space complexity**: # of nodes simultaneously in memory *(worst case)*

Time/space complexity variables
  - \(b\), maximum branching factor of search tree
  - \(d\), depth of the shallowest goal node
  - \(m\), maximum length of any path in the state space (potentially \(\infty\))
Introduction to \textit{space} complexity

You know about:
\begin{itemize}
  \item "Big O" notation
  \item \textit{Time complexity}
\end{itemize}

\textit{Space complexity} is analogous to time complexity

Units of space are arbitrary
\begin{itemize}
  \item Doesn't matter because Big O notation ignores constant multiplicative factors
  \item Plausible Space units:
    \begin{itemize}
      \item One Memory word
      \item Size of any fixed size data structure
        \begin{itemize}
          \item For example, size of fixed size node in search tree
        \end{itemize}
    \end{itemize}
\end{itemize}
Breadth-First Search and Depth-First Search
Breadth-first search

Idea:
• Expand *shallowest* unexpanded node

Implementation:
• *frontier* is FIFO (First-In-First-Out) Queue:
  • Put successors at the *end* of *frontier* successor list.
Breadth-first search

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

node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then return node

frontier ← a FIFO queue, with node as an element
reached ← {problem.INITIAL}

while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then
            add s to reached
            add child to frontier
    
return failure
```
Breadth-first search

```
function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(State=s’, Parent=node, Action=action, Path-Cost=cost)
```

Node data structure contains variables like the state, a pointer to its parent node, the action that was used to create this state, and the path cost.

The Python yield keyword means that we don’t have to pre-compute a list of all successors.
Breadth-first search

Function **BREADTH-FIRST-SEARCH**(problem) returns a solution node or failure

\[\text{node} \leftarrow \text{NODE}(\text{problem.INITIAL})\]

\[\text{if problem.IS-GOAL(node.STATE)} \text{ then return node}\]

\[\text{frontier} \leftarrow \text{a FIFO queue, with node as an element}\]

\[\text{reached} \leftarrow \{\text{problem.INITIAL}\}\]

\[\text{while not IS-EMPTY(frontier) do}\]

\[\text{node} \leftarrow \text{POP(frontier)}\]

\[\text{for each child in EXPAND(problem, node) do}\]

\[s \leftarrow \text{child.STATE}\]

\[\text{if problem.IS-GOAL(s) then return child}\]

\[\text{if s is not in reached then}\]

\[\text{add s to reached}\]

\[\text{add child to frontier}\]

\[\text{return failure}\]

Subtle: Node inserted into queue only after testing to see if it is a goal state
Properties of breadth-first search

**Complete?**
Yes (if $b$ is finite)

**Optimal?**
Yes, if cost = 1 per step
(not optimal in general)

**Time Complexity?**
$1 + b + b^2 + b^3 + ... + b^d = O(b^d)$

**Space Complexity?**
$O(b^d)$ (keeps every node in memory)

$b$: maximum branching factor of search tree
$d$: depth of the least cost solution
$m$: maximum depth of the state space ($\infty$)
Exponential Space (and time) Is Not Good...

- Exponential complexity uninformed search problems cannot be solved for any but the smallest instances.
- *(Memory requirements are a bigger problem than execution time.)*

<table>
<thead>
<tr>
<th>DEPTH</th>
<th>NODES</th>
<th>TIME</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>0.11 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>10^6</td>
<td>1.1 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>10^8</td>
<td>2 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>10^10</td>
<td>3 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>10^12</td>
<td>13 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>10^14</td>
<td>3.5 years</td>
<td>99 petabytes</td>
</tr>
</tbody>
</table>

Assumes b=10, 1M nodes/sec, 1000 bytes/node
Depth-first search

Idea:
• Expand *deepest* unexpanded node

Implementation:
• *frontier* is LIFO (Last-In-First-Out) Queue:
  • Put successors at the *front* of *frontier* successor list.

Image credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu
Properties of depth-first search

**Complete?**  No. It fails in infinite-depth spaces, spaces with loops
If we modify it to avoid repeated states along path, then it is complete in finite spaces

**Optimal?**  No

**Time?**  $O(b^m)$. This is terrible if $m$ is much larger than $d$
  • but if solutions are dense, may be much faster than breadth-first

**Space?**  $O(b \times m)$, i.e., linear space!

\[ b: \text{maximum branching factor of search tree} \]
\[ d: \text{depth of the least cost solution} \]
\[ m: \text{maximum depth of the state space} (\infty) \]
Depth-first vs Breadth-first

Use depth-first if
- *Space is restricted*
- There are many possible solutions with long paths and wrong paths are usually terminated quickly
- Search can be fine-tuned quickly

Use breadth-first if
- *Possible infinite paths*
- Some solutions have short paths
- Can quickly discard unlikely paths
Search Conundrum

Breadth-first
- Complete,
- Optimal
- but uses $O(b^d)$ space

Depth-first
- Not complete unless $m$ is bounded
- Not optimal
- Uses $O(b^m)$ time; terrible if $m \gg d$
- but only uses $O(b^*m)$ space

How can we get the best of both?
Depth-limited search: A building block

Depth-First search *but with depth limit* \( l \).  
- i.e. nodes at depth \( l \) *have no successors*.  
- No infinite-path problem!

If \( l = d \) (by luck!), then optimal  
- But:  
  - If \( l < d \) then incomplete 😞  
  - If \( l > d \) then not optimal 😞

Time complexity: \( O(b^l) \)  
Space complexity: \( O(bl) 😊 \)
Iterative deepening search

A general strategy to find best depth limit \( l \).

- **Key idea:** use *Depth-limited search* as subroutine, with increasing \( l \).

\[
\text{For } l = 0 \text{ to } \infty \text{ do}
\]

- \text{depth-limited-search to level } l
- if it succeeds
  - then return solution

- **Complete & optimal:** Goal is always found at depth \( d \), the depth of the shallowest goal-node.

*Could this possibly be efficient?*
Nodes constructed at each deepening

Depth 0: 0 (Given the node, doesn’t construct it.)

Depth 1: $b^1$ nodes

Depth 2: $b$ nodes + $b^2$ nodes

Depth 3: $b$ nodes + $b^2$ nodes + $b^3$ nodes

...
Total nodes constructed:

Depth 0: 0 (Given the node, doesn’t *construct* it.)
Depth 1: \(b^1 = b\) nodes
Depth 2: \(b\) nodes + \(b^2\) nodes
  - Depth 3: \(b\) nodes + \(b^2\) nodes + \(b^3\) nodes
  - ...

Suppose the first solution is the last node at depth 3:
Total nodes constructed:

\(3b\) nodes + \(2b^2\) nodes + \(1b^3\) nodes
ID search, Evaluation: Time Complexity

- More generally, the time complexity is
  - $(d)b + (d-1)b^2 + ... + (1)b^d = O(b^d)$

*As efficient in terms of $O(\cdot)$ as Breadth First Search:*
- $b + b^2 + ... + b^d = O(b^d)$
ID search, Evaluation

Complete: YES (no infinite paths)

Time complexity: \(O(b^d)\)

Space complexity: \(O(bd)\)

Optimal: YES if step cost is 1.
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Depth-First</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Next Up: Informed Search

AIMA 3.5-3.6
Informed Search

An informed search strategy uses domain-specific information about the location of the goals in order to find a solution more efficiently than uninformed search.

Hints will come as part of a heuristic function denoted $h(n)$.

One of the most famous informed search algorithms is $A^*$ which was developed for robot navigation.

Shakey the robot was developed at the Stanford Research Institute from 1966 to 1972.

https://www.youtube.com/watch?v=7bsEN8mwUB8