Search and Planning

AIMA Chapters 3 and 7
Problem-Solving Agents

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

1. take pole
2. go out
3. go south
4. catch fish with pole
5. go north
6. pick rose
7. go north
8. go up
9. get branch
10. go down
11. go east
12. give the troll the fish
13. go east
14. hit guard with branch
15. get key
16. go east
17. get candle
18. go west
19. go down
20. light lamp
21. go down
22. light candle
23. read runes
24. get crown
25. go up
26. go up
27. go up
28. unlock door
29. go up
30. give rose to the princess
31. propose to the princess
32. down
33. down
34. east
35. east
36. wear crown
37. sit on throne
Review of Search Problems

AIMA 3.1-3.3
Formal Definition of a Search Problem

1. **States**: a set $S$
2. An **initial state** $s_i \in S$
3. **Actions**: a set $A$
   \[
   \forall s \text{ Actions}(s) = \text{the set of actions that can be executed in } s.
   \]
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**: Goal($s$)
   $s$ is a goal state if Goal($s$) is true.
   Can be implicit, e.g. **checkmate($s$)**
**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 $n \times 2^n$ states.

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

Graph showing the movement of a robot in a grid world, with arrows indicating right or left moves and a central action of sucking.
Vacuum World

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

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

Initial state

Suck
Right
Left

Right
Left

Solution:
A path from the initial state to a goal state

Goal states
Search Algorithms
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 $s_f$ 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: *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
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
8-Puzzle Search Tree

Start State

Max Branching Factor = 4

Action: Move Blank
Tie Left

Action: Up

Action: Right

Action: Down
8-Puzzle *Search Tree*

Repeated State
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
Several classic search algorithms differ only by the order of how they expand their search trees.

You can implement them by using different queue data structures:

- **Depth-first search** = LIFO queue
- **Breadth-first search** = FIFO queue
- **Greedy best-first search** or **A* search** = Priority queue
8-Puzzle Breadth-first search

Start State

Action: Move Blank Tie Left

Action: Up

Action: Right

Action: Down

Penn Engineering
Search Algorithms

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 \(\infty\))
## Properties of breadth-first search

<table>
<thead>
<tr>
<th>Property</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes (if $b$ is finite)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes, if cost = 1 per step (not optimal in general)</td>
</tr>
<tr>
<td>Time Complexity?</td>
<td>$1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$</td>
</tr>
<tr>
<td>Space Complexity?</td>
<td>$O(b^d)$ (keeps every node in memory)</td>
</tr>
</tbody>
</table>

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 $\infty$)
BFS versus DFS

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 $m >> d$
- but only uses $O(b^m) space$

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 $\infty$)
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 ms</td>
<td>107 KB</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 ms</td>
<td>10.6 MB</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 s</td>
<td>1 GB</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 h</td>
<td>10 TB</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 d</td>
<td>1 PB</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 y</td>
<td>99 PB</td>
</tr>
</tbody>
</table>

Assumes b=10, 1M nodes/sec, 1000 bytes/node
Action Castle
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
Let’s consider the sub-task of navigating from one location to another.

Formulate the *search problem*

- **States:** locations in the game
- **Actions:** move between connected locations
- **Goal:** move to a particular location like the **Throne Room**
- **Performance measure:** minimize number of moves to arrive at the goal

Find a *solution*

- Algorithm that returns sequence of actions to get from the start state to the goal.
The frontier tracks order of unexpanded search nodes. Here we’re using a FIFO queue.

The visited dictionary prevents us from revising states.

get_available_actions() to return all commands that could be used here.

The parser can execute this command to get the resulting state.

Check to see if this state satisfies the goal test, if so, return the command sequence that got us here.

Todo: implement get_state()

Todo: implement get_available_actions()

Todo: implement goal_test()
def BFS(game, goal_conditions):
    command_sequence = []
    if goal_test(game, goal_conditions): return command_sequence

    frontier = queue.Queue()
    frontier.put(((game, command_sequence)))

    visited = dict()
    visited[get_state(game)] = True

    while not frontier.empty():
        (current_game, command_sequence) = frontier.get()
        current_state = get_state(current_game)
        parser = Parser(current_game)
        available_actions = get_available_actions(current_game)

        for command in available_actions:
            # Clone the current game with its state
            new_game = copy.deepcopy(current_game)
            # Apply the command to it to get the resulting state
            parser = Parser(new_game)
            parser.parse_command(command)
            new_state = get_state(new_game)
            # Update the sequence of actions that we took to get to the resulting state
            new_command_sequence = copy.copy(command_sequence)
            new_command_sequence.append(command)
            if not new_state in visited:
                visited[new_state] = True
                if goal_test(new_game, goal_conditions):
                    frontier.put(((new_game, goal_conditions)))
            # Return None to indicate there is no solution.
            return None
Action Castle

Let’s consider the full game.

Actions

Start State

Transitions

State Space

Goal test
Actions

**Go**
- Move to a location

**Get**
- Add an item to inventory

**Special**
- Perform a special action with an item like “Catch fish with pole”

**Drop**
- Leave an item in current location
State Info

Location of Player

Items in their inventory

Location of all items / NPCs

Blocks like

• Troll guarding bridge,
• Locked door to tower,
• Guard barring entry to castle
goal_conditions = {
"at_location" : "Throne Room",
"inventory_contains" : "crown (worn)"
}

game = build_game()
solution = BFS(game, goal_conditions)
print("SOLUTION:", solution)

----

Found solution at depth 36.
Expanded 4138 nodes. Trimmed 18632 nodes.
There are 83 nodes on the frontier.
Classical Planning

AIMA Chapter 11
Classical Planning

The task of finding a sequence of action to accomplish a goal in a deterministic, fully-observable, discrete, static environment.

If an environment is:

- **Deterministic**
- **Fully observable**

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

In environments that are:

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

The solution must recommend different future actions depending on the what percepts it receives. This could be in the form of a *branching strategy.*
Representation Language

Planning Domain Definition Language (PDDL) express **actions** as a **schema**

```
(define (domain action-castle)
  (:requirements :strips :typing)
  (:types player location direction item)

  (:action go
    :effect (and (at ?p ?l2) (not (at ?p ?l1)))
  )

  (:action get
    :parameters (?item - item ?p - player ?l1 - location )
    :precondition (and (at ?p ?l1) (at ?item ?l1))
    :effect (and (inventory ?p ?item) (not (at ?item ?l1)))
  )

  (:action drop
    :parameters (?item - item ?p - player ?l1 - location )
    :effect (and (at ?item ?l1) (not (inventory ?p ?item)))
  )
)
```

- **Action name**
- **Variables**
- **Preconditions**
- **Effects**

Preconditions and effects are **conjunctions** of logical sentences.

These logical sentences are **literals** – positive or negated atomic sentences.
In PDDL, a **state** is represented as a **conjunction** of logical sentences that are **ground atomic fluents**. PDDL uses **database semantics**.

- **Ground** means they contain no variables.
- **Atomic sentences** contain just a single predicate.
- **Fluent** means an aspect of the world that can change over time.

**Action Schema** has variables:

```plaintext
(:action go
 :effect (and (at ?p ?l2) (not (at ?p ?l1)))
)
```

**State Representation** arguments are constants fluents may change over time:

```plaintext
(:init
 (connected cottage out gardenpath)
 (connected gardenpath in cottage)
 (connected gardenpath south fishingpond)
 (connected fishingpond north gardenpath)
 (at npc cottage)
)
```

Closed world assumption. Any fluent not mentioned is false. Unique names are distinct.
Successor States

A **ground action** is **applicable** if if every positive literal in the precondition is true, and every negative literal in the precondition is false.

**Ground Action**

```
(action go
  :parameters (out, npc, cottage, gardenpath)
  :precondition (and (at npc cottage) (connected cottage out gardenpath))
  :effect (and (at npc gardenpath) (not (at npc cottage))
)
```

**Initial State**

```
(init
  (connected cottage out gardenpath)
  (connected gardenpath in cottage)
  (connected gardenpath south fishingpond)
  (connected fishingpond north gardenpath)
  (at npc cottage)
)
```

**Result**

```
(connected cottage out gardenpath)
(connected gardenpath in cottage)
(connected gardenpath south fishingpond)
(connected fishingpond north gardenpath)
(at npc gardenpath)
```

Negative literals in the effects are kept in a **delete list**, and positive literals are kept in an **add list**.
Domains and Problems

Domain

```
(define (domain action-castle)
  (:requirements :strips :typing
  (:types player location direction item)

  (:action go
    :effect (and (at ?p ?l2) (not (at ?p ?l1)))
  )

  (:action get
    :parameters (?item - item ?p - player ?l1 - location )
    :precondition (and (at ?p ?l1) (at ?item ?l1))
    :effect (and (inventory ?p ?item) (not (at ?item ?l1)))
  )

  (:action drop
    :parameters (?item - item ?p - player ?l1 - location )
    :effect (and (at ?item ?l1) (not (inventory ?p ?item)))
  )
)
```
Algorithms for Classical Planning

We can apply **BFS** to the **initial state** through possible states looking for a **goal**. An advantage of the **declarative representation** of action schemas is that we can also **search backwards**.

**Start with a goal** and work backwards towards the initial state.

*In our Action Castle example, this would help us with the branching problem that the **drop** action introduced. If we work backwards from the goal, then we realize that we don’t ever need to drop an item for the correct solution.*