STATE SPACE SEARCH AND HEURISTIC TECHNIQUE

State Space Search: is a method of A I where a problem is solved by exploring all possible states (situations) the system can be in, until it find the goal state (solution)

State : situation or condition at a point in time

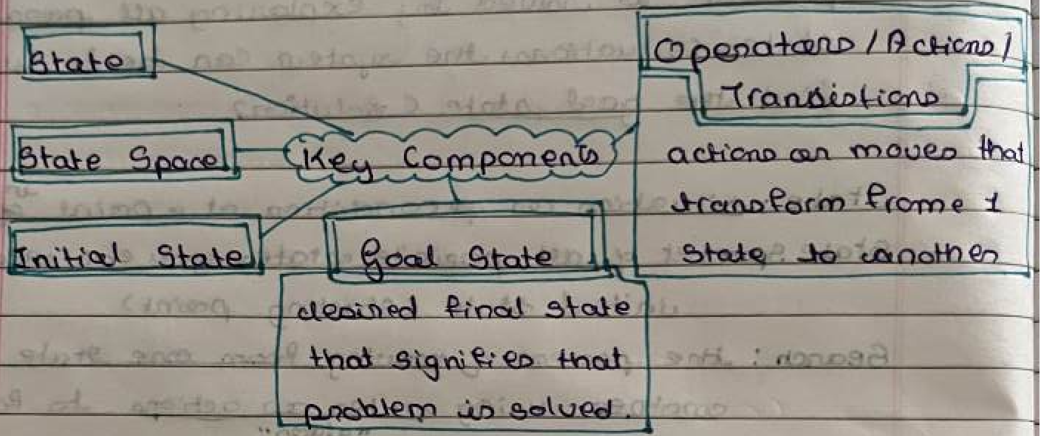
State Space set of all possible states you can reach from initial state (starting point)

Search : the process of moving from one state to another using rules or actions "algos" to find the goal

Solving Problems As State Space Search

The Core idea of State Space Search is representing a problem as a State Space

Imagine a problem as a collection of possible configurations or STATES. you start at one state (the initial state) and want to reach another (the goal state) by performing valid actions. The entire of these possible states and the connections between them forms STATE SPACE.



The Search Process

Solving a problem as state space search means finding a sequence of operators (actions) that leads from the initial state to (goal) final state

This sequence is called "SOLUTION PATH"

Eg. Finding a Route on Map

State: Your current location

State Space: All possible cities, towns, junctions

Operators: Travelling along a road from from location to another

Initial state: your starting city
Goal state: city you wanna reach
solution: the sequence of roads (ops) and cities (states) that takes you from the start to the destination.

PRODUCTION SYSTEM

· type of AI architecture or framework commonly used for implementing rule-based systems, expert systems and for solving problems that can be represented as state space search problems.

Components

1. Global Database / working memory
   · central data structure that holds all the current info on facts about the problem at hand.
   In a robot navigation it may have data like robot_at (x,y), obstacle_at (x+1, y), goal_at (gx, gy)
   · data in this can be modified by production rules
2. Production Rules / Knowledge Base
   · heart of the system representing knowledge
   · each rule is typically in an IF-THEN or CONDITION-ACTION format.

- IF part: set of conditions that must be true for the rule to be applicable
- THEN: set of actions to be performed if the condition is true (met).

e.g. IF robot at (x, y) AND no obstacle ahead then move forward (x, y)

3. Interpreter (Control Strategy) (Inference Engine)
   - this is the brain that decides which rule to apply and when.
   - It manages the execution of the production rule
   - It's primary task is to recognize act cycle.

   a. Recognize (scan global database, find production rules with satisfied IFs)
   (the IF's that match are called ACTIVE or CONFLICT SET RULES)

   b. conflict Resolution: if multiple rules are active, the interpreter uses a strategy to select only one rule to execute.
   Strategies: · first rule · rule with ↑ specifics
   · ↑ priority rule
   · rule that refers to the most recently addeds facts.

   c. Act: execution of the selected rule, modifying global database with changes leading to new state

• It repeats until goal state achieved
no rules are applicable
or predefined termination

Production System in State Space Search Context.

global database → current state
production rules → operations / actions
interpreter → search

e.g. Imagine a cooking recipe book.
· Global Database: kitchen counter with ingredients
· Production rules: each step
· Interpreter : you checking if conditions match
then taking next action

PROBLEM CHARACTERISTICS

· Features of a problem that help us understand its nature and more details
· it guids us to select suitable search strategy.

Key Characteristics

1. Is the problem decomposable?
· can the problem be broken down into smaller independent sub problems, each of which can be

thoroughly before starting and ensure that you understand all the steps and concepts involved from underpinning theory.

e

solved seperatly? If so, solutions of sub problems can be combined to form original solution.

- Decompable problems can often be solved more efficiently by tackling parts independently.

2. Can solution steps be ignored, undone, or must be irreversible.

- Ignorable: If a step taken turns out to be wrong, it can simply be ignored or forgotten, and another path can be tried without any significant cost or consequences.

  e.g. Proving a theorem - if one line of reasoning doesn't lead to a solution, you can simply abandon it and try another approach without having to undo everything.

- Recoverable: if a step proves incorrect, it's possible to backtrack and undo the move to return to a previous state.

  e.g. Playing chess (in casual games, you might retract a move) or solving a Rubiks Cube you can reverse moves

- Irreversible: once step is taken, it cannot be undone. mistake have permanent consequences

  e.g. Suppose robot dropping a fragile object.

3 Is the universe predictable?

do we know the exat outcome of every action we take? if yes then CERTAINTY SUDOKU
if no then UNCERTAINTY DRIVING

4. Is a good solution absolute or relative?

- Are we looking for any solution that meets criteria, or the best possible solution acc to some measures
  - L if yes then Satisficing (reed req min) any route to desti
  - L if no then Optimality (best) best route to destination

5. Is solution a state or path?

- is goal simply reaching a particular state, or sequence of action
  - L State Solution Proving theorem
  - L Path Solution chess maybe, navigation

6. what is the role of knowledge

- does the problem provides additional info beyond its basic definition that can guide the search?
  - INFORMED ~~BLIND~~ HEURISTIC L if yes: WITH KNOWLEDGE ~~BLIND / UNINFORMED~~ solving maze without layout
  - UNINFORMED BLIND ~~HEURISTIC~~ L if no: WITHOUT KNOWLEDGE ~~INFORMED / HEURISTIC~~ solving maze without layout
    - L DFS
    - L BFS

7. Does problem require interaction?

- is the entire solution planned before any action is taken, or is planning iteractively with acting and seeing the real world
  - if planned before action then OFFLINE 8 tile
  - if no the ONLINE driving car

4) Depth First Search

search strategy that explores as far as possible along each branch before backtracking. (going deeper in every child where they exist, if not backtrack dive into another side)

working

1. Start at the initial state (root node)
2. Expand one of its children
3. Keep expanding deeper into the tree/graph until
   - read the goal state OR
   - reach the dead (leaf) end
4. If a dead end is reached, backtrack to the most recently unexplored state.
5. Repeat until the goal state is found OR all states are explored.

Data Structure used : Stack (LIFO)

e.g maze with many paths
- DFS picks one direction (say left) and keeps left until exit or dead end.
- If dead end, go back where right was available
- continue till exit found

Characteristics

1> Completeness: x guaranteed (if ∞ state space)

2> Optimality : (might not give shortest path)

3> Time Complexity: worse : might have to visit all nodes

$O(b^m)$ b = branching factor

m = maximum paths

4> Space Complexity: $O(bm)$ comparatively memory efficient

Advantages	Disadvantages
1> memory efficient	1> can be trapped in ∞ state space
2> explores everything	2> x optimal (shortest path)
3> faster than BFS	3> can miss shallow solutions

Breadth First Search

search strategy which explores states level by levels, ensuring that it fully examines all states at a given depth before moving to next deeper level.

Working

1. Start at the initial state
2. Put it in a queue
3. While the queue is not empty.

suppose A
- remove the front node. (removing means visiting)

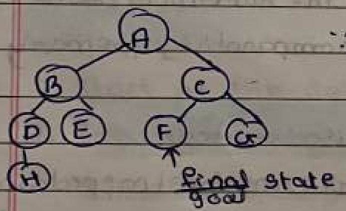
check for A
- check if it's the goal, yes → stop.

suppose B C
- else enqueue all neighbors (unvisited) childs

4. Repeat unit goal found or queue is empty.

Data Structure: Queue, FIFO (to ensure nodes are explored in the exact order they are discovered)

Eg.



∵ note when we say we dequeue a node (remove), we mean we visit it.

0. Start at root (A):
queue = [A]

1. Dequeue A → visit A
queue = []
Enqueue A's children: [B, C]
visited order: A

2. Dequeue B (visit it)
Queue: [C]
Enqueue B's children : [C D E]

visit order: A → B

3. Dequeue C (visit it)
Queue: [D, E]
enqueue children: [D, E, F, G]
visited order: A → B → C

4. Dequeue D (visit it)
Queue: [E, F, G]
enqueue children: [E, F, G, H]
visited order: A → B → C → D

5. Dequeue E
Queue: [E, G, H]
~~visit~~ enque children: x children
visited order: A → B → C → D → E

6. Dequeue F
Queue: [G H]
this is the goal state.
visited order: A → B → C → D → E → F

so path A → B → C → D → E → F

Properties

1. Completeness: yes (guaranteed to find a solution)
2. Optimality: yes finds the shortest path (in terms of number of steps, if all steps cost same)
3. Time complexity $O(b^d)$ b = branching factor
d = depth of the shallowest solution

4. Space complexity $O(b^d)$

Advantages

1. shortest path
2. complete
3. simple (maybe)

Disadvantages

1. ↑ memory
2. ↓ speed for deep problems
3. not suitable for ↑↑↑ state spaces

Heuristic Function

- estimate of cost or distance from a given state (n) to nearest goal state
- It is a informed (with knowledge) ~~pa~~ search algo
- denoted by h(n)
- It doesn't guarentee best path but gives
  1. guess or 2. rule of thumb
  to guide the flow
- It returns numeric value.
- For understand imagine states and everything on a graph, states are point
  flow is line
  this ~~path~~ h(n) is also a point.

Work Flow

1. For each node (state), calculate h(n)
2. use this value to decide the order of exploration
   - Smaller h(n) → seems closer to the goal
3. The search continues until the goal state is reached

Properties

1. Admissibility: never overestimates the true cost to reach the goal state.

Date
Page

2. Consistency (monotonicity) : estimates always respects the triangle inequality (like ensures that h(n) is well behaved)
3. Informedness: a good heuristic provides an estimate that is as close as possible to the true cost.
4. Computability: it should be easy and fast to compute h(n)

Advantages		Disadvantages
1. efficient	4. optimal	1. highly depends on h(n)
2. informed		2. not always easy
3. ↓ time		3. may be problem specific

Hill Climbing

- local search algorithm that continuously moves in the direction of increasing (or decreasing) h(n) until it reaches a peak (local minimum / maximum)
- It's like climbing uphill - at each step, you look at the neighbours and moves to the one that looks better according to h(n) until you can't improve further
- Stops when no better neighbour is found → risk of getting stuck in local optima.

Working

1. Start with a initial state
2. Evaluate its heuristic value
3. look at the neighbouring states and their h(n)
4. move to the neighbour with best heuristic value
5. Repeat until
    - No better neighbour exists (local max, min) or
    - Goal state reached

Types

↳ Simple Hill Climbing :
It moves to first neighbour state that is better than current

↳ Steppest Ascent Hill Climbing :
examines all neighbours and moves to best

↳ Stochastic Hill Climbing :
examines all neighbours chooses randomly from better neighbours

Challenges & Limitations

1. Local maxima (or minima) : algo might reach a state that is better than all neighbours, but no globally best (true highest point)

(narrow path)
2. Ridges (narrow peaks hard to reach)
3. Plateus (flat areas with no improvement) i.e all (current, neighbour) have same h(n)
4. x complete   3. x optimal

Advantages
1. memory efficient
2. simple heuristic
3. fast for simple ones
4. works well when search space is smooth and has single peak.

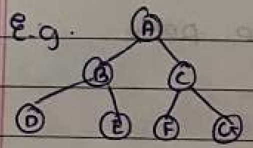
Best First Search
search strategy that always expands the node which appears best according to some evaluation (mainly h(n))
"It looks at all possible states picks the one that seems closest to goal state"

Components
Heurisitic Function (h(n)): same, same, no difference
Priority Queue (open List): stores all generated but not yet explored (expanded) nodes.
Closed List : keeps track of nodes that are expanded

Working

1. Initialize the open list with start node.
2. While open list is not empty
   - Remove the node with lowest h(n) (visit it)
   - If it's goal → stop
   - Else, expand it, add children (acc to order)
3. Continue until the goal is found or queue is empty

E.g.



h(A) = 6   h(D) = 7   h(G) = 0
h(B) = 4   h(E) = 2
h(C) = 3   h(F) = 6

1. Start at A = (h=6)
2. Expand → B(h=4), C(h=3)
   - Queue = { C(3), B(4) }
3. pick C (lowest h=3). Expand → F(6), G(0)
   - Queue = { G(0), B(4), F(6) }
4. Pick G(h=0) → Goal reached.

Advantages
1. Faster
2. priority base
3. efficient

Disadvantages
1. × optimal
2. may loop if × closed list
3. ↑ memory required.

Properties

- ✗complete (may loop if graph cycles unhandled)
- ✗optimal (may not find shortest path)
- time complexity $O(b^m)$ (b = branching factor, m = depth)
- Space complexity $O(b^m)$