Chapter 1-3 (14)
Innleiðing í tilgjørdum viti

Notes

Contents

A search problem is the foundational concept in AI problem-solving. The agent must decide which actions to take to transition from the initial state to a goal state, while considering the path cost and possible state transitions.

1 Elements of a Search Problem (Common exam question)

- Initial State : The starting point of the agent in the problem space.
- Actions : A set of all possible actions the agent can take from a state.
- Transition Model: A function $Result(s, a)$ that returns the resulting state from performing action a in state s.
- States: All possible configurations or situations the agent may encounter.
- Goal Test: A function that checks if a given state is a goal state.
- Path Cost: A numerical value that defines the cost associated with a path from the initial state to a given state.

Key point: A solution is a sequence of actions that leads from the initial state to the goal state.

1.1 Additional Key Concepts

State space: The complete set of all possible states reachable from the initial state, given the actions and transition model.

Search Tree: A structure where nodes represent states, and branches represent actions leading to those states.

Search Graph (vs. Tree): Includes cycles and shared states — more efficient than trees but requires cycle detection.

Node (in a search tree/graph): A data structure that typically contains:

- State
- Parent Node
- Action taken to reach this state
- Path cost so far
- Depth (optional)

Branching Factor (b): The average number of successors (child nodes) per state.

Belief State: A representation of all the information an agent has about the world (useful in partially observable environments). Example: tracking visited locations.

Directed Acyclic Graph (DAG): A graph with directed edges and no cycles.

2 Search Strategies

2.1 Uninformed Search

Uninformed (Blind) Search has no domain-specific knowledge is used.

Strategies:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Uniform-Cost Search (Lowest-Cost-First)
- Iterative Deepening DFS

2.2 Search Algorithm Structure

Generic Graph Search

- Initialize:
  - Frontier = Initial State
  - Explored set = empty set
- Loop
  - If Frontier is empty return no solution
  - Remove node from Frontier
  - If node.state is goal return the solution
  - add node.state to Explored set
  - Expand node and add resulting nodes to Frontier (Only if not in Frontier or Explored set)

Depth-First Search (DFS):

- Frontier: Stack (Last in first out)
- Explores the deepest node first.
- Pros: Low memory usage
- Cons: Can get stuck in deep or infinite paths
- Time complexity: $O(b^m)$
- Space complexity: $O(b \cdot m)$

Where b is the branching factor and m is the maximum depth.

Iterative Deepening DFS (IDDFS)

- Combines benefits of BFS and DFS.
- Repeatedly runs DFS with increasing depth limits.
- Pros: Finds optimal solution with less space
- Redundant expansions

Simplified Pseudocode:

```
for depth = 0 to infinity:
    result = Depth-Limited-Search(initial, depth)
    if result not equal failure:
        return result
```

Breadth-First Search (BFS)

- Frontier: Queue (FIFO)
- Explores the shallowest nodes first.
- Guarantees: Finds the solution with the fewest steps (minimum depth)
- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$

Where b is the branching factor and d is depth of the shallowest solution.

Uniform-Cost Search (UCS)

- Expands the node with the lowest path cost.

- Uses a priority queue, ordered by path cost.
- Guarantees:Finds optimal solution (lowest cost)
- Time complexity: Depends on cost granularity (can be exponential)

-

Informed (Heuristic) Search

- Uses domain-specific knowledge (heuristics) to guide search.
- Examples: $A*$ Search and Greedy Best-First Search

3 Heuristic (Informed) Search

Heuristic search leverages problem-specific knowledge to guide exploration more efficiently than uninformed strategies.

3.1 Heuristic Function $h(n)$

A heuristic is a non-negative estimate of the cost from node n to the goal.

- Example: Euclidean (straight-line) distance in a map.

3.2 Admissibility

A heuristic is **admissible** if it never overestimates the true cost:

$$h(n) \leq \text{true cost from } n \text{ to goal}$$

- Examples: Euclidean distance, Manhattan distance (in grid environments without obstacles).

3.3 Consistency (Monotonicity)

A heuristic is **consistent** if:

$$h(n) \leq c(n, n') + h(n')$$

where $c(n, n')$ is the cost of moving from n to n'.

- Equivalent to the triangle inequality.
- Ensures the total estimated cost $f(n) = g(n) + h(n)$ is non-decreasing along a path.

4 Summary of Key Terms

- Agent: Entity making decisions
- Environment: The world the agent operates in
- Action: Operation an agent can take
- Frontier: Set of nodes available for expansion
- Explored Set: Set of visited states
- Goal State: Target state for the agent
- Path Cost: Numerical cost of reaching a node
- Optimal Solution: The solution with the lowest total path cost

Exam Tips:

- List and explain the 5 elements of a search problem
- Distinguish between search tree and search graph
- Compare time and space complexity for BFS, DFS, IDDFS
- Know pros/cons of different search methods

5 Types of Heuristic Search Algorithms

5.1 Greedy Best-First Search

- Expands the node that appears closest to the goal (lowest $h(n)$).
- Frontier: Priority Queue ordered by $h(n)$.
- Pros: Fast in many cases.
- Cons: Not optimal; can fail with misleading heuristics.

5.2 A^* Search

Evaluates nodes using:

$$f(n) = g(n) + h(n)$$

where $g(n)$ = cost so far, $h(n)$ = estimated cost to goal.

- Frontier: Priority Queue ordered by $f(n)$.
- Combines UCS (optimality) and Greedy Best-First (efficiency).
- Guarantees:
  - Completeness: Finds a solution if one exists.
  - Optimality: Finds least-cost path if $h(n)$ is admissible.

5.3 Time Complexity

- Generally exponential in solution depth.
- Strongly affected by branching factor b.

6 Search Complexity Concepts

- **Forward Branching Factor**: Number of successors from a node.
- **Backward Branching Factor**: Number of predecessors.
- Lower branching factor $\Rightarrow$ better time complexity.
- Example: $2 \cdot b^{k/2} \ll b^k$

7 Multi-Agent Systems & Game Theory

7.1 Overview

Game theory studies the behavior of multiple agents that may be cooperative, competitive, or mixed.

7.2 Utility

Utility measures future rewards from a state or action.

- Agents may have different utility functions.
- They act autonomously with limited/incomplete information.
- Outcomes depend on other agents' actions.

8 Types of Games

8.1 Cooperative Games

Agents coordinate to achieve shared goals.

8.2 Competitive Games

Agents have conflicting goals. Often modeled as zero-sum games.

9 Game Representations

9.1 Normal Form (Strategic Form)

- Payoff matrix shows utilities for each agent given action profiles.
- Action Profile: Assignment of an action to each agent.
- Utility Function: Maps action profiles to payoffs.

9.2 Extensive Form (Game Tree)

Represents sequential games with perfect information.

9.3 Imperfect Information & Simultaneous Actions

- Partially Observable Games: Agents lack full information (e.g., card games).
- Simultaneous Move Games: Players act without knowing others' choices.
- Information Set: States indistinguishable to a player at decision time.

10 Strategy and Planning

- **Strategy**: A full plan of actions for all possible situations.
- **Strategy Profile**: One strategy per agent.
- Strategies may be deterministic or probabilistic.

11 Adversarial Search

11.1 Minimax Algorithm

- MAX tries to maximize utility; MIN tries to minimize it.
- Works recursively from terminal states backward.

Key Functions:

- s_0: Initial state.
- Player(s): Returns whose turn it is.
- Actions(s): Legal actions.
- Result(s, a): Next state.
- Terminal(s): Checks if game is over.

- Utility(s): Value of terminal state.

Simplified Pseudocode:

```
def minimax(s):
    if Terminal(s):
        return Utility(s)
    if Player(s) == MAX:
        return max(minimax(Result(s,a)) for a in Actions(s))
    else:
        return min(minimax(Result(s,a)) for a in Actions(s))
```

12 Optimizations

12.1 Alpha-Beta Pruning

Prunes branches of the minimax tree that cannot affect the final decision, greatly reducing node evaluations.

13 Evaluation Function

Used when search is cut off early (e.g., depth limit). Estimates utility of a non-terminal state for approximate reasoning.