COMP3211: Fundamentals of AI

Hong, Lanxuan

Spring 2025

This is a lecture note for Fundamentals of AI.

- UCB CS188: notes, website, playlist

Last updated on Tuesday 16th September, 2025.

Contents

1 Introduction

1.1 Rational Agents

One of the most important areas in AI is to create **rational agents**, i.e. entities that have goals or preferences and tries to perform a series of actions that yield the **optimal** expected outcome given these goals. There are various ways to achieve intelligence, including how the agents learn, optimize, and interact with the **environment**.

The following shows a boundary-following robot example with multiple solutions to make it smart and follow the boundary.

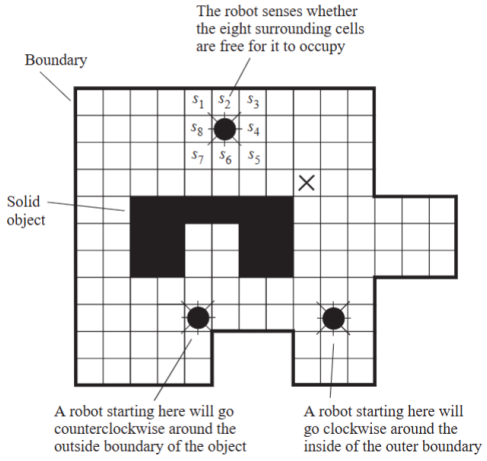


Figure 1: Boundary-Following Robot

The environment is modeled by a 2-d grid world with no tight space. The agent can get 8 sensory input from cells nearby to check whether they are occupied. These inputs are fed into **action function**, which decides its action. In this situation where agent does not know the whole map, **SR agent** response only according to current input, while **reflex agent** consider its "state"(past input) as well.

1. rule-based method: **production system**

   A production system can be derived from weight vector W of each sensory inputs from learning. It can also be designed by considering all production combinations. ⚠**need memory?**

2. learning-based method: **supervised learning**

The action function is learned from past experience, i.e. the match pattern between sensory input and decisions. The decision is made by TLU:

$$\texttt{decision is true} = \sum_{W,X} W^T X \geq \texttt{threshold}$$

The prediction is then compared to real decision to update weight vector W through the error-correction procedure.

$$W \leftarrow W + c(d - f)X$$ ⚠ **Weight vector W is the action function**

In general, TLU finds a linear decision boundary in the perceptual space. It is possible that patterns cannot be learned(even after many iterations) if the pattern is not **linear separable**.

3. optimize by **genetic programming**

   Instead of optimizing by error correction, evolution provides another way for system adaptation: generations of descendants are produced that perform better than ancestors. The process start with a population of random programs using functions, constants and sensory inputs. The (i+1)-th generation is constructed from the i-th one by copy, crossover, and mutation.

4. memory block: **state machine**

   If the environment cannot be sensed at the time the agent needs them, a memory block can be introduced to a S-R agent.

1.2 Environment

As shown by the example, the design of an agent heavily depends on the type of **environment** the agents acts upon. The complexity of the environment depends on whether it is **fully observable**, **deterministic**, and **static**.

Note ⚠ For the boundary-following agent, the environment is **partially observable**, thus the agent must have an internal estimate(by rules or learning) of the state of the world to support its action. Without the need to consider stochastic result and multi-agent situation, the rules can be very simple.

2 Searching

Searching is an important task for an agent to find certain **strategy**, which is a sequence of actions, with optimal cost to reach the goal state.

We can formulate the search problem as, given the current **state** of the agent, the agent takes **action** with **cost**, and reaches a new state in the **state space** by a (deterministic) **transition model**. The agent starts from **start state** and tries to determine the optimal sequence of actions to reach the **goal state**.

2.1 State space

The **state space** represents the information about the world that's necessary for planning, for example, it's location, the goal state, "obstacles" on the road, etc. The estimation of computational runtime of solving a search problem depends on the **size of state space**, which can be calculated easily by the fundamental counting principle.

The search problem can be represented by **state space graph** and **search trees**.

> **Note** ⚠ A state space graph is similar to a finite state machine(FSM) with states as nodes and actions (cost) as edges between states. For search trees, each node encodes not just the state itself, but the entire path from the start state to the given state in the state space graph. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees.

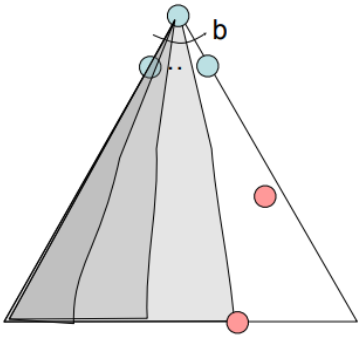
The general strategy for searching can be described in the following process.

```
function TREE-SEARCH(problem, frontier) return a solution or failure
    frontier = initial state
    while (!IsEmpty(frontier) do
        node = POP(frontier)
        if problem.IsGoal(node.state) then
            return node
        for each child-node in EXPAND(problem, node) do
            add child-node to frontier
    return failure
```

```
function EXPAND(problem, node) yields nodes // search strategy
    s = node.state
    for each action in problem.action(s) do
        s′ = problem.result(s, action)
    yield NODE(state=s′, parent=node, action=action)
```

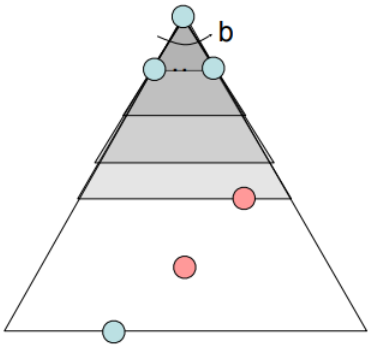
2.2 Uninformed search strategy

There are 3 commonly used **uninformed search strategy**: **depth first search**. **breadth-first search**, and **uniform cost search**. They will be compared(and with informed search) in terms of **completeness**, **time complexity**, **space complexity**, and **optimality**.



DFS

1. completeness: Yes
2. time complexity: $\mathbb{O}(b^m)$
3. space complexity: bm
4. optimally: No



BFS

1. completeness: Yes
2. time complexity: $\mathbb{O}(b^m)$
3. space complexity: $\mathbb{O}(b^m)$
4. optimally: Yes

2.3 Informed search

Uniform cost search is good because it's both complete and optimal, but it can be fairly slow because it expands in every direction from the start state while searching for a goal. The idea of **heuristic** provide the direction in which the agent should focus on search.

The heuristic function take in a state as input and output a corresponding estimate of how close a state is to a goal(e.g. by Manhattan distance). The heuristic strategy chooses nodes that have rather small value to expand. Different design of heuristic function shows a trade-off of completeness, optimal, and efficiency.

1. **greedy search** is a strategy for exploration that always selects the frontier node with the lowest heuristic value for expansion.

2. **A* search** is a strategy for exploration that always selects the frontier node with the lowest estimated total cost for expansion, where total cost $f(n)$ is actual path cost $g(n)$ from the start node and estimated cost $h(n)$. **A* search can be both complete and optimal, given an appropriate heuristic** $h(n)$. The algorithm terminates when the goal is dequeue from the fringe.

Note ⚠ The difference between A* algorithm and Dijkstra algorithm.

It's an important topic to discuss what constitutes a good heuristic. The condition required for optimality when using A* tree search is known as **admissibility**, i.e. neither negative(for monotonicity) nor an overestimate(larger than actual cost).

$$0 \leq h(n) \leq h^*(n) \text{ ⚠ less than the true cost to a nearest goal}$$

The following proofs that $A*$ search gives the optimal solution while satisfying completeness. It also gives the best solution for heuristic searching.

$$f(n) = h(n) + g(n) \leq f^*(n)$$
$$f(G) = h(G) + g(G) \geq g(G) \geq f^*(n) \geq f(n)$$
$$f(G_2) = h(G_2) + g(G_2) \geq g(G_2) \geq f^*(n) \geq f(n)$$
all ancestor of G will be expanded before G_2 expands
$$f(G) = g(G) \leq g(G_2) = f(G_2)$$
the optimal goal will be expanded first

Note ⚠ Consistency problem in graph search

2.4 Constraint satisfaction problem

In some applications where the path to the goal is less important, we focus in identifying the variable assignment of goal state directly. The goal test is a set of **constraints** specifying allowable combinations of values for variables(e.g. four-color problem, n-queens problem). Constraint satisfaction problems are NP-hard, but we can formulate CSPs as search problems, defining states as partial assignments (variable assignments to CSPs where some variables have been assigned values while others have not).

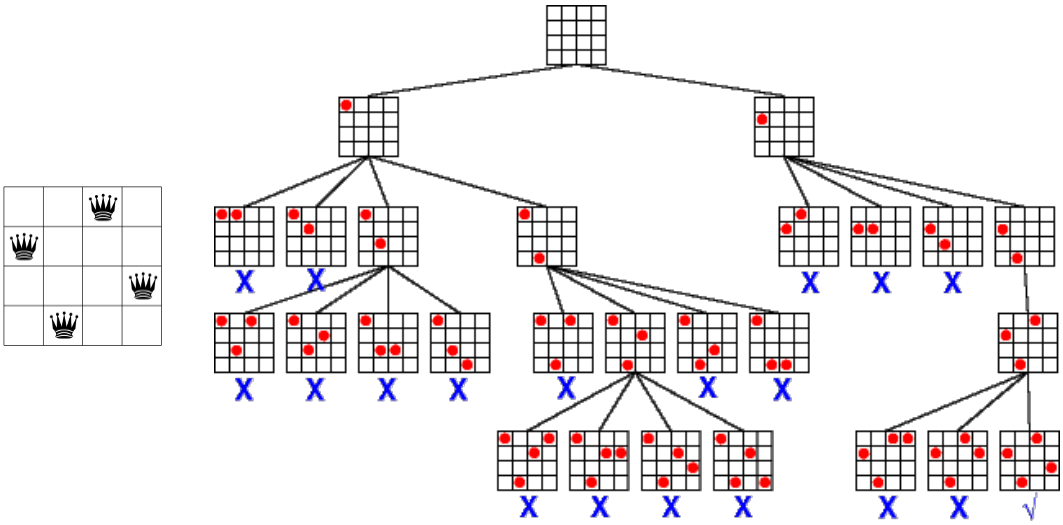


Figure 2: N-Queens Problem

The N-Queen Problem can be formulated into a CSP problem as follows.

Variables q_1 to q_4 representing the position of a queen in columns 1 to 4.
Constraints: no 2 queens should be on the same diagonal.

$$q_i \neq q_j$$
$$|q_i - q_{i+k}| \neq k, k = 1, 2, 3, 4; i = 1, ..., 4 - k$$

These constraints between variables can be formulated by **graph** where variables are nodes and constraints are edges, where we can constructively apply the searching algorithm. We can apply **constraint propagation** according to local consistency property to eliminate the search space. The algorithm, however, is not

completed yet: **we need a procedure to decide the order of searching**.

2.4.1 DPLL Procedure

The constraint satisfaction problem can be reduced to **satisfiable problem**, i.e. SAT problem. SAT problem is known to be NP hard, thus lots of approximation algorithm are invented to find a "good" solution.

2.5 GSAT Procedure

2.6 Game Trees

3 Game Theory

Multi-Agent System is one where more than one agent co-exist and interact. New issues in MAS including agent communication, cooperation, and adversity need to be considered. **Game theory** studies how self-interested agents interact.

- **Preference** refers to an order by which an agent, while in search of an "optimal" choice, ranks alternatives based on their respective **utility**.
- **Utility function** measures from situations to real numbers for each agent.

According to the properties of preferences and utility functions, we can come up with the following theorem to construct a equivalence between utility and preference.

Theorem 3.1 If a preference relation satisfies completeness, transitivity, substitutability, decomposability, monotonicity, and continuity, then there exist a function u mapping from the outcome set to a real number in $[0,1]$ with the properties:

- $u(o_1) \geq u(o_2)$ iff $o_1 \succeq o_2$ $\forall o_1, o_2 \in \mathcal{O}$
- The expected utility $u([p_1 : o_1, ...p_k : o_k]) = \sum_{i-1}^{k} p_i u(o_i)$