CT421

Artificial Intelligence

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–01–17

Contents

1 Introduction

1.1 Assessment

- Exam: 60%.
- 2 projects: 20% each.

1.2 Introduction to Artificial Intelligence

The field of Artificial Intelligence has evolved & changed many times over the years, with changes focussing both on the problems & approaches in the field; many problems that were considered typical AI problems are now often considered to belong in different fields. There are also many difficulties in defining intelligence: the **Turing test** attempts to give an objective notion of intelligence and abstracts away from any notions of representation, awareness, etc. It attempts to eliminate bias in favour of living beings by focusing solely on content of questions & answers. There are many criticisms of the Turing test:

- Bias towards symbolic problem-solving criticisms;
- Doesn't test many aspects of human intelligence;
- Possibly constrains notions of intelligence.

1.2.1 Approaches to AI

- **Classical AI** uses predicate calculus (& others) and logical inference to infer or find new information. It is a powerful approach for many domains, but issues arise when dealing with noise or contradictory data.
- **Machine learning** learns from data and uses a distributed representation of learned information. It is typified by neural networks & deep learning approaches.
- **Agent-based systems** view intelligence as a collective emergent behaviour from a large number of simple interacting individuals or *agents*. Social systems provide another metaphor for intelligence in that they exhibit global behaviours that enable them to solve problems that would prove impossible for any of the individual members. Properties of agent-based systems / artificial life include:
  - Agents are autonomous or semi-autonomous;
  - Agents are situated;
  - Agents are interactional;
  - Society is structured;
  - Intelligence is emergent.

2 Search

Many problems can be viewed as a **search** problem; consider designing an algorithm to solve a sudoku puzzle: in every step, we are effectively searching for a move (an action) that takes us to a correct legal state. To complete the game, we are iteratively searching for an action that brings us to legal board and so forth until completion. Other examples include searching for a path in a maze, word ladders, chess, & checkers. The problem statement can be formalised as follows:

- The problem can be in various states.
- We start in an initial state.
- There is a set of actions available.
- Each action changes the state.
- Each action has an associated cost.

- We want to reach some goal while minimising cost.

More formally:

- Set of states S.
- Start state $s_0 \in S$.
- Set of actions A and action rules $a(s) \to s'$.
- A goal test $g(s) \to \{0, 1\}$.
- Cost function $C(s, a, s') \to \mathbb{R}$.
- Search can be defined by the 5-tuple (S, s, a, g, C).

We can then state the problem as follows: find a sequence of actions $a_1 \dots a_n$ and corresponding states $s_0 \dots sn$ such that:

- $s_0 = s$
- $s_i = a_i(S_{i-1})$
- $g(s_n) = 1$

while minimising $\sum_{i=1}^{n} c(a_i)$.

The problem of solving a sudoku puzzle can be re-stated as:

- Sudoku states: all legal sudoku boards.
- Start state: a particular, partially filled-in, board.
- Actions: inserting a valid number into the board.
- Goal test: all cells filled with no collisions.
- Cost function: 1 per move.

We can conceptualise this search as a **search tree**: a node represents a state, and the edges from a state represent the possible actions from that state, with the edge pointing to the new resulting state from the action. Important factors of a search tree include:

- The breadth of the tree (branching factor).
- The depth of the tree.
- The minimum solution depth.
- The size of the tree $O(b^d)$.
- The **frontier:** the set of unexplored nodes that are reachable from any currently explored node.
- Choosing which node to explore next is the key in search algorithms.

2.1 Uninformed Search

In **uninformed search**, no information is known (or used) about solutions in the tree. Possible approaches include expanding the deepest node (depth-first search) or expanding the closest node (breadth-first search). Properties that must be considered for uninformed search include completeness, optimality, time complexity (the total number of nodes visited), & space complexity (the size of the frontier).

```
visited = {};
frontier = {$s_0$};
goal_found = False;

while (not goal_found):
    node = frontier.next();
    frontier.delete(node);

    if (g(node)):
        goal_found = True;
    else
        visited.add(node);
        for child in node.children():
            if (not visited.contains(child)):
                frontier.add(child);
```

Listing 1: Pseudocode for an uninformed search

The manner in which we expand the node is key to how the search progresses. The way in which we implement `frontier.next()` determines the type of search; otherwise the basic approach remains unchanged.

Depth-first search is good regarding memory cost, but produces suboptimal solutions:

- Space: $O(bd)$.
- Time: $O(b^d)$.
- Completeness: only for finite trees.
- Optimality: no.

Breadth-first search produces an optimal solution, but is expensive with regards to memory cost:

- Space: $O(b^{m+1})$, where m is the depth of the solution in the tree.
- Time: $O(b^m)$.
- Completeness: yes.
- Optimality: yes (assuming constant costs).

Iterative deepening search attempts to overcome some of the issues of both breadth-first and depth-first search. It works by running depth-first search to a fixed depth of z by starting at $d = 1$ and if no solution is found, incrementing d and re-running.

- Low memory requirements (equal to depth-first search).
- Not many more nodes expanded than breadth-first search.
- Note that the leaf level will have more nodes than the previous layers.

Thus far, we have assumed each edge has a fixed cost; consider the case where the costs are not uniform: neither depth-first search or breadth-first search are guaranteed to find the least-cost path in the case where action costs are not uniform. One approach is to choose the node with the lowest cost: order the nodes in the frontier by cost-so-far (cost of the path from the start state to the current node) and explore the next node with the smallest cost-so-far, which gives an optimal and complete solution (given all positive costs).

2.2 Informed Search

Thus far, we have assumed we know nothing about the search space; what should we do if we know *something* about the search space? We know the cost of getting to the current node: the remaining cost of finding the solution is the cost from the current node to the goal state; therefore, the total cost is the cost of getting from the start state to the current node, plus the cost of getting from the current node to the goal state. We can use a (problem-specific) **heuristic** $h(s)$ to estimate the remaining cost: $h(s) = 0$ is s is a goal. A good heuristic is fast to compute and close to the real costs.

Given that $g(s)$ is the cost of the path so far, the **A* algorithm** expands the node s to minimise $g(s) + h(s)$. The frontier nodes are managed as a priority queue. If h never overestimates the cost, the A* algorithm will find the optimal solution.

2.3 Adversarial Search

The typical game setting is as follows:

- 2 player;
- Alternating turns;
- Zero-sum (gain for one, loss for another);
- Perfect information.

A game is said to be **solved** if an optimal strategy is known.

- A **strong solved** game is one which is solved for all positions;
- A **weak solved** game is one which is solved for some (start) positions.

A game has the following properties:

- A set of possible states;
- A start state;
- A set of actions;
- A set of end states (many);
- An objective function;
- Control over actions alternates.

The **minimax algorithm** computes a value for each node, going backwards from the end-nodes. The **max player** selects actions to maximise return, while the **min player** selects actions to minimise return. The algorithm assumes perfect play from both players. For optimal play, the agent has to evaluate the entire game tree. Issues to consider include:

- Noise / randomness;
- Efficiency – size of the tree;
- Many game trees are too deep;
- Many game trees are too broad.

Alpha-beta pruning is a means to reduce the search space wherein sibling nodes can be pruned based on previously found values. Alpha represents the best maximum value found so far (for the maximising player), and beta represents the best minimum value found so far (for the minimising player). If a node's value proves irrelevant (based on the alpha & beta values), its entire subtree can be discarded. In reality, for many search scenarios in games, even with alpha-beta pruning, the space is much too large to get to all end states. Instead, we use an **evaluation function** which is effectively a heuristic to estimate the value of a state (the probability of a win or a loss). The search is ran to a fixed depth and all states are evaluated at that depth. Look-ahead is performed from the best states to another fixed depth.

Horizon effects refer to the limitations that arise when the algorithm evaluates a position based on a finite search depth (the horizon):

- What if something interesting / unusual / unexpected occurs at horizon $+ 1$?
- How do you identify that?
- When to generate and explore more nodes?
- Deceptive problems.