CT421

Artificial Intelligence

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

2025–02–11

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:

- There is a set of (possible/legal) states S;
- There is some start state $s_0 \in S$;
- There is a set of actions A and action rules $a(s) \to s'$;
- There is some goal test $g(s) \to \{0, 1\}$ that tests if we have satisfied our goal;
- There is some cost function $C(s, a, s') \to \mathbb{R}$ that associates a cost with each action;
- 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 $s_0 \dots s_n$ such that:

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

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

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.

2.4 Monte Carlo Tree Search

Minimax and alpha-beta pruning are both useful approaches, and alpha-beta pruning effectively results in computing the square root of the branching factor. In many cases, however, game trees have too high a **blocking factor** and evaluating the leaf nodes is not possible; instead, an **estimation function** is used instead.

Monte Carlo tree search continually estimates the value of tree nodes. It combines ideas from classical tree search with ideas from machine learning, and balances the exploration of unseen space with exploitation of the best known move. Monte Carlo tree search works by exploring all potential options at the time. The best move is identified and other option are searched for while validating how good the current action is. It uses **play out**, which is simulation considering random actions.

1. **Selection:** pick a suitable path.
2. **Expansion:** expand from that path.
3. **Simulation:** simulation / play outs.
4. **Back propagation:** update states.

There is a trade-off between exploration and exploitation; the balance is usually controlled by the number of parameters / factors, including the number of wins, simulations, & exploration cost. Monte Carlo grid search is used in a large number of domains, including AlphaGo, DeepMind, & many video games.

2.5 Local Search Algorithms

In many domains, we are only interested in the goal state: the path to finding the goal is irrelevant. The main idea in **local search algorithms** is to maintain the current state and repeatedly try to improve upon the current state. This results in little memory overhead and can find reasonable solutions even in large or infinite spaces. Local search algorithms are suitable only for certain classes of problems in which maximising (or minimising) certain criteria among a number of candidate solutions is desirable. Local search algorithms are **incomplete algorithms**, as they may not find the optimal solution.

2.6 Hill Climbing

Hill climbing involves moving in the direction of increasing value and stopping when a peak is reached. There is no look-ahead involved, and only the current state & the objective function are maintained. One problem with hill climbing algorithms is local maxima & minima; therefore, the success of the algorithm depends on the shape of the search space. One approach to solve this problem is **random restart**.

```
current_node = initial_state;

while (stopping_criteria):
    neighbour = current_node.fittest_neighbour;

    if neighbour.value > current_node.value:
        current_node = neighbour;
```

Listing 2: Hill climbing pseudocode

One problem that can be solved with a hill climbing algorithm is the **8 queens problem**: place 8 queens on a chess board so that no two queens are attacking each other. The problem can be stated more formally as: given an 8×8 grid, identify 8 squares so that no two squares are on the same row, column, or diagonal. The problem can also be generalised to an $N \times N$ board. An algorithm can be generated using hill climbing to find a solution. However, it must be noted that the algorithm may get caught at a local maximum.

Simulated annealing attempts to avoid getting stuck in local maxima by randomly moving to another state. The move is based on the fitness of the current state, the new state, and the **temperature** (a parameter which decreases over time and reduces the probability of changing states).

2.7 Well-Known Optimisation Problems

2.7.1 Knapsack Problem

There are a set of items, each with a value & a weight. The goal is to select a subset of items such that the weight of these items is below a threshold and the sum of the values is optimised / maximised. The problem is how to select those items. More formally, given two n-tuples of values $< v_0, v_1, \ldots, v_n >$ and $< w_0, w_1, \ldots, w_n >$, choose a set of items i from the $n$items such that $\sum_i v(i)$ is maximised and $\sum_i w(i) \leq T$.

The brute force approach is to enumerate all subsets and keep the subset that gives the greatest payoff: $O(2^n)$. The greedy approach is more efficient but won't give the best solution.

There are many variations on the knapsack problem; the variation described above is the 0/1 knapsack problem; there are other scenarios where part of an item may be taken, variations wherein there are constraints over the items where the value of an item is dependent on another item being chosen or not, and variations with multiple knapsacks.

2.8 Travelling Salesman Problem

Given N cities, devise a tour that involves visiting every city once. The distance, or *cost*, between pairs of cities is known and the goal is to minimise the cost of the tour. There are many applications of this problem in scheduling & optimisation.

The brute force approach is to enumerate all tours and choose the cheapest, and is computationally intractable.

3 Genetic Algorithms

Genetic algorithms are directed search algorithms inspired by biological evolution, developed by John Holland in the 1970s to study the adaptive processes occurring in natural systems. They have been used as search mechanisms to find good solutions for a range of problems. At a high level, the algorithm is as follows:

1. Produce an initial population of individuals.
2. Evaluate the fitness of all individuals.
3. While the termination condition is not met, do:

1. Select the fitter individuals for reproduction.
2. Recombine between individuals.
3. Mutate individuals.
4. Evaluate the fitness of the modified individuals.
5. Generate a new population.

There are many potential options for the representation of chromosomes, including bit strings, integers, real numbers, lists of rules/instructions, & programs. To initialise the population, we typically start with a population of randomly generated individuals, but we can also use a previously-saved population, a set of solutions provided by a human expert, or a set of solutions provided by another algorithm. As rules of thumb, we generally use a data structure as close as possible to the natural representation, write appropriate genetic operators as needed, and, if possible, ensure that all genotypes correspond to feasible solutions. Selection can be fitness-based (roulette wheel), rank-based, tournament selection, or elitism.

Crossover is a vital component of genetic algorithms that allows the recombination of good sub-solutions and speeds up search early in evolution. **Mutation** represents a change in the gene; the main purpose is to prevent the search algorithm from becoming trapped in a local maxima.

As a simple example, suppose that one wishes to maximise the number of `1`s in a string of l binary digits. Similarly, we could set the target to be an arbitrary string of `1`s and `0`s. An individual can be represented as a string of binary digits. The fitness of a candidate solution is the number of `1`s in its genetic code. A related problem, maintaining the same representation as before, is to modify the fitness function as follows: for strings of length l with x `1`s, if $x > 1$, then fitness is equal to x; else, fitness is equal to $l + k$ for $k > 0$.

3.1 Schema Theorem

Consider a genetic algorithm using only a binary alphabet. $\{0, 1, *\}$ is the alphabet, where $*$ is a special wildcard symbol which can be either 0 or 1. A **schema** is any template comprising a string of these three symbols; for example the schema $[1 * 1*]$ represents the following four strings: $[1010]$, $[1011]$, $[1110]$, $[1111]$.

The **order** of a schema S is the number of fixed positions (0 or 1) presented in the schema. For example, $[01 * 1*]$ has order 3, $[1 * 1 * 10 * 0]$ has order 5. The order of a schema is useful in estimating the probability of survival of a schema following a mutation.

The **defining length** of a schema S is the distance between the first and last fixed positions in the schema. For example, the schema $S_1 = [01 * 1*]$ has defining length 3. The defining length of a schema indicates the survival probability of the schema under crossover.

Given selection based on fitness, the expected number of individuals belonging to a schema S at generation $i + 1$ is equal to the number of them present at generation i multiplied by their fitness over the average fitness. An "above average" schema receives an exponentially increasing number of strings over the evolution. The probability of a schema S surviving crossover is dependent on the defining length: schemata with above-average fitness with short defining lengths will still be sampled at exponentially increasing rates. The probability of a schema S surviving mutation is dependent on the order of the schema: schemata with above-average fitness with low orders will still be sampled at exponentially increasing rates.

The **schema theorem** states that short, low-order, above-average schemata receive exponentially increasing representation in subsequent generations of a genetic algorithm. The **building-block hypothesis** states that a genetic algorithm navigates the search space through the re-arranging of short, low-order, high-performance schemata, termed *building blocks*.