CS2109S Midterms

by Zong Xun

Introduction to AI

An **agent** perceives its environment through **sensors** and act upon that environment through **actuators**. The **agent program** runs on the **physical architecture** to produce the agent function, f,which maps percept histories to actions.

We can specify a task environment with the **P.E.A.S** framework. **Rational agents** use percept sequence and built-in knowledge to maximize its performance measure, regardless of environment. They aim to maximize expected performance, whereas **Omniscience** maximizes actual performance.

Nature of Environments

- **Fully observable vs Partially Observable.** Able to detect all aspects that are relevant to decision making. "Relevance" depends on performance measure.
- **Single-agent vs Multi-agent.** Object with behaviour best described as maximising a performance measure whose values depends on another agent's behaviour.
- **Deterministic vs Strategic vs Stochastic.** Deterministic if next state is determined by current state and action executed. Strategic if environment is deterministic except for actions of other agents. Stochastic if the model deals with probabilities i.e Autonomous Driving.
- **Episodic vs Sequential.** Experience is divided into atomic "episodes", and the choice of action in each episode depends only on the episode itself. For sequential, the current action may affect all future decisions.
- **Static vs Dynamic.** Dynamic if environment can change as agent is deliberating. Semi-dynamic if environment is static but performance measure decrease with time.
- **Discrete vs Continuous.** Discrete means distinct and clearly defined inputs. Applies to the state of the environment, to how time is handled, and to the percepts and actions of the agent.

Structure of Agents

- **Table lookup agent.** Each percept sequence is mapped to an action. (1) storage, (2) inflexible, (3) long time to compute, (4) can we even build it?
- **Simple reflex agent.** Select actions based on current percepts, **ignoring** percept history. Uses condition-action (if-then) rules. May loop **infinitely** in a partially observable environment. (1) similar to table lookup
- **Model-based reflex agent.** Keep track of the world it cannot see. This internal state is updated through the transition model of the world (i.e. how the world changes independently from the agent) and the sensor model, (i.e. how the agent's actions affect the world).
- **Goal based agent.** In addition to tracking the state of the world, it also tracks a set of goals, then pick an action that brings it closer to the goal.
- **Utility-based agent.** Uses utility function to assign score to percept sequence. If utility function and performance measure aligns, the agent is rational.
- **Learning agent.** Uses *performance element* to select external actions, *critic* to give feedback, *learning element* for making improvements, and *problem generator* to suggest actions that lead to new and informative experiences.

Notes

1. Agent program takes in **current percept**, whereas the agent function takes in **entire percept history**.
2. Not all agent functions are implementable by programs, such as the halting problem or an intractable problem of arbitrary size in constant time.
3. Not all state abstractions can be mapped back to the real world i.e inaccurate representation.
4. All agents are rational if permutations of actions are reward-invariant i.e. random selection
5. Time affects rationality of agent decisions e.g. maximizing returns over 999 years, vs 99 years.
6. Performance measure should indicate **correctness** and **efficiency**. It is generally hard to define, and should focus on what one wants to achieve in the environment.

Solving problems by searching

Types of problems: Single state, sensorless (conformant), contingency, exploration.

A single state search problem can be defined formally as:

- **State space.** Legal states and possible dimensions.
- **Successor function.** A finite set of actions that can be executed at state S.
- **Transition model.** Result(s, a) returns the state that results from doing action a in state s.
- **Action cost.** Action-Cost(s, a, s') or c(s, a, s') gives a numeric cost of applying action a in state s.
- **Goal test.** Explicit e.g. n = 10, or implicit e.g. Checkmate(x)

Search Strategies

The order of node expansion. They are evaluated based on:

- **Completeness**: Does it find a solution if it exists?
- **Time Complexity**: Number of nodes generated
- **Space Complexity**: Number of nodes in memory
- **Optimality**: Does it find a least cost solution?

General Tree Search - Best First Search

Pick a node from the **open list** with min value of an evaluation function, *f(n)*. Child nodes are added only if they are not visited or previously visited with a higher path cost. By employing different *f(n)* functions, we get specific algorithms.

Uninformed Search

An uninformed search algorithm is given no clue how close a state is to the goal(s).

- **Breath first search**. Use early goal test for optimization. Complete if finite branching factor. Not optimal if nodes on the same level have different "costs" (2008 MT)
  - Optimal if all nodes on **same level** have same cost. **Intuition**: all the nodes at depth d is generated before nodes at depth d + 1.
- **Uniform cost search** (Best first search with f(n) = path cost) - Special case of A*
  - $h(n) = 0 \implies$ always Optimal
- **Depth first search** - Special case of best first search. Optimal if all solutions have the same depth.
- **Depth limited search** (DFS with depth limit, *l*)

$$N_{DLS} = b^0 + b^1 + \cdots + b^d$$

- **Iterative deepening search** (Slightly more overhead compared to DLS, but with DFS memory)

$$N_{IDS} = (d+1)b^0 + db^1 + \cdots + 2b^d - 1 + b^d$$

of depth d and branching factor, b.

Bidirectional search

Intuition: $2O(b^{\frac{d}{2}})$ is smaller than $O(b^d)$. Different search strategies can be employed for either half. However, *pred(succ(n))* and *succ(pred(n))* must be equal. Optimal if the evaluation function is the path cost.

Resolving redundant paths

(1) Memoize (2) Better heuristic (3) Pruning

Informed Search

Informed search strategy uses domain-specific hints about the location of goals to find solutions more efficiently than an uninformed strategy. These are known as heuristics.

Greedy Best First Search

Uses the function $f(n) = h(n)$. Expand the first node that appears to be the closest to goal. Tree: not complete, Graph: Complete only if finite search spaces.

A* Search

Uses the function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost so far to reach n.

Assuming all action costs are greater than $\epsilon > 0$ and state space is finite, A* search is complete. If there are infinitely many nodes with $f(n) \leq f(goal)$, then it is not complete. Whether it is **cost-optimal** depends on:

- **Admissibility.** $h(n) \leq h^*(n)$. If $h(n)$ is admissible, then A* using TREE-SEARCH is optimal.
- **Consistency.** $h(n) \leq c(n, a, n') + h(n')$ i.e. the triangle inequality holds. If h(n) is consistent, A* using GRAPH-SEARCH is optimal.

$$Consistent(h_1) \Rightarrow f(n') \geq f(n)$$

This means that f(n) is **non-decreasing** along any path.

- **Dominance.** h_1 if $h_2(n) \geq h_1(n)$ for all $n \Rightarrow h_2$ is better for search. Intuition: "Larger" steps towards the goal state, visiting fewer nodes, $\implies$ faster and quicker.

Notes on A*

1. Admissibility is a sufficient condition for optimality, but not necessary. If $h(n)$ overestimates by a constant factor, A* search is still optimal.
2. If the optimal solution has cost $C*$ and the second-best has cost C_1 and if h(n) overestimates some costs, but never by more than $C * -C_1$ then A* is guaranteed to return cost-optimal solution.
3. All nodes with $f(n) < C*$ is expanded. No nodes with $f(n) > C*$ is expanded. A* is efficient because it prunes away nodes that are not necessary for finding an optimal solution.
4. If $h(n) = -2g(n)$, then $f(n) = -g(n)$ which means $A* = DFS \implies$ not complete.

Memory Bounded Algorithms

The main issue with A* is its use of memory.

- **IDA***. Instead of using depth, use f-cost for cutoff. Min f-cost among unexplored nodes is used as the next cutoff.
- **Recursive Best-First Search**. Tracks f-value of the best alternative path available in the open list. If the current node exceeds this value, the recursion unwinds back to the alternative path. The f-value of each node along the original path is replaced with the current node's f-value. Optimal if h(n) is admissible. Space complexity is linear in the depth of the deepest optimal solution.

  Complain: Both IDA* and RBFS use too little memory. They may end up exploring the same state many times over.
- **Simplified MA* (SMA*).** Expand newest best leaf until memory is full. Then drop the oldest node with the worst f-value and backup the forgotten node's value to its parent. SMA* will only regenerate the forgotten sub-tree when all other paths have been shown to look worse.

Heuristics

- **Effective branching factor,** b. The smaller b is, the better.
- **Effective depth,** d. The smaller d is, the better.
- The $h_3 = max\{h_1, h_2\}$, where h_1 and h_2 are both admissible heuristics, is a dominant admissible heuristic.
- The cost of an **optimal solution** to a **relaxed problem** is an **admissible heuristic** for the original problem.
- Consider proving by exhuastion

Local Search Algorithms

Find best state according to **objective function**.

Hill Climbing Search Find local maxima by travelling to neighbouring states with steepest ascent. It can get stuck on **local maxima** and **plateaus**. Solutions are:

- **K-Sideways Move**.
- **Stochastic**. Chooses a random uphill move, with probabilities based on the steepness of the move.
- **Random-Neighbours**.Useful when a state has thousands of successors. May escape shoulders, but little chance of escaping local optima.
- **Random-Restart**. Perform a fixed number of steps from some randomly generated initial steps, then restart if no maximum found. Can escape shoulders, and high chance of escaping local optima.

Simulated Annealing Escape local maxima by allowing some bad moves, but gradually reduce their frequency. Accept bad moves with probability that decrease exponentially with the "badness" of the move.

Beam Search Perform k hill-climbing searches in parallel. In **local beam search**, information is shared between parallel threads, allowing the algorithm to abandon unfruitful searches. However, the k states may end up clustering together, making it k-times slower. **Stochastic beam search** alleviate this problem by choosing successors with probability proportional to their value.

Genetic algorithms A successor state is generated by combining two parent states. (1) selection (who gets to be parent), (2) crossover point (recombination procedure), (3) mutation (randomly flip bits).

Formulating local search problems

- Define an **objective function** (e.g. min-conflicts)
  - If there are more than 1 constraints, can adopt linear combination
- Define an initial candidate solution (e.g. greedy/randomized)
- Define transition function (e.g. swapping conflicts)
- Goal test (e.g. total conflicts = 0)

Notes on Local vs Informed

- Informed search is preferred if the search space is small, and there is **always** a solution.
- Local search is preferred in constraint satisfaction problems, and that **path to goal does not matter**.

Adversarial Search

Adversarial search problems arise from competitive environments wherein two or more agents have conflicting goals, such as in games.

Minimax Search Algorithm

Depth first exploration. Includes the state of the game, and the player's turn.

$$\text{MM(s)} = \begin{cases} \text{UTILITY(S, MAX)}, & \text{if IS-TERMINAL(S)} \\ max_{a \in A(s)} MM(R(s,a)), & \text{if TM(S)=MAX} \\ min_{a \in A(s)} MM(R(s,a)), & \text{if TM(S)=MIN} \end{cases}$$

Alpha-Beta Pruning

To optimize minimax when both b and m is large, we can introduce pruning. Pruning occurs when $\alpha \geq \beta$. Pruning **does not** affect the final outcome. With good move ordering, time complexity can be reduced to $O(b^{m/2})$, allowing us to examine twice as deep given the same computation power. Generally, the "best" nodes should be visited first. We can further optimize by using **transpositions table** to cache previously seen states or pre-computing the Opening and Closing moves.

Making use of heuristics

Computation of the entire search space is not practical. Instead, utility of states is estimated with **evaluation functions**. We can also make use of **cut-off depth** for the terminal test. This allows us to do iterative deepening, where each iteration gives rise to a better move ordering (assuming we keep past entries in the transposition table). Use the features of the state to compute an expected utility value. The contribution of each feature is assumed to be independent from other feature, which is not always true. For this reason, many programs use nonlinear combinations, so two or more features are combined.

Notes on Minimax

1. It is optimal only against an optimal opponent.
2. As long as the **relationship between leaves are retained**, a translation or scale will not affect the final outcome (not true if multiplying by negative value).
3. Assume left-to-right ordering for $\alpha - \beta$ pruning, the leftmost branch is always evaluated.

Machine Learning

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Decision Trees

A **decision tree** is a representation of a function that maps a vector of attributes to a single output value.
A **boolean decision tree** is equivalent to

$$output \leftrightarrow (Path_1 \vee Path_2 \vee \dots)$$

where each $Path_i$ is a conjunction of the form $(A_i = a_i \wedge A_n)$ of attribute value tests corresponding to a path from the root to a true leaf. We want to find the most compact decision tree. However, it is computationally intractable to construct all 2^{2^n} distinct decision trees with n Boolean attributes.
There are a total of 3^n distinct conjunctive hypotheses where each attribute in n is either in (positive), in (negative), or out.

A more **expressive hypothesis** space
- increases the chance that target function can be expressed
- increases the number of hypotheses consistent with training set

but may also result in worse predictions, due to "overfitting".

Information Theory

To find the most compact tree, we need to choose a root that offers the most information gain. Entropy is defined to be the measure of **impurity**. Entropy can be found using the following:

$$I(P(v_1), \dots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) log_2 P(v_i)$$

A chosen attribute, A, splits the training set E into subsets $E_1, \dots, E_v$, where A has v distinct vales.

$$remainder(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

Information gain is given by

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

The main intuition is that even partitions offer more information gain than uneven partitions.

Generalisation and Overfitting

Overfitting becomes more likely as the number of attribute grows, and less likely as we increase the number of training examples. We can prune certain nodes in the decision tree by removing those that are clearly irrelevant. We can detect that a node is irrelevant by using a statistical significance test. This can be done via χ^2 tests, also known as χ^2 pruning. Another way would be **cut-off sample**. (Will add more after midterms)

Notes on Decision Trees

1. Generating a decision tree, t_2 from a training set formed by another decision tree, t_1 will generate a tree that is logically equivalent, not equal! Two attributes might have the same IG, so its 50/50 who gets picked as root.
2. If the training set is small, even a reasonable hypothesis may overfit.
3. The simplest and smallest consistent DT is preferred.
4. Wrong predictions might indicate errors in some of the observations or missing a critical attribute. Another reason is some of the attributes may not be relevant.

Linear Regression and Classification

Types of loss functions

Loss function gives us an indication of how far off our predictions are compared to the actual values.

- Mean Squared Error

$$J(w_o, w_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2)$$

- Mean Absolute Error

$$J(w_o, w_1) = \frac{1}{m} \sum_{i=1}^{m} |h_w(x^{(i)}) - y^{(i)}|$$

Univariate Linear Regression

Form of $y = w_1 x + w_0$, where x is the input and y is the output. The loss function is minimized when its partial derivatives w.r.t w_0, w_1 is 0.

Gradient Descent

We can't always find partial derivatives that are zero. Gradient descent (i.e. reverse hill climbing) allows us to compute an estimate of the gradient at each point and move a small amount in the steepest downhill direction, until we converge on a point with minimum loss. Since the loss surface is convex, we will always arrive at the global minimum.

$$w_0 = w_0 - \alpha \frac{1}{m} \sum_{j=1}^{m} (w_0 + w_1 x_1^{(j)} + \dots - y_j)$$

$$w_i = w_i - \alpha \frac{1}{m} \sum_{j=1}^{m} (w_0 + w_1 x_1^{(j)} + \dots - y_j) \times x_i^{(j)}$$

This is otherwise known as **batch gradient descent**. Stochastic gradient descent updates weights by considering one point at a time, which is faster but does not guarantee global minima due to its randomness.

Multivariate linear regression

Gradient descent does not work well if the features have significantly different scales. We can perform

- **mean normalisation**.

$$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$$

  where x_i refers to the ith feature.
- **min-max normalisation**

$$x_i^j = \frac{x_i^j - min(x_i)}{max(x_i) - min(x_i)}$$

We can perform **polynomial regression** with linear regression. Make sure to do the necessary conversion by replacing all power terms with their corresponding linear terms.

Normal Equation

Non-iterative approach, that works well when the number of features, n is small.

$$w = (X^T X)^{-1} X^T Y$$

where $X^T X$ is invertible. Suppose not, perform the ERO/ECO and identify the linear dependency. Remove the necessary rows and cols and recompute the result.

Summary of algorithms

Criterion	BFS	Uniform cost	DFS	DLS	IDS	A*	Greedy BFS	Minimax
Complete	Yes	Yes	No	No	Yes	Yes	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil \frac{C^*}{\epsilon} \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^m)$	$O(b^m)$	$O(b^m)$
Space	$O(b^{d+1})$	$O(b^{\lceil \frac{C^*}{\epsilon} \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^m)$	$O(b^m)$	$O(bm)$
Optimal	Yes	Yes	No	No	Yes	Yes	No	Yes

The answer to all questions is that it depends.
Problem sets are free marks.
Life's hardest question is answering what to eat.
- Ben leong