1 Introduction

Agent percepts the environment using **sensors**, when then go into **functions**, which map into actions and **actuators** perform actions that modify the environment. A rational agent will choose the action that maximises its performance measure.

1.1 Environment

- **Fully Observable vs Partially Observable** Agent sensors has access to the complete state of the environment
- **Deterministic vs Stochastic** Next state is determined by current state and action. If deterministic but there is another agent, then **strategic**
- **Episodic vs Sequential** Each episode consists of agent perceiving and performing action, and choice of action depends only on episode. In sequential, current action affects all future actions
- **Static vs Dynamic** Environment doesn't change while agent is delibertaing. Semi dynamic if performance score changes with time
- **Discrete vs Continuous** Limited no of distinct, defined percepts and actions
- **Single Agent vs Multi Agent** Agent operating by itself.

1.2 Structure of Agents

- **Reflex Agents** Select actions on current percept, condition to action mapping
- **Model Based** Tracks world it can't see and updated through transitions
- **Goal Based** Tracks goals and picks action that brings it closer to goal
- **Utility Based** Score the next state, and pick the most optimal score

Agent must choose between

- **Exploitation** Maximising utility according to knowledge of the world
- **Explore** Learn more about the world

2 Solving Problems by Searching

Problem Solving Agents *Plan ahead* to consider sequence of actions that form a path to a goal, through **search**

2.1 Problem Formulation

- **States** Set of possible states for the environment to be in
- **Initial State** Starting State
- **Goal State** End State
- **Actions** Given State `s`, `actions(s)` returns finite set of actions that can be executed in `s`.
- **Transition Model** `transition(s, a)` returns the next state when the action has been applied on state
- **Action cost function** `cost(s, a, s')` gives the cost of applying action `a` to state `s` to reach state `s'`

2.2 Search Algorithms

Evaluation Criteria

- **Time Complexity** No of nodes expanded
- **Space Complexity** Max no of nodes in memory
- **Completeness** Does it always return a solution?
- **Optimality** Does it always find least cost solution?

Measure the above using: branching factor **`b`**, depth **`d`**, max depth **`m`**

`f(n)` `f` is the evaluation function, `n` is the node.

2.2.1 Uninformed Search / Tree Search

- **Breadth First Search**
  - Frontier: Queue
  - **`f(n)`**: `d` (depth of next node)
  - **Time**: $O(b^d)$
  - **Space**: $O(b^d)$
  - **Completeness**: Yes if B is finite
  - **Optimality**: Yes, if step cost is same
- **Uniform Cost Search** (Dijkstra)
  - Frontier: Priority Queue(cost from root to state)
  - **`f(n)`**: `tc + c(s, a, s')` (total cost + cost to next node)
  - **Time**: $O(b^{C*/e})$, $C*$ is the optimal cost, e is the min edge cost
  - **Space**: $O(b^{C*/e})$, $C*$
  - **Completeness**: Yes if $e > 0$ and $C*$ is finite
  - **Optimality**: Yes, if $e > 0$
- **Depth First Search**
  - Frontier: Stack
  - **Time**: $O(b^m)$
  - **Space**: $O(bm)$
  - **Completeness**: No, if depth is incomplete / loops
  - **Optimality**: No
- **Depth Limited Search**
  - Frontier: Stack, backtrack when depth limit **`l`** is reached
  - **Time**: $O(b^l)$
  - **Space**: $O(bl)$
  - **Completeness**: No
  - **Optimality**: No
  - $N_{dls} = b^0 + b^1 + ... + b^d$
- **Iterative Deepening Search**
  - Frontier: Stack, DLS with max depth from $0..N$
  - **Time**: $O(b^d)$
  - **Space**: $O(bd)$
  - **Completeness**: Yes
  - **Optimality**: Yes if step cost is same.
  - $N_{ids} = (d+1)b^0 + (d)b^1 + ... + (1)b^d$
- **Bidirectional Search**
  - Search from initial state and goal state at same time since $b^{\frac{d}{2}} + b^{\frac{d}{2}} < b^d$.

```
frontier = []; initial_state = x
visited = set() # if graph search
frontier.add(initial_state)
while len(frontier) != 0:
  state = frontier.pop()
  if state in visited: continue # if graph search
  visited.add(next_state) # if graph search
  for action in actions(state):
    next_state = transition(state, action)
    if next_state == goal: return solution
    frontier.add(next_state)
return failure
```

2.3 Informed Search Algorithms

- **Greedy Best Fit Search**
  - Frontier: Priority Queue(`f(n)`)
  - **`f(n)`**: `h(n)`: Heuristic: estimated cost from `n` to goal.
  - **Time**: $O(b^m)$, Good heuristic improves
  - **Space**: $O(b^m)$ Keeps all nodes in memory
  - **Complete**: No
  - **Optimal**: No (Doesn't consider cost so far)
- **A***
  - Frontier: Priority Queue(`f(n)`)
  - **`f(n)`**: `g(n) + h(n)`, `g` : total cost, `h` : estimated cost
  - **Time**: $O(b^m)$, Good heuristic improves
  - **Space**: $O(b^m)$ Keeps all nodes in memory
  - **Complete**: Yes
  - **Optimal**: Yes (Doesn't consider cost so far)

2.4 Heuristics

2.4.1 Admissible

Admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach goal state from n. An admissible heuristic never over-estimates the cost to reach the goal, i.e. its a conservative estimate

If $h(n)$ is admissible, A* using **tree search** is optimal

2.4.2 Consistent

If it obeys the triangle inequality, $\forall n, h(n) \leq c(n, a, n') + h(n')$, or estimated cost of reaching goal node through node n is $\leq$ to the estimated cost of reaching goal node through n' + cost of going to node n' from n.

If our heuristic is consistent, then the first time we visit a node, the estimated cost to the goal $h(n)$ is guaranteed to be smallest. Therefore, we do not need to visit the node again as any other path we might visit it with has a larger overall cost, and graph search (i.e. tree search with memoisation) is optimal. However, if our heuristic is not consistent, then consider a path that we might visit later where $h(n) \geq c(n, a, n') + h(n')$. Then, if we have visited $h(n)$ already, we still need to revisit it as we have found another shorter path later on in our traversal, hence making graph search sub-optimal.

2.4.3 Dominance

If $\forall n, h_2(n) \geq h_1(n)$, then h_2 dominates h_1 and h_2 would be better for search.

2.4.4 Creating Admissible Heuristics

Cost of an optimal solution to a relaxed problem (problem with fewer restrictions) is an admissible heuristic for the original problem

2.5 Local Search

When path to solution not important, state is the solution (chess, sudoku, bin packing)

Keep the current state, and iteratively try improving using heuristic we define. Local search can get stuck in local minima/maxima, so random restarts can help get better results

2.5.1 Trivial Algorithms

- **Random Sampling** Random sample a state until solution is found
- **Random walk** Go to random neighbours until solution found

2.5.2 Non-trivial Algorihthms

- **Hill Climbing** Pick the best among neighbours, repeat
- **Simulated Annealing** Hill climbing but allows bad moves
- **Beam Search** k-kill climbing in parallel
- **Genetic** Marry the best, mutate, repeat

2.6 Adversarial Search

When trying to find moves against rational agent.

2.6.1 Minimax

heuristic $h(n)$ that defines "goodness" of current state. We want to *maximise* $h(n)$, whereas opponent wants to *minimise* $h(n)$.

We can optimise minimax algo by introducing α/β pruning, where α is best value for max player, and β is best value for min player. If at any node, α and β does not overlap($\beta \leq \alpha$), we can prune that node.

- **Time**: $O(b^m)$
- **Space**: $O(bm)$
- **Complete** Yes, if tree is finite
- **Optimal** Yes, against optimal opponent

```
def alpha_beta_search(state):
  _ = max_value(state, -INF, INF)
  return action in succesors(state) with value v

def max_value(state, alpha, beta):
  if is_terminal(state): return utility(state)
```

```
  v = -INF
  for action, next_state in successors(state):
    min_v = min_value(next_state, alpha, beta)
    v = max(v, min_v)
    if v >= beta: return v
    alpha = max(alpha, v)
  return v

def min_value(state, alpha, beta):
  if is_terminal(state): return utility(state)
  v = INF
  for action, next_state in successors(state):
    max_v = max_value(next_state, alpha, beta)
    if v <= alpha: return v
    beta = min(beta, v)
  return v
```

3 Machine Learning and Decision Trees

3.1 Supervised Learning

Learns from being given the right answers

- **Regression**: Predict Continuous outputs
- **Classification**: Predict discrete outputs

Assumption: y is generated by true mapping function $f : x \to y$. We want to find a hypothesis $h : x \to \hat{y}$

3.2 Performance Measure

We can measure error for regression:

- Absolute Error: $|\hat{y} - y|$
- Squared Error: $(\hat{y} - y)^2$

For a set of N examples, we can compute average error for regression:

- Mean Squared Error: $\frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}_i - y_i\right)^2$
- Mean Absolute Error: $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$

Average correctness for classification: Accuracy $= \frac{1}{N}\sum_{i=1}^{N} 1_{\hat{y}_i = y_i}$

Confusion Matrix:

Accuracy $= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}}$

Precision P $= \frac{\text{TP}}{\text{TP} + \text{FP}}$ (Maximise if FP is very costly, e.g. email spam)

Recall R $= \frac{\text{TP}}{\text{TP} + \text{FN}}$ (Maximise if FN is bad, e.g. Cancer)

F1 $= \frac{2}{(\frac{1}{\text{P}}) + (\frac{1}{\text{R}})}$ (Maximise if FN is bad, e.g. Cancer)

3.3 Decision Trees

Choosing an attribute to split a decision tree: Ideally, select attribute that splits all examples into 2 distinct groups

The amount of information at a given node is the entropy, $I(P(v_1), ..., P(v_n)) = -\sum_{i=1}^{n} P(v_I)\log_2 P(v_i)$, where v_i are the different classifications of the dataset. For a binary classification, (Positive, Negative), $I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{n}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$

When an attribute is divided into subsets, we can calculate the *Information Gain* (reduction in entropy) by

$$\text{IG(Attrib)} = I\left(P\left(\tfrac{p}{p+n}\right), P\left(\tfrac{n}{p+n}\right)\right) - \sum_{i=1}^{v}\frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

4 Misc

4.1 Minimax Intuition

- In Maximiser:
  - When $v \geq \alpha$, update α.
  - If $v \geq \beta$, prune the rest
- In Minimiser:
  - When $v \leq \beta$, update β.
  - If $v \leq \alpha$, prune the rest

4.2 Proof of Admissibility

- If a relaxed version of the problem's optimal solution is H_a, then H_a is an admissible heuristic for the current problem
- Check if $h_a(\text{Goal}) > 0$. If it is, then its inadmissible
- Compare cost of moves vs maximum rate of loss of heuristic. This is the intuition when checking consistency because if one move improves heuristic "too much", it's likely to be inconsistent. For example, in the snake question, if the snake can eat the apple and improve heuristic by 5 when the cost of the move to eat the apple is only 1, we can immediately say it's inconsistent.
- Proof that it fails by counterexample / (contradiction / contrapositive)
- Proof that it passes by contradiction / contrapositive
- Proof that it passes by abuse of inequalities
- Proof that it passes by proving the correctness for extreme cases, and proving that everything in between is therefore correct (dodgy proof technique)
- Lemma that guarantees admissibility as long as we have consistency and h(Goal)=0
- Contrapositive of this Lemma: inconsistent as long as inadmissible and h(Goal)=0

4.3 Proof of InAdmissibility

- Show that $\exists n, H_a(n) > H^*(n)$

4.4 Proof of Inconsistency

- First show that $H_a(\text{Goal}) = 0$
- Try show that $H_a(n) > c(n, a, n') + H_{a(n')}$, assuming n' is goal, and c is the true cost function. Then show $H_a(n) > c(n, a, n')$, that the cost calculated is >
- If cost is always 1, then try to ensure that $H_a(n)$ is always <=1.

4.5 Information Gain

- Pick the one with the lowest remainder