AI

Aamod Varma

CS - 3600, Spring 2025

Contents

Chapter 1

Search

We will go over **uninformed** (given no information except problem definition) and **informed** search (given some guidance on where solutions are) algorithms.

1.1 Problem solving agents

We need to formulate the problem first by deciding what actions and states to consider, given a goal. Process of looking for a sequence of actions to reach goal is called **search**.
An **incremental formulation** starts with the empty state and adds each state incrementally. A **complete-state formulation** starts with the set of things that can be on and then tries to optimize them.

1.2 Searching

We start at the root node. We then expand the current state, so the branches of the current node we are at. Then we choose which of these three children to consider. Each child is called a **leaf node** and set of all leaf nodes to be expanded is called a **frontier**.
Some graphs have loopy paths such that one ends up traversing a loop in the graph indefinitely.
The general approach to a graph search algorithm is,

```
func GRAPH-SEARCH(problem) returns solution,failure
    init frontier with initial state
    init explored set as empty
    loop do
        if frontier empty return with failure
        choose leaf node, remove from frontier
        if node is goal return solution
        add node to explored set
        expand node
            if children not in explored add to frontier
```

1.2.1 Measuring performance

- **Completeness**: Is the also guaranteed to find solution if it exists
- **Optimality**: Does it find the optimal solution
- **Time complexity**: How long to find solution
- **Space Complexity**: How much memory is required

b, **branching factor** is the max num of successors of any node and d, **depth** is the depth of the shallowest goal and m, is the **max length** of any path in the state space.

1.3 Uninformed Search

1.3.1 Breadth-first search

Root node is expanded first, all successors are next and their successors. So all nodes expanded at a given depth before we go on to the next level.
Need a **FIFO** (first in first out) queue.
Metrics are,

- Complete: Yes. If the goal is in some finite depth, BFS will eventually reach it.
- Optimal: No. Shallowest goal is not necessarily the most optimal
- Time Complexity: At depth d we will have b^d nodes so we have,

$$b + \cdots + b^d = O(b^d)$$

- Space complexity: We store every expanded in explored. So we have $O(b^{d-1})$ nodes in the explored and $O(b^d)$ in the frontier. Latter dominates so that is the space complexity.

Main problems are memory requirements.

Uniform-cost search

Is optimal with any step-cost function. Instead of expanding the shallowest node, UCS expands the node with the lowest path cost $g(n)$. Stores frontier as a priority queue (heap).
The goal test on a node is when we expand the node and not when we add it to the frontier. Because when we add we may be on a sub optimal path and a better path may exist in the current frontier.

- Complete: Yes if provided cost is non-zero, if not it can get stuck in an infinite loop.
- Optimal: Yes. Optimal in general.
- Time Complexity: Worse than breadth search. If all cost is same it is equivalent to breadth in terms of function however even if it finds the goal node as it expands all of its children its b^{d+1}

1.3.2 Depth-first search

Expands deepest node in frontier. Basically deepest level until nodes have no children. Then goes one up to next deepest node with unexplored successors and so on. Uses a **LIFO** queue or a **STACK**.
Generally implemented using a recursive function.

- Complete: In finite state space if Graph-search is used its complete (as we have explored set to prevent loops). If Tree-search is used its not complete. However in finite state space both fail.
- Optimal: Both versions are non-optimal
- Time Complexity: Depends on state space. We have $O(b^m)$ where m is the max depth. Can be greater than the state space itself (if finite).

Depth-first only advantage is its space complexity which is $O(bm)$ as you don't need to store all children. Can delete if a node has been expanded, can be removed after all its children are explored.
Backtracking search is a variant uses less memory. Only one successor is generated at a time than all successors. Each partially expanded node knows what to expand next.

1.3.3 Depth-limited search

We limit the depth to l. Assume nodes at level l have no children. Is obviously incomplete as if goal exists at level $> l$ it won't be found.

1.3.4 Iterative deepening depth-first search

Iteratively increase the limit - first 0, then 1,2, ... until goal is found. Combines benefit for DFS and BFS searches. Makes it complete. Space complexity is $\mathbf{O(bd)}$. Time complexity is $\mathbf{O(b^d)}$
Is the preferred uninformed search algo.

1.3.5 Bidirectional search

We run two simultaneous searches - from forward from initial state and backward from goal. Motivation is $b^{d/2} + b^{d/2}$ is better than b^d.
Search ends when frontiers of the two searches intersect (The first sol might not be the most optimal). Additional search required.

- Complete: Is complete
- Optimal: Depends on stopping condition.
- Time Complexity: $O(b^{\frac{d}{2}})$

1.3.6 Summary of uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

1.4 Informed (Heuristic) Search

Uses problem-specific knowledge beyond definition of problem - more efficient than uninformed.
General approach - **best-first search**. Node is selected based on an **evaluation function** $f(n)$ (a cost estimate). So node with lowest $f(n)$ is chosen. Choice of f determines the search strategy. $h(n)$ is the heuristic function which is a part of $f(n)$.

$$h(n) = \text{estimated cost of the cheapest path from } n \text{ to a goal.}$$

$h(n)$ is similar to $g(n)$ but $h(n)$ is dependent on the state and not the path. For instance one node in UCS might have different $g(n)$ depending on the path it takes. However it can only have the same $h(n)$.

1.4.1 Greedy best-first

Expands the node closest to the goal. So $f(n) = h(n)$. Eg. for maps our $h(n)$ can just be the straight line distance from node to goal. It is not optimal - rather greedy.
Graph version is complete in finite spaces but not in infinite, tree is not complete (can be stuck in loops). Space and time complexity are $O(b^m)$ where m is max depth. Different heuristics can make it better.

1.4.2 A* search

Our evaluation function is,

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is path cost from start node to n and $h(n)$ is cost of cheapest path from n to goal.
So we try the node with lowest $g(n) + h(n)$. Works only if $h(n)$ satisfies certain conditions. It is both **Complete and Optimal**

Conditions for optimality

- $h(n)$ needs to be a **admissible heuristic**: One that never overestimates the cost to reach the goal. Here $g(n)$ is the actual cost till current node. So if $h(n)$ overestimates then $f(n)$ overestimates. If this happens then good paths could never be explored (non-optimal).
- $h(n)$ needs to be **consistent** (relevant for graph search only): For every node n and every successor n' of n generated by action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching goal from n'.

$$h(n) \leq c(n, a, n') + h(n')$$

Optimality of A*

The tree search version is optimal if $h(n)$ is admissible and the graph search is optimal if $h(n)$ is consistent.
If $h(n)$ is consistent then the values of $f(n)$ along any path is non decreasing,

$$f(n) \leq f(n')$$

$A*$ is **optimally efficient** for any given consistent heuristic. No other optimal algo is guaranteed to expand fewer nodes than A^* because A^* expands all nodes with $f(n) < C^*$ where C^* is true cost of the optimal solution.
So A^* is **complete, optimal and optimally efficient** among all such algorithms.

Chapter 2

More Searches

We explore purely **local searches** in the state space, evaluating and modifying one or more current states rather than exploring paths from an initial state.

2.1 Local Search

In some problems, path to goal is irrelevant and only the final configuration matters (eg. place 8 queens such that they don't attack each other).
Local search algorithms operate using a single current node and move only to neighbors of that node. Paths are not retained. Two advantages,

- Use little memory
- Can find reasonable solutions in large or infinite (continuous) state spaces.

Local search is useful for solving pure **optimization problems** in which aim is to find the best state according to an **objective function**.
We can consider the **state-space landscape**. It has a location (the state) and elevation (value of objective function or heuristic). If elevation is heuristic want to find the lowest valley - a **global minimum**, if its an objective function we want to find the highest peak - a **global maximum**. A **complete** local search algo always finds a goal, an **optimal** one always finds a global max/min.

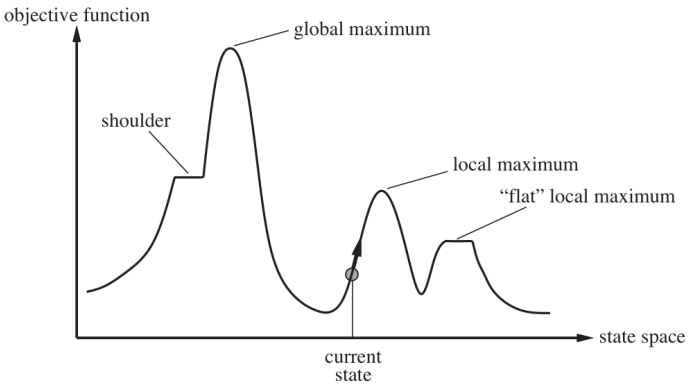

Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

A loop that continually moves in the direction of increasing value - uphill. Terminates when there is no neighbor with higher value. Does not maintain a search tree, data structure only maintains current node and value of objective function. Does not look ahead beyond immediate neighbors.

Example. In the **8-queens problem** we start with all 8 queens on the board. Successor of a state are all possible states by moving a single queen to another square in the same column (each state has $8 \times 7 = 56$ successors). Heuristic h is num of pairs of queens that are attacking each other. Global minimum is when no queen are attacking each other so when $h = 0$ $\diamond$

Sometimes called **greedy local search** as it grabs a good neighbor without thinking anything ahead.

1. Local maxima: A peak that is higher than each of its neighbors but lower than global minimum. Hill-climbing algos will reach here but will be stuck here as all its neighbors are worse than its current position.
2. Ridges: Result in a sequence of local maxima that is difficult to navigate.
3. Plateaux: A flat area of the state-space landscape. No uphill from which progress is possible. Algo might get lost.

In each case algo can't make any more progress. For the 8 queen problem, hill-climbing algos get stuck 86% of the time.
There are other variants such as **stochastic hill climbing** - chooses at random from the uphill moves: probability can vary with the steepness. Converges slowly but sometimes finds better solutions. **First choice hill climbing** uses stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. All these algos are incomplete (as we saw with the 8 queen problem).
Random-restart hill climbing randomly restarts to another initial state space if the default gets stuck. Is able to solve the 8 queens problem.

2.1.1 Simulated annealing

A hill-climbing algo that never makes a downhill move toward states with lower value is guaranteed to be incomplete (as it can get stuck at a local maxima or minima). A pure random walk for instance is complete but inefficient. A solution might be **combining** hill climbing and random walk so its both efficient and complete.
We switch pov from maximizing to minimizing (gradient descent). Analogy is pingpong ball on a bumpy surface. Letting the ball roll, it will stop at a local minima. If stuck we can shake the surface and hopefully bounce the ball out of there. So **simulated annealing** is shaking hard and then gradually reducing the intensity of the shaking.
So instead of picking the best move (like in hill climbing) it picks a **random move**. If the move improves the situation it is accepted, else the algo accepts the move with some probability less than 1 (decreases exponentially with how bad the move is). Probability also decreases as the temperature goes down. (If T goes down slowly, probability of finding global optimum approaches 1).

2.1.2 Local beam search

Initialized with k random states rather than just one in memory. At each step, all successors of all k searches are generated. If any one finds goal, algorithm halts. Else it selects k best successors from the complete list and repeats. **Not** the same as k random restarts in parallel. In local beam search useful information is **passed among the parallel search**. For instance if the kth search is stuck and the $k-1$ search finds two good successors, the kth search with move to one of the two.
Can cause a lack of diversity where they converge to being concentrated in a small region making it a little more than an expensive hill climbing. **Stochastic beam search** is a variant - chooses k successors at random (instead of the best) probability of choosing is a function of its value.

2.1.3 Genetic algorithms

Variant of stochastic beam search in which successor states are generated by combining two parent states than by modifying a single state.
So we start with k random initial states (the population). To produce the next generation of states we have an objective function (in GA terms the **fitness function**). The function returns higher value for better states. For instance for the 8 queen problem we can construct a 8 digit string each number representing the position of the queen in that column. We can mate two strings by using the first half of the first and second half of the second (we decide which strings to mate using their fitness function).
Crossover often takes **long steps** early in and short steps **later on**.

Chapter 3

Adversarial Search

We are in competitive environments where agents goals are in conflict. Most common games are **zero-sum games** of **perfect information**. Which means its **deterministic** in a fully **observable environment**. The utility values at the end are always equal and opposite (eg. in chess one wins and other loses). Games are hard to solve. Chess games can go upto 50 moves by each player, if branching factor is around 35 that means the search tree has 35^{100} or 10^{154} nodes (search graph has 10^{40}). Optimality and efficiency must be balanced for games.

Pruning allows us to ignore portions of the search tree that make no difference to the final choice and **heuristic evaluation functions** allow us to approximate the true utility of a state without doing complete search.

Consider a game with two players, MIN and MAX. MAX moves first and they take turns until the game is over. At the end, points are given to the winner and taken from the loser. For instance tic-tac-toe. Initially MAX has 9 possible moves. Plays alternative between MAX and MIN. Number on each leaf node is the utility value of the terminal state from the pov of MAX; so high value are good for MAX and bad for MIN.

3.1 Optimal decisions

In adversarial search, MIN prevents us from reaching the goal state (pov of MAX). So, MAX needs to find a contingent strategy to figure out a move. So MAX must consider every possible response by MIN to its moves. Given a game

tree, the optimal strategy can be determined from the **minimax value** of each node. The minimax value of a node is the utility of being in the corresponding state assuming that both players play optimally from there to the end.

3.1.1 Minimax algorithm

Computes the minimax decision from the current state. Uses recursive computation of the minimax values of each successor. Time complexity is O^{b^m} as it does depth first search to identify value for all possible paths.

In terms of efficiency it is like DFS with time complexity of $O(b^m)$ and space complexity of $O(bm)$.

3.1.2 Alpha-Beta Pruning

In the general case we are pruning children of the MIN node. We compute the MIN value at some node n and loop over n's children. We keep track of MAX's best value (α) so far from any choice point along the current path from the root. If n becomes worse than MAX's best then we can prune $n's$ other children as we know n wont be played anyways.

Similarly we keep track of MIN's best value (β) so far from any choice along the current path from the root. If $n's$ children are worse (greater than β) then we don't need to explore its children because we know we won't explore n anyways.

- The pruning has **no effect** on minimax value computed for the root.
- Values of intermediate nodes might be wrong
  - Children of the root may have the wrong value
  - Can't do any action selection
- Good child ordering improves effectiveness of pruning
- With perfect ordering
  - Time complexity becomes $O(b^{m/2})$
  - Doubles solvable depth

3.1.3 Evaluation Functions

Score non-terminals in depth-limited search. The ideal functino returns the actual minimax value of the position. However in depth limited we don't have access to the leaves. So it's typically weighted sum of features, eg. num white queens - num black queens

3.1.4 Expectimax Search

Instead of worse-case outcomes we want values to reflect the average-case. **Expectimax serach** computes average score under optimal play.

- Max nodes as in minimax search
- Chance nodes are like min nodes but the outcome is uncertain
- Calculate their expected utilities - take weighted average of children.

So the max value calculation is the same, but instead of min value calculation we use our expected value using probabilities based on the actual value returned.

If your opponent is running depth 2 minimax, using the result 80% of the time and moving randomly otherwise we use **Expectimax**.

3.1.5 Multi-Agent Utilities

If we have multiple players or don't have a zero-sum game. Then we have,

- Terminals have **utility tuples**
- Node values are tuples as well
- Each player maximize its own component in the tuple

3.1.6 Monte Carlo Tree Search

Methods that are based on alpha-beta search assume a fixed horizon. Combines two ideas,

- Evaluation by rollouts - play multiple games to termination from a state s and count wins and losses.
- Selective search - explore parts of tree that will help improve the decision at root, regardless of depth.

Rollouts

For each rollout we,

- Repeat until terminal - play a move according to a fast policy
- Record the result

The fraction of wins correlate with the true value of the position.

MCTS 0

- Do N rollouts from each child of the root, record fraction of wins
- Pick the move that gives best outcome by this metric

MCTS 0.9

- Allocate rollouts to more promising nodes.

MCTS 1

- Allocate rollouts to more promising nodes.
- Allocate rollouts to more uncertain nodes.

UCB (Upper Confidence Bound) heuristic

$$UCB(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\log N \frac{PARENT(n)}{N(n)}}$$

Where,

- $N(n)$ = number of rollouts from node n
- $U(n)$ = total utility of rollouts (e.g. # wins)

3.1.7 MCTS 2

Repeat until out of time,

- Given current tree, apply UCB to choose a path down to a leaf node n
- Add a new child c to n and run a rollout from c
- Update the win counts from c back up to the root

Choose the action leading to the child with the highest N

3.1.8 No min or max

- Value of a node $U(n)/N(n)$ is a weighted sum of child values.
- as $N \to \infty$, the vast majority of rollouts are concentrated in the best children. So the weighted average is essentialy the max/min
- As $N \to \infty$ UCT selects the minimax move.

3.1.9 MCTS + ML

- MCTS can be paired with rollout policies that are neural networks with reinforcement learning and expert human moves.
- Also have a trained value function to better predict node's utility.

3.1.10 Summary

- When games require decisions when optimally is impossible,
  - Bounded-depth search and approximate evaluation functions.
- Games force efficient use of computation
  - Alpha-beta pruning, MCTS

Chapter 4

Logic

We have,

1. Propositional Logic I
2. Propositional Logic II
3. First-order Logic

4.1 Knowledge

- Agents quire knowledge through perception, learning, language
  - Knowledge of the effects of actions
  - Knowledge of how the world affects sensors
  - Knowledge of the current state world

Your knowledge base is the set of sentences in a formal language.

4.2 Logic

Syntax: What sentences are allowed?
Semantics:

- What are the possible worlds?
- Which sentences are true in which worlds?
- Propositional Logic
  - Syntax: $P \vee (\neg Q \wedge R)$; $x_1 \Leftrightarrow$ Raining $\Rightarrow \neg$ Sunny
  - Possible worlds: P, Q, R, S is true or false
  - Semantics: $\alpha \wedge \beta$ is true if both alpha and beta is true
- First-order Logic
  - Syntax: $\forall x \exists y P(x, y) \wedge \neg Q(\text{Joe}, f(x)) \Rightarrow f(x) = f(y)$

Other kinds of logic are, **Relational databases:**

- Syntax: relational sentences, e.g. Sibiling(A, B)
- Sentences: Every sentence in the DB are true, everything else is false
  - SQL is some variant of first-order logic
  - Cannot express disjunction, implication, universals, etc

4.3 Inference

4.3.1 Entailment

Entailment: $\alpha \models \beta$ means that α entails β or β follows from α iff in any world where α is true, β is also true or that,

$$models(\alpha) \subseteq models(\beta)$$

For instance if,

$$\alpha_1 \text{ is } \neg Q \wedge R \text{ and } \alpha_2 \text{ is } \neg Q$$

Then we know that,

$$\alpha_1 \models \alpha_2$$

4.3.2 Proofs

A proof is a demonstrating of entailment between α and β.
Sound algorithm: Everything it claims to prove is entailed
Complete algorithm: Everything that is entailed can be proved

1. Method 1: Model checking
   - For every possible world, if α is true make sure that β is true too.
   - Works for propositional logic (finite many worlds), not easy for first-order logic (because of quantifiers)
2. Method 2: Theorem Proving
   - Search for a sequence of proof steps that lead from α to β
   - E.g. $P \wedge (P \Rightarrow Q)$ we can infer Q by Modus Ponens

4.4 Propositional logic syntax

Given a set of proposition symbold $\{X_1, \ldots, X_n\}$

- X_i is a sentence
- If α is a sentence then $\neg\alpha$ is a sentence
- If α and β are sentences then $\alpha \wedge \beta, \alpha \vee \beta, \alpha \Rightarrow \beta, \alpha \Leftrightarrow \beta$ are sentences.
- There are no other sentences
- Prop logic has limited expressive power
- We can't generate all the sentences by hand - let the code generate it
- In first order logic we need $O(1)$ transition model sentences

4.5 Entails vs Implies

Entails: $\alpha \models \beta$
Implies $\alpha \Rightarrow \beta$
KB is a set of sentences then,

- If $\alpha \Rightarrow \beta \in$ KB then $\alpha \wedge$ KB $\models \beta$ (modus ponens)

4.6 Reasoning tasks

- Localization with a map and local sensing
  - Given an initial KB, plus a sequence of percepts and actions where am I?
- Mapping with a location sensor
  - Given an initial KB, plus a sequence of percepts and actions what is the map?

4.7 Summary

- One possible agent architecture: knowledge + inference
- Logics provide a formal way to encode knowledge
- A simple KB for pacman covers the initial state, sensor model, and transition model

4.8 Decision Networks

4.9 Value of Perfect Information

The value of perfect information is,

$$VPI(E'|e) = MEU(e, E') - MEU(e)$$

Here $MEU(e)$ is the maximum expected utility given the current evidence e or,

$$MEU(e) = \max_a \sum_s P(s|e)U(s, a)$$

Essentially it is the maximum of the expected utility given the initial evidence. For instance consider we want to decide where to drill a oil rig, we have at location 1 (a_1) or at location 2 (a_2). In this case e would reprint the initial evidence. Say we don't have any evidence other than the probability that the oil has fifty percent chance of being in either we could compute the maximum expected utility as,

$$EU(a_1) = P(s_1)U(s_1, a_1) + P(s_2)U(s_2, a_1) = .5 \times k + .5 \times 0 = .5k$$
$$EU(a_2) = P(s_1)U(s_1, a_2) + P(s_2)U(s_2, a_2) = .5 \times 0 + .5 \times k = .5k$$
$$MEU = max(EU(a_1), EU(a_2)) = .5k$$

$MEU(e, E')$ is the max utility given the current evidence e and new set of evidence E' of which let e' be a specific piece of evidence so,

$$MEU(e, e') = \max_a \sum_s P(s|e, e')U(s, a)$$

and,

$$MEU(e, E') = \sum_{e'} P(e'|e)MEU(e, e')$$

Lets assume our only piece of new evidence in this is where the actual oil rig and let that be e' so we need now $MEU(e, e') = MEU(e, E')$. We need,

$$EU(a_1|e, e') = P(s_1|e, e')U(s_1, a) + P(s_2|e, e')U(s_2, a)$$

In this case if a_1 is we drill in oil rig 1. Consider our new evidence tell us that the oil rig is actually in the first location then we have $P(s_1|e, e') = 1$ and $U(s_1, a_1) = k$ we get,

$$EU(a_1|e, e') = k$$

and

$$EU(a_2|e, e') = 0$$

So our,

$$MEU(e, E') = k$$

And hence our,

$$VPI(E'|e) = k - k/2 = k/2$$

Chapter 5

Utilities

Utilities are functions from outcomes to real numbers that describe an agent's preferences.

- In a game, may be simple (+1/-1)
- Summarize the agent's goals
- Theorem: any "rational" preference can be summarized as a utility function

We hard-wire utilities and let behaviors emerge

5.1 Maximum Expected Utility

- A rational agent should choose the action that maximize its expected utility, given its knowledge.
- For average case expectimax reasoning, we need magnitudes to be meaningful

5.2 Preferences

An Agent must have preferences among:

- Prices: A, B, etc
- Lotteries: situations with uncertain prizes $L = [p, A; (1 - p), B$

5.2.1 Rational Preferences

We want some constraints on preferences before we call them rational eg,

$$\text{Axiom of Transitivity: } (A > B) \wedge (B > C) \Rightarrow (A > C)$$

There are costs to irrationality. An agent with intransitive preferences can be induced to give away all its money
Axioms of rationality,

1. Orderability: $(A > B) \vee (B > A) \vee (A \sim B)$
2. Transitivity $(A > B) \vee (B > C) \vee (A > C)$
3. Continuity $(A > B > C) \Rightarrow \exists p[p, A; 1 - p, C] \sim B$
4. Substitutability $(A \sim B) \Rightarrow [p, A; 1 - p, C] \sim [p, B, 1 - p, C]$
5. Monotonicity $(A > B) \Rightarrow (p \geq q) \Leftrightarrow [p, A; 1 - p, B] >^{\sim} [q, A; 1 - q, B]$

5.3 Utilities of Sequences

What preferences should an agent have over prize sequences?
More or less? $[1, 2, 2]$ or $[2, 3, 4]$
Now or later? $[0, 0, 1]$ or $[1, 0, 0]$

5.3.1 Stationary preferences

If we assume stationary preferences,

$$[a_1, a_2, \dots] > [b_1, b_2, \dots] \Leftrightarrow [c, a_1, a_2, \dots] > [c, b_1, b_2, \dots]$$

Additive discounted utility:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

where $\gamma \in (0, 1]$ is the discount factor

Chapter 6

Markov Decision Processes

Say we're working with a maze like problem - a grid for instance. Movements are noisy, so we have a probability that a specific action leads you to a specific place. For instance the action North might only take you North 80% of the time. At each step the agent receives a reward, reward can be negative. The big rewards come at the end. The overall goal is to maximize the sum of rewards.

6.1 Introduction

An MDP is defined by,

1. Set of states $s \in S$
2. Set of actions $a \in A$
3. A transition function $T(s, a, s')$ - the dynamics
    - Probability that action a from s leads to s'.
4. A reward function $R(s, a, s')$ (sometimes $R(s)$ or $R(s')$)
5. A start state and maybe a terminal state

One way to solve it is using expectimax search.

6.2 Markov

Means that given the present, the future and past are independent. So essentially our action outcomes only dependent on the current state, so we have,

$$P(S_{t+1} = s' | (S_t = s_t, A_t = a_t), (S_{t-1} = S_{t-1}, A_{t-1} = a_{t-1}), \ldots, S_0 = s_0)$$

$$=$$

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

So only the current state matters and not the history.

6.3 Policy

A policy π gives an action for each state. So an optimal policy is one that maximizes the expected utility if followed. Expectimax for instance did not compute entire policies, it computed the action for a single search only.

6.4 Discounting

It's reasonable to maximize the sum of rewards. But in some cases we prefer rewards now to rewards later. We can exponentially decay the value of later rewards.

Each time we descend a level, we multiply in the discount once. For instance we have,

$$U([a, b, c]) = a + \gamma b + \gamma^2 c$$

Discounting also helps in convergence as in some cases we can get infinite rewards. Some other solutions are,

- Finite horizon: Terminate after fixed T steps.
- Discounting: use $0 < \gamma < 1$
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached.

6.5 Optimal Qualities

1. Value utility of a state (s): $V(s)$ = expected utility starting in s and acting optimally.
2. Value utility of a q-state (s, a): $Q(s, a)$ = expected utility starting out having taken action a from state s and acting optimally.
3. Optimal policy: $\pi(s)$ = optimal action from state s

6.6 Bellman Equations

The recursive definition of value is,

$$\begin{aligned} V(s) &= max_a Q(s, a) \\ Q(s, a) &= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')] \\ V(s) &= max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')] \end{aligned}$$

Essentially V is the max q value across all the actions we can take and the q-value of a state given an acting is summing the quantity $T(s, a, s')[R(s, a, s') + \gamma V(s')]$ across all reachable states given that action (for instance given action North we might have $T_N = 0.8, T_S = 0.1, T_E = 0.1, T_W = 0$)

6.7 Value Iteration

We start with all $V_0(s) = 0$ at time step 0. And for each s we do,

$$V_{k+1}(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

We repeat this until convergence (until the max change is smaller than our change margin). The complexity of this would be,

$$O(S^2 A)$$

6.8 Policy Evaluation

In value iteration we max over all the possible actions from a given state. But in policy evaluation we use the optimal policy (fixed policy) and calculate our q-value like that. So we have,

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

here $V^\pi(s)$ = expected total discounted rewards starting in s and following π. We can use the bellman equations into updates (like value iteration),

$$V_0^\pi(s) = 0$$
$$V_{k+1}^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

Efficieny is $O(S^2)$ per iteration. Because we don't have the maxes, our equations are just a linear system that we can solve.

6.9 Policy extraction

To extract the optimal policies after finding the optimal values we just need to find the action that maximizes the q-values the state given the action, so we have,

$$\pi(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

So if we have the optimal q-values for each state it goes down to being,

$$\pi(s) = \arg\max_a Q(s, a)$$

6.10 Problems with value iteration

It repeats the bellman updates so we have the following issues,

1. Slow - $O(S^2 A)$ per iteration
2. Arg max at each state rarely changes
3. Policy often converges long before the values

6.11 Policy Iteration

Policy iteration helps solve some of these issues,

- Policy evaluation: calculates utilities for some fixed policies (not optimal utilities).
- Policy improvement: We update the policy using one-step look-ahead with resulting converges utilities as future values.
- We repeat until policy converges

It is optimal and can converge much faster in some conditions.
We have the following,

- Evaluation: For some fixed current policy π, find values with policy evaluation,

$$V_{k+1}^{\pi_i}(s) = \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')]$$

- Improvement: For fixed values, get better policy using policy extraction,

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')]$$

6.12 Comparison

Both compute the same thing (all optimal values) but,

In value iteration,

- Every iteration updates both the values and implicit the policy.
- We don't track the policy, but taking the max over actions basically recomputes it.

In policy iteration,

- We do several passes that update utilities with fixed policy (fast because we only consider 1 action)
- After evaluation, we choose new policy (slow because we need to find arg max)

6.13 Summary

We have,

1. Compute optimal values: Use value iteration of policy iteration
2. Compute values for a particular policy: Use policy evaluation
3. Turn values into policy: Use policy extraction

All these are variations of the bellman updates that use one step look ahead expect max fragments.