Artificial Intelligence Lecture Notes

Ryan Vaz

Graduate Center 2024

Contents

1 Artificial Intelligence

1.1 Introduction to Artificial Intelligence

- Artificial Intelligence is making machines that can...
  - Think like people (philosophical approach)
  - Think rationally (pass the turing test)
  - Act like people (mimic human behaviour)
  - Act rationally (the focus of our course)
- To be rational is to maximally achieve pre-defined goals. A better title for this course would be computational rationality, but that sounds less interesting.

  Remark 1. *Maximize your expected utility*

- An agent is an entity that perceives and acts, our rational agents will try to maximize a utility function.

1.2 Uninformed Search

- A *reflex agent* is one that chooses it's actions based on it's perceptions and maybe some memory, it will not consider the future consequences of it's actions instead *consider how the world IS.*

  Remark 2. *In our world, we will punish our agents over time to force it being optimal.*

- When developing planning agents, we need to consider:
  - Ask "what if"

- Make decisions based on the predicted consequences of our actions
- Must have a model of how the world evolved to respond to actions
- Must formulate a goal(test)
- Consider how the world WILL BE

- Additionally there are different approaches to planning we can take,
  - Optimal Planning (Achieve the goal in minimum cost)
  - Complete Planning (When there exists a solution it is found)

- A **search problem** consists of:
  - A state space (Set of possible configurations of the board)
  - A successor function (How the world works, possible actions/cost of action)
  - A start state and a goal state

- A solution is a sequence of actions(a plan) which transforms the state space to a goal state

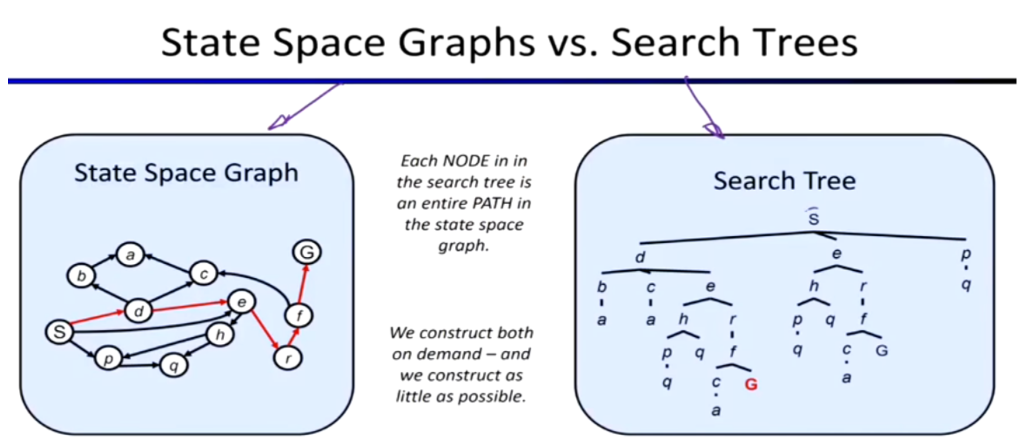
- Example Problem: Traveling in NYC
  - State space: Subway Stations
  - Successor Function: Going to the adjacent station with cost = Time
  - Start State: 34th Street Herald Square
  - Goal Test: JFK Airport

- A **world state** includes every last detail about the environment

- A **search state** will only keep the details needed for planning (abstraction)

- Example Problem: Pacman Eat-All-Dots
  - States: ((x,y), dot flags) location + boolean if dot was eaten
  - Actions: UDLR
  - Successor: Update location and one of the dot flags
  - Goal Test: Dot Flags all flipped sign

- From these search problems we can make a **State Space Graph/Trees** or a mathematical representation of a search problem where the nodes are abstracted world configurations.



```
function Tree-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the cooresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- There's a problem with this strategy, we need some way to picking the fringe candidates for expansion.
- One such approach would be a **Depth-First Search**
  - Strategy: Expand the deepest node first
  - Implementation: Fringe is a LIFO Stack
- For the properties of the DFS:
  - Let each node have factor b (each node spawns b children)
  - Let m be the maximum depth of the tree
  - Starting from the root node, we have one node on layer 1. On layer 2, the one root node will spawn b children. On layer 3, the b nodes will each have another b children therefore there will be b^2 nodes. At the end, we can state that there will be b^m nodes in the last layer.
  - Time complexity will be $O(b^m)$ if m is finite.
  - Each element on the fringe will be m deep in the tree, thus it will take up m space.
  - Space complexity will be $O(b * m)$
  - Assuming the tree cannot be infinite, then it will always find a solution.
  - Does not find the optimal solution.
- Another approach we can use is **Breadth-First Search**
  - Strategy: Expand the shallowest node first
  - Implementation: Fringe is a FIFO Queue
- For the properties of the BFS:
  - Let each node have factor b (each node spawns b children)
  - Let depth of the shallowest solution be s
  - Time complexity will be $O(b^s)$
  - BFS will expand the most amount of nodes onto the fringe, as it expands each node on every layer.
  - Space complexity will be $O(b^s)$
  - Assuming the tree cannot be infinite, then it will always find a solution.
  - Does not find the optimal solution.
- BFS outperforms DFS if the goal is shallow or on the left side of the tree
- DFS outperforms BFS memorywise

- What if we want the DFS's space advantage and the BFS's time/shallow solution advantages? Introducing **Iterative Deepening**
  - Run a DFS with depth limit 1. If there's no solution...
  - Run a DFS with depth limit 2. If there's no solution...
  - Run a DFS with depth limit 3. ...
- While this method is redundant, generally most of the work happens in the lowest level so it isn't nearly as bad.
- When cost is present, we can use **Uniform Cost Search**
  - Strategy: Expand the cheapest node first
  - Implementation: Fringe is a Priority Queue (Priority: cumulative cost)

  For the properties of UCS:
  - Let the optimal solution has the cost C^*
  - Let each individual action cost ϵ
  - We can say that the effective depth of the solution is C^*/ϵ actions to get there, Furthermore the size of the fringe will also be C^*/ϵ.
  - Time complexity will be $O(b^{C*/\epsilon})$
  - Space complexity will be $O(b^{C*/\epsilon})$
  - Assuming the tree cannot be infinite, then it will always find a solution.
  - Will find the optimal solution! Everything on the fringe will have a higher cost than it.
- All these searches explore in every direction and doesn't think about where the goal will be.

1.3 A* and Heuristics

- A **heuristic** is a function that estimates how close a state is to a goal. Some examples include the Manhattan Distance and the Euclidean Distance(L1 and L2 norms).
- Some characteristics of Heuristics
  - These are measures, we want h(x) = 0 when we are at the goal state (If y = goal, $h(x) = |x - y| = 0$)
- Greedy Search
  - Strategy: Expand the node that seems the closest to the goal state
  - Heuristic: Estimate of the distance to nearest goal for each state
  - A common case: Best-First takes you straight to the (wrong) goal
- What if we could combine the features of UCS and Greedy search?
  - Uniform-Cost orders by path cost, or backward cost g(n)
  - Greedy orders by goal proximity, or forward cost h(n)
  - A^* Search orders by the sum f(n) = g(n) + h(n)

Remark 3. *We should not stop when we enqueue a goal, but only once we pop the goal off the queue*

- A* is sometimes optimal, but this relies on us choosing a good heuristic.

  Definition 1. *A heuristic h is* **admissible** *(optimistic) if*

  $$0 \leq h(n) \leq h^*(n)$$

  where $h^*(n)$ is the true cost to a nearest goal. An example of a admissible heuristic is the L1 Norm.

- Optimality of A^* Proof

- Assume:
  - A is an optimal goal node, B is a suboptimal goal node, h is admissible
  - Claim: A will exit the fringe before

- Proof:
  - If B is never on the fringe, Q.E.D.
  - Imagine if B is on the fringe
  - We know some ancestor n of A is on the fringe, too (possibly A). This is because the only way for A's ancestors to be gone is if A was already expanded.
  - Claim: n will be expanded before B

  1. $f(n) \leq f(A)$
     - $f(n) = g(n) + h(n)$
     - $f(n) \leq g(A)$ by admissibility
     - $g(A) = f(a)$ since h=0 at goal
  2. f(A) is less than f(B)
     - $g(A) < g(B)$ since B is suboptimal
     - $f(A) < f(B)$ since h=0 at goal
  3. n expands before B
     - $f(n) \leq f(A) < f(B)$

  - Repeat for all ancestors of A until A itself is on the fringe therefore A^* search is optimal

- Uniform cost will equally fast in all directions, meanwhile A* will tunnel directly towards a goal in an environment

- Properties of the different search algorithms
  - BFS will find a path in the least amount of moves expanding outwards in each direction.
  - Greedy will find the thing closest to the goal based on the heuristic, does not care about cost.
  - UCS will explore forward quickly in regions where cost is lower, but does now orient itself towards the goal.
  - DFS will explore the whole search space.
  - A* will explore both the heuristic and in the minimal cost direction.

- As heuristics get closer to the true cost, the fewer nodes will be expanded but the more work per node to compute the heuristic will be done

- **Graph Search**

- Idea: Never expand a state twice

  Tree Search + set of expanded states (the closed set)

  Only expand a node if it has never been expanded before

  If it has been expanded before, skip the expansion and add it to the closed set
- Important: store the closed set as a set and not a list
- Graph search is still complete

- Issue: If we explore an optimal node that we already found unoptimally, than the graph search will not explore it even if it would lead to a better result.

- Main Idea: Estimated Heuristics $\leq$ actual costs

  - Admissibility: heuristic cost $\leq$ actual cost to goal
  - **Consistency**: heuristic "arc" cost $\leq$ actual cost for each arc. This statement is stronger than Admissibility.
  - Consequences: The f(A) value along the path never decreases

- Consistency implies Admissibility

- A^* uses both backwards and (estimates) forwards costs

- A^* is optimal with an admissible/consistent heuristic

- Heuristic design is key: often use relaxed problems

1.4 Constraint Satisfaction Problems

- Standard Search Problems

  - State is a black box with little knowledge of what's inside it
  - Goal test can be any function over states
  - Successor Function can also be anything

- **Constraint Satisfaction Problems** (CSPs)

  - A special subset of search problems
  - State is defined by variables X, with values from a domain D.
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.

- CSPs are a Simple example of a formal representation language

- A solution to a CSP will be one where all the assignments satisfy all the constraints

- Arcs can be used to make Constraint graphs, however they do not tell information about the constraints themselves.

  Example: Sudoku

  - Variables: Each open square
  - Domains: {1,2,3,4,5,6,7,8,9}

- Constraints: alldiff(columns), alldiff(row), alldiff(region)

- Discrete Variables
  - Finite Domains
    - Size of d means $O(d^n)$ complete assignments
  - Infinite domains (int, str, etc)
    - Linear constraints solvable, nonlinear decidable

- Continious Variables
  - Linear constraints solvable in polynomial time by LP methods

- Constraint Types
  - Unary Constraints (single variable, x != 0)
  - Binary Constraint (pair of variables, x != y)
  - High Order Constraints involve 3 or more variables

- Standard Search Formulation
  - Standard Search formulation of CSPs
  - States defined by the values assigned so far (partial assignments)
    - Initial State: the empty assignment
    - Sucessor function: assign a value to an unassigned variable
    - Goal test: the current assignment is complete and satisfies all the constraints

- Backtracking Search
  - Variable assignments are commutative, so fix ordering(x = 1 then y = 2 is the same as y = 2 then x = 1)
  - Only consider assignments to a single variable at each step
  - Check constraints as you go
  - Incremental goal test
  - Might have to do some computation to check the constraints

```
function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)
function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var = Select-Unassigned-Variable(Variables[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given Constraints[csp] then
            add {var = value} to assignment
            result = Recursive-Backtracking(assignment, csp)
            if result != failure then return result
            remove {var = value} from assignment
    return failure
```

- An arc $X \rightarrow Y$ is consistent $\iff$ $\forall$ x in the tail $\exists$ y in the head which could be assigned without violating a constraint
- On each assignment if we enforce consistency then we can make better filtering algorithms (This method is called Arc-Consistency)
- Variable Ordering: Minimum Remaining Values(MRV) - Pick the variable with the fewest legal left domain values in its domain. It's also called fail fast ordering.

1.5 Searching with Minimax

- We can classify games as (deterministic or stochastic), (player count), (zero sum), (perfect information)
- We want algorithms that calculate a strategy(policy) which recommends a move from each state.
- In this lecture we will focus on **Deterministic games**
  - States: S(start at s_0)
  - Players: P={1,...,N} (usually take turns)
  - Actions: A (may depend on player/state)
  - Transition Function: $S \times A \rightarrow S$
  - Terminal Test: $S \rightarrow \{t, f\}$
  - Terminal Utilities: $S \times P \xrightarrow{R}$
- Solution for a player is a **policy**: $S \rightarrow A$
- **Zero-Sum Games**
  - Agents have opposite utilities (values on outcomes)
  - Let us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- **General Games**
  - Agents have independant utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
- To solve Zero-Sum games, we need to think about what opportunities the opponent will make, then what
- Single Agent Trees are ones where a single agent wants to achieve the best achievable outcome(utility) from that state.
- On an Adversarial Game Tree, the player and the enemy take turns doing actions. At the end of the game, we assign a utility.
  - States under Agent's Control
    $V(s) = \max_{s' \in \text{successors}(s)}$ V(s')
  - States under Enemy Control
    $V(s') = \min_{s \in \text{successors}(s')}$ V(s)

- If we have two players playing against each other optimally, we know that the game will result in a draw in the case of Tic-Tac-Toe and Checkers.

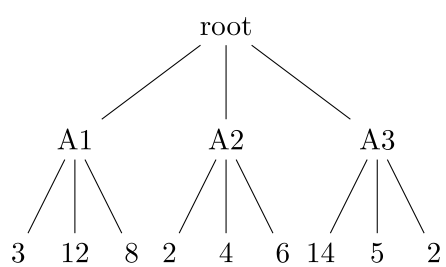
- **Minimax** is when we assume the enemy will also be playing optimally.

```
def max-value(state):
    initialize v = -inf
    for each successor of state:
        v = max(v, min-value(successor))
    return v

def min-value(state):
    initialize v = +inf
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

- This is a recursion where we jump from min to max, but we will need a base case.

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```



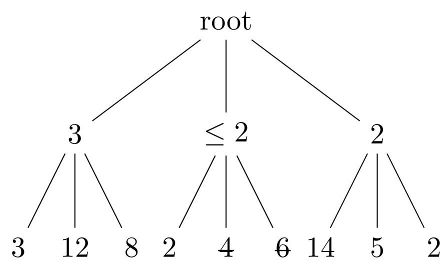
- Assuming root = level 1 = player's turn = max, level 2 = enemy turn = min, and level 3 = utility we can state
  - A1 = $\min\{3, 12, 8, \infty\} = 3$
  - A2 = $\min\{2, 4, 6, \infty\} = 2$
  - A3 = $\min\{14, 5, 2, \infty\} = 2$
  - root = $\max\{3, 2, 2, -\infty\} = 3$

- Minimax is good only if both players are playing optimally, but in cases when they aren't it might be worth taking risks to gain an even bigger rewards

- Properties of Minimax:
  - Minimax is like an exhaustive DFS
  - Time Complexity is $O(b^m)$
  - Space Complexity is $O(bm)$

- However we don't always need to look at the full tree, we can prune some of the tree to reduce the search space drastically.

- Let's walk through the Minimax Pruning of this tree



  - We solve out the A1 branch to determine that our current max is 3
  - We take a look inside A2 and find a 2, since we 2 is less than 3, we would never pick this option since it is less than our currently assumed max
  - Inside A3, we see a 14 and that's fine. We continue exploring and see a 5, that's still better than 3 so we continue exploring. Lastly we see a 2 and compute the min between the 3 values and infinity.
  - Inside this tree we skip A22 and A23 but in bigger trees there is more to save

- This technique specifically is called **Alpha-Beta Pruning**, this is a example configuration for the MIN variant

  - We compute the Min value at some node n
  - We're looping oven n's children
  - n's estimate of the childrens' min is dropping
  - Who cares about n's value? MAX
  - Let α be the best value that MAX can get at any choice point along the current path from the root.
  - If n becomes worse than α, MAX will avoid it, so we can stop considering n's other children (it's already bad enough that this won't be played)

- α = max best option currently available, β = min best option currently available

- Formally with alpha-beta pruning:

```
def max-value(state, alpha, beta):
    initialize v = -inf
    for each successor of state:
        v = max(v, min-value(successor, alpha, beta))
        if v >= beta return v
        alpha = max(alpha, v)
    return v

def min-value(state):
    initialize v = +inf
    for each successor of state:
        v = min(v, max-value(successor, alpha, beta))
        if v <= alpha return v
        beta = min(beta, v)
    return v
```

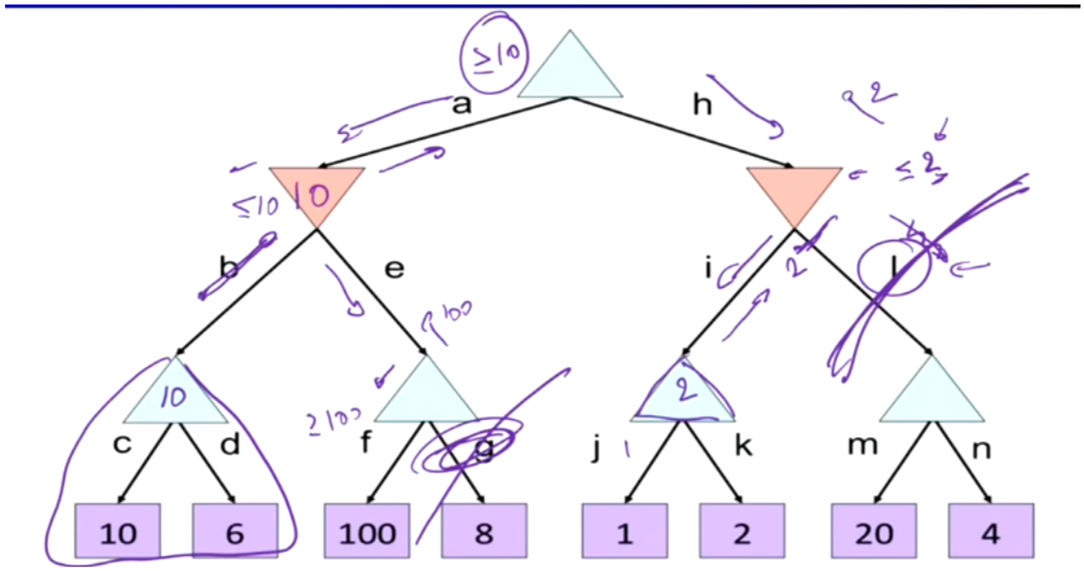


Figure 1: Example $\alpha - \beta$ Pruning

- Alpha and Beta get passed down, only v gets passed up
- Pruning has no effect on the minimax value computed for the root
- With "perfect ordering", the time complexity drops to $O(b^{n/2})$ and it doubles solvable depth!

1.6 Searching with Expectimax and Utilities

- Idea: Uncertain outcomes controlled by chance, not an adversary. Sometimes the enemy doesn't play optimally and the penalty isn't too bad.

  Remark 4. *Up Triangle = Maximizer, Down Triangle = Minimizer, Circle = Chance Nodes*

- Expectimax Search
  - Maybe we don't know everything about the game, sometimes there is explicit randomness
  - In this case we should model it with a probability distribution
- Values should now reflect the average case(expectimax) outcome, not worst-case(minimax) outcomes
- When we run expectimax search we should compute the average score under optimal play
- The max nodes work the same as usual, but the chance nodes work by taking the weighted average of the outcomes
- Expectimax Psuedocode

```
def max-value(state):
    initialize v = -inf
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

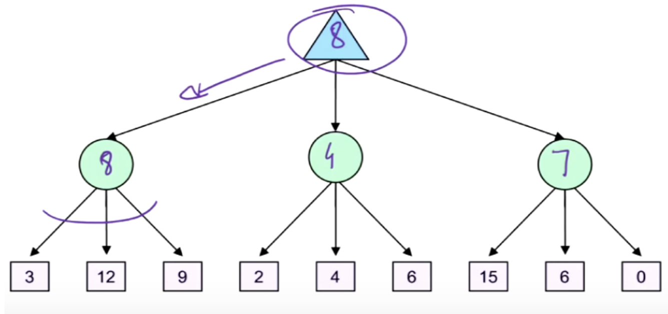


Figure 2: Expectimax

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(sucessor)
        v += p * value(sucessor)
    return v

def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes
  - Dangerous Optimism - Assume chance when the world is adversarial
  - Dangerous Pessimism - Assume the worst case when it's not likely
- In Expectiminimax the environment is an extra "random agent" player that moves after each min/-max nodes

  Definition 2. *Principle of maximum expected utility: A rational agent should choose the action that maximizes its expected utility, given it's knowledge*

- An agent must have preferences among:
  - Prizes A, B
  - Lotteries: situations with uncertain prizes
    - L = $[p, A; (1-p), B]$
- Notation
  - Preference: $A \succ B$
  - Indifference: $A \sim B$
- Rational Preferences:
  - Axiom of Transitivity: $(A \succ B) \wedge (B \succ C) \implies (A \succ C)$

- Orderability: $(A \succ B) \vee (B \succ A) \implies (A \sim C)$
- Continuity: $A \succ B \succ C \implies \exists p[p, A; 1-p, C] \sim B$
- Substitutability: $A \sim C \implies [p, A; 1-p, C] \sim [p, B; 1-p, C]$
- Monotonicity: $A \succ B \implies (p \geq q \iff [p, A; 1-p, B] \succeq [q, A; 1-p, B]$

- MEU(Maximum Expected Utility) Principle: Given any preferences satisfying these constraints, there exists a real-valued function U such that:
  - Choose the action that maximizes expected utility
  - Note: an agent can be entirely rational without very representing or manipulating utilities and probabilities
  - Example: A lookup table for perfect tic-tac-toe, a reflex vacuum cleaner
  - $U(A) \geq U(B) \iff A \succeq B$
  - $U([p_1, S_1; ...; p_n, S_n]) = \sum_i p_i U(S_i)$
- i.e. values assigned by U preserve preference of both prizes and lotteries!

1.7 Markov Decision Processes

- Grid World
  - A Maze-Like Problem where the agent lives in a grid and wall block the agent's path
  - Noisy Movement: Actions do not always go as planned
  - 80% of the time, the action North takes the agent north if there is no wall
  - 10% of the time, North goes West, 10% it will go east
  - If there is a wall in the direction the agent would have moved, it will stay still
  - Small living reward for each step
  - Big reward comes at the end
  - Goal: maximize sum of rewards
- A MDP is defined by
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s, a, s')$
    - Probability that a from s leads to s' i.e. P(s' — s,a)
    - Also called the model or the dynamics
  - A reward function R(s,a,s')
  - A start state
  - Maybe a terminal state
- In a deterministic single-agent search problem, we wanted an optimal plan, or sequence of actions from start to goal
- MDPs are non-deterministic search problems

- Markov generally means that given the present state, the future and past are independant
- For Markov decision processes, "Markov" means action outcomes depend only on the current state
- For MDPs, we want an optimal policy $\pi^* : S \to A$
- An optimal policy is one that maximizes utility when followed
- It's also reasonable to maximize the sum of the rewards
- However we can also prefer rewards now to rewards later
- One solution is to make the values of rewards decay exponentially
- Suppose we make the discounted reward less than one, then we can say reward at step 1 = 1, reward at step 2 = γ, reward at step 3 = γ^2
- Example: $\gamma = .5$, then reward for each future stage will decrease in half
- Example Problems
  - Actions: East, West, Exit
  - Q: For $\gamma = 1$, what is the optimal policy? $10, \leftarrow, \leftarrow, \leftarrow, 1$
  - This is because $(10, 1), (10, 1), (10, 1), (10, 1), (10, 1)$
  - Q: For $\gamma = 0.1$, what is the optimal policy? $10, \leftarrow, \leftarrow, \rightarrow, 1$
  - This is because $(10, 0.0001), (1, 0.001), (0.1, 0.01), (0.01, 0.1), (0.001, 1)$
- What if we have infinite utilities? We can use finite horizon which terminates episodes after a fixed T steps. This gives non stationary policies (π depends on time left) aka just declare an end.
- The value of a state s is: $V^*(s) = \max_a Q^*(s, a)$, expected utility starting in s and acting optimally
- The value(utility) of a q-state (s,a): $Q^*(s, a) = \sum_{s'} T(s, a, R(s, a, s') + \gamma V^*(s')$ expected utility starting out having taken action a from state s and acting optimally. The reward of the action leading to s' plus it's discounted score
- The optimal policy: $\pi^*(s)$ = optimal action from state s
- Propositional Logic is logic based on (and, or, not, iff, implies)
- Entailment: ($a| = b$ is read as a entails b, or b follows from a) and it means every world where a is true, b is also true
- First order logic also includes (forall and exists)

1.8 Notes for Self

- When a heuristic cannot be defined correctly, and we need to fully explore a search space with all solutions on the lowest level, it stands to reason that A* will expand the same amount of nodes as DFS
- In the case of Pacman being in the same world as a single Box, both being at some coordinates (M,N), we can state that the state space would be $(MN)^2$ since pacman can be in MxN spots and the box can be in MxN spots
- The rules for a heuristics are the same as the rules for a measure
- In forward checking we only prune the domain that a assignment directly affects.