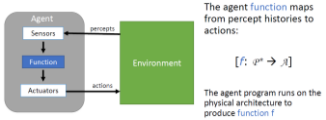
CS2109S AY23/24 MidTerms CheatSheet

Chapter 0: Intro to AI

- AI is the creation of computer systems that can perform tasks and make decisions similar to/ greater than human intelligence.
- Intelligent Agents:
  - Maps input to output via agent function.
  - The agent program runs on physical architecture to produce function.



- The performance of the Intelligent Agents are based on PEAS:

P	Performance Measure	A rational agent will choose actions that MAX P • Best for whom?, What are we optimizing? • What information is available? • Any unintended effects?, What are the costs? • Defines "goodness" of a solution
E	Environment	Define what the agent can or cannot do
A	Actuators	Outputs of a system (E.g Signals, Screen etc)
S	Sensors	Inputs to the system (E.g Camera, Mic etc)

Properties of Task Environment

Fully Observable (vs Partially Observable)	An agent's sensors give it access to the complete state of the environment at each point in time.
Deterministic (vs Stochastic/ Strategic)	• The next state of the environment is completely determined by the current state and the action executed by the agent. • If the environment is deterministic except for the actions of other agents, then the environment is strategic
Episodic (vs Sequential)	The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
Static (vs Dynamic)	The environment is unchanged while an agent is deliberating. (The environment is semi-dynamic if the environment itself does not change with the passage of time, but the agent's performance score does)
Discrete (vs Continuous)	A limited number of distinct, clearly defined percepts and actions.
Single Agent (vs Multi-Agent)	An agent operating by itself in an environment.

** An agent can exhibit multiple task environment properties.
E.g Chess: Full, Sequential, Discrete, Strategic. Car → Continuous, Partial…

Structure of Agents

Agent is completely specified by the agent function mapping percept sequences to actions.

Simple Reflex Agent Selects action based on current perception, ignoring history. Uses condition-action (if-then) rules	Model-Based Agent Keep track of the part of the map it cant see. This internal state is then updated through the transition model of the map (i.e knowledge on how the map changes) and the sensor model (i.e info from perceptions.)
Goal-Based Agent On top of tracking the state of the map, it also track a set of goals, then pick the action that brings it closer to goal	Utility-Based Agent Uses a utility function to score any given percept sequence (i.e internal performance measure). If the utility function and performance measure are aligned → agent is rational
Learning Agent Uses a performance element to select external actions, a *critic* to give feedback on how the agent is doing, and how the performance element can be improved. A *learner* to improve, a *problemGEN* to give suggestions.	

Exploitation vs Exploration

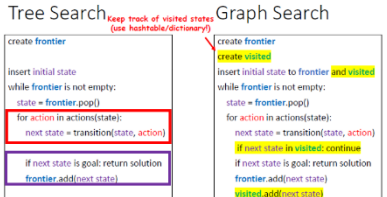
- An agent operating in the real world must often choose between:
  - **Exploiting**: MAX its expected util according to its curr knowledge about the world
  - **Exploring**: Trying to learn more about the world

Chapter 1: Searching

Problem Formulation

- **States (state space)**
- **Initial State**: The initial state of the agent
- **Goal State(s)/ Test**: Final State
- **Actions**: Things the agent can do
- **Transition Model**: What each action does
- **Action Cost Function**: The cost of performing an action (e.g Σ time taken)

A Sequence of action form a **Path/Trajectory**
The **Solution** is a path to a goal (state)



Uninformed Search (Search Blindly, no info on state)

- For uninformed search, no clue is given about how close a state is to goal

BFS

- For Breadth First Search, we use a QUEUE.
- Explores layer by layer

Time	$O(b^d)$
Space	$O(b^d)$, worst case: expand till last child, leaf
Complete?	Yes, if #branches is finite
Optimal?	Yes, if step cost is same everywhere

** BFS is a special case of UCS (step cost = 1)
** b → branching factor, d → depth of OPTIMAL SOLUTION

DFS

- For Depth First Search, we use a STACK
- Go deep first, then backtrack

Time	$O(b^m)$
Space	O(bm)
Complete?	NO. When depth is infinite. Or DFS can get stuck in loops, go back n forth
Optimal?	NO

**b → branching factor, m → MAX depth (from root to leaf)

UCS

- For Uniform Cost Search, we use PRIORITY QUEUE

Time	$O(b^{C^*/\epsilon})$
Space	$O(b^{C^*/\epsilon})$
Complete?	Yes, if $\epsilon > 0$ and C^* is finite
Optimal?	Yes, if $\epsilon > 0$, $\epsilon = 0$ may cause 0 cost cycles

** Swap the for action(red), and if state (purple).
** C^* → Cost of optimal solution, ϵ → MIN edge cost
** b → Branching factor, C^*/ϵ → "Tiers"
Note:

- Each tier is the same cost.
- If the edge has a 0 edge cost → zero-cost cycles → ∞ loop
- UCS applies goal-test when POPPING a state from frontier

Variants of Uninformed Search

DLS

- For Depth-Limited Search, similar to DFS
- Limit the search depth + backtrack once depth is reached

Time	$O(b^l)$ $[1 + b + b^2 + \cdots + b^l = O(b^l)]$
Space	O(bl)
Complete?	NO. When depth is infinite. Or DFS can get stuck in loops, go back n forth
Optimal?	NO

**b → branching factor, l → depth limit

IDS

- For Iterative Deepening Search, similar to DLS
- Do DLS with max-depth increasing iteratively 0… to … N
  - Return solution if found, increase depth otherwise
  - Has a lot of overhead ☹ → $(n\_ids - n\_dls)/n\_dls$

Time	$b^0 + (b^0 + b^1) + \ldots + (b^0 + \ldots + b^d)$ $= (d+1)b^0 + db^1 + (d-1)b^2 + \ldots + 2b^{d-1} + b^d$ $= O(b^d)$ **yellow → overhead
Space	O(bd)
Complete?	Yes
Optimal?	Yes, if step cost is the same everywhere

BDS

- For Bidirectional Search, we combine forward & backward search, stop when the 2 searches meet. (Meet "middle")

$$2 \times O(b^{d/2}) < O(b^d)$$

Speeds up by AT MOST 2 times ONLY. Uses more space

Schema:

```
create frontier_forward: queue
create frontier_backward: queue
explored = set()
insert initial state for front and back
while frontier_forward&frontier_backward is not empty:
    state = frontier.pop()
    if state is goal:
        return solution
    if state not in explored and depth(state) < depth_limit:
      explored.add(state)
    for action in actions(state):
      next state = transition(state, action)
          frontier.add(next state)
    ** do 2 times, 1 for backward, 1 forward
return failure
```

** Use graph search to keep track of visited nodes → can speed up program

Takes more **time**.

Graph Search #1

create frontier
create visited
• Expand less states
• May skip states with less cost
insert initial state to frontier and visited
while frontier is not empty:
state = frontier.pop()
for action in actions(state):
next state = transition(state, action)
if next state in visited: continue
if next state is goal: return solution
frontier.add(next state)
visited.add(next state)
return failure

Takes more **space**.

Graph Search #2

create frontier
create visited
• Expand more states
• Will not skip states with less cost
insert initial state to frontier
while frontier is not empty:
state = frontier.pop()
if state is goal: return solution
if state in visited: continue
visited.add(state)
for action in actions(state):
next state = transition(state, action)
frontier.add(next state)
return failure

Informed Search

- Use Domain information to guide the search. Uses heuristics (something like a method of solving problem)
  - E.g To predict temperature: 1. Past data + noise, or 2. RNG
  - Different Heuristics give different results, and we want a good heuristic that best mimic/best predict our desired dataset.

Admissible Heuristics

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

- A heuristic h(n) is admissible if for every node n, h(n) ⩽ h*(n), where h*(n) is the true cost to reach the goal state from n.
- Theorem: if h(n) is admissible, A* using tree search is optimal
- It **never overestimates the cost** to reach the goal from any given state.
  - The heuristic value is **always less than or equal to the actual cost**. (true cost of the relaxed problem)

Consistent Heuristics

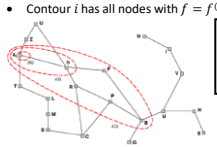
$$h(n) \leq c(n,a,n') + h(n')$$

- A heuristic h(n) is consistent if for every node n, every successor n' of n generated by any action a,
  $h(n) \leq c(n,a,n') + h(n')$, and $h(G) = 0$.
- Consistent Heuristic **ensures that the heuristic is reliable** and doesn't lead to unnecessary exploration of suboptimal paths.
  - **Satisfies Triangle Equality**
- A heuristic is considered consistent if the estimated cost from a current state to a successor state + the heuristic value of the successor state, is always greater than or equal to the heuristic value of the current state.

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) = f(n) \end{aligned}$$

Optimality

- **A* expands nodes in order of increasing f-cost**
- Gradually adds "f-contours" of nodes
- Contour i has all nodes with $f = f^{(i)}$, where $f^{(i)} < f^{(i+1)}$



- BFS like
- Triangle Equality → Shows contour

Dominance Heuristics

if $h_1(n) \leq h_2(n), h_2$ dominant

- Dominance heuristics involve comparing heuristics for multiple agents or entities to determine which one dominates the others.
- A heuristic dominates another if it **provides better or equal information for every state**.
- Dominance heuristics are often used in multi-agent systems or scenarios where different agents may have different heuristics.
  - i.e fight it out to see which heuristic is better. Then use that one. **Survival of the strongest** (Let the heuristics compete)

- We can also combine heuristics. Why would we do that?
  - Sometimes, there may be a collection of admissible heuristics $h_1(n)$, $h_2(n)$, …, $h_m(n)$, but **NONE of them dominate** each other.
  - As such, we can **combine them together**, and **nitpick** the best of the best via:

$$h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$$
$$OR$$
$$h(n) = \min\{h_1(n), h_2(n), \ldots, h_m(n)\}$$

- These set of heuristics all provide valuable information, but since none of them dominate the others, **creating a composite or combining these heuristics can lead to a creation of a better heuristics** that benefit from the strengths of each individual heuristic.
  - E.g A fast searching + A pruning heuristic == Super-efficient heuristic
- Generally, a better heuristic → faster, more accurate, explores less nodes.

- A problem with **fewer restrictions** on the actions is called a **relaxed problem**.
- The cost of an **optimal solution** to a relaxed problem is an **admissible heuristic** for the original problem.
  - E.g find number of misplaced tiles, vs find Manhattan distance as heuristic

Informed Search Algorithms:

Schema:

```
create frontier : priority queue f(n)
insert initial state
while frontier is not empty:
  state = frontier.pop()
  if state is goal: return solution
  for action in actions(state):
    next state = transition(state, action)
    frontier.add(next state)
return failure
```

- GBFS, and A* are special cases of BFS.
- They use a different state check approach via heuristics.

BFS

- Best-First Search.
- We evaluate the "goodness" of a function f(n) on the state.
- Return BEST State **GBFS, and A* are special cases of BFS

GBFS

Instead of evaluating f(n), evaluate f(n) = h(n), where h(n) is the estimated cost from n to goal
** Doesn't consider the cost so far (Unlike A*)

A*

Instead of evaluating f(n), evaluate **f(n) = g(n) + h(n)**, where g(n) is the cost so far. h(n) is the estimated cost from n to goal
** A* sorts PQ by heuristic + cost, pop until goal state to ensure optimality

GBFS & A*:
Time: Good heuristic gives improvement
Space: Keep all nodes in memory → Can prune to save memory!

All search strategies. C*=cost of best path.

Algorithm	Complete?	Optimal?	Time complexity	Space complexity	Implement the Frontier as a…
BFS	Yes	If all step costs are equal	O(b^d)	O(b^d)	Queue
DFS	No	No	O(b^m)	O(bm)	Stack
UCS	Yes	Yes	Number of nodes w/ g(n) ≤ C*	Number of nodes w/ g(n) ≤ C*	Priority Queue sorted by g(n)
Greedy	No	No	Worst case: O(b^m) Best case: O(bd)	Worse case: O(b^m) Best case: O(bd)	Priority Queue sorted by h(n)
A*	Yes	Yes	Number of nodes w/ g(n)+h(n) ≤ C*	Number of nodes w/ g(n)+h(n) ≤ C*	Priority Queue sorted by h(n)+g(n)

Chapter 3: More Searches (Local + Adversarial Search)

- Use local search when path is NOT impt, and the state is the solution.
  - Since the path to a goal might not be the solution we want.
  - E.g Show me best outcome for X, we just want the best state of X, not how to get from initial to best X (i.e path).
- Local search **DOES NOT maintain a complete search tree**. It focuses on the current solution and its immediate neighbors.
  - Goal: Find an optimal/ satisfactory solution without exhaustively exploring the entire solution space.
- Set up the 4 objectives:
  - State (State Space) → How to represent your problem, array?
  - Initial State → What is the current state
  - Goal State → Usually unknown, we want to find this
  - Successor → Successive states >= Current state
    - E.g nxt state shouldn't have less points than current state
- We can use either heuristic function, or objective functions to aid us

H

- The **heuristic function** provides an estimate of the desirability of a state based on some domain-specific knowledge.
- **Heuristic functions** are often used when the **problem is complex** and an **exact solution is hard to find**, providing a shortcut to guide the search.

O

- We can opt to optimize the algorithm/ find a solution for the problem mathematically. E.g $f(x) = -x^2$
- In this case, an optimal solution is maximize/minimize J(x) based on the context of the question
- **Objective functions** are more **precise** and may involve **mathematical optimization** techniques to find an optimal solution.

Local Search Algorithms

HC

- Hill Climbing → slowly compare with neighbors, choose best
- Might get stuck in local max/min (cant find global)
  - This issue can be resolved via simulated annealing, or random jumps

```
current= initial state
loop:
  neighbor= a highest value successor of current
  if value(neighbor) <= value(current):
    return current
  current= neighbor
```

Pick the BEST among the neighbours, repeat

BS

- Beam Search is similar to HC, but we perform k-HC in parallel
  - **Local Beam Search** → Choose k successors deterministically.
  - **Stochastic Beam Search** →Choose k successors probabilistically.
  - k → #states kept at each level of the search

SA

- In Simulated Annealing, its similar to HC, just that it gives the algorithm a chance to escape local min/max.
  - Via randomly selecting the successor (instead of choosing the BEST) → No tunnel vision 😊
- If T decreases slowly enough, simulated annealing WILL find a GLOBAL OPTIMUM with HIGH PROBABILITY

```
current = initial state
for t = 1 … ∞:
  T = schedule(t)
  if T = 0:
    return current
  next = a randomly selected successor of current
  if value(next) > value(current) or Prob(next, current, T):
current = next
```

Hill Climbing, but allow bad moves from time to time

GA

- In Genetic Algorithm, its basically Darwin's Evolution.
- Each node crossbreed (combination of traits), then get mutated abit.
- Do this many times, compete, survival of the fittest.

Marry the best, mutate and repeat

Random sampling: random sample a state until solution is found
Random walk: go to random neighbours until solution is found

Adversarial Search

- Find the BEST strategy for a player in a competitive realm.

MM

- Min-Max involves minimizing the possible loss for a worst-case scenario while maximizing potential gain.
- The algorithm alternates between two players: the maximizing player (Max) and the minimizing player (Min).
  - Assuming both parties plays optimally, we are trying to minimize the opponent's value, while maximizing our own

```
def minimax(state):
  v = max_value(state)
  return action in successors(state) with value v
def max_value(state):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
```

```
    v = max(v, min_value(next_state))
  return v
def min_value(state):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
  return v
```

Time	O(b^m)
Space	O(bm)
Complete?	Yes, if tree is finite
Optimal?	Yes, **against optimal opponent**. Else NOT optimal

- Similar to DFS, go deep till the goal state, then back tracks.
- Note, this is suitable for small games with few states. For scary games like chess, there may be a lot of states. E.g b = 35, m = 100 → 35^{100} possibilities → slow

Note:
- We can limit the Min-Max to a cut-off, by changing "is_terminal" to "is_cutoff", and returning "eval(state)" instead of "utility(state)"
- This enables us to handle large/infinite game trees 😊

Aß

- Alpha-Beta **pruning** is a technique to optimize the Minimax algorithm by **pruning branches of the game tree that won't affect the final decision.**
- This **reduces the number of nodes evaluated** and speeds up the search process.
  - Evaluating a node is sometimes not useful, as it DOES NOT change the final decision outome/ branch path chosen.

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v
def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v
def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state, α, β))
    if v <= α: return v
    β = min(β, v)
  return v
```

- Pruning doesn't affect final result
- Good move ordering improves effectiveness of pruning

Perfect Ordering": $O\left(b^{\frac{m}{2}}\right)$

An e.g of meta-reasoning: reasoning on which computations to do 1st

** The min for the max player is the max for the min payer vice versa.

Optimizing the Search

- Optimize the search process to speed up time in generating sol^n & attain better results.

Transposition Table	• Store previously computed results of positions in the game. - Avoid redundant computations during the search. • Game-playing algorithms often encounter the same positions multiple times • The transposition table allows the algorithm to retrieve this information instead of recalculating it. → faster
Precomputation of Best Moves in Openings & Closing	• In many games, certain sequences of moves in the opening and closing phases are well-known and can be precomputed. - By **precomputing the best moves** for these known positions, the algorithm can save time during actual gameplay by relying on this precomputed knowledge. • Similar for end/closing

Chapter 4: Intro to ML & Decision Trees

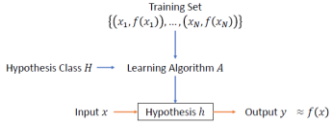
A computer program is said to learn from experience, E, wrt some class of tasks, T, and performance measure, P, if its performance at tasks in T, as measured by P, improves with experience E.

Types of Feedback in ML

Supervised Learning	Agent observes input-output pairs and learns a function that maps from input to output. Feedback given to improve
Unsupervised Learning	Agent learns patterns in the input without any explicit feedback. Most common task is clustering. No feedback, self study
Reinforcement Learning	Agent learns from a series of reinforcements: rewards and punishments. (Trial and Error)

Supervised Learning:

- **Regression** →To predict continuous output; **Classification** →To predict discrete output.
- In supervised learning, we assume that y is generated by a **true mapping function** $f: x \to y$
- We want to find a **hypothesis** $f: x \to \hat{y}$ (from a **hypothesis class** H) s.t $h \approx f$ given a **training set** $\{(x_1, f(x_1)), \ldots, (x_N, f(x_N))\}$



- Use MAE if we want to ignore outliers. (i.e outliers don't cause too big an error)
- Use MSE if we want to PUNISH outliers. (the square error will exaggerate the error)

Performance Measure

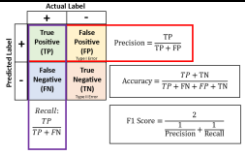
Regression

- If the output of the hypothesis is a continuous value → measure its **error**
- For an input x with a true output y, we can compute:

Squared Error $= (\hat{y} - y)^2$	*Absolute Error* $= \lvert\hat{y} - y\rvert$
Mean Squared Error (MSE)	**Mean Absolute Error (MAE)**
For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, we can compute the (mean) squared error: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$	For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, we can compute the (mean) absolute error: $MAE = \frac{1}{N}\sum_{i=1}^{N}\lvert\hat{y}_i - y_i\rvert$
** where $\hat{y}_i = h(x_i), y_i = f(x_i)$	** where $\hat{y}_i = h(x_i), y_i = f(x_i)$

Classification

Classification is correct when the prediction $\hat{y} = y$ (true label)

Correctness & Accuracy	**Confusion Matrix**
For a set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, we can compute the average correctness (accuracy): $Accuracy = \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}_{\hat{y}_i = y_i}$ This is equivalent to: $Accuracy = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}1 & , \hat{y}_i = y_i \\ 0 & , \hat{y}_i \neq y_i\end{cases}$ ** where $\hat{y}_i = h(x_i), y_i = f(x_i)$	 • Maximise **PRECISION** if **false positives (FP)** are very costly (E.g email spam) • Mazimise **RECALL** if **false negatives (FN)** are very costly (cancer prediction)

Decision Trees (DT)

Decision trees are a powerful tool in ML for both **classification** and **regression** tasks.

Pros	✓ **Expressiveness** - Decision trees are capable of representing complex decisions. - They can capture non-linear relationships between features and target variables, without any explicit assumption about the data. ✓ **Interpretability** → Easy to interpret, tree structure is intuitive. ✓ **Feature Importance** - Provides insight on the most important feature/variable. - Can narrow down the feature most relevant for making predictions
Cons	× **Overfitting** - Overfitting esp when tree grows deep/ dataset is noisy - DT's performance is perfect on training data, but worse on test data - Can use pruning/limit depth/random forest to mitiage these issues ➢ Prunning can be done via **Min sample** or **Max-depth** Rule of thumb Pruning: • If there is a split with a LARGE majority, we can merge it with the minorities. • These minorities can be due to noise/ this 1 dataset is unlikely to affect the decision outcome - Use Occam's Razor: Preference for short/simple hypothesis ➢ More complex → Coincidence / 1 trick pony, we don't want that × **Instability** - Small changes in training data → significant change in tree structure - Decision trees are sensitive to noise → may reduce general performance in the decisions made → poorer prediction. × **Bias towards Features with Many Levels** - Decision trees tend to favor features with many levels → overfitting

Size of Hypothesis Class

- Size of hypothesis class → #possible DT that can be generated from a given data set.
- DT can potentially have ∞ hypothesis class, as there are countless ways to partition the feature space and create decision trees of varying depths and complexities.
- Size of hypothesis class grows exponentially with the number of features and the max depth of the tree. Given n Boolean Attributes, the size of the hypothesis class = #Boolean functions = #distinct Truth Tables with 2^n rows = 2^{2^n}
  - With 6 Boolean Attributes, there will be: 18,446,744,073,709,551,616 trees

Decision Tree Learning (DTL) → Greedy, Top-Down, recursive algorithm

```
def DTL(examples, attributes, default):
  if examples is empty: return default
  if examples have the same classification:
    return classification
  if attributes is empty:
    return mode(examples)
  best = choose_attribute(attributes, examples)
  tree = a new decision tree with root best
  for each value vi of best:
    examples_i = {rows in examples with best = vi}
    subtree = DTL(examples_i, attributes - best, mode(examples))
    add a branch to tree with label vi and subtree subtree
```

Note: We will need to define the examples to the DTL

- In a DTL, we will need to choose the attributes that we want.
- Ideally, this choosing of attribute can be split into binary (i.e all positive, or all negative). However, this is not very likely in the real world, as most datasets come in a spectrum.
  - We can classify these dataset to simplify the issue (e.g A grade: > 70, B: 60 – 70 etc)
  - This can be done via splitting, balancing, normalizing, partitioning, etc ...
- Afterwards, these decisions are made via 2 mechanisms: Entropy and Information Gain
  - The lower the entropy, the easier it is to make decisions
  - The higher the information gained, the better the decision

Choosing the Best Attribute	• Select the best attribute to split on. • Via some heuristic, such as Information Gain(IG) or Entropy

Creating Subtrees	For each possible value of the selected attribute: - Partition the examples into subsets that have the same value for the selected attribute. - Recursively build a subtree using the remaining attributes and the subset of examples corresponding to that attribute value. - Add the subtree as a branch to the decision tree with the label corresponding to the attribute value.

Entropy

- Entropy is a measure of randomness: (n factors)

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

- For a dataset containing p positive and n negative examples: (only 2 factors)

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\left(\frac{p}{p+n}\log_2\frac{p}{p+n}\right) - \left(\frac{n}{p+n}\log_2\frac{n}{p+n}\right)$$

- The higher the entropy, the higher the randomness
  - The higher the randomness, the harder it is to make concrete decisions by the algorithm.
  - Lower entropy → Easier to make decision
  - There is a pattern identified → likely to be a good enough decision

Information Gain (IG)

- A chosen attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A has v distinct values.

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

- Information Gain (or Reduction in Entropy):

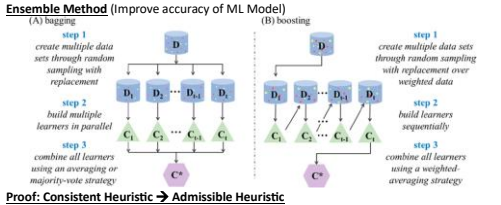
$$IG(A) = \underbrace{I\left(\frac{p}{p+n}, \frac{n}{p+n}\right)}_{\text{Entropy of this node}} - \underbrace{remainder(A)}_{\text{Entropy of child node}}$$

Different Types of Attributes

Many Values	• IG select attribute with many values as it splits the data "well". • In the extreme case, each branch will have a single example, so "all positive" or "all negative". • Hence, to remedy this issue, we balance the Information Gain (IG) with the number of branches: $Split\ Information(A) = -\sum_{i=1}^{d}\frac{\lvert E_i\rvert}{\lvert E\rvert}\log_2\frac{\lvert E_i\rvert}{\lvert E\rvert}$ $Gain\ Ratio(A) = \frac{IG}{SI}$ ** E → Examples, SI → Split Information Split Information → Sum of Entropy
Differing Costs	• **Certain attributes may have costs**. (Lose money or Lose Life) • Make decision trees to **use low-cost attributes** where possible using **Cost-Normalized-Gain**: $\frac{IG^2(A)}{Cost(A)}$ OR $\frac{2^{IG(A)} - 1}{(Cost(A)+1)^w}, w \in [0,1]$ ** w → Determine the importance of cost (weight)
Continuous Valued Attribute	Define a discrete-valued input attribute to **partition the values into a discrete set of intervals.** E.g: Grades: A→ 70+, B: 60-70, ...
Missing Values	Many ways to clean the data: • Assign the most common value of the attribute • Assign the most common value of the attribute w same output • Assign probability to each possible value and sample • Drop the attribute, Drop the rows, Etc ...

- Models could represent the decision too perfectly, which may result in ineffectiveness, since noise are also taken into account. (Keep things simple! Occam's Razor)

Ensemble Method (Improve accuracy of ML Model)



Proof: Consistent Heuristic → Admissible Heuristic

Let v_n be an arbitrary node and $v_n, v_{n-1}, \ldots, v_1, G$ be a least cost path to the goal.

Applying consistency definition along the path gives:

$h(v_n) \leq c(v_n, a, v_{n-1}) + h(v_{n-1})$
$h(v_{n-1}) \leq c(v_{n-1}, a, v_{n-2}) + h(v_{n-2})$
$\ldots$
$h(v_1) \leq c(v_1, a, G) + h(G)$

CONSISTENCY: $h(v) \leq c(v, a, u) + h(u)$
ADMISSIBILITY: $h(v) \leq h^*(v)$

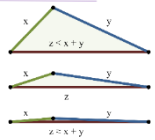
For graphs, we need the heuristic to be both admissible & consistent

Thus,

$h(v_n) \leq c(v_n, a, v_{n-1}) + h(v_{n-1})$
$\leq c(v_n, a, v_{n-1}) + c(v_{n-1}, a, v_{n-2}) + h(v_{n-2})$ Total Cost to Goal
$\ldots$
$\leq c(v_n, a, v_{n-1}) + c(v_{n-1}, a, v_{n-2}) + \ldots + c(v_1, a, G)) = h^*(v_n)$
$h(v_n) \leq h^*(v_n)$ **Admissible!**

Triangle Inequality

$\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$



Derivation:

$z + x \geq y$
$z + y \geq x$
$z \geq x - y$
$x \geq y - x$
$\}\ Z \geq |X - Y|$

Hence:

$Z \geq |X - Y|$

h(n) = (g, h, f)

- g → cost to get here so far; h → heuristic; f → Σ h + y

Alpha-Beta Pruning:

- α → worse value so far for MAX PLAYER, ß →best value so far MAX PLAYER
- ONLY α value is changed in MAX turn (i.e lower bound for MAX player)
  - α will be updated to max{curr_α, child_α, child_ß}
- ONLY ß value is changed in MIN turn (i.e upper bound for MIN player)
  - ß will be update to min{curr_ß, child_α, child_ß}
- Prune Condition: IF α ≥ ß → Can prune, as current branch WONT affect final results.
  - Note: If α ≥ ß, but no more child → nothing to prune. Continue process
- If we sort the leaf nodes → Possible improvement, as we prune more leaf nodes → faster algorithm... However, in practice, we can't infer anything solely based on the leaf nodes. → We won't know who the min/max is based on leaf nodes.
- Yes, we get max performance (=pruning) when the first leaf you get to has the best score. But if we know which leaf had the best score → no need to do alpha-beta.
- In practice, we want to order the immediate children of a node during the search in decreasing estimated score order, where your estimates are the best guess you can come up with without actually performing a search.

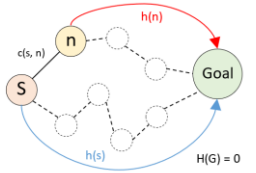
Local Search vs Informed Search

- **Informed Search**: Informed search algorithms, such as A*, use heuristic information to guide the search process towards the goal state efficiently.
  - Explore the most promising paths first, based on estimated costs or distances to the goal. Ideal when we want to find the optimal solution or path to a problem.
- **Local Search**: Local search algorithms, such as hill climbing or simulated annealing, explore the search space by making incremental changes to the current solution, aiming to improve it iteratively.
  - Don't guarantee finding the optimal solution but focus on finding a satisfactory solution within a reasonable amount of time. Suitable for problems with large search spaces where exhaustively exploring all possibilities is not feasible.
- Ideally, to have the best solution, we should use informed search. However, if the search space is HUGE O(2^{N^*N}), local search may be more appropriate.
- If we just want to obtain the solved state, and not the most optimal path, we can opt for local search as well. → Save time and space.

Linear Regression Normal Equation: $\theta = (X^T X)^{-1} X^T Y$

Heuristics

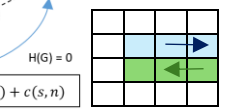
- Consistent → admissible. BUT it is not necessarily true that admissible → consistent
  - Each step does not decrease heuristic by more than 1/ at most 1→ Consistent.
  - As long as the it is not possible for the heuristic to increase or decrease by more than 1 in each step, then triangle equality will not be violated.
- If a heuristic uses max{}, high chance it is NOT admissible, as it either overestimates the actual cost → it ignores other potential path.
- If a heuristic involves sharing of a state (e.g 3 balls can fill 1 slot) → probably not consistent.
- If a heuristic involves comparing to FURTHEST goal state → probably not admissible/consistent. (If closest goal state → both admissible + consistent)
- A heuristic that is admissible for a relaxed problem must also be admissible for a more constrained problem.
- For one heuristic to dominate another, all of its values must be greater than or equal to the corresponding values of the other heuristic.
- h_A dominates h_B as it provides a closer estimate to true cost, with X factor.



Consistent: $h(s) \leq h(n) + c(s, n)$

How to test admissibility?

- Check ALL cardinal directions
  - If overestimate → NOT Admissible
- If it's a 2D-Goal state
  - Check direction, in out. + rotation



E.g : #moves to goal state = rotate + move down = 2 moves.

If heuristic counts that we need to loop 1 round s.t move > 2. NOT Admissible

- If Manhattan Distance used, USUALLY consistent (check context)

Local Search

- Initial Representation(candidate) → Usually randomized/ empty state
- Transition Function → Usually involves shifting/swap pieces purposefully
- Heuristic Function → Used to gauge the performance of the algorithm. Should converge to the goal state eventually.
  - Goal Test: Usually, check if H(n) = 0
- Note: Sorting might affect the optimality of solution.
  - E.g [1,1,1,2], find sum = 2 → [1,1] is correct, but [2] is a more optimal solution.
  - If we sort: [2,1,1,1], we get the answer faster, less branches traversed.
- If the search tree has no bounded depth → INFINITE Search tree (can ping pong states)
- If there are many methods to get towards a goal state → multiple goal states+repeated
- Local search is better for solving constraint satisfaction problem, as it is faster.
- Informed search may be preferred in problems for which local search often ends up in a local minimum or if it is not easy to find a good heuristic function for local search. Informed search is also preferred if the path cost matters.
- Local search is preferred in problems for which path cost does not matter and local search is often faster.