Artificial Intelligence

* Rational Decision
  - rational：最大化的你要達成的目標，只考慮做了什麼決定
    maximizing your expected utility
  - utility：對於目標的完成度、好壞，用來評估的指標 ⇒ $\boxed{\text{Computational Rationality}}$

* Designing Agent
  - agent：is an entity that perceives and acts
    透過一些sensor來感知環境，做出一些決策來讓utility最大化

* Agent　(最直覺的方法，好處：較快)
  1. reflex agent：根據當下的world做出action，不會考慮後果，做完or卡住就停了
  2. planning agent：考慮一連串的事、後果
     - planning：思考一連串的動作後再做
     - replanning：先完成當下的動作後再重新規劃一次
     - optimal：最佳化所有的action
     - complete：只要能完成任務就可以

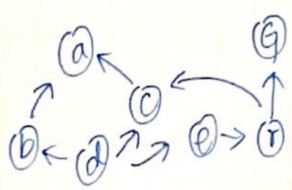
⇒ 要如何知道要take which action?

* Search Problem：在一個world，要找到一個動作可有較好的結果(透過"演算法")
  - a state space
  - a successor function
  - a start state and a goal test

  } 一個world (定義好world後就可用某種演算法來解決)

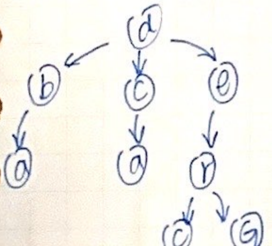
⇒ a solution is a sequence of actions (a plan) which transform the start state to a goal state through "algorithm"

* State space
  有2種展現state的方法
  1. State space graph



在graph裡，每個node只會出現一次
但rarely build it in memory (too big)

  2. Search tree



each node in the search tree is an entire path in the state space graph
每個node可能會重複出現 ⇒ tree無深度，depth = ∞

⇒ 我們只看我們要的就好

⇒ fringe：保留在記憶体的部份(末梢)

Search method : 1. Uniformed Search 2. Informed Search

*Uniformed search: 不知終點在哪 ⇒ 盲目尋找

Depth-First Search — Complete: ?, Optimal: No

以深度為主, fringe 乃 a LIFO stack

Breadth-First Search — Complete: yes, Optimal: 不一定

以廣度為主, fringe 乃 a FIFO queue

但DFS會有"cycle走不出來"的問題 ⇒ 限制每次iteration的深度

Depth-First Search - Iterative Deepening

get DFS's space advantage with BFS's time advantage

∴但因DFS & BFS都無考慮到cost

Uniform Cost Search — Complete: yes, Optimal: yes.

找"累積"到該node後最小的cost的node展開

Search method: 1. Uninformed Search 2. informed search

* Informed Search: 知道目標在什麼位置

Heuristics (啟發式) — 一種工具

有某個"concept"且依據"concept"去做 (concept是自己定義的)

Greedy Search — Complete: ?, Optimal: No.

每次做的決策都是對當下最有利的 (by heuristics)

A* Search (UCS + greedy) — Complete: yes, Optimal: with admissible heuristic (tree)

因為UCS像BFS會一直尋找,無目標; greedy知道目標但cost可能會太大)

⇒ UCS考慮"走的成本", greedy考慮"到終點的成本" ($h(x)$)

⚠ 但因$h(x)$是自己估計的,需要一些條件,否則不是optimal

⇒ Admissible Heuristic (可接受的h) $0 \le h(n) \le h^*(n)$, $h^*(n)$是true cost to goal

⚠ 但tree search會有重複node的問題 ⇒ extra work

Graph Search

將走過的node存起來才不會重複走

⚠ 但因為有heuristic所以又會有一些問題

⇒ Consistency of heuristics $h(A) - h(C) \le cost(A \text{ to } C)$

※ Summary

- Tree A*: is optimal with admissible heuristics
- Graph A*: is optimal with consistent heuristics

Constraint Satisfaction Problem 條件滿足的問題
達成某些條件才是重點(目標在滿足條件的狀況下完成)

* Constraint Satisfaction Problem
- state defines by variables x_i
- with domain D
- some <u>constraints</u>
  - Unary: involves one variable
  - Binary: involves two variable
  - high-order: involves three or more

* Constrain Satifaction Method

也可用DFS、BFS…但效果非常差

<u>Backtracking Search</u> = DFS + variable-ordering + fail-on-violation

is the basic uninformed algorithm

一個一個加,再不違反條件的情況下繼續走

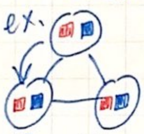
※但等到發現錯誤時通常來不及了 ⇒ detect failure early

- <u>Filtering</u>

assign一個variable時將和它有相關constrain的也刪掉,但它也無法全部檢查到

⇒ Arc consistency: 雙向看是否有違規 (if x loss a value, neighbors needs to be checked

※但也不是都可fixed ex.



- <u>Ordering</u>

1. 將剩餘值少的先處理 minimum remaining value (MRV)
2. 選會造成較少限制的 least constraining value (LCV)

※但也還是有 limit arc consistency的問題, 且若是K-consistency會需大量計算

<u>Structure</u>

重新排列,把不相關的&中間點切開

Tree-Structured

切開某部分讓它成為一個tree

<u>Iterative Algorithm</u>

將所有地塗色,再慢慢修改到無不合規定的

☆No fringe, Not complete & Not Optimal

<u>Local Search</u>

improve a single option until you can't make it better. 會再找到更好的解

☆No fringe, incomplete but <u>suboptimal</u>

Adverserial method：1. Deterministic　2. non-deterministic
希望演算法計算一個strategy(policy)可推薦採取的動作
Types of game：deterministic or stochastic?/one or more play?/zero sum?

Single Agent Tree
☆ value of a state：在該位置有可能拿到的最高分

Adversarial Game Tree - MiniMax search
計算每個node的minimax value，Just like DFS
※ 但若是一開始直接做，則有可能為了optimal就放棄一些机会(对手可能不是optimal)
且like DFS則計算量大

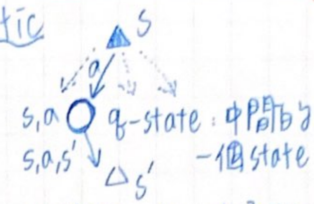
Game tree Pruning - Alpha-Beta Pruning
在minimax search中，已知拿到最高分了，則某些子樹就不須要經過
[α：max's best option on path to root
[β：min's best option on path to root
但在部分過程中算的node的value可能是錯的，不過對結果不會影響
※ 但在真實狀況中，無法計算到最後
⇒ depth-limited (限制計算minimax、α、β的深度)
用evaluation function取代terminal utilities (依照現在的情況給分)
※ 要這個是因為若d限制太淺会有thrash情況
※ 但上面的演算法是取最安全的情況下，但若是事情發生有机率……

Expectimax Search
計算平均而非minimax，且無法pruning (因為計算平均)

Markov Decision Process
- Utilities：將喜好轉為數字来表現
- Rational Preferences：$(A>B)\wedge(B>C)\Rightarrow(A>C)$
- Maximum expected utility (MEU)：choose the action that maximizes expected utility
- Utility Scales：衡量utility的方式

Adversarial Method: "Deterministic" → non-deterministic

當有些action發生時不是固定的,是有機率的



做某個action有部分機率會跳到另個state

Markov Decision Process — Expectimax

→ 接下來做的決定,只和現在的狀態有關

在deterministic時,我們要的是一個"計畫"

在non-deterministic中,我們要的是一個"policy"

one way to solve them is with "expectimax" but it didn't compute entire policies

※但若是reward先拿到會較好?(因有些沒拿到reward前就結束了)

⇒ discounting: 得到的rewards隨時間下降(→愈早拿到愈好) → 助於收斂

限制深度: 似depth-limited search

Markov decision Problem
- set of states $s \in S$
- set of actions $a \in A$
- transition function $T(s,a,s')$
- reward function $R(s,a,s')$

Optimal Quantities
- $V^*(s)$: 最佳動作得到的值
- $Q^*(s,a)$: 期望做a後拿到最好的值
- $\pi^*(s)$: 最佳的policy

☆ V^*不一定和Q^*相同,若是optimal才會相同

※但用expectimax會花太多時間,且有太多重覆的state, tree會無限深

Time-limited Value

計算k步的每個state的值就好,不用一直無限算 (it's what a depth-k expectimax)

⇒ 愈深則state的值會慢慢收斂 (due to discounting)

Value Iteration

從底層開始做,由累積的方式慢慢加值上去, repeat until convergence

⇒ will converge to unique optimal value

到目前為止
- policy: 每個state採取的action
- utility: reward (with discounting) 的總和
- value: 在該點期望得到的最大reward (max node)
- q-value: 在該點期望得到的最大reward (chance node)

The Bellman Equations: 定義"optimal utility" via expectimax
- $V^*(s) = \max_a Q^*(s,a)$
- $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
- $V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

※但要怎麼知道什麼時候會收斂?

Policy Iteration: policy evaluation + policy extraction
- evaluation: 依照你的policy給一個評分 → fixed policy π: 給固定的policy
- extraction: 把好的policy解出來

⇒ 先給(部分)固定的policy, 一直evaluation再慢慢更新修改到optimal, 加快收斂

Reinforcement Learning - online planning
前面都是已經知道所有的規則，但若是不知道，只能一直try，然後修改
在不知T.R.P的情況下，慢慢去學習

*Reinforcement Learning
還是MDP的問題，但只是不知道T.R.P

Model-based Learning：經過多次try後可計算T.R.P

Model-Free Learning：不去管T.R.P，直接算state的avg

- Passive reinforcement learning (固定policy去採樣學習)
  - direct evaluation：直接計算每個state的avg under π
    ∵但每個state都是獨立的學習 ⇒ 要花較久時間
    ⇒那為什麼不用policy evaluation呢？因為我們沒有T.R
  - sample based policy evaluation：在某個state不斷repeat做來取樣
    ∵但我們不一定能回到上一步
  - Temporal Difference Learning
    不斷的回到原點重複做後學習再取動態平均
    加上exponential moving average：較晚學習到的資料比例占越大
    (因為晚學到的較好)
    ∵但我們的policy還是固定的... ⇒ "learn Q-value instead of value"
- Active reinforcement Learning (學習optimal的policy/values)
  - Q-value iteration ⇒ Q-learning
    學習q-value再和前面的合併
    ∵但在學習過程中或許還有別的路，要不要走走看？⇒ explore
    - off-policy learning
    - converges to optimal policy even if you're acting suboptimally
    ∵要不要去試試其它路？
    Exploration：加上ε讓state可能亂跳
    ∵但有的ε不管走多久都會亂走 ⇒ 讓ε隨時間↓ ∴ exploration function
  - exploration function：去過的地方傾向用原來的action, or explore new place
    ∵但沒辦法通用？(改變門的位置就出不去了)
    ⇒ feature-based representation - linear value function
    describes a state using a vector of features，好處是經驗會被統整起來
    壞處是雖然share同一個feature但值會很不一樣
  - policy search: start with an ok solution (eg. q learning) then fine-tune by hill climbing on feature weights
    ⇒ learn the policy that maximize the rewards, not value