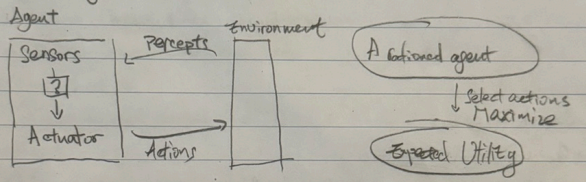
Cmpt 310

Lecture 2

- Rational Decisions
  - ↳ Rational : maximally achieving pre-defined goals
  - ↳ Rationality only concerns what decisions are made
  - ↳ Goals : are expressed in terms of Utility of outcomes
  - ↳ Being rational means maximizing your expected Utility

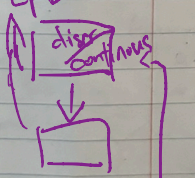
- Rational Agents



Percepts, environment, action space ⇒ Characteristics → decide techniques for selecting rational action

What is Utility is -X for goal

Episodic



Discrete: Chess
Continuous: Self-driving

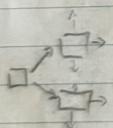
Environment properties
- ↳ Observability : Agent sensor full access (partial access)
- ↳ Multigent / Sigle Agent : Cooperation, decisions are influenced by other agent. One agent makes decisions & takes actions
- ↳ Deterministic / non Deterministic : Every action predictable . Current state (certain) → Next state. Actions not certain, involve probabilistic reasoning to make decisions
- ↳ Stochastic : Deals with probability
- ↳ Episodic / Sequential : Agent's Experience divided into discrete, 1 Action → (no affect) 2 Action. current decision → ~~affect~~ subsequent decision/outcome, long-term strategies
- ↳ Static / dynamic : no Change while agent deliberating. no Environment might change when agent acts compared to when it began deliberation, agent needs to be responsive and adapt to changes
- ↳ Discrete/Continuous : time and percepts and actions are divided into seperate interval time and percepts flow and uninterrupted manner

Agent Design

- Environment type determines Agent Design
  - ↳ Partially observable → agent requires memory (internal state)
  - ↳ Stochastic → agent may have to prepare for contingencies
  - ↳ Multi-agent → an agent might behave randomly at times to be unpredictable to other
  - ↳ Static → If the environment no change while agent deliberating, agent has luxury time to Compute rational decision
  - ↳ Real-Time ⇒ In real-Time environment, the agent operates as a continuously active controller
  - ↳ Unknown Physics: Unknown environment, the agent need to explore to understand how to operate
  - ↳ Unknown Performance Measure: When criteria for success unclear, agent might observe and interact with humans

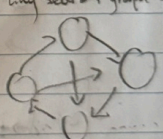
Search Problems

- Search Problem Consist
  1. ↳ A state space
  2. ↳ A successor function (with actions, cost)
  3. ↳ start / goal state
- Solution: A sequence of actions (a plan) transform start → goal state



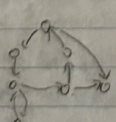
- State Space graph: (A mathematical representation of search problem)
  - ↳ Node: are (abstracted) world Configuration
  - ↳ Arc represent successors (action results)
  - ↳ The goal Test is a set of goal node (maybe only one)
  - → each state occurs only once

Tiny search graph for tiny search problem



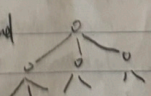
- State Graph / Search Trees

State Graph

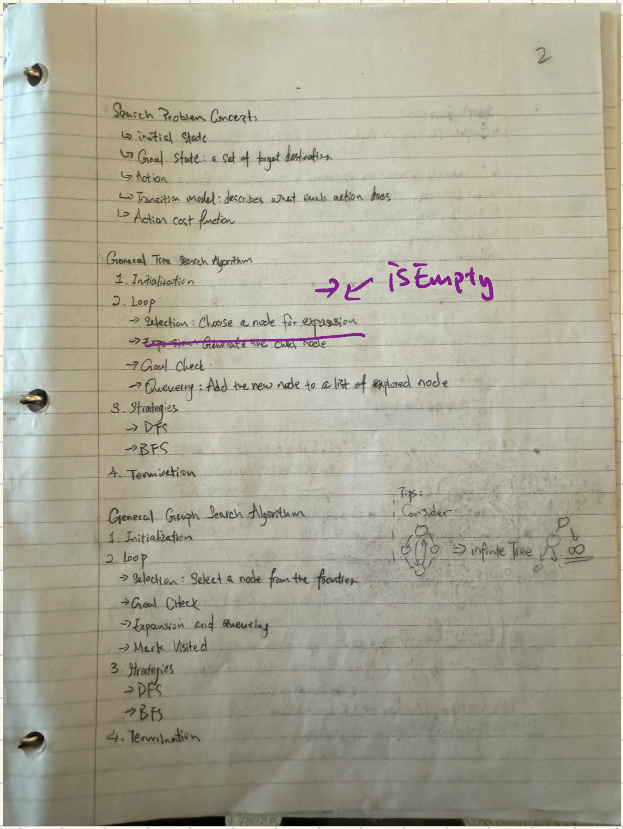


Construct both on demand
Construct As Small As Possible

Search Tree



Each Node is tree is an entire PATH in problem graph



2

Search Problem Concepts
↳ initial State
↳ Goal State: a set of target destination
↳ tation
↳ Transition model: describes what each action does
↳ Action cost function

General Tree Search Algorithm
1. Initialization
2. Loop
→ Selection: Choose a node for expansion
→ ~~Expansion: Generate the child node~~
→ Goal Check
→ Queueing: Add the new node to a list of explored node
3. Strategies
→ DFS
→ BFS
4. Termination

→ ↙ isEmpty

General Graph Search Algorithm
1. Initialization
2. Loop
→ Selection: Select a node from the frontier
→ Goal Check
→ Expansion and Queueing
→ Mark Visited
3. Strategies
→ DFS
→ BFS
4. Termination

Tips:
Consider
=> infinite Tree

Tree:
① Initialize
② Loop
↳ IsEmpty
↳ Selection (Stack or Queue)
↳ Goal Check
↳ Expansion

Graph:
① Initialize (visited Set)
② Loop
↳ IsEmpty
↳ Select
↳ Goal
↳ Mark Node
↳ Expansion

Search Properties

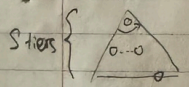
- Complete Planning Vs Replanning : a full plan or solution before EXECUTION
  plan just enough to move Next Step. Entails planning increment
- Completeness : whether the Algorithm is guaranteed to find a solution if exists (or no solution)
- Cost Optimality : Algorithm find lowest cost
- Time Complexity
- Space Complexity
- Computation Complexity

Search Type:

① Uninformed Search Types
- BFS
- DFS
- Uniform Cost Search

② Informed Search Type
- A*
- greedy search

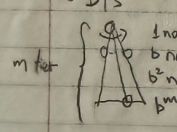
BFS



Characteristics:
Completeness / Optimality (serve path Cost)

time Complexity/runtime memory $O(b^s)$
s: tiers, b: children

DFS



Characteristics:
Completeness (not infinite) / may not optimality

time and Space Complexity $O(b^m)$
Space fringe take (Only has siblings on path to root, so $O(bm)$)

DFS : → Memory Efficiency → Solutions are Deep → Graph Many Node and Few Solution
→ Detecting Cycles → Safe for loop

BFS : Shortest path → Shallow Solution → Tree Level Order Traversal → Find All solution
→ limit memory

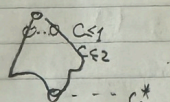
can be implemented Priority Queue

Uniform Cost Search (UCS)

→ Strategy: expand a cheapest node first

→ Fringe is a priority queue (priority: cumulative cost)

effective depth C^*/ε



Characteristic:

- Processes all nodes with cost less than cheapest solution.
- Solution cost C^*, arc (action) cost at least ε then "effective depth" is C^*/ε
- Time $O(b^{C^*/\varepsilon})$

Tips:

- fringe take how much space, → last tier $O(b^{C^*/\varepsilon})$
- Complete, if finite cost, minimum arc cost positive
- good: Complete / Optimal bad: Explores options in every "direction" no info about goal location.

Informed Search (has Compass Person)

↳ Use a heuristic function $h(s)$ → estimate how "far" the agent from goal

↳ $h(s)$ is optimistic (no overestimate) → called admissible heuristics

↳ UCS try to minimize the cost function $g(s)$

↳ Greedy Search try to minimize heuristic $h(s)$

↳ A* try minimize $f(x)$ (Combination function ⇒ $f(x) = g(s) + h(s)$)

→ A heuristic is:

↳ A function, estimates how close a state is to a goal

↳ for a particular search problem

↳ (Manhattan distance, Euclidean distance for pathing for example)

Greedy Search

local optimality

only see next step

↳ Expand the node seems closest

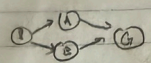
↳ not optimal (not accounting for cost). (Best-first takes you straight to wrong state)

↳ Worst-Case: like a badly-guided DFS

A* Search

↳ A* Search: $f(n) = g(n) + h(n)$

orders by the Sum = Uniform-Cost $g(n)$ (cost) + Greedy $h(n)$ (goal proximity)

↳ 

Q: Stop when enqueue a goal?
A: No, when dequeue

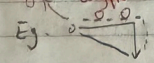
↳ not optimal ?

○ Admissible Heuristics

↳ A heuristic h is admissible (optimistic), only if

$0 \le h(n) \le h^*(n)$ → true cost to goal (from original)

original, augments

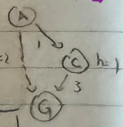
Eg. 

+ = Optimal A*

○ Consistency of Heuristics (slightly stricter)

(or h cost ~~h estimate cost~~ 不会变惜)

→ main idea: estimated heuristic cost ≤ actual cost



Consider optimal

↳ Admissibility: heuristic cost ≤ actual cost to goal

$h(A) \le$ actual cost A to G

↳ Consistency: heuristic "arc" cost ≤ actual cost for each arc

$h(A) - h(C) \le \text{cost}(A \text{ to } C)$

→ Consequences of Consistency

h() ↓, g() ↑ <- - - - -> The f value along a path never increase

h() faster g()

$h(A) \le \text{cost}(A \text{ to } C) + h(C)$

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

$= h(C) + g(A) + \text{Cost}(A \text{ to } C)$

$\ge h(C) + g(A) + h(A) - h(C)$

↳ A* graph search is optimal (by one of two)

Consistent Heuristic: ensures heuristic estimate is ~~always increasing~~ decrease/same

↳ Optimal

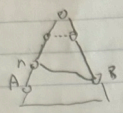
$h(A) \le \text{Cost}(A, C) + h(C)$

to goal; actions by us $\bigcirc \to \bigcirc$; $h() \downarrow$ $g() \uparrow$

Heuristic design often use relaxed problem

Optimality of A* Tree Search

A → optimal
B → Suboptimal



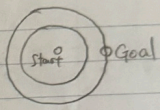
Claim: n will be expanded before B

1. f(n) is less or equal to f(A)
2. f(A) less than f(B)
3. n expands before B

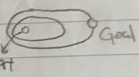
All ancestors of A expand before B

UCS vs A* Contour

→ Uniform-Cost expands equally in all "directions"



→ A* expands mainly toward the goal, but does hedge its to sure optimality



Constraints Satisfaction Problems (CSP)

Conditions: 1 → N variables 2 → domain D 3 → constraints

Agent: Search for: Planning, Identification

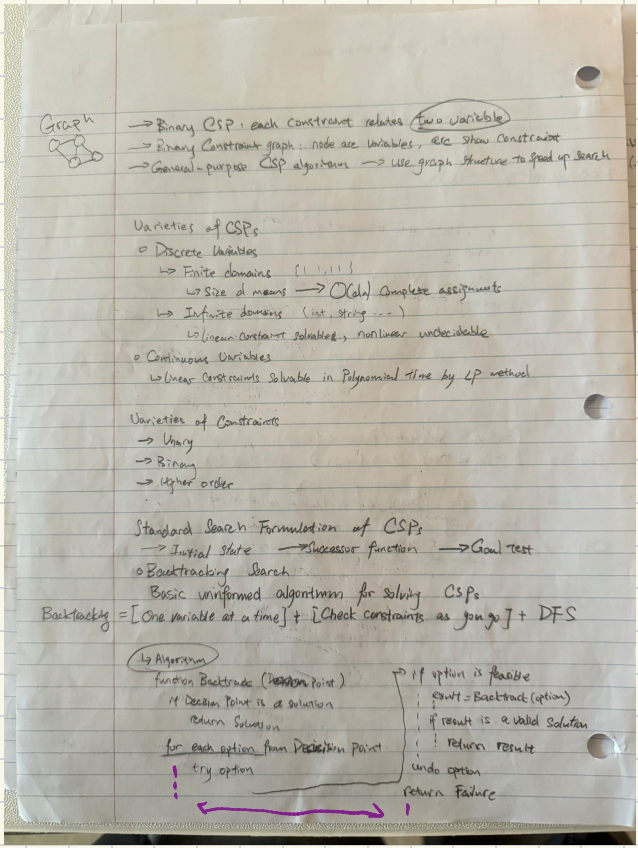
↳ planning: sequences of actions → path to the goal (important)

CSP → ↳ Identification: assignment to variables → goal itself (important), not path (All path same depth)

Standard search problems:
- state is "black box": arbitary data structure
- Goal test can be any function over states
- Successor function can be anything

CSP:
- A special Subset of search problem
- State defined by variables X_i with value from domain D
- Goal test is a set of constraints

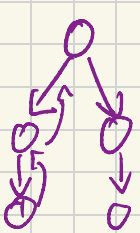
Backtracking




```
function Backtrack(DecisionPoint):
    if DecisionPoint is a solution:
        return Solution
    for each option from DecisionPoint:
        try option
        if option is feasible:
            result = Backtrack(option)
            if result is a valid solution:
                return result
        undo option
    return Failure
```

```
Algorithm BackTrack(Sol)
  If isSol(solution)
     Output(solution)
     return

for choice in GetChoice(sol)
    If IsValid(choice, sol)
       ApplyChoice(sol, choice)
       BackTrack(sol)
       UndoChoice(sol, choice)
```



- Improving Backtracking

General-purpose ideas give gains in speed

→ ordering

→ Filtering

→ structure

↳ Filtering: ~~Forward checking~~

track of domains for Unassigned Variable & Cross off bad option

↳ Forward checking | propagates information but no early detection for failures

↳ Cross off ✗ violated constraint, when added to assignment

↳ Consistency of a single Arc: An arc $X \to Y$ is consistent iff every x in tail there is some y in the head which could assign without violating

↳ Arc Consistency detects failure earlier than forward checking

↳ If X loses a value, neighbors of X need to be rechecked

↳ Can be run as a preprocessor or after each assignment

$X\{x_1, x_2 \dots\}$

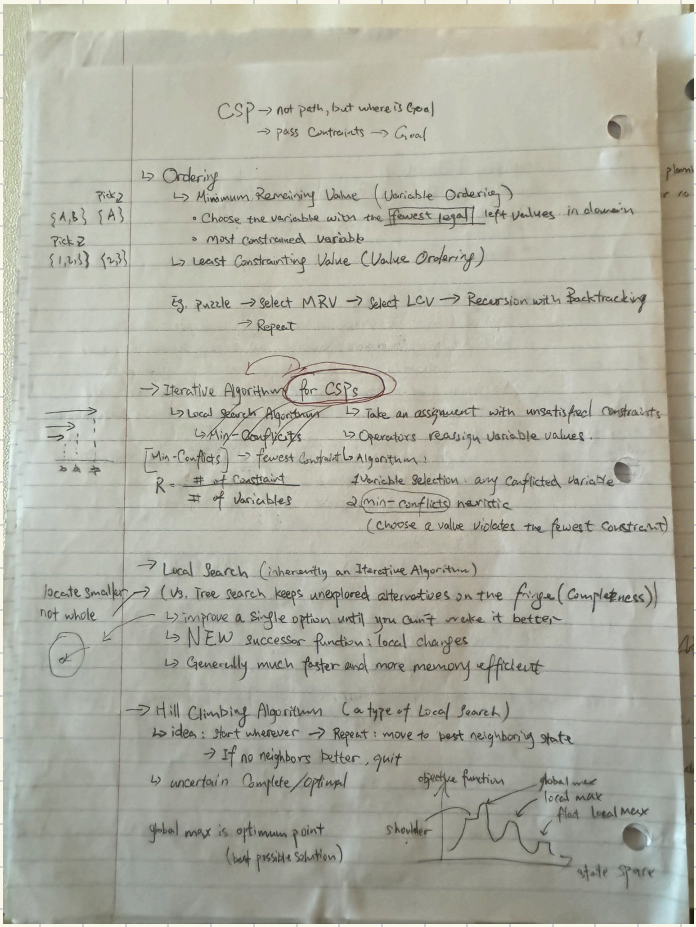
$Y\{y_1, y_2 \dots\}$

Can be assigned (x_i, y_j) without violating Constraint

↳ ~~Exact~~ Algorithm

```
function AC-3(csp)
queue = all arcs in csp
while queue is not Empty
    (x, Y) = queue.pop()
    if remove_inconsistent_value(X, Y)
        for each Z in neighbors of X except Y:
            queue.add((Z, X))

function remove_inconsistent_value(X, Y)
removed = false
for each x in domain of X
    if no value y in domain of Y allows (x, y) to satisfy the Constraint in X and Y
        domain of X = domain of X - {x}
        removed = true
Return removed
```



迭代算法伪代码（使用最小冲突启发式）：

Iterative algorithm

1. 开始时，随机为CSP中的每个变量分配一个值，即使这个初始分配可能违反一些约束。

2. while (当前分配未能解决所有约束) do
   a. 选择一个冲突变量。冲突变量是指参与了至少一个约束冲突的变量。
      可以随机选择，或者选择冲突最多的变量。

   b. 为选定的变量选择一个新值。这个新值应该是最小化该变量当前约束冲突的值。
      即，选择一个新值，使得改变后违反的约束数量最小。

   c. 如果找不到能减少冲突的新值（意味着当前变量在所有可能值中冲突都一样多），
      可以选择保持当前值，或随机选择一个新值尝试打破僵局。

   d. 更新变量的值为步骤b中选择的新值。

3. 当所有变量的分配不再产生任何冲突时，算法结束，当前的分配被认为是CSP的解。

↳ Hill Climbing Algorithm (most basic local search)

function H-C (Problem) returns a state is local max
  Current = Make-Node (Problem. Initial_State)
  loop
    neighbor = a highest-valued successor of current
    If neighbor. Value ≤ current. Value
      then return Current. State
    current = neighbor

→ Lecture 5 Adversarial Search

→ Types of Game (many kinds of game)
- Game = task environment > 1 agent
- Axes: Deterministic, Perfect information, multiple player, turn-taking, Zero-Sum

(Want Algorithm for calculating a strategy (policy)) not plan

Policy = Strategy

→ "standard" Games (solution is policy not plan)
- "Standard" → Deterministic, observable, 2 player, turn-taking, Zero-Su
- Game formulation: ◦ Initial state S (s0)
  - ◦ player P = {1 ... N}
  - ◦ Actions A = (depends)
  - ◦ Transition Function: S X A => S
  - ◦ Terminal / goal Test: S → {t, f}
  - ◦ Terminal Utilities (value): S X P → R
- ◦ Solution for player is policy: S → A

A's utility + B's utility = 0

→ zero-Sum Games
- ◦ Agents have opposite utilities
- ◦ Pure Competition:
  A maximizes, B minimizes it
  Vice versa

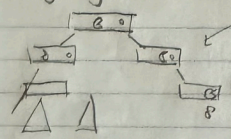
→ General Sum Games
- Agent have independent utilities
- Cooperation, indifference, competition, shifting alliances, etc

→ Team games - Common payoff for all team member

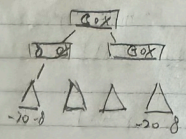
→ Adversarial Search

A → Goal ← B
optimum get

- Single-Agent Trees (go for max)



- Adversarial Game Trees



→ Minimax Values

State under Agent's Control: $V(s) = \max_{s' \in successors(s)} V(s')$

under Opponent's: $V(s') = \min_{s \in successors(s')} V(s)$

→ Minimax

- Deterministic, Zero Sum game:

  Tic-Tac-toe → (one maximizes result, one minimizes result)

- Minimax Search

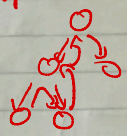
  A state-space search tree

  Player alternate turns

  Compute each node's minimax value (max myself, min opponent)

- Algorithm

traversal



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

def max-value (state)
  initialize $V = -\infty$
  for each successor of state:
    $V = \max(V, \text{value}(\text{successor}))$
  return V

def min-value (state)
  initialize $V = +\infty$
  for each successor of state
    $V = \min(V, \text{value}(\text{successor}))$
  return V

[DFS, Time: $O(b^m)$ Space: $O(bm)$]

- Generalized minimax for not zero-sum but multiple players
  Terminal has utility tuples [1,1,1] [1,1] ... (Node value also utility tuple)
  Each player maximizes its own component
  Can Cooperate and Competition for own goal

- Resource limits (realistic, won't search to leaves)
  → Depth-limited search (use evaluation function)
    (no guarantee of optimal play)
    ↳ use iterative deepening for an anytime algorithm

- Evaluation Function
  $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
  (weight + 特徵 function)

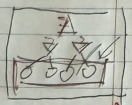
~~Alpha-Beta~~ Game Tree Pruning

→ Alpha-Beta Pruning

α = Max player 目前为止 得到 lowest score

- The order of generation matters: more pruning is possible if good moves come first

β = Min player 目前为止 得到 highest score



For Node

↳ 1. initialize

2. For search, check Max or Mini to upload α or β

3. pruning, if a node α ≥ β, don't have to keep search (for Max player) ~~already has a biggest point~~ Same

(for min player) if a node β ≤ α, don't have to search them

(With perfect ordering: $O(b^{m/2})$ Time, Doubles depth solvable)

Traversal