1 Introduction to AI

1.1 What is AI

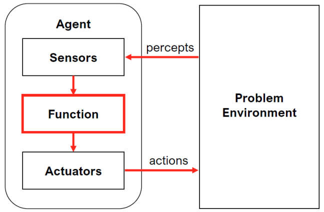
- Intelligent mechanisms that solve problems to help humans
  - concerned with human thinking
  - assessed based on generality (more dynamic solutions that is able to deal with many cases) and performance (perform at least as well as humans)

1.2 Kinds of AI

- Strong AI
  - General problem solver → very dynamic program that solves many problems
- Weak/Narrow AI
  - Less dynamic program → typically solves 1 problem, easier to formalize

1.3 Rational Agent

- An agent is an entity that perceives its **environment** through **sensors** (what is captured about environment) and acts through **actuators** (how agent affect change in environment)
- An agent's **percept sequence** is the complete history of everything the agent has ever perceived.
- What is rational depends on: (1) quantifiable performance measure that defines success, (2) prior knowledge of the env, (3) actions available, (4) percept sequence to date.
- For each possible percept sequence, a Rational Agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.



1.4 AI as Graph Search

- Percept → state/vertex
- Desired states → goals
- Actions → edges
- Search space → graphs
- Solved using graph search algorithms

1.5 Environment Properties

- **Fully observable** vs **Partially observable**: Partially observable agent does not have access to all information (e.g. fully observable maze VS slowly expanding maze based on actions taken). Requires dealing with **uncertainty** i.e. backtracking algos
- **Deterministic** vs **Stochastic**: if the next state of the env is completely determined by the current state and the action executed **VS** otherwise. (A fully observable environment that has randomness with action is stochastic) (e.g. Sudoku VS Poker)
- **Episodic** vs **Sequential**: actions only impact current state **VS** action impact future decisions
- **Discrete** vs **Continuous**: in terms of state of env, time, percepts and actions (tend to discretize continuous environments)
- **Single agent** vs **Multi-agent**: whether there are any other agent in the environment whose actions directly influence the performance of this agent, multi-agent further divided into **competitive** and **cooperative**
- **Static** vs **Dynamic**: if the environment is unchanged while an agent is deliberating **VS** otherwise

1.6 Agent Types, in increasing generality

- **Simple Reflex Agent**: narrow agents, follows set of rules (if-else statements) to make decision, direct mapping of percepts to actions, mostly domain specific, impractical with large search space (requires iterating through all cases)
- **Model-based Reflex Agents**: Makes decision based on internalized model (typically logical agents or bayesian networks)
- **Goal-based/Utility-Based Agent**: given state and available actions, determines a sequence of actions to reach goal/maximize utility (typically solved using search algo, local search, CSP or adversarial search)
- **Learning Agent**: Agents that learn how to optimize performance

2 Solving Problems by Searching

2.1 Path Planning Problem Properties

- Environment assumed to be fully observable, deterministic, discrete and episodic (assume that we can see the whole problem, plan a path and then execute it)
- Plan is formed sequentially, each action in the plan impacts the next action in the plan, one plan (path) is independent from another plan

2.2 Problem Formulation

State (abstract data types that describes the environment), **Actions** (function that returns set of actions possible given a particular state), **Transition Models** (description of each action), **Goal Test** (determines whether a state is a goal state), **Path/Action Cost** (assigns a numeric cost to each path) A **node** includes state, parent node, action, and path cost, depth

2.3 Evaluation criteria

- **Completeness**: always find solution if one exists and correctly report failure when there is no solution
- **Optimality**: finding a least-cost solution
- **Time complexity**: no of nodes generated
- **Space complexity**: max. no of nodes in memory

2.4 Problem parameters

- b: max. no of successors of any node (branching factor)
- d: depth of shallowest goal node
- m: max. depth of search tree

2.5 Uninformed Search

Uninformed Search Strategies

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes*	Yes**	No***	No	Yes*
Optimal	No*	Yes	No	No	No*
Time	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$

*: BFS, IDS – complete if b/state space is finite or if there is a solution, optimal if step costs are identical
**: UCS is complete if b is finite and action cost $> \epsilon > 0$
***: DFS is complete only on finite depth & branching factor graphs
C^* is the optimal cost

2.5.1 Breadth-First Search (BFS)

Expand shallowest unexpanded node, frontier is FIFO. Takes $O(b^{d+1})$ space if using late goal test. Typically use **Graph Search**

2.5.2 Uniform-Cost Search (UCS)

Expand least-path-cost unexpanded node, frontier is PQ by path cost. Equivalent to BFS if all step costs are equal

2.5.3 Depth-First Search (DFS)

- Expand deepest unexpanded node, frontier is LIFO.
- **Backtraking Search**, space can be $O(m)$ if successor is expanded one at a time (partially expanded node remembers which successor to generate next)
- Typically use **Tree Search**

2.5.4 Depth-Limited Search (DLS)

Run DFS with depth limit l, to solve the infinite-path problem

2.5.5 Iterative Deepening Search (IDS)

- Perform DLS with increasing depth limit.
- Preferred if search space is large and depth of solution is not known
- Properties of completeness from BFS with space complexity of DFS, disadvantage of reruning top levels many times

2.6 Tree vs Graph Search

Graph search could contain cycles & redundant paths and requires a **visited** hashmap to prevent revisits
Tree search on the other hand allows revisits

2.6.1 Graph Search Versions

```
frontier = {Node(initial state, NULL)}
reached = {initial state: Node(initial state)}
while frontier not empty:
        current = frontier.pop()
        if isGoal(current.state): return current.getPath()
        for a in actions(current.state):
                successor = Node(T(current.state, a), current)
                if successor.state not in reached:
                        frontier.push(successor)
                        reached.insert(successor.state: successor)
return failure
```

```
frontier = {Node(initial state, NULL)}
reached = {initial state: Node(initial state)}
while frontier not empty:
        current = frontier.pop()
        if isGoal(current.state): return current.getPath()
        for a in actions(current.state):
                successor = Node(T(current.state, a), current)
                if successor.state not in reached or
                   successor.getCost() < reached[successor.state].getCost():
                        frontier.push(successor)
                        reached.insert(successor.state: successor)
return failure
```

Graph search V1 ensures nodes are not revisited which could omit optimal paths
Graph search V2 solves that by allowing revisits provided cost is lower

2.6.2 Graph Search Properties

- Time and space complexity are both $O(|V|+|E|)$

2.7 Informed Search

Informed (Heuristic Search Strategies)

Idea is to use domain knowledge to determine cost required to go from particular state to nearest goal which reduces search space

2.7.1 Greedy best-first search

- $f(n) = h(n)$ = estimated cost of cheapest path from n to goal
- Expands nodes that appear to be closest to the goal
- **Incomplete** → can get stuck in loop between nodes where h values are lowest, **Non-optimal** under both tree and graph search, **Time** $O(b^m)$, **Space** $O(b^m)$

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node          Late Goal Test
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then     Graph-search (v2)
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do                   Utilises search problem definitions
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

2.7.2 A* Search

- $f(n) = g(n) + h(n)$, $g(n)$ = path cost from start node to node n
- Avoids expanding paths that are already expensive
- **Admissible Heuristic** never overestimates the cost to reach the goal: $\forall n, h(n) \le h^*(n)$ where $h^*(n)$ = true cost
  - Consequence: all paths with actual costs less than P must be searched
- **Consistent Heuristic**: $h(n) \le c(n, n') + h(n')$, will make $f(n)$ monotonically increasing along a path
- Consistency $\implies$ Admissibility
- $h(n)$ is **admissible**:
  - Optimal under **Tree Search** and **Graph Search V2**
  - Non-optimal under **Graph Search V1**
- $h(n)$ is **consistent**:
  - Optimal under **Tree Search**, **Graph Search V2 and V3** (insert into visited when popped)
  - Non-optimal under **Graph Search V1**
- **Complete** if b & m finite OR has a solution and all action cost $> \epsilon > 0$, **Optimal**, **Time** $O^{h^*(s_0) - h(s_0)}$ where $h^*(s_0)$ is the actual cost of getting from root to goal, **Space** $O(b^m)$
- **Dominant heuristic**: if $\forall n, h_2(n) \ge h_1(n)$ then h_2 **dominates** h_1
- More dominant heuristics incur lower search cost

3 Goal Search

- Path to goal is irrelevant, the goal state itself is the solution.
- Advantages: (1) use very little $O(b)$/constant memory, (2) can find reasonable solns in large/infinite continuous state spaces
- Useful for **pure optimization problems**: objective is to find the best state according to an **objective function**. e.g. Vertex cover, TSP, Boolean Satisfiability Problem (SAT), Timetabling/scheduling

3.1 Hill-climbing Algorithms

3.1.1 Problem Formulation

- Start with **complete** state i.e. no partial state (removes build up stage and start checking on 1st iteration)
- Each state is a possible solution

3.1.2 Steepest Ascent - Greedy

- Start with random initial state, in each iteration find successor that improves on current state
- Requires **actions** and **transition** to determine successors
- Requires some heuristic to give value to each state e.g. $f(n) = -h(n)$ and find maxima
- Algorithm **can get stuck at local maxima** and return non-goal state
- Problems arises if met with **shoulders/plateau, local maxima or ridge**

```
current = initial_state
while true:
        neighbour = highest_valued_successor(current)
        if value(neighbour) ≤ value(current): return current
        current = neighbour
```

3.1.3 Stochastic Hill Climbing

- Instead of choosing highest-valued-successor in each step, **choose randomly among states with better values** instead
- Idea is to make the choosing of next state less deterministic to give to algo more chance of finding global maxima
- Takes longer but could lead to better solutions

3.1.4 First-choice Hill Climbing

- Handles high branching factor by randomly generating successors until one with better value is found
- Possible to achieve $O(1)$ space with this

3.1.5 Sideways Move

- Replace the ≤ sign in steepest ascent with <
- Allows algo to traverse shoulders/plateaus

3.1.6 Random-restart

- Adds outer loop which randomly pick new starting state and keep attempting restarts until solution is found (up to a certain threshold)
- Also allows for sideways move

```
current = NULL
while current is NULL or not isGoal(current):
        current = random_initial_state()
        while true:
                neighbour = highest_valued_successor(current)
                if value(neighbour) < value(current):
                        break
                current = neighbour
return current
```

generally want to have some way of limiting # of times algo restart
local search
finding for maxima

3.2 Analysis of Hill Climb

- Hill climbing (via steepest-ascent) with random restarts
  local minima which is not goal
  - Solution: p_1 = 14% (expected solution in 4 steps; expected failure in 3 steps)
  - Expected computation = 1×(steps for success) + ((1 - p_1) / p_1)×(steps for failure)
    = 1×(4) + (0.86/0.14) ×(3)
    = 22.428571428571427 steps

  (1 - p_1) / p_1) determines the expected number of failed attempts
- Adding sideways moves
  - Solution: p_2 = 94% (expected solution in 21 steps; expected failure in 64 steps)
  - Expected computation = 1×(steps for success) + ((1 - p_1) / p_1)×(steps for failure)
    = 1×(21) + (0.06/0.94) ×(64)
    = 25.085106382978722 steps
- 8-Queens possible states = 8^8 = 16777216

  Extremely efficient for such a large space

3.3 Local Beam Search

- Stores k states instead of 1
- Algo begins with k random restarts which generates successors for all k states
- Next iteration will repeat the above step with best k among ALL generated successors found (unless goal is found)
- Better than k parallel random restarts, Since best k among ALL successors taken (not best from each set of successors, k times)
- Also has a stochastic variant to increase probability of escaping from local maxima

4 Search Problem Representation

1. State Representation
   - State how the problem is represented e.g. $m \times n$ grid
   - State values that each cell/state can take e.g. "1" means action performed, "0" means valid, "X" means invalid
   - How the position is encoded e.g. (x, y)
2. Initial State
   - How each cell/state is initially labeled i.e. how do you determine what is initially labeled "X" or "0" or "1"
   - Starting position
3. Actions
   - Movements e.g. Up down left right
   - Possible actions to take e.g. clean cell, eat cell etc.
4. Transition
   - How to update cells
5. Step cost
6. Goal test

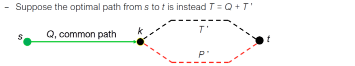
5 Local Search Problem Representation

- Initial state
  - How do you get the first "complete" state
  - What heuristic is being used?
  - How do we assign a value to the initial state?
- Finding next state
  - How do we transition from one state to another? What actions are taken?
- Stopping State
  - What happens if $val(next_state) \le val(curr_state)$?
  - What happens if $val(next_state) > val(curr_state)$?

6 Proofs

6.1 Why is UCS optimal?

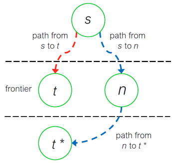
- UCS traverses paths in order of path cost
  - This is because path costs from the initial state are always increasing (given ε)
    - i.e., whenever a node, n, is added to a path, P, the new path, P' must have a path cost that is at least ε greater than the past cost of P
- UCS finds the optimal path to each node
  - Suppose UCS outputs path $P = Q + P'$ as the solution for s to t
  - Suppose the optimal path from s to t is instead $T = Q + T'$



  - UCS must skip shorter paths between k and t for it to have chosen P, which is a contradiction since it always chooses shorter paths to explore first

6.2 Why is A* optimal?

- Assume non-optimal *t* expanded before *n* (on optimal path)
  - f(*t*) < f(*n*)
- Assuming tree-search
  - All paths searched
  - All sub-paths* along a single path to a goal must be searched before path including that goal
    - For non-goal *m*, f(*m*) ≤ g(*m*) + h*(*m*) (since admissible)
    - If goal *m** on path from *m*, f(*m*) ≤ f(*m**) (given ε)
  - Since *t* * is goal on optimal path
    - f(*n*) < f(*t**) < f(*t*)
    - *CONTRADICTION*



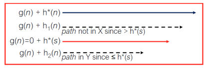
* Consider a path, *P*, from an initial state *s* to a goal state *t*, to be $s > n_1 > n_2 > \ldots > n_k > t$
Let a sub-path to *P*, *P* ' be any path $s > n_1 > n_2 > \ldots > n_i$, where $1 \leq i \leq k$

6.3 Why dominant heuristic is more efficient?

- Proving a dominant heuristic improves efficiency?
  - All paths with approximated path cost ≤ optimal path cost are explored
  - A* is more efficient under h_1 since |X| ≤ |Y|, where
    - X denotes the set of paths with costs below h*(*s*) based on h_1
    - Y denotes the set of paths with costs below h*(*s*) based on h_2
    - And *s* is the initial state
  - Consider example path, node *n* ; assume
    - *n* not on optimal path and (g(*n*) + h*(*n*)) > h*(*s*)
    - possible that $h_2(n) \leq h^*(s) \leq h^*(n)$ // so $n \in Y$
    - and also that $h^*(s) < h_1(n) \leq h^*(n)$ // so $n \notin X$
  - Consider the reverse of the above:
    - $h_1(n) \leq h^*(s) \leq h^*(n)$ & $h^*(s) < h_2(n) \leq h^*(n)$
  - This is impossible!
    - As we have: $h_1(n) \leq h^*(s)$ & $h^*(s) < h_2(n)$
    - Or: $h_1(n) < h_2(n)$ which contradicts our dominance assumption

Assume h_1 dominates h_2

Note that h*(*s*) is the optimal path cost!



Note: *path* refers to node *n*

6.4 Why UCS time and space complexity is $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$

- In the worst case, we assume step costs of all actions are ϵ. Since, at each state, the number of possible actions is b, we may assume the progression:

$$1 \text{ (initial state)} + b \text{ (nodes with path cost } \epsilon) + \\ b^2 \text{ (nodes with path cost } 2\epsilon) + \ldots + b^{\lfloor C^*/\epsilon \rfloor}$$

- However, note UCS performs a late goal test, which means that by the time it checks the node at depth $\lfloor C^*/\epsilon \rfloor$, it would have also generated, in the worst case, all the nodes in the following level as well, which requires an additional term $b^{\lfloor C^*/\epsilon \rfloor + 1}$.
- This series sums to $(b^{\lfloor C^*/\epsilon \rfloor + 2} - 1)/(b-1) = O(b^{\lfloor C^*/\epsilon \rfloor + 1})$.
- It should also be noted that we take the floor since step size is lower bounded by ϵ - i.e., since we cannot have a step cost $< \epsilon$, the remainder from C^*/ϵ must come from some steps costs in the optimal path having values higher than ϵ, and not from one additional step that had cost $< \epsilon$.

6.5 Why deterministic search algo will search entire state space

- Idea is that it is possible to randomly pick a goal node g_1, run the algo and if it doesn't search the entire tree, pick another goal node g_2 that is in the unexplored set when g_1 is used
- Can keep repeating these steps to force algo to search entire tree since it is a deterministic algorithm

7 Miscellaneous

7.1 Episodic vs Sequential

In most cases, when a human tries to solve a puzzle for e.g. Sudoku, the actions themselves are considered sequential. However, in the case of an intelligent agent, it would plan all the moves ahead. This makes the intelligent agent actions "episodic" where each action is the entire plan to solve one case of the maze.

7.2 Defaults for Graph Search

- All default to graph search V1 unless otherwise stated
- BFS defaults to early goal test (stop when goal is found)

7.3 Calculating Effective Branching Factor of Heuristics

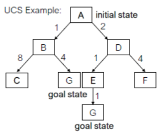
- Given empirical results of N nodes explored and solution depth of d
- Solve for b^* using $N + 1 = ((b^*)^{d+1})/((b^*) - 1)$

7.4 Calculating size of search tree

- Geometric series: $a + ar + ar^2 + \cdots + ar^{p-1} = a(r^p - 1)/(r - 1)$
- Size of search tree: $(r^p - 1)/(r - 1)$

8 Counter Examples

8.1 Non-Optimality under Graph Search V1



Tie-breaking: nodes ordered alphabetically when priority is the same

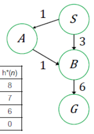
	Frontier:	Reached:
ITR1 =	[A((-),0)]	[A]
ITR2 =	[B((A),1), D((A),2)]	[A,B,D]
ITR3 =	[D((A),2), G((A,B),5), C((A,B),9)]	[A,B,C,D,G]
ITR4 =	[E((A,D),3), G((A,B),5), F((A,D),6), C((A,B),9)]	[A,B,C,D,E,F,G]
ITR5 =	[G((A,B),5), F((A,D),6), C((A,B),9]	[A,B,C,D,E,F,G]
ITR6 =	DONE (A,B,G) Non-optimal path!	

- Will A* be optimal under graph search Version 1?
- Example

T02 Q3b - construct an alternative example

Assume this admissible h:

n	h(*n*)	h*(*n*)
S	8	8
A	7	7
B	0	6
G	0	0



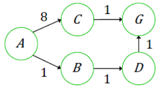
Trace:
ITR1 = [S((-),0+8)]
ITR2 = [B((S),3+0), A((S),1+7)]
ITR3 = [A((S),1+7), G((S,B),9+0)]
ITR4 = [G((S,B),9+0)] as B popped before, do not revisit
ITR5 = DONE (S,B,G) not the optimal path!

We need a tighter constraint on h

Similar to what UCS offers → contours of search progression

8.2 Non-Optimality using Greedy Best First Search

- Example (tree-search)



Assume this h:

n	h(*n*)	h*(*n*)
A	3	3
B	2	2
C	1	1
D	1	1
G	0	0

Trace:
ITR1 = [A((-),3)]
ITR2 = [C((A),1), B((A),2)]
ITR3 = [G((A,C),0), B((A),2)]
ITR4 = DONE (A,C,G)

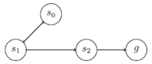
Notice that even with the perfect heuristic, we may not get the optimal solution. Why?

h*(*n*) = true path cost from *n* to nearest goal

Algorithm never exploits information on path already travelled.

8.3 Non-Completeness using Greedy Best First Search

Solution: Consider the following search space with initial state s_0, goal state g, and where $h(s_0) = 3$, $h(s_1) = 4$, $h(s_2) = 5$, and $h(g) = 0$.

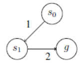


Each time s_0 is explored, we add s_1 to the front of the frontier, and each time s_1 is explored, we add s_0 to the front of the frontier. Notice that s_2 is never at the front of the frontier. This causes the greedy best-first search algorithm to continuously loop over s_0 and s_1.

8.4 Admissible but not consistent heuristic

Solution: An example of an admissible heuristic function that is not consistent is as follows.

Consider a heuristic function h, such that $h(s_0) = 3$, $h(s_1) = 1$, and $h(t) = 0$ for the following graph.



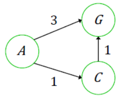
h is admissible since

$$h(s_0) \leq h^*(s_0) = 1 + 2 = 3$$
$$h(s_1) \leq h^*(s_1) = 2$$

However, h is not consistent since $3 = h(s_0) > c(s_0, s_1) + h(s_1) = 1 + 1 = 2$.

8.5 Non-Optimality of A* using consistent heuristic under Graph search V1

- Refer to the UCS example on Slide 15
- Assume consistent heuristic h*
  - With this example, we have
    - F = {A(g(A)=0 + h(A)=2)(-)}; R = {A}
      - pop A(0+2)(-), push C(1+1)(A) and G(3+0)(A)
    - F = {C(2)(A), G(3)(A)}; R = {A, C, G}
      - pop C(2)(A), cannot push G(2+0)(A,C) since already in R!
    - F = {G(3)(A)}; R = {A, C, G}
      - pop G(3)(A), path is A → G



Considering G as reached too early under graph search for A* is similar to applying Early Goal Test to UCS!