CS 486 Personal Notes

Marcus Chan

Taught by Pascal Poupart & Sriram Ganapathi Subramanian

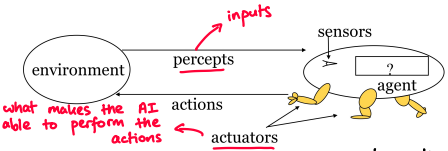
UW CS '25

Chapter 1: Introduction

REINFORCEMENT LEARNING PROBLEM

Our goal is for the AI to learn to choose actions that maximize rewards.

AGENTS & ENVIRONMENTS



The "agent function" maps percepts to actions; ie $f: P \to A$.

The "agent program" runs on the physical architecture to produce f.

RATIONAL AGENTS

A "rational agent" chooses whichever action that maximizes the expected value of its performance measure given the percept sequence to date.

Note that rationality is not omniscience, but rather learning & autonomy.

PEAS

"PEAS" helps us specify the task environment:

1. Performance measure;
   eg safety, destination, etc
2. Environment;
   eg streets, traffic, etc
3. Actuators; &
   eg steering, brakes, etc.
4. Sensors.
   eg GPS, engine sensors, etc.

PROPERTIES OF TASK ENVIRONMENTS

Task environments can be:

1. fully vs partially observable;
   - fully observable: agent knows state of the world from the stimuli
   - partially observable: agent does not directly observe the world's state
2. deterministic vs stochastic;
   - deterministic: next state is observable at any time
   - stochastic: next state is unpredictable
3. episodic vs sequential;
   - episodic: agent's current action will not affect a future action
   - sequential: agent's current action will affect a future action
4. static vs dynamic;
   - static: model is trained once
   - dynamic: model is trained continuously
5. discrete vs continuous; &
6. single agent vs multiagent.

* the former option is "easier" than the latter.

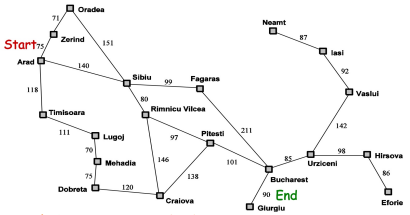
Chapter 2: Uninformed Search Techniques

SIMPLE PROBLEM SOLVING AGENT

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

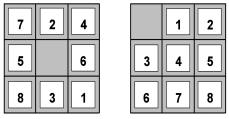
This can only tackle problems that are
1. fully observable;
2. deterministic;
3. sequential;
4. static;
5. discrete; &
6. single agent.

EXAMPLE: TRAVELLING IN ROMANIA



- initial state: In Arad
- actions: drive between cities
- goal test: In Bucharest
- path cost: distance between cities

EXAMPLE: 8-TILE PUZZLE



- states: locations of 8 tiles & blank
- initial state: any state
- actions: up, down, left, right
- goal test: does State match desired configuration
- path cost: # of steps

SEARCHING

We can visualize a state space search in terms of trees or graphs;
1. nodes correspond to states; &
2. edges correspond to taking actions.

These "search trees" are formed using "search nodes", which have
1. the state associated with it;
2. parent node & operator applied to the parent to reach the "current" node;
3. cost of the path so far; &
4. depth of the node.

EXPANDING NODES

"Expanding a node" refers to applying all legal operators to the state contained in the node & generating nodes for all corresponding successor states.

ie

GENERIC SEARCH ALGORITHM

Algorithm:
1. Initialize search with initial problem state.
2. Then repeat:
   - if no candidate nodes can be expanded, return failure
   - otherwise, choose a leaf node for expansion according to our search strategy.
   - if the node contains a goal state, return the solution.
   - otherwise, expand the node by applying the legal operators to the state associated within the node, & add the resulting nodes to the tree.

EVALUATING SEARCH ALGORITHMS

We can use the following properties when evaluating search algorithms:

1. "completeness" — is the algorithm guaranteed to find a solution (if it exists?)
2. "optimality" — does the algorithm find the optimal solution (ie lowest path cost)?
3. time & space complexity.

We consider the following variables:

1. "branching factor" (b) — the # of children each node has
2. depth of shallowest goal node (d); &
3. max length of any path in the state space (m).

BREADTH-FIRST SEARCH

Refer to CS341 notes for details; we expand all nodes on a given level before any node on the next level is expanded.

Evaluating the algorithm:

1. Completeness: yes if $b < \infty$
2. Optimality: yes if all costs are same
3. Time: $1 + b + \cdots + b^d \in O(b^d)$
4. Space: $O(b^d)$.

* all uninformed search methods have exponential time complexity.

UNIFORM COST SEARCH

Idea: we expand the node with the lowest path cost.

We can implement this using a priority queue.

Let C^* = cost of optimal solution & ε = min action cost. Then

1. Completeness: yes if $\varepsilon > 0$
2. Optimality: yes
3. Time: $O(b^{\lceil C^*/\varepsilon \rceil})$
4. Space: $O(b^{\lceil C^*/\varepsilon \rceil})$

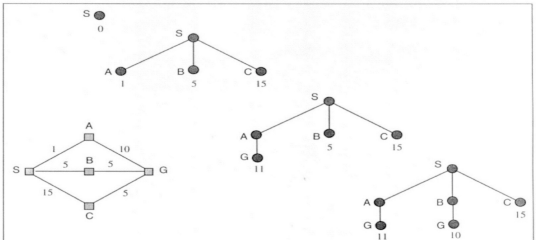


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

DEPTH-FIRST SEARCH

Refer to CS341 notes for details; we expand the deepest node in the current fringe of the search tree first.

Evaluation:

1. Complete: no
   - may get stuck going down a long path
2. Optimal: no
   - might return a solution which is deeper (ie more costly) than another solution
3. Time: $O(b^m)$
   - note we might have $m > d$
4. Space: $O(bm)$

DEPTH-LIMITED SEARCH

Idea: Treat all nodes at depth ℓ as if they have no successors.
- try to choose ℓ based on the problem

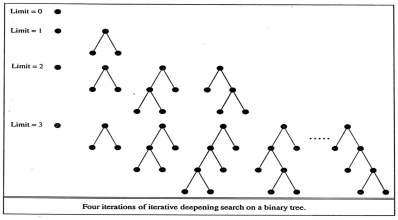
This avoids the problem of unbounded trees.

Evaluation:

1. Time: $O(b^\ell)$
2. Space: $O(b^\ell)$
3. Complete: no
4. Optimal: no

ITERATIVE-DEEPENING

Idea: repeatedly perform depth-limited search, but increase the limit each time.



Four iterations of iterative deepening search on a binary tree.

Evaluation:

1. Complete: yes
2. Optimal: yes
3. Time: $O(b^d)$
4. Space: $O(bd)$

Time:

(limit = 1) 1

(limit = 2) $1 + b$

(limit = 3) $1 + b + b^2$

$\vdots$

(+) (limit = d) $1 + b + b^2 + \cdots + b^d$

$d + (d-1)\,b + (d-2)\,b^2 + \cdots + b^d \in O(b^d)$

Chapter 3: Informed Search Techniques

MOTIVATION

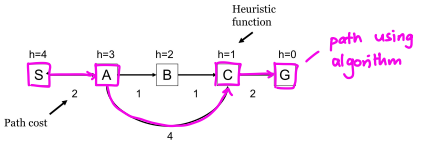
1. In search problems, we often have additional knowledge about the problem.
   - eg with the "travelling around Romania", we know dist. bw cities
     - ↳ so we can find the overhead in going the wrong direction
2. Our knowledge is often about the "merit" of nodes.
3. Notions of merit:
   1. how expensive it is to get from a state to a goal;
   2. how easy it is to get from a state to a goal.

HEURISTIC FUNCTIONS · h(n)

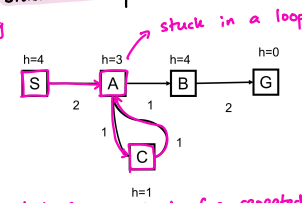
1. We need to develop domain specific "heuristic functions" h(n), which guess the cost of reaching the goal from node n.
2. In general, if $h(n_1) < h(n_2)$, we guess reaching the goal is cheaper from n_1 than from n_2.
3. We also need
   1. $h(n) = 0$ if n is a goal node
   2. $h(n) > 0$ if n is not a goal node.

GREEDY BEST-FIRST SEARCH

1. Idea: Use h(n) to rank the nodes in the fringe & expand the node with the lowest h-value.
   (ie "greedily" trying to find the least-cost solution).
2. Example:



3. Note greedy best-first is not optimal.
   - eg in the above example,
     - path found has cost = 6.
     - but cheaper path is S→A→B→C→G with cost = 6.
4. It is also not complete, as it can be stuck in loops.
   - eg



   - but if we check for repeated states then we are okay
5. This algorithm uses exponential space & worst-case time.

A* SEARCH

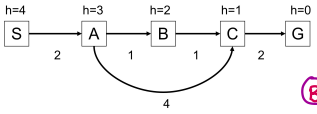
Idea: Define $f(n) = g(n) + h(n)$, where

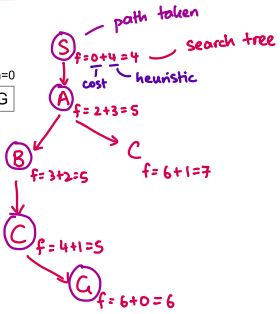
① $g(n)$ = cost of path to node n

② $h(n)$ = heuristic estimate of cost of reaching goal from node n.

We then iteratively expand the node in the fringe with the lowest f value.

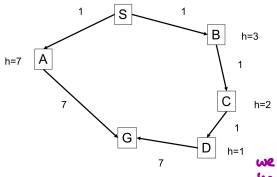
Example:

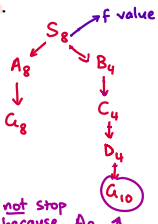




A* should terminate only when the goal state is popped from the queue / fringe.

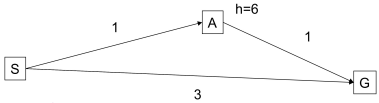
eg





we do not stop here because A_8 is still in the queue.

Note A* is not optimal.



- A* finds S→G with cost 3
- but there is a cheaper path S→A→G with cost 2.
- A* performed poorly because our heuristic was not good.

ADMISSIBLE [HEURISTICS]

Let $h^*(n)$ be the true minimal cost to the goal from node n.

Then, we say the heuristic $h(n)$ is "admissible" if

$$h(n) \leq h^*(n) \quad \forall n.$$

In particular, admissible heuristics never overestimate the cost to the goal.

h(n) IS ADMISSIBLE ⇒ A* IS OPTIMAL

If $h(n)$ is admissible, then A* with tree-search is optimal.

Proof. Let G be an optimal goal state, & $f(G) = f^* = g(G)$.

Let G_2 be a suboptimal goal state, ie $f(G_2) = g(G_2) > f^*$.

(since $h(G_1) = h(G_2) = 0$)

Assume, for a contⁿ, that A* selects G_2 from the queue; ie A* terminates with a suboptimal solution.

Let n = node currently a leaf node on an optimal path to G.

G • • G_2

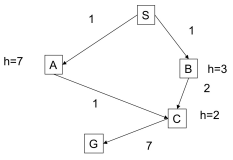
Since h is admissible, $f^* \geq f(n)$.

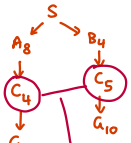
If n is not chosen for expansion over G_2, then $f(n) \geq f(G_2)$.

Thus $f^* \geq f(G_2)$. Since $h(G_2) = 0$, thus $f^* \geq g(G_2)$, a contⁿ.

REVISITING STATES IN A*

Motivation:





* the left path is cheaper than the right!

If we allow states to be expanded again, we might get a better solution!

CONSISTENT [HEURISTICS]

We say $h(n)$ is "consistent" if

$$h(n) \leq \text{cost}(n, n') + h(n') \quad \forall n, n'.$$

Note that A* graph-search with a consistent heuristic is optimal.

PROPERTIES OF A*

Note that A* is

① Complete if $h(n)$ is consistent;
- f always increases along any path

② Has exponential time complexity in the worst-case; &
- but a good heuristic helps a lot
- $O(bm)$ if heuristic is perfect

③ Has exponential space complexity.

ITERATIVE DEEPENING A* (IDA*)

1. Idea: Like iterative deepening search, but change f-cost rather than depth in each iteration.
2. This reduces the space complexity.

SIMPLIFIED MEMORY-BOUNDED A* (SMA*)

1. Idea: Proceeds like A* but when it runs out of memory it drops the worst leaf node (ie one with highest f-value).
2. If all leaf nodes have the same f-value, then drop the oldest & expand the newest.
3. This is
   ① optimal; &
   ② complete if depth of shallowest goal node < memory size.

OBTAINING HEURISTICS

1. One approach to get heuristics is to think of an easier problem & let h(n) be the cost of reaching the goal in the easier problem.
2. We can also
   ① precompute solution costs of subproblems & store them in a pattern database; or
   ② learn from experience with the problem class.

EXAMPLE: 8-PUZZLE GAME

1. We can relax the game in 3 ways:
   ① We can move tile from position A→B if A is next to B (ignore whether position is blank)
   ② We can move tile from position A→B if B is blank (ignore adjacency)
   ③ We can move tile from position A→B regardless.
2. ③ leads to the "misplaced tile heuristic". (h_3)
   - to solve this problem we need to move each tile into its final position.
   - # of moves = # of misplaced tiles
   - admissible
3. ① leads to the "manhattan distance heuristic". (h_1)
   - to solve this we need to slide each tile into its final position
   - admissible
4. Note h_1 "dominates" h_3; ie $h_3(n) \leq h_1(n)$ $\forall n$.

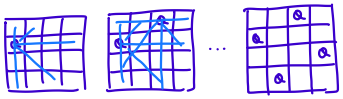
Chapter 4: Constraint Satisfaction

INTRODUCTION

1. These are useful for problems where the state structure is important.
2. In many problems, the same state can be reached independent of the order in which the moves are chosen.
3. So, we can try to solve problems efficiently by being smart about the action order.

4-QUEENS CONSTRAINT PROPAGATION

Idea: Remove conflicting squares from consideration when we put a queen down.



CONSTRAINT SATISFACTION PROBLEM (CSP)

A "constraint satisfaction problem" is defined by some $\{V, D, C\}$, where

1. $V = \{V_1, \dots, V_n\}$ is a set of variables;
2. $D = \{D_1, \dots, D_n\}$ is a set of domains, where D_i is the set of possible values for each V_i; &
3. $C = \{C_1, \dots, C_m\}$ is the set of constraints.

STATE

A "state" is an assignment of values to some or all of the variables;

ie $V_i = x_i$, $V_j = x_j$, etc.

CONSISTENT [ASSIGNMENT]

We say an assignment is "consistent" if it does not violate any constraints.

SOLUTION

A "solution" is a complete, consistent assignment.

EXAMPLE: 8 QUEENS AS A CSP

8 queens as a CSP:

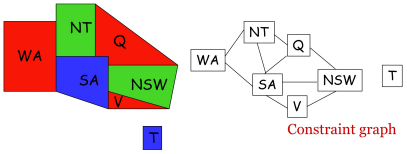
- variables: V_{ij}, $i,j = 1, \dots, 8$
- domain of each var: $\{0,1\}$
- constraints: $V_{ij} = 1 \Rightarrow V_{ik} = 0 \quad \forall k \neq j$

  $V_{ij} = 1 \Rightarrow V_{kj} = 0 \quad \forall k \neq i$

  similar constraint for diagonals

  $\sum_{i,j} V_{ij} = 8$

EXAMPLE: MAP COLORING



- variables: WA, NT, ..., T (the regions)
- each var has the same domain: {red, green, blue}
- no 2 adjacent variables have the same value

  (ie $WA \neq NT$, $WA \neq SA$, etc)

PROPERTIES OF CSPs

Types of variables:

① Discrete & finite;
eg 8-queens, map coloring
* we focus on this in this course.

② Discrete with infinite domains; &
eg job scheduling

③ Continuous domains.

Types of constraints:

① "Unary constraint": relates a single variable to a value
- eg Queensland = blue

② "Binary constraint": relates two variables

③ "Higher order constraints": relates ≥3 variables.

CSPs & SEARCH

We can formulate CSPs as a search problem:

① we have N variables $V_1, \dots, V_n$;

② a valid assignment is $\{V_1 = x_1, \dots, V_k = x_k\}$, $0 \le k \le n$ where values satisfy the variable constraints.

③ states: valid assignments

④ initial state: empty assignment

⑤ successor: $\{V_1 = x_1, \dots, V_k = x_k\} \rightarrow \{V_1 = x_1, \dots, V_k = x_k, V_{k+1} = x_{k+1}\}$

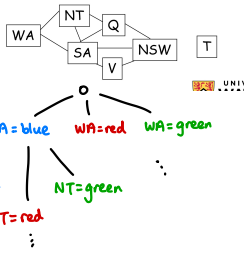
⑥ goal test: complete assignment

BACKTRACKING SEARCH

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or 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 according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

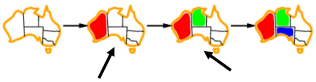
- this is DFS that choose values for one variable at a time
- we "backtrack" when a variable has no legal values to assign

EXAMPLE: MAP COLORING



MOST CONSTRAINED VARIABLE HEURISTIC

Idea: Choose the variable which has the fewest "legal" moves.



$D_{NT} = \{green, blue\}$ $D_{SA} = \{blue\}$
$D_{SA} = \{green, blue\}$ $D_{Q} = \{blue, red\}$
$D_{others} = \{red, green, blue\}$ $D_{others} = \{red, green, blue\}$

In a tie, choose the variable with the most constraints on the remaining variables.
(ie "most constraining variable").

LEAST CONSTRAINING VALUE HEURISTIC

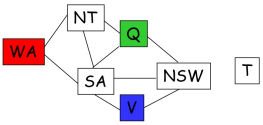
Idea: Given a variable, choose the "least constraining value", ie the one that rules out the fewest values in the remaining variables.

FORWARD CHECKING

Idea_1: We keep track of remaining legal values for unassigned variables, & terminate search when any variable has no legal values.

This_2 helps us detect failure early.

EXAMPLE: MAP COLORING



	WA	NT	Q	NSW	V	SA	T
	RGB	RGB	RGB	RGB	RGB	RGB	RGB
WA = R	R	GB	RGB	RGB	RGB	GB	RGB
Q = G	R	B	G	RB	RGB	B	RGB
V = B	R	B	G	~~RB~~	B	~~B~~	RGB

this is the empty set;
⇒ the current assignment does not lead to a solution.

Chapter 5: Uncertainty

1. Refer to STAT 231/330 for more details.
2. We use "~" to denote the complement of an event (ie ~A).

BAYES RULE

1. For 2 events A, B, note

$$P(B|A) = \frac{P(A|B)\,P(B)}{P(A)}$$

Proof. $P(A)P(B|A) = P(A \wedge B) = P(B)P(A|B)$.

$\therefore P(B|A) = \frac{P(B)P(A|B)}{P(A)}$. ∎

2. In particular, it allows us to compute a belief about hypothesis B given evidence A.
3. More general forms:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B|A)P(A) + P(B|\sim A)P(\sim A)}$$

$$P(A|B \wedge X) = \frac{P(B|A \wedge X)\,P(A|X)}{P(B|X)}$$

$$P(A = v_i|B) = \frac{P(B|A=v_i)\,P(A=v_i)}{\sum_{k=1}^{n} P(B|A=v_k)\,P(A=v_k)}$$

PROBABILISTIC INFERENCE

1. Idea: Given a prior distribution P(X) over variables X of interest & given new evidence E=e for some variable E, revise our degrees of belief: ie the "posterior" P(X|E=e).

ISSUES

1. Specifying the full joint distribution for $X_1, \ldots, X_n$ requires an exponential number of possible "worlds".
2. So, inference is also slow since we need to sum over these exponential number of worlds.

$$P(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n | X_i) = \frac{P(X_1, \ldots, X_n)}{\sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_n} P(X_1, \ldots, X_n)}$$

CONDITIONAL INDEPENDENCE

1. Two variables X, Y are "conditionally independent" given Z if

$$P(X=x \mid Z=z) = P(X=x \mid Y=y, Z=z)$$
$$\Leftrightarrow P(X=x, Y=y \mid Z=z) = P(X=x|Z=z)\,P(Y=y|Z=z)$$
$$\Leftrightarrow \forall x \in \text{dom}(X),\ y \in \text{dom}(Y),\ z \in \text{dom}(Z)$$

2. If we know the value of Z, nothing we learn about Y will influence our beliefs about X.

VALUE OF INDEPENDENCE

1. If $X_1, \ldots, X_n$ are mutually independent, then we can specify the full joint distribution using only n parameters (ie linear) instead of $2^n - 1$ (ie exponential).
2. Although most domains do not exhibit complete mutual independence, they do instead exhibit a fair amount of conditional independence.

NOTATION: P(X)

1. We define "P(X)" as the marginal distribution over X.
   - P(X=x) is a number, P(X) is a distribution.

NOTATION: P(X|Y)

1. We define "P(X|Y)" as the family of conditional distributions over X; one for each $y \in \text{dom}(Y)$.

EXPLOITING CONDITIONAL INDEPENDENCE : CHAIN RULE

Consider a story:

- If Pascal woke up too early E, Pascal probably needs coffee C; if Pascal needs coffee, he's likely grumpy G. If he is grumpy then it's possible that the lecture won't go smoothly L. If the lecture does not go smoothly then the students will likely be sad S.

E - Pascal woke up too early G - Pascal is grumpy S - Students are sad
C - Pascal needs coffee L- The lecture did not go smoothly

S is independent of E, C, G given L
L is independent of E, C given G & so on.

$\Rightarrow P(S \mid L, G, C, E) = P(S \mid L)$
$P(L \mid G, C, E) = P(L \mid G)$
$P(G \mid C, E) = P(G \mid C)$

Then

$$P(S, L, G, C, E) = P(S \mid L, G, C, E)\, P(L \mid G, C, E)\, P(G \mid C, E) \cdot P(C \mid E)\, P(E)$$
$$= P(S \mid L)\, P(L \mid G)\, P(G \mid C)\, P(C \mid E)\, P(E).$$

In this, we can specify the full joint distribution by specifying the five local conditional distributions.