

CS3243 CheatSheet

by Zachary Chua

Agents and Environment

Agent Components

Sensors and Actuators

- Sensors: information about the env

- Percept data at timestep t, p_t

- Percept History, $P = p_1, \dots, p_t$

- Actuators: how agent interacts with environment

- set of Actions A

Agent function $f : P \rightarrow a$, where a is a selected action given P

Agent is rational if optimises performance measure (implies quantifiable objective), given

- percept seq

- prior knowledge

- set of actions

- performance measure

Note: do not assume agent is omniscient.

Types of agents

1. Reflex Agents

- uses if-else to make decisions, direct mapping of percepts to actions
- domain specific, impractical for large search space

2. Model-based reflex agent

- makes decisions based on an internalised model
- eg logical agents / bayesian networks

3. Goal Utility-based agents

- Given state, actions and goals / utility
- Determine seq of actions to reach goal / max utility
- uninformed / informed search, Local Search, CSP, Adversarial Search

AI as Graph Search

- Each percept corresponds to a state in the problem (state \rightarrow vertex)

- Define desired (goal) states

- After action, arrive at new state (actions are edges)

- Search space is graph

Problem Env

Fully observable	Partially observable	Whether can sense all information
Deterministic	Stochastic	Whether intermediate state can be determined based on action and state
Episodic	Sequential	episodic: action impacts current state, sequential: actions impact future states
Discrete	Continuous	for state information
Single	Multi agent	other entities that influence agent behaviour, competing or cooperating
Known	Unknown	Knowledge of agent (knowing where is goal state)
Static	Dynamic	If env will change while agent is deciding action

Path Planning

Properties

Environment

- Fully observable
- Deterministic
- Discrete
- Episodic (plan solution, not execute, but plan is formed sequentially)

Search Space

State: representation of instance of environment

Node: Element in frontier representing current path traversed

- State
- Parent Node
- Action
- Path Cost
- Depth

Goal Test function

Actions function

Action costs function (≥ 0)

Transition Model function: takes in a state, applies action, returns new state

Algo Criteria

Efficiency - space and time

Complete - find a solution when one exists and correctly report failure when it does not

Optimal - solution with **lowest** path cost among all solutions

Types of search

Tree Search

- No restrictions on revisiting states, allows redundant paths, including cycles
- Incomplete, if there are cycles

Performance under tree-search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes ¹	Yes ^{1,2}	No	No	Yes ¹
Optimal Cost?	Yes ³	Yes	No	No	Yes ³
Time	$O(b^d)$	$O(b^{1+ C^*/\epsilon })$	$O(b^m)$	$O(b^d)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+ C^*/\epsilon })$	$O(bm)$	$O(b^d)$	$O(bd)$

1. Complete if b finite and state space either finite or has a solution

2. Complete if all actions costs are $> \epsilon > 0$

3. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test on BFS may improve runtime practically

- UCS must perform a Late Goal Test to be optimal (this also accounts for the +1 in the index of its complexity)

- DFS is not complete (even under 1) as it might get caught in a cycle

- DFS space complexity may be improved to $O(m)$ with backtracking (similar for DLS and IDS)

Graph Search

- Maintain reached hash table
- Add nodes to reached hash table on push
- Only add new node to frontier (and reached) if
 1. state represented has not been reached before
 2. path to state already reached is cheaper than the stored one

Performance under graph-search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes ¹	Yes ^{1,2}	Yes ¹	No	Yes ¹
Optimal Cost?	Yes ³	Yes	No	No	Yes ³
Time					$O(V + E)$
Space					

1. Complete if b finite and state space either finite or has a solution

2. Complete if all actions costs are $> \epsilon > 0$

3. Cost optimal if action costs are all identical (and several other cases)

- DFS under graph search is complete, assuming a finite state space

- Time and space complexities are now bounded by the size of the state space

- i.e., the number of vertices and edges, $|V| + |E|$

- Note that we **do not** need to allow cheaper paths under graph-search for BFS and DFS since costs play no part in algorithm and they cannot guarantee an optimal solution anyway

Limited Graph Search V1

- Uses reached hash table

- Adds to reached on push to frontier

- Only pushes to frontier when not in reached

Limited Graph Search V2

- Similar to V1

- Adds notes to reached on pop from frontier

- Does not check if path is cheaper than stored one

- Not optimal for UCS or A*

Uninformed Search

BFS - Queue

Time: $O(b^d)$

Space: $O(b^d)$

Complete: Yes if finite search space or contains solution

Optimal: No unless costs uniform

b: branching factor

d: depth of shallowest goal

Can be optimised with **early goal test** (goal test on push to frontier instead of pop)

UCS aka Dijkstra - PQ, path cost

Time: $O(b^e)$

Space: $O(b^e)$

Complete: Yes if finite search space or contains solution

Optimal: Yes, tree, graph and limited V2

e: $1 + \lfloor \frac{C^*}{\epsilon} \rfloor$, where C^* is the optimal path cost

ϵ : small positive constant, smallest action cost

DFS - Stack

Time: $O(b^m)$

Space: $O(bm)$

Complete: No, unless finite search space

Optimal: No

m: max depth

Note: space can be improved to $O(m)$ by using backtracking (assume fixed order of actions)

Depth Limited Search (DLS)

DFS with depth limit l

Time: $O(b^l)$

Space: $O(bl)$

Complete / Optimal: No

Iterative Deepening Search (IDS)

Use DLS recursively, each time increasing l by 1

- Completeness of BFS with space complexity of DFS

Overhead: rerun top levels many times ($\sim 11\%$)

- Nodes generated by DLS: $O(b^0 + b^1 + \dots + b^l)$

- Nodes generated by IDS: $O((d+1)b^0 + db^1 + \dots + b^d)$

Time: $O(b^d)$

Space: $O(bd)$

Complete: same as BFS

Optimal: same as BFS

Informed Search

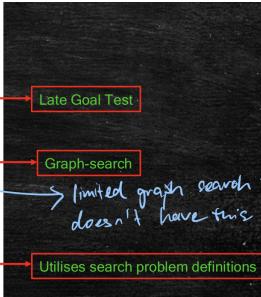
Use domain knowledge to estimate cost from s to G using heuristic function h

Define **evaluation function** f to be used as priority function in PQ

UCS: $f = g$

```
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
        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
                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
        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)
```



Greedy Best First Search

Explore state that you estimate is closest to a goal

Evaluation fn: $f(n) = h(n)$

Optimal: **not optimal** as does not take into consideration cost of path travelled (one node very close to goal but high cost to get to adjacent to start node)

Tree Search: **Incomplete**, stuck between 2 nodes with lowest h values

Graph Search: **Complete**, if search space is finite

A* Search

Evaluation fn: $f(n) = g(n) + h(n)$

Optimality:

Admissible heuristic: Tree Search, Graph Search

Consistent heuristic: Tree, Graph, Limited Graph V2

A* and Graph Search **costly** due to updates (need update descendants)

Heuristics

Admissible Heuristics

$\forall n, h(n) \leq h^*(n)$

- $h(n)$ never overestimates the cost

- $h(n) = 0$ at goal

Consistent Heuristics

$\forall n, n', h(n) \leq cost(n, a, n') + h(n')$

- f costs monotonically increasing along a path

Dominance

if $\forall n, h_1(n) \geq h_2(n)$, then h_1 dominates h_2

- if h_1 is also admissible: h_1 closer to h^* , h_1 more efficient than h_2

Note: for 3243, h_1 and h_2 need not be admissible

Creating heuristics

- Want **efficient** h , close to $O(1)$

- Want **admissible**, consistent harder

Relaxing the problem

- relaxing constraints gives admissible heuristics

- relaxing more constraints creates **less** dominant heuristics

Top Down Approach

- What dependent variable do I want to model / approx

- What are the independent variables that help to calculate these

Bottom up Approach

- What variables can you efficiently calculate?

- What can these variables model?

Local Search Problems

- Only have goal test, but not values in goal state

- Want goal state values, not interested in path

- Greedy **not** systematic

- Space Complexity: $O(b)$, only store node and successors

State formulation

- All states have all components of solution

- no partially filled states

- Each state is a potential solution

- "guess" a solution

- "check" its value

- make "systematic guess" by moving to states of higher value (higher value closer to goal)

Steepest Ascent Hill Climbing Algorithm

```
current = initial state
while true:
    neighbour = highest-valued successor of current
    if value(neighbour) ≤ value(current) return current
    current = neighbour
```

- Only store current state

- On each iteration, find successor that **improves** on current state

- Requires **actions** and **transitions** to determine successor

- Requires **value**: a way to assign each state a value ($f(n) = -h(n)$)

- If none exists, return current state as the best option

- **Can fail**, may return non-goal state

- may fail at **local maxima**, **shoulder**, **plateau**, **ridge**

Variants

- Sideways move: replace \leq with $<$

- **Can traverse shoulder**

- Stochastic hill climbing: chooses randomly among states with values better than current

- try to prevent getting stuck at local maxima

- First Choice: handles **high b** by randomly generating successors and choosing first one that is better

- Similar to stochastic, handles high b

- Random restart: add outer loop which randomly picks new start state until solution found

Guaranteed to find solution

Analysis

let p_1 be probability of success, n_1 be number of steps of expected solution, n_2 be no. of steps of expected failure

$$\text{Expected Computation} = n_1 + \frac{1-p_1}{p_1} * n_2$$

Local Beam Search

Algo:

1. Begin with k random starts
2. Each iteration generate successors for **all** k states

3. Repeat with best k among **all** successors unless goal found

Better than k parallel random restarts → best k among **all** successors, not best from each set of successors, k times

Stochastic Beam Search

Similar to stochastic hill climbing

Constraint Satisfaction Problem

Systematic search unlike LSP greedy search

Formulating CSP

State Representation:

- Variables: $X = x_1, x_2, \dots, x_n$

- Domains: $D = d_1, d_2, \dots, d_n$

- such that x_i has domain d_i

- Initial State: all variables unassigned

- Intermediate State: partial assignment

Goal Test

- Constraints: $C = c_1, c_2, \dots, c_m$

- Each c_i describes necessary relationship rel , between set of variables $scope$

- eg. $scope = (x_1, x_2)$ and $rel = x_1 > x_2$

- constraints can be **unary**, **binary**, **global**

Actions and Transitions

- costs and evaluation function not utilised

Objective: find **complete** and **consistent** assignment

- complete: all variables assigned

- consistent: all constraints in C satisfied

CSP search algo

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})
```

```
function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
```

```
var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
```

```
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
```

```
            add inferences to csp
            result ← BACKTRACK(csp, assignment)
            if result ≠ failure then return result
            remove inferences from csp
            remove {var = value} from assignment
```

```
return failure
```

We will look into making these choices in the next lecture

- Use DFS as solutions are **all at** max depth

- Each level assign to only **one variable**, **not all** because order does not matter (prunes tree)

- Backtrack when there are no legal assignments

