CS3243 Cheatsheet

Ng Xuan Jun

Miscellaneous:

Arithmetic Series: If $a_n = a_{n-1} + c$, where c is a constant

$$\sum_{i=1}^{n} a_i = a_1 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

Geometric Series: If $a_n = ca_{n-1}$, where $c \neq 1$ is a constant

$$\sum_{i=0}^{n} a_i = a_1 + \cdots + a_n = a_1\left(\frac{c^n - 1}{c - 1}\right)$$

If $0 < c < 1$, then the sum of the infinite geometric series is

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c}$$

Environment Properties:

Property	Description
Fully Observable vs Partially Observable	(***States***) Concerned with whether the agent has full knowledge of the search space. Requires handling of uncertainty
Deterministic vs Stochastic	(***Action and State***) Whether given an initial state and action, we can determine the intermediate state (i.e. if we start from the same place and take the same action, will we get to different results every time we run the program) Note that stochastic may still be fully observable (sense all) but there is randomness with the actions (e.g. Monopoly)
Episodic vs Sequential	(***Action***) Whether the previous moves affects the next moves or not Note that we can model an episodic environment into a sequential search space
Discrete vs Continuous	(***Action/State***)Refers to state information, time, percepts, actions Most things in life are continuous and we will try to make them discrete
Single vs Multi-Agent	(***Action***)Do other entities exist within the environment that are themselves agents whose actions directly influence the performance of this agent
Known vs Unknown	(***Problem Knowledge/Rules***)Refers to the knowledge of the agent/designer about the problem Includes performance measure
Static vs Dynamic	(***Environment/State***)Does the environment change while the agent is deciding on an action

Taxonomy of Agents:

Agent Type	Description
Reflex Agent	Uses rules in the form of if-else statements to make decisions
Model-based reflex agent	Internalised model for rules to make decisions.
Goal-based agent	***Given***: State and action representation + definition of goal ***Determines***: (1) Sequence of action to reach goal OR (2) Final state that satisfies goal
Utility-based agent	***Given***: State and action representation + definition of utility ***Determines***: (1) Sequence of action to maximise utility OR (2) Final state that maximises utility
Learning agents	Agents that learn how to optimise performance

Search Problems:

Path Planning: Path to a goal is necessary and path cost is important

Properties:

Note that if we need to formulate a search problem, we should try to ensure these properties are met

- Fully observable
- Deterministic
- Discrete
- Episodic

This means that we have **complete information**, it is **fully deterministic** and we are able to **PLAN**, i.e. can look ahead at what to do and execute the plan once we have defined it

Uninformed Search:

Correctness Definition:

- ***Completeness***: An algorithm is **complete** if it will find a solution when one exists and correctly report failure when it does not
- ***Optimality***: An algorithm is **optimal** if it finds a solution with the lowest path cost among all solutions (i.e. path cost optimal)

Nodes: We can store the following information in our nodes to improve performance

- ***State***: Each node represents one state
- ***Parent node and action***: Useful for DFS because we want to perform backtracking and assuming a static sequence over the possible actions so that the space complexity can be shrunk to $O(m)$
- ***Depth***: Useful for DLS and IDS to know if the depth limit has reached
- ***Path cost***: To efficiently update the path cost when extending upon the current path

Algorithm	Breadth-First Search (BFS)	Depth-First Search (DFS)	Uniform-Cost Search (UCS)	Depth-Limited Search (DLS)	Iterative Deepening Search (IDS)
Data Structure (Frontier)	Queue (FIFO)	Stack (LIFO)	Priority Queue **Priority**: Path Cost	Stack	Stack
Early / Late Goal Test	Early Goal Test	Late Goal Test	Late Goal Test		
Complete?	Yes[1]	No[3]	Yes[1, 2]	No	Yes[1]
Optimal?	Yes[4]	No	Yes	No	Yes[4]
Time (Tree)	$O(b^d)$	$O(b^m)$	$O\left(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor}\right)$	$O(b^\ell)$	$O(b^d)$
Space (Tree)	$O(b^d)$	$O(bm)$	$O\left(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor}\right)$	$O(b\ell)$	$O(bd)$
Time (Graph)	$O(\lvert V\rvert + \lvert E\rvert)$				
Space (Graph)					
Notes	Early Goal Test is carried out to save on the computational time so we don't have to branch out the final layer which will be b^{d+1} nodes If we want BFS to be optimal, we need the sequence of nodes explored to be monotonically increasing in terms of action cost. Not concerned with path cost	Not complete under (1) as even if the solution exists, it may infinitely traverse down a path without a solution (Needs both so b and m to be finite) DFS space complexity can be improved to O(m) with backtracking Take note that when we pop out it is in reverse.	Late Goal Test is required for UCS to be optimal. Because we can only know that a state is optimal after popping it out since still needs to be pushed into the Priority Queue	Variation of DFS with max depth Space complexity can be improved to $O(\ell)$ with backtracking	Iterative version of DLS Space complexity can be improved to $O(d)$ with backtracking

1. Complete if b finite and either has a solution **or** m finite
2. Complete if all action costs are $> \epsilon > 0$
3. DFS is incomplete unless the search space is finite – i.e., when b is finite and m is finite
4. Cost optimal if action costs are all identical

Proof that UCS is optimal so long as each action cost exceeds some small positive constant ϵ

- Whenever UCS expands a node n, the optimal path to that node has been found. If this was not the case (i.e. the path to n was not optimal – lets us denote this path T), there would have to be another frontier node n' on the optimal path from the start node to n (denote this optimal path U). By definition, $g(n)$ via U would be less than $g(n)$ via T which implies that n' should have been selected first.
- If action costs are non-negative path costs, then g values would never get smaller with more nodes being added. Therefore, UCS expands nodes in the order of the optimal path cost.

Tree Search Algorithm:

- Can revisit nodes

Algorithm:

```
frontier = {Node(initial state, NULL)}
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)
                frontier.push(successor)
return failure
```

Graph Search Algorithm:

- Does not allow for revisiting of states (unless there is some specified condition like Version 2)

Typical practice:

- Maintain a reached (or visited) ***Hash Table***
- Add reached states
- Only add new nodes to frontier and reached if: State represented by node not previously reached

Algorithm:

```
frontier = {Node(initial state, NULL)}
reached = {}
while frontier not empty:
        current = frontier.pop()
        reached.insert(current)
        if isGoal(current.state): return current.getPath() (Late Goal Test)
        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
```

Assume that Version 1 is used unless stated otherwise

Version 1: Ensures that nodes are never revisited (Omits all redundant paths and may omit optimal path)

Version 2: More relaxed constraint on paths, also considers paths with lower path cost (Includes **purple** part of the algorithm). Non redundant paths are never skipped

Version 3: Only adds a node to reached when it is popped. (Add in the **blue** part and omit the **green** part. For Version 1 and 2, include the **green** and omit the **blue**)

Informed Search:

- Make use of domain knowledge to estimate the cost to the goal
- Make use of a heuristic function, $h(n)$ to estimate the cost of going from the current node to the nearest goal state. However, it should be an efficient function to compute, time complexity should not be high

Heuristic Theorems:

Admissible: $h(n)$ is admissible if $\forall n, h(n) \leq h^*(n)$

- $h(n)$ never overestimates the true cost
- By the time we visit a path to a goal, P, all paths with actual costs less than P must be searched
- Paths not ending at a goal are never over-estimated. (At non-goal, n, $f(n) = g(n) + h(n) \leq g(n) + h^*(n)$)
- Paths ending at a goal are exact, $h(g) = 0$. Only when we pop off the goal node can we be sure that the optimal path to the goal node is found. If we pop off any other node, we cannot say that it is the optimal path to that node since there is no ordering

Consistent: $h(n)$ is consistent fi $\forall n$, and successor $n, n', h(n) \leq cost(n, a, n') + h(n')$

- Ensures that f costs are monotonically increasing along a path
- Note that for this, all paths are in increasing order and once we get to a state, we must have the optimal path to that node.
- Consistency ⇒ Admissibility

Algorithm	Uniform-Cost Search (UCS)	Greedy Best-First Search	A* Search
Data Structure (Frontier)	Priority Queue **Priority**: Evaluation Function	Priority Queue **Priority**: Evaluation Function	Priority Queue **Priority**: Evaluation Function
Early / Late Goal Test	Late Goal Test	Late Goal Test	Late Goal Test
Evaluation Function	$f(n) = g(n)$	$f(n) = h(n)$	$f(n) = h(n) + g(n)$
Complete under Tree?	Yes[1,2]	No	Yes[1, 2]
Optimal under Tree?	Yes	No	Yes[3]
Complete under Graph?	Yes[1, 2]	Yes[1]	Yes[1, 2]
Optimal under Graph?	Yes (Version 2, 3)	No	Yes[3] (Version 2) Yes[4] (Version 2 & 3)
Notes	It is the worst case if we are considering admissible heuristic. Because we are just taking $h(n) = 0$	It does not exploit the information of cost of path already taken. Might get us to a solution faster but may not be the optimal one.	**Theorem**: If $h(n)$ is admissible, tree search is optimal **Theorem**: If $h(n)$ is consistent, graph search is optimal

1. Complete if b finite and either has a solution or m finite
2. Complete if all action costs are $> \epsilon > 0$
3. If the heuristic $h(n)$ is admissible
4. If the heuristic $h(n)$ is consistent

Algorithm:

```
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
                                    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')
                        yields NODE(State=s',Parent=node,Action=action, Path-Cost=cost)
```

Proof that Tree Search Implementation for Greedy Best-First Search is incomplete

- Just need to make sure that we have 2 nodes with the same $h(n)$ and we can travel bidirectional

Proof that Graph Search Implementation for Greedy Best-First Search is complete

- Assuming that we have a finite search space, a graph search will not revisit states and therefore, would visit all states within the state space. Therefore, it will either report a solution or failure

Example that Tree/Graph Search Greedy Best-First Search is not optimal:

- Idea is just to give estimation for states with higher cost to be lower than that of those with lower cost and we will be able to get a non-optimal solution

Proof that tree search for A* Search is optimal when Admissible Heuristic is utilised:

- Let s be the initial state, n be an intermediate state along the optimal path, t be a suboptimal goal state and t^* be the goal along the optimal path
- Optimal solution means that n must be expanded before t since we need to travel to n first before t
- ***Proof by Contradiction***:
  - Assume that a suboptimal solution is found (i.e. t is expanded before n which means $f(t) \leq f(n)$)
  - Assuming that both t and n are in the frontier, we need $f(t) < f(n)$ for it to be popped off first
  - However, since t is not on the optimal path but t^* is then we must have: $f(t) > f(t^*)$
  - $f(t) > g(t^*)$ since $h(t^*) = 0$ for goal node $\Rightarrow f(t) > g(n) + p(n, t^*)$ where $p(n, t^*)$ is cost of going from n to $t^* \Rightarrow f(t) > g(n) + h(n)$ by Admissibility $\Rightarrow f(t) > f(n)$ which is a contradiction
  - Note that we do not consider $f(t) = f(n)$ since this would mean that $f(t)$ is equally optimal

Proof that graph search (V3) for A* Search is optimal when Consistent Heuristic is utilised:

- ***Main Idea***: We want to show that when a node is popped from the frontier, we have found the optimal path to it
- Let $f(s_k) = g(s_k) + h(s_k)$ be the minimum f value for s_k we have observed when s_k is popped off
- Let the optimal path from the start node s_0 to any node s_g, be $P = s_0, s_1, \cdots, s_g$. We will show that when we pop s_g, $f(s_g) = g(s_g) + h(s_g) = g^*(s_g) + h(s_g)$, where g^* is the optimal cost going from initial state to s_g via P
- ***Proof by Induction***:
  - **Base Case**: Start node, $f(s_0) = g(s_0) + h(s_0) = g^*(s_0) + h(s_0) = h(s_0)$
  - **Induction Hypothesis**: Assume that for all s_0 to s_k, when we pop off s_i, $g(s_i) = g^*(s_i)$
  - **Induction Step**:
    Since $g^*(s_{k+1})$ is the minimum path cost from s_0 to s_{k+1}, we have that for any path going to s_{k+1}
    $$g(s_{k+1}) + h(s_{k+1}) \geq g^*(s_{k+1}) + h(s_{k+1}) \Rightarrow g(s_{k+1}) \geq g^*(s_{k+1})$$
    To make sure that each s_{k+1} is popped off after we pop s_k, the condition $f(s_k) \leq f(s_{k+1})$ needs to be meant and we also have $f(s_{k+1}) = g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1})$
    $$\Rightarrow g(s_k) + h(s_k) \leq g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \Rightarrow h(s_k) \leq c(s_k, s_{k+1}) + h(s_{k+1})$$
    Note that $h(s_k) \leq c(s_k, s_{k+1}) + h(s_{k+1})$ is our consistent heuristic definition
    When we just pop off s_k, from consistency
    $$g(s_{k+1}) + h(s_{k+1}) = \min\{g(s_{k+1}) + h(s_{k+1}), g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1})\}$$
    $$\Rightarrow g(s_{k+1}) = \min\{g(s_{k+1}), g(s_k) + c(s_k, s_{k+1})\} \leq g(s_k) + c(s_k, s_{k+1})$$
    $$= g^*(s_k) + c(s_k, s_{k+1}) \ (By\ Inductive\ Hyp) = g^*(s_k, s_{k+1})$$
    Note that because we are using V3, only nodes that are popped off the frontier are added to reached and therefore, the optimal path is not excluded.

Proof that if h is consistent, it is also admissible given $h(t) = 0$ when t is a goal node

- ***Main Idea***: Proof by Induction on $k(n)$ which is the number of actions required to reach the goal from a node n to the goal node t
- ***Proof by Induction***:
  - **Base Case**: ($k = 1$) The node is one step from t.
    Since the heuristic is consistent, $h(n) \leq c(n, a, t) + h(t)$ and since $h(t) = 0$, we have $h(n) \leq c(n, a, t) = h^*(n)$ (Therefore, h is admissible)
  - **Induction Hypothesis**: Suppose the assumption holds for every node that is $k - 1$ actions away from t, where the least-actions optimal path from n to t has $k > 1$ steps
  - **Induction Step**: If we look at node n which is k actions away and considering its optimal path
    Optimal path from n to t is: $n \rightarrow n_1 \rightarrow \cdots \rightarrow n_{k-1} \rightarrow t$
    Since h is consistent: $h(n) \leq c(n, a, n_1) + h(n_1)$
    Since n_1 is on the least-cost path from n to t, we must have the path $n_1 \rightarrow \cdots \rightarrow n_{k-1} \rightarrow t$ is a minimal cost path from n_1 to t as well. By the induction hypothesis, $h(n_1) \leq h^*(n_1)$
    $$\Rightarrow h(n) \leq c(n, a, n_1) + h^*(n_1)$$
    Note that $h^*(n_1)$ is the cost of the optimal path from n_1 to t; we have that the cost of the optimal path from n to t, $h^*(n) = c(n, a, n_1) + h^*(n_1)$. Therefore, the heuristic is admissible

Admissibility under Max/Min

Cases	Admissible/Inadmissible/Indeterminate
max(Admissible, Admissible)	Admissible
max(Admissible, Inadmissible)	Inadmissible
max(Inadmissible, Inadmissible)	Inadmissible
min(Admissible, Admissible)	Admissible
min(Admissible, Inadmissible)	Admissible
min(Inadmissible, Inadmissible)	Indeterminate

Efficiency of Heuristic:

Dominance: If $h_1(n) \geq h_2(n)$ for all n, then h_1 dominates h_2

- If h_1 is also admissible, (also means that h_2 is admissible). Then it means that h_1 is closer to h^* and therefore it should be more efficient.
- h_1 will be more efficient because the set of paths that it needs to check is lesser

Effective Branching Factor:

- Makes use of empirical results of: N nodes explored and solution path is at depth d
- We try to approximate the amount of times we need to branch out given the number of nodes explored and the depth of the solution.
- The branching factor allows us to know an approximate number of nodes that is expanded by the algorithm as if we branch out lesser, it means that we have more efficient algorithm. Because if they branch out lesser, it means that even if we go deeper, the number of states to search is lesser.

Solve for b^* using the solve:

$$N + 1 = (b^*)^0 + (b^*)^1 + \cdots + (b^*)^d$$
$$N + 1 = \frac{(b^*)^{d+1} - 1}{(b^*) - 1}$$

Proof that a dominant heuristic improves efficiency:

Admissibility (Assumed under dominance):

- All paths with approximated path cost ≤ optimal path cost are explored

Let the optimal path be $s_0, s_1, \cdots, s_k$ (s_k is a goal node)
Assume that we have 2 heuristics whereby $h_1(n) \geq h_2(n)\ \forall n$ (Which means h_1 dominates h_2)

By admissibility and dominance:

$$f_{h_2}(n) = h_2(n) + g(n) \leq f_{h_1}(n) = h_1(n) + g(n) \leq h^*(n) + g(n)$$

By admissibility: If s_k is the goal node (Since $h(s_k) = 0$ for admissible heuristic)

$$f_{h_2}(s_k) = h_2(s_k) + g(s_k) = f_{h_1}(s_k) = h_1(s_k) + g(s_k) \leq h^*(s_k) + g(s_k)$$

Also, all paths to n such that $f(n) < f(s_k) = g(s_k)$ will be expanded first since it is admissible and no paths that has larger cost than the optimal path to the goal state will be expanded first.

- X denotes the set of paths with cost below $h^*(s)$ based on h_1. i.e. $f_{h_1}(n) \leq g(s_k)$
- Y denotes the set of paths with cost below $h^*(s)$ based on h_2 i.e. $f_{h_2}(n) \leq g(s_k)$
- $h^*(s)$ denotes the optimal path cost of going from the initial state s to the nearest goal node

If we consider $X \backslash Y = \{n_i | f_{h_1}(n_i) \leq g(s_k) < f_{h_2}(n_i)\}$ but since h_1 dominates h_2, we know that $f_{h_2}(n_i) \leq f_{h_1}(n_i)$ and therefore, $X \backslash Y = \emptyset$
If we consider $Y \backslash X = \{n_i | f_{h_2}(n_i) \leq g(s_k) < f_{h_1}(n_i)\} \neq \emptyset$ (possible, since $f_{h_2}(n_i) = h_2(n) + g(n) < f_{h_1}(n) = h_1(n) + g(n)$ may still hold given that h_1 dominates h_2
Therefore, $|X| < |Y|$. This means that the effective branching factor using h_1 must be less than the effective branching factor using h_2 since effective branching factor is calculated by solving $b^0 + \cdots + b^d = |N| + 1$

Alternative: Another way is to consider a node n that is not the optimal path ($g(n) + h^*(n)) > h^*(s)$
Possible that $h_2(n) \leq h^*(s) \leq h^*(n)$ so $n \in Y$
Also consequently possible that $h^*(s) < h_1(n) \leq h^*(n)$ so $n \notin X$
However, the opposite is not possible $h_1(n) \leq h^*(s) \leq h^*(n)$ and $h^*(s) < h_2(n) \leq h^*(n)$ which means that $h_1(n) \leq h^*(s) < h_2(n) \Rightarrow h_1(n) < h_2(n)$ which contradicts the dominance assumption

Relaxation of Problems:

- ***General Idea***: If we are able to find the actual heuristic for the relaxed problem, it must mean that the heuristic is admissible for the original problem. This is because the original problem will need more constraints and therefore, the actual cost will be higher and the heuristic we found would be admissible. More we relax, the further we are from the goal and the more stringent, the closer we are to the goal

Local Search:

- When we are only concerned with what the goal state is. Not concerned with how to get there

Advantages:

- Can just store the current and immediate successor states:

Space Complexity: $O(b)$ – Can be reduced to O(1) if successors may be processed one at a time

- Applicable to very large or infinite search spaces

Problem Formulation for Local Search:

States: Each state will be a complete assignment
Initial State: Probably just some randomly initialised complete assignment
Next State (Actions): Perturbs the current state by 1 move but still a complete assignment
Stopping criteria: Stop when the optima is found or after a specified number of iterations n, whereby the value of the current state is better than all its neighbours or next states. We can use hill-climbing with random restarts as well.

Hill Climbing (Steepest Ascent) Algorithm

```
current = initial_value
while true:
            neighbour = highest_valued_successor(current)
            if value(neighbour) <= value(current): return current
            current = neighbour
```

Explanation:

- Start with a random initial state
- Only store the current state
- In each iteration, find a successor that improves on the current state
  - Requires actions and transition to determine successor
  - Requires value; a way to value each state e.g. $f(n) = -h(n)$
- If none exists, return the current state as the best option
  - Note that the algorithm can fail; return a non-goal state

Issues:

- May get stuck at Local Maxima, Shoulder or Plateau, Ridge (Sequence of local maxima)

Variants:

Variant	Details
Stochastic hill climbing	Changes the highest_valued_successor(...) Chooses randomly among states with values better than the current one (*Not choosing the highest one but any that is better*) May take longer to find a solution but sometimes leads to better solutions
First-choice hill climbing	Changes the highest_valued_successor(...) Handles high b by randomly generating successors until one with better value than current is found (rather than generating all possible successors) Space complexity will be O(1) since don't have to keep track of all of them
Sideways move	Replaces ≤ with <; allows for continuation when value(neighbour) == value(current) Can help to traverse shoulders/plateaus
Random-Restart hill climbing	Adds an outer loop which randomly picks a new starting state Keeps attempting random restarts until a solution is found Note that this is the default one that we should be using

Random-Restart Hill Climbing Algorithm

```
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
```

Local Beam Search:

Stores k states instead of 1 now

- Hill Climbing stores only the current state but beam stores k states

Algorithm:

Begin with k random initial states
Each iteration generate all successors of the current k states
Repeat with the best k among ALL generated successors unless goal is found

Advantages:

- Better than k parallel random restarts because we just take the best k among all successors and not just the best from each set of k sets
- No longer a constraint to have 1 best successor from each set of random initial state

Stochastic Beam Search

- Original variant may get stuck in a local cluster
- Adopt stochastic strategy similar to stochastic hill climbing to increase state diversity