CS3243 Cheatsheet
by Yiyang, AY21/22

Introduction

Intelligent Agent

An intelligent agent consists of: (1) **Sensors**, for capturing data (known as **Percepts**, p_i of **Percept History**, $P = \{p_1, ..., p_n, ...\}$) from the environment; (2) **Agent Functions**, f, for making decisions based on percepts, and **Actuators**, for performing actions, $a_1, ..., a_m \in A$, based on agent functions.
In short, an agent is a function $f : P \to A$.

Types of Agents and Environments

Agents

Reflex Agent - Agents that use IF-ELSE rules to make decisions.
Model-based Reflex Agent - Agents that use an internalised model to make decisions. (e.g. state graph models for search AI)
Goal-/Utility-based Agent - Agents that determine sequences of actions to reach goals / maximise utility.
Learning Agent - Agents that learn to optimise. (not covered)

Properties of the Problem Environments

Fully / **Partially Observable** - whether agents can access all information of the environment.
Deterministic / **Stochastic** ~ whether transition of states is certain.
Episodic / **Sequential** - whether actions impact only current states or all future decisions.
Discrete / **Continuous** ~ of state info., time, percepts and/or actions.
Single / **Multi-agent**
Known / **Unknown** ~ of knowledge of the agent.
Static / **Dynamic** - whether the environment changes while the agent is deciding actions.

The real world is partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

Uninformed Search

General Search

Search Problem Definitions

A Search problem consists of 1) **State representation**, s_i, for each environment instance, 2) **Goal test**, isGoal $: s_i \to \{0, 1\}$, that determines if a state is a goal, 3) **Action Function**, action $: s_i \to A$, that returns possible actions for every state, 4) **Action Cost**, cost $: (s_i, a_j, s_i') \to V$, that returns cost v of taking action a_i of state s_i to reach s_i', and 5) **Transition Model**, $T : (s_i, a_j) \to s_i'$ representing the state transition.

A transition model describes the problem in a dynamic and efficient way, as it does not list all states' actions.

Uninformed Search are search algorithms without domain knowledge beyond the search problem formulation.

Generic Search Algorithm

The only difference is how each algorithm implements the `frontier` for searching.

Correctness:: 1) **Completeness** - whether an algorithm will find a solution when one exists **AND** correctly report failure if not exists, and 2) **Optimality** - whether an algorithm finds a solution with lowest path cost among all solutions. Note: An optimal solution must be complete.

Implementation wise, there are 1) **Tree-Search** that allows revisiting of nodes, and 2) **Graph-Search** that do not allow revisit to states unless the new cost is smaller than current one (by maintaining a `reached` hash table upon adding nodes to `frontier`.

Search Algorithms

Breadth-First Search (BFS) : Use `Queue<>` for `frontier`. Possible improvement is to perform **Goal checking on pushing to frontier** to reduce storage.
Depth-First Search (DFS) : Use `Stack<>` for `frontier`.
Uniform-Cost Search (UCS) : Use `PriorityQueue<>` for `frontier`. Essentially Dijkstra's Algorithm that always explores the node with shortest path cost in the frontier. Note: Need to ensure all costs are larger than some constant $\epsilon > 0$. Therefore, it cannot be used for negative / zero costs (just like Dijkstra).
Depth-Limited Search (DLS) : DFS but with a limit on the maximum depth, l, which may be determined using domain knowledge.
Iterative Deepening Search (IDS) : Perform DLS repeated with $l = 1, 2, ...$ Intuitive, the algorithm compromises running time for better memory usage.

Informed Search

Heuristics

Heuristic Function, $h = h(n)$, approximates the shortest distance from a state to the nearest goal.
$h(n)$ tries to approximate the actual distance function $h^*(n)$.
A heuristic is **admissible** if for any state n, $h(n) \leq h^*(n)$, which means the heuristic might under-estimate but never over-estimates.
A heuristic is **consistent** if for all states n and its successor n', $h(n) \leq \text{cost}(n, a, n') + h(n')$, which means priority f in A* is non-decreasing along a path if a consistent heuristic is chosen. Note:Consistent heuristics are always admissible.
Two heuristics, h_1 **dominates** h_2 iff. $h_1(n) \geq h_2(n)$ for all states n.
Note: dominance is defined for all heuristics. However, for two admissible heuristics, the dominating one is preferred.

Informed Search Algorithms

Based on UCS, but incorporate domain knowledge via h.

Greedy Best-First Search : Use **Evaluation Function**, $f(n) = h(n)$ as priority. Intuitively, it picks state "seemingly" closest to goal.
Analysis: **Incomplete** under Tree-Implementation, and **Complete** under Graph-Implementation. **Not optimal** under both implementations.

A* Search : Use **Evaluation Function**, $f(n) = g(n) + h(n)$ as priority where $g(n)$ is the current path cost.
Limited-Graph Search (LGS) : A modified Graph-Implementation of A* that adds nodes to `reached` table on pop instead of pushing.
Analysis: Tree-Implementation of A* is **Optimal** for admissible h.
LGS is **Optimal** for consistent h.

Local Search

Local Search only concerns with goal state(s) but not how it is found or its cost.
Local Search is **Incomplete**, but it uses less space ($O(b)$ for branching factor b) and is applicable to larger and finite search space.
Complete-State Formulation - Every state has all components of a solution. Each state is a potential solution.

Local Search Algorithm

Hill Climbing (aka. **Steepest Ascent - Greedy Strategy**) - It stores only current state. In each iteration, find a successor that improves - 1) use actions and transitions to determine successors, and 2) use "heuristic-liked" values (e.g. $f(n) = -h(n)$) to evaluate each state. The algorithm terminates with a state when the value f cannot be improved. Note: The algorithm may fail as it can terminate at a local maximum / plateau.

Hill Climbing Variations

Slideways Move - Replace $<$ with $\leq$, to allow continuation with neighbours of same value and to traverse plateaus.
Stochastic ~ - Choose randomly a state with better value (not the best value) to explore. This takes longer time to find a solution but gives more flexibility and randomness. Relieve "local maximum" issue.
First-Choice ~ - Generate successors until one with better value than current is found.
Random-Restart ~ - Use a loop to randomly pick a new starting state. Keep running until a solution is found.

Local Beam Search

Local Beam Search - Similar to Hill Climbing but start with k random starting states. Each iteration generates successors of all k states and choose new k ones to explore. Stochastic elements can also be incorporated into this algorithm. Note: [1] It is not equivalent to k-parallel Hill Climbing. [2] Local Beam requires problems with different possible starting states, otherwise it cannot be run.

Constraint Satisfication Problem (CSP)

CSP Formulation

A CSP consists of 1) **State Representation**, for variables $X = \{x_1, ..., x_n\}$ and set of domains $D = \{d_1, ..., d_n\}$, so that x_i has domain d_i, 2) **State**, where the initial / intermediate / goal state(s) have variables unassigned / partially assigned / all assigned, 3) **Goal Test**, with constrains $C = \{c_1, ..., c_m\}$ to satisfy, and 4) **Actions** and **Transitions**. Note: Costs are not utilised in CSP.

Depending on the **Scope**, number of variables involved, of constraints, they can be categorised into 1) **Unary**, **Binary**, and **Global** which involves 1, 2, and ≥ 3 variables respectively.

A **Constraint Graph** is a representation for constraints where each variable is a vertex, each binary constraint is an edge between two variables, and each global constraint is expressed as multiple binary constraints using linking a vertex.

CSP Heuristics

Minimum-Remaining-Values Heuristic (MRV) - Choose the variable with fewest legal values to be assigned. Intuitively, it means to consider nodes with fewer branches first, so that it tends to prone larger invalid sub-trees.
Degree Heuristic - Choose the variable with fewest number of constraints with other non-determined variables (smallest **degree**) to be assigned. Typically used as a tie-breaker after MRV Heuristic.
Least-Constraining-Value Heuristic (LCV) - Choose value that rules out fewest choices (in terms of domain sizes of "neighbour variables") to be assigned. Intuitively, it avoids failure.

Variable Order Heuristics: MRV & Degree. **Value Order Heuristics**: LCV

CSP Algorithms

Backtracing Algorithm uses DFS to examine value assignments of one variable at a time, and backtrack when no possible legal assignments with current assignment of the variable.
Heuristics can be used for `SELECT-UNASSIGNED-VARIABLE()` and `ORDER-DOMAIN-VALUES`. Constraint propagation algorithms are used for `INFERENCE()`.

Constraint Propagation Algorithms

Forward Checking tracks remaining legal values (reduces domain) of each unassigned variable after current variable's assignment. It makes search terminate when any variable has no legal values (in its domain).

AC3 Algorithm updates all arcs corresponding to binary constrains of that variable with others, whenever the domain of the variable is updated. It ensures **Arc-Consistency** between variables X_i and X_j that is for every $x \in D_i$ there exists some value $y \in D_j$ that satisfies the binary constraint(s) on arc (X_i, X_j). Note: Arcs are **directed**.
Analysis: Time $O(n^2 d^3)$ for number of states n and domain size d.
Preprocessing to reduce domain size is therefore important.

Adversarial Search

Game Model

A game model consists of 1) **State representation**, 2) `TO-MOVE(s)`, a function that returns the player to move for state s, 3) `ACTIONS`, legal moves in state s, 4) `RESULT(s, a)`, the transition model that returns resultant state after taking action a at state s, 5) `IS-TERMINAL(s)`, whether the game has ended at state s, and 6) `UTILITY(s, p` that defines the final numeric value to p when the game ends in terminal state s.
Also, we define two players: 1) **MAX**, the agent with goal to maximise value, and 2) **MIN**, the opponent with goal to minimise (agent's) value.

Search Strategies

MiniMax Optimal Strategy - MAX chooses the move that maximise the minimum payoff while MIN chooses to minimise the maximum payoff.
Implementation wise, it checks from leaves to root - for every node of MAX / MIN, takes the maximum / minimum utility value of its children nodes. DFS in nature.
Analysis: It is **Complete** if game tree finite; **Optimal**, assuming both players play optimally; time $O(b^m)$ and space $O(bm)$.

Nash Equilibrium - a condition where whether one player knows the strategy the other player has does not affect the game outcome.

Pruning Techniques

Alpha-Beta Pruning defines $\alpha(n)$ and $\beta(n)$ as highest and lowest observed value found on path from n respectively. ($\alpha_0 = -\infty$ and $\beta_0 = \infty$ Initially.) The pruning follows:

- Given a MIN node n, stop searching below n if for some ancestor i of n, $\alpha(i) \geq \beta(n)$
- Given a MAX node n, stop searching below n if for some ancestor i of n, $\beta(i) \leq \alpha(n)$

Analysis: Pruning might not improve efficiency but it does not affect correctness. Limitations: 1) Effectiveness depends on move ordering and 2) Does not reduce maximum depth of tree. Note: Also pruned when taking '='!

Cutting-Off Search with **Heuristic MiniMax** prunes by replacing DFS with LDS and using a heuristic to choose (non-terminal) nodes to explore. For modelling, it 1) replaces `IS-TERMINAL(s)` with `CUTOFF-TEST(s, d)` that checks if current depth has reached the depth limit, and 2) replaces `UTILITY(s,p)` with `EVAL(s,p)`.

Knowledge-Based Agents

Knowledge-Based Agent - agent making inferences on existing info to infer new ones. It consists of an **Inference Engine** (domain-independent algorithms) and a **Knowledge Base** (domain-specific content), which is a set of sentences (logical formulae).

Logic

A value assignment ν **Models** a sentence α iff. *alpha* is true under ν. A **Model** can also mean a value assignment.
Define $M(\alpha)$ as the set of all possible models for α.

For two sentences α and β, α **Entails** β iff.

$$\alpha \models \beta \equiv M(\alpha) \subseteq M(\beta)$$

Intuitively, entailment is a more abstract form of "implication".

Inference Algorithm

Inference algorithms assume a query α is given and check if KB entails it. α usually comes from `action = AS(...)`.

Properties

Let $KB \vdash_{\mathcal{A}} \alpha$ means "Sentence α is derived from Knowledge Base, KB, using inference algorithm $\mathcal{A}$".
Soundness - $KB \vdash_{\mathcal{A}} \alpha \implies KB \models \alpha$, i.e. $\mathcal{A}$ will not infer incorrect sentences.
Completeness - $KB \models \alpha \implies KB \vdash_{\mathcal{A}} \alpha$, i.e. $\mathcal{A}$ will arrive at all possible conclusions.

Truth Table Enumeration Algorithm

An algorithm that lists all 2^n truth assignments to verify KB entails α, by using DFS.
Analysis: **Sound** and **Complete**. Time and space $O(2^n)$ and $O(n)$.

Theorem Proving Methods

Logical Agent

A sentence α is **Valid** if it is true for **all** possible truth value assignments. It is linked to entailment by **Deduction Theorem**:

$$(KB \models \alpha) \equiv ((KB \implies \alpha) \text{ is valid })$$

A sentence α is **Satisfiable** if it is true for **some** truth value assignments. It is related to entailment:

$$(KB \models \alpha) \equiv ((KB \wedge \neg\alpha) \text{ is unsatisfiable })$$

A valid sentence is a tautology. Proof by contradiction uses the idea that something is true $\equiv$ its negation is unsatisfiable.

Resolution

Conjunctive Normal Form - Conjunctions of Disjunctive. i.e. clauses joined by AND, where each clause uses only OR to join literals. KB is a CNF where each clause is a sentence in KB.

Resolution - Simplifies KB to prove entailment of query α. $(R_1 \vee x) \wedge (R_2 \vee \neg x)$ is resolved to $R_1 \wedge R_2$.

Resolution Algorithm

Procedure: 1) Make a clause list (KB in CNF, with **Negation of Query** α). 2) Repeatedly resolve two clauses and add resolvent to list. 3) Stop when empty clause $\emptyset$ found, which implies α CAN be inferred, or when no more resolutions possible, which implies α CANNOT be inferred.
Note: [1] Cannot be inferred $\neq$ Is incorrect / negation can be inferred. [2] The negation of query may be a conjunction of 2 clauses (, e.g. if query has $\implies$). [3] When clauses have universe qualifier $\forall$, can still resolve by setting variable to certain value.
Analysis: **Sound** and **Complete**.

Bayesian Inference

Probability Basics

Independent Events - $P(ABC) = P(A)P(B)P(C)$.
Causal Chain - $P(ABC) = P(C|B)P(B|A)P(A)$, where C depends on B and B depends on A.
Independent Causes - $P(ABC) = P(C|AB)P(A)P(B)$, where C is affected by both A and B, which are independent.
Conditionally Independent Effects - where A affects both B and C separately, i.e. $P(ABC) = P(C|A)P(B|A)P(A)$.

Bayesian Network

Bayesian Network is a model that represents joint distributions with a Directed Graph, where vertices are random variables, and an edge from X to Y indicates X **directly affects** Y. Analysis: A Bayesian Network is a DAG.

Standard Algorithm Pseudocode

Generic Search

```
frontier = {initial state}
while frontier not empty:
    current = frontier.pop()
   // checking
   if isGoal(current):
      return path found
   // exploration
   for a in actions(current):
      frontier.push(T(current, a))
return failure
```

CSP Backtracking

```
def BT-SEARCH(csp) -> solution
    return BT(csp, {})
def BT(csp, assign) -> solution
    if assign complete:
        return assign
    var = select-unassigned-var(csp, assign)
    for val in order-dom-val(csp, var, assign):
        if val is consistent:
            assign.add((var, val))
            inf = inference(csp, var, assign)
            if inf != NULL:
                csp.add(inf)
                result = BT(csp, assign)
                if result != NULL:
                    return result
                csp.remove(inf)
            assign.remove((var, val))
    return NULL
```

AC3

```
def AC3(csp): // true for consistent
    queue = csp.get_all_arcs()
    while queue not empty:
        (i, j) = queue.pop()
        if REVISE(csp, i, j):
            if len(D[i]) == 0:
                return FALSE
            for k in (i.neighbours() - j):
                queue.add(i,j)
    return TRUE
// true if domain revised/modified
def REVISE(csp, i, j):
    revised = FALSE
    for x in D[i]:
        if no y in D[j] that (x,y) satisfy (i,j):
            D[i].remove(x)
            revised = TRUE
    return revised
```

Alpha-Beta Pruning

```
def AB-SEARCH(game,state) -> action:
    player = game.TO-MOVE(state)
    value,move =
        MAX-VALUE(game,state,MIN_INT,MAX_INT)
    return move
// MAX-VALUE, MIN-VALUE symmetric
def MAX-VALUE(game,state,a,b) -> (util,move):
    if game.IS-TERM(state):
        return game.UTIL(state,player),NULL
    v = MIN_INT
    for ac in game.ACTIONS(state):
        v2,act2 =
            MIN-VALUE(game,game.RES(state,ac),a,b)
        if v2 > v:
            v, move = v2, ac
            a = MAX(a, v)
        if v >= b: return v, move
    return v, move
```

KB Agent Function

```
global: KB, t = 0 // t for time
def KB-Agent(percept) -> action:
    TELL(KB, to_sentence(percept, t))
    action = ASK(KB, make-action-query(t))
    TELL(KN, to_sentence(action, t))
    t = t + 1
    return action
```

Truth Table Enumeration

```
def TT-ENTAILS(KB, a):
    symbol = list of prop. symbols in KB & a
    return TT-CHECK-ALL(KB, a, symbols, {})
def TT-CHECK-ALL(KB, a, symbols, model):
    if symbol is empty:
        if PL-TRUE?(KB, model):
            return PL-TRUE?(a, model)
        else:
            return TRUE
    // i.e. non-emtpy
    p, rest = symbols.first(), symbol.rest()
    return
        (TT-CHECK-ALL(KB,a,rest,model+{P=TRUE})
        and
        (TT-CHECK-ALL(KB,a,rest,model+{P=FALSE})
```

Resolution

```
def PL-RESOLUTION(KB, a) -> bool:
    clauses = get_CNF(KB+(!a))
    new = {}
    while TRUE:
        for (i, j) in clause:
            res = PL-RESOLVE(i, j)
            if res is empty:
                return TRUE
            new = union(new, res)
        if new subset of clauses:
            // cannot further resolve
            return FALSE
        clauses = union(clauses, new)
```

Intermediate Results / Lemmas

From Tutorials & Quizzes

(Quiz1 Qn9) Testing goal upon pushing (than poppig) to frontier can save at most $(b^{d+1} - b)$ nodes in BFS.
(Quiz2 Qn12) For Uninformed Search problems with the goal node near the root, the branching factor finite, and all action costs equal, **BFS** is preferred.
(Quiz2 Qn13) For Uninformed Search problems with all nodes at a certain depth being goal nodes and all action costs equal, **DFS** is preferred. Note: Distinguish between pushing order and popping order in DFS.
(Quiz6 Qn8) In AC3, each arc is inserted at most d times for domain size d.
(Tut10 Qn2) If T affects on D then it is true that D also depends on T as our knowledge of T affects info. about D.

From Past Questions

(AY19/20Sem1 Midterm Qn1) A* Graph Search with consistent heuristic is guaranteed to visit no more nodes than UCS.
(AY19/20Sem2 Midterm Qn1) In a A* Graph Search problem, leaving a consistent heuristic $h(n)$ unchanged:

- After adding edges to the transition graph, h might not be still consistent.
- After removing edges from the transition graph, h is still consistent.

Summary

Uninformed Search Algorithms

For Tree-Search implementation:

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

For Graph-Search implementation, **all have time and space complexity** $O(V + E)$.

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1][2]	Yes[1]	No	Yes[1]
Optimal?	Yes[1]	Yes	No	No	Yes[3]

[1] If b finite **AND** (state space finite **OR** has a solution)
[2] If the ϵ assumption is satisfied for all costs
[3] If all costs are identical