CS3243 Finals Cheatsheet v1.0 (2019-09-29)
by Julius Putra Tanu Setiaji, page 1 of 2

1 Introduction to AI

1.1 Rational Agent

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

1.2 Task Environment

- **PEAS**: Performance, Environment, Actuators, Sencsors
- **Fully observable** vs **Partially observable**:: an agent's sensors give it access to the complete state of the env at each point in time **VS** if the sensors detect all aspects that are relevant to the choice of action
- **Single agent** vs **Multiagent**: whether there are any other agent in the environment, multiagent further divided into **competitive** and **cooperative** where **communication** and **randomised behaviour** are the typical rational behaviours respectively
- **Deterministic** vs **Stochastic**: if the next state of the env is completely determined by the current state and the action executed by the agent **VS** otherwise. (partially observable env may appear stochastic)
- **Episodic** vs **Sequential**: the choice of current action does not depend on prev actions **VS** otherwise
- **Static** vs **Dynamic**: if the environment is unchanged while an agent is deliberating **VS** otherwise
- **Discrete** vs **Continuous**: in terms of state of env, time, percepts and actions

1.3 The Structure of Agents

- An **agent program** (takes in current percept) implements the **agent function** (percept sequence): mapping from percept sequence to actions.
- **Table-Driven-Agent**: persists the percept sequence from the current percept, and looks up action from table.
- Drawback: Hube table to build and store (time and space), no autonomy (impossible to learn all correct table entries from experience), no guidance on filling in the correct tabel entries

1.4 Agent Types, in increasing generality

- **Simple Reflex Agent**: passive, only selects actions on the basis of the current percept (ignoring percept history). Updates state based on percept only
- The rest updates state based on percept, current state, most recent action and model of the world.
- **Model-based Reflex Agents**: passive, (to handle partial observability, need to build model of the world)
- **Goal-based Agent**: achieve goal (binary: achieve goal/not).
- **Utility-based Agent**: maximises utility function (measure of happiness: more than binary).
- **Learning Agent**: **Learning element** responsible for making improvements with feedback from the **critic**, **performance element** (what was agent) responsible for selecting external actions, **problem generator** responsible for suggesting actions (do suboptimal now to explore better actions in the long run).

2 Solving Problems by Searching

2.1 Problem Formulation

Initial State, **Actions** (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 Cost** (assigns a numeric cost to each path)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

A **node** includes state, parent node, action, and path cost.

2.2 Evaluation criteria

- **Completeness**: always find solution if one exists
- **Optimality**: finding a least-cost solution
- **Time complexity**: no of nodes generated
- **Space complexity**: max. no of nodes in memory

2.3 Problem parameters

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

Uninformed Search Strategies

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

*: BFS, IDS – complete if b is finite, optimal if step costs are identical
**: UCS is complete if b is finite and step cost $\geq \epsilon$
***: DFS is complete only on infinite depth graphs
C^* is the optimal cost

2.4 Breadth-First Search (BFS)

Expand shallowest unexpanded node, frontier is FIFO

2.5 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

2.6 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)

2.7 Depth-Limited Search (DLS)

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

2.8 Iterative Deepening Search (IDS)

- Perform DLS with increasing depth limit.
- Preferred if search space is large and depth of solution is not known

Informed (Heuristic Search Strategies)

Use an **evaluation function** $f(n)$ for each node n

2.9 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
- **Complete** if b is finite, **Non-optimal**, **Time** $O(b^m)$, **Space** $O(b^m)$

2.10 A* Search

- $f(n) = g(n) + h(n)$ where $g(n)$ = 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: $\forall n, h(n) \leq h^*(n)$ where $h^*(n)$ = true cost
- **Consistent Heuristic**: triangle inequality – $h(n) \leq c(n, n') + h(n')$
- Every consistent heuristic is also admissible.
- Theorem: If $h(n)$ is admissible, then A* tree-search is optimal
- Theorem: If $h(n)$ is consistent, then A* graph-search is optimal (from lemma consistent heuristic always follow optimal path)
- **Complete** if there is a finite no of nodes with $f(n) \leq f(G)$, **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 $\forall n, h_2(n) \geq h_1(n)$ then h_2 **dominates** h_1
- More dominant heuristics incur lower search cost

3 Beyond Classical Search

- Path to goal is irrelevant, the goal state itself is the solution.
- Advantages: (1) use very little/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**

3.1 Hill-climbing Search (aka Greedy Local Search)

- Continually moves in the direction of icnreasing value, terminate when reaching a "peak"
- Possible to get stuck in local maxima, only use if OK with approximate solutions.

4 Adversarial Search

- 2 players, zero-sum game
- **Game formulation**:
  - S_0: The **initial state**
  - PLAYER(s): which player has the move in a state
  - ACTIONS(s): returns the set of legal moves in a state.
  - RESULT(s, a): The **transition model**, defines the result of a move
  - TERMINAL-TEST(s): A **terminal test**, true when game is over
  - UTILITY(s, p): A **utility function**), defines final numeric value for a game that ends in terminal state s for a player p

4.1 Optimal Decisions in Games

- **Winning** strategy for one player if for any strategy played by the other player, the game ended with the former as the winner. Similar for **non-losing** strategy.
- **Nash Equilibrium** – when players know the strategies of all opponents, no one wants to change their strategy.
- **Subgame Perfect Nash Eq** – every subgame is a **Nash Eq**

4.2 Minimax

- Optimal strategy can be determined from the **minimax value** of each node: utility (for MAX) of being in the state, assuming both players play optimally from there to the end of the game.
- Minimax returns a subperfect Nash equilibrium
- Minimax is **Complete** (with finite game tree), **Optimal**, **Time** $O(b^m)$, **Space** $O(bm)$

4.3 $\alpha - \beta$ Pruning

- MAX node n: $\alpha(n)$ = highest observed value found on path from n, initially $-\infty$
- MIN node n: β = lowest observed value found on path from n, initially $+\infty$
- If a MIN node has value $v \leq \alpha(n)$, can prune
- If a MAX node has value $v \geq \beta(n)$, can prune

4.4 Imperfect Real-time Decisions

- Although very large search space in typical games is pruned by $\alpha - \beta$ pruning, minimax still has to search all the way to the terminal states.
- Replace utility function with heuristic **evaluation function** that estimates the position's utility, and replace the terminal test with a **cutoff test** that decides when to apply EVAL.

4.5 Evaluation Functions

- A mapping from game states to real values.
- Should be cheap to compute; for non-terminal states, must be strongly correlated with actual chances of winning
- Modern eval function: weighted sum of position features
- Need not return actual expected values, just maintain relative order of states, typically from statistically probabilities

4.6 Cutting off Search

Stop after a certain depth, can be combined with iterative deepening

5 Constraint Satisfaction Problems

Consists of 3 components:

- X is a set of variables, $\{X_1, ..., X_n\}$
- D is a set of domains, $\{D_1, ..., D_n\}$, one for each variable
- C is a set of constraints that specify allowable combinations of values

5.1 Terminologies

- **Consistent** assignment = does not vilate any constraints
- **Complete** assignment = every variable is assigned
- Goal: find a consistent and complete assignment
- **Binary constraint** relates 2 variables
- **Global constraint** involve an arbitrary number of variables
- Every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced.
- **Constraint graph**: nodes are variables, links are constraints

5.2 Variants

- Domain can be **discrete** (both **finite** and **infinite**) or **continuous**
- For discrete, infinite domains, a **constraint language** must be used without enumeration

5.3 Constraint propagation: Inference in CSP

Try to infer illegal values for variables by performing constraint propagation

For unary constraints, node consistency; For binary constraints, arc consistency

Arc Consistency = a variable X_i in CSP is arc-consistent with another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A network is arc-consistent if every variable is arc-consistent with every other variable.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Time $O(n^2 d^3)$ where n is number of vars, d is max domain size

K-**consistency** = if, for any set of $k - 1$ vars and for any consistent assignment to thsoe variables, a consistent value can always be assigned to any k-th var (arc-consistency is 2-consistency)

5.4 Backtracking Search for CSPs

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

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

- Better an just doing search, because CSPs are **commutative**
- DFS that chooses values for one variable at a time, and backtracks when a var has no legal values left to assign.
- For SELECT-UNASSIGNED-VARIABLE: use **Most Constrained Variable** choose the var with fewest legal values (Minimum Remaining Values (MRV) heuristic)
- Once a variable is selected, to decide the order to examine its values, use **Least Constraining Value** heuristic: prefer value that rules out the fewest choices for the neighbouring variables in the constraint graph

5.5 Local Search for CSPs

- Similar to hill-climbing, but instead with complete states, allow states that violate constraints, then reassign variable values
- In choosing a new value for a variable, herustic: select the value that results in the minimum number of conflicts with other variables

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

5.6 The structure of problems

- Theorem: if CSP constraint graph (with binary constraints) is a **tree**, then we can compute a satisfying assignment (or determine one does not exist) in $O(nd^2)$ time (no need to backtrack)
- Proof: Pick any variable to be the root of tree, and choose an ordering of vars such that each var appears after its parent in the tree (Toposort: $O(n)$, each of which must compare up to d possible domain values for the two variables)