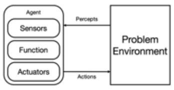
CS3243 Introduction to Artificial Intelligence

AY2021/22 Semester 2

1. Introduction

- Agent Function: $f: P \rightarrow a_t$, where P is the sequence of percepts captured by sensors and $a_t \in A$ is the selected action by activators.



- Rational agent optimises performance measure.
  - An agent that senses only **partial** information can also be perfectly rational.
- Environment Properties:
  - Fully Observable vs Partially Observable
  - Deterministic vs Stochastic: Whether immediate state can be determined based on action.
  - Episodic vs Sequential: Whether actions only impact current state or all future states.
  - Discrete vs Continuous
  - Single-agent vs Multi-agent: Opponents might be competitive or cooperative.
  - Known vs Unknown: Refers to the agent/designer.
  - Static vs Dynamic
- Taxonomy of Agents:
  - Reflex Agents: Uses if-statements.
  - Model-based Reflex Agents: Makes decisions based on an internal model.
  - Goal-based/Utility-based Agents
  - Learning Agents

2. Uninformed Search

- Formulation of search problem:
  - State Representation (s_i): ADT containing data describing an instance of the environment.
  - Initial State (s_0): Initial values of the data above.
  - Action
  - Transition Model: How each data change corresponding to the action given.
  - Step Cost
  - Goal Test
- Uninformed Search: No domain knowledge beyond search problem formulation.
- General Search Algorithm:

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

BFS – queue
DFS – stack
UCS – pq

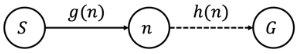
- State vs Node:
  - State: A representation of the environment at some timestamp.
  - Node: Includes state, parent node, action, path cost (for UCS), depth.
- Algorithm Criteria:
  - Time/Space Complexity
  - Completeness: Complete if can find a solution when one exists and report failure if it does not.
  - Optimality: Optimal if it finds a solution with the lowest path cost among all solutions.
- Tree Search vs Graph Search: In graph search, we only add nodes to `frontier` and `reached` if (1) state represented by node not previously reached and (2) path to state already reached is cheaper than the one stored.
- Performance Summary:

1. If b is finite and state space is finite or contains a goal.
2. Same as 1.
3. If action costs are all equal.
4. b is branching factor, d is depth of shallowest goal. Can be improved by early goal test (when pushing): Assuming the worst case, we can save the time and space associated with $(b^d - b)$ nodes.
5. May get caught in a cycle.
6. If search space is finite.
7. Where m is the maximum depth. Can be improved to $O(m)$ by backtracking.
8. If BFS is complete and all action cost $> \varepsilon > 0$. Must perform late goal test (when popping).
9. $e = 1 + \lfloor C^*/\varepsilon \rfloor$, where C^* is the optimal path cost and ε is some small positive constant.
10. Where l is the limited depth.
11. Number of nodes explored: $(d+1)O(b^0) + dO(b^1) + \cdots + O(b^d)$.

Criterion	BFS	DFS	UCS	DLS	IDS[11]
Complete (Tree/Graph)	✓[1]	X[5]	✓[8]	X	✓
	✓[2]	✓[6]	✓	X	✓
Optimal	X[3]	X	✓	X	✓
Time (Tree/Graph)	$O(b^d)$[4]	$O(b^m)$	$O(b^e)$[9]	$O(b^l)$[10]	$O(b^d)$
	$O(\lvert V\rvert + \lvert E\rvert)$				
Space (Tree/Graph)	$O(b^d)$	$O(bm)$[7]	$O(b^e)$	$O(bl)$	$O(bd)$
	$O(\lvert V\rvert + \lvert E\rvert)$				

3. Informed Search

- Heuristic Function (h): Approximates the path cost from `n.state` to its nearest goal G.



  - $h^*(n)$: True path cost from n to G.
- Evaluation Function (f): Priority for a node n.
- Best-First Search Algorithm:

```
frontier = {initial state}
reached = {}
while frontier not empty:
     current = frontier.pop()
     if current is goal:
         return path found
     for a in actions(current):
         next = T(current, a)
         s = next.state
         if s not reached or
            next has a smaller path cost:
             reached[s] = next
             frontier.push(next)
return failure
```

- Greedy Best-First Search:
  - $f(n) = h(n)$
  - Tree search version is incomplete (may get stuck in a loop between nodes where h values are lowest). Graph search version is complete if search space is finite.
  - Optimal: No.
- A* Search:
  - $f(n) = g(n) + h(n)$
  - Limited Graph Search Version 1: No exceptions on lower path costs.
  - Limited Graph Search Version 2: Adds to `reached` when popping.
  - Complete if UCS is complete.
  - Tree search and graph search version is optimal if h is admissible. Limited graph search version 2 is optimal if h is consistent.
- Heuristics
  - Admissible: $h(n)$ is admissible if $\forall n, h(n) \leq h^*(n)$. If $h(n)$ is admissible, then A* Search using tree/graph search is optimal.
  - Consistent: $h(n)$ is consistent if $\forall n, n', h(n) \leq \text{cost}(n, a, n') + h(n')$, where n' denotes all successors of n. If $h(n)$ is consistent, then A* Search using limited graph search version 2 is optimal.
  - If h is consistent, then it is admissible.
  - If $\forall n, h_1(n) \geq h_2(n)$, then h_1 dominates h_2. If h_1 is admissible, then it is more efficient than or as efficient as h_2.

4. Local Search

- Hill-Climbing Algorithm:

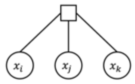
```
current = initial state
while true:
     neighbour = highest-valued successor
     if neighbour.value <= current.value:
         return current
     current = neighbour
```

  - $f(n) = -h(n)$
  - May get stuck at (1) local maxima, (2) shoulder or plateau and (3) ridge (sequence of local maxima).
  - Sideways Move: Replaces <= with <. This allows the algorithm to traverse shoulders.
  - Stochastic Hill Climbing: Chooses randomly among states with values better than `current`. May take longer to find a solution but sometimes leads to better solutions.
  - First-choice Hill Climbing: Randomly generating successors until one better than `current` is found.
  - Random-restart Hill Climbing: Adds an outer loop which randomly picks a new starting state. Keeps attempting random restarts until a solution is found.
- Local Beam Search:
  - Always stores k states instead of 1.
  - Begins with k random starts and chooses best k among all successors until a solution is found.

    - May be improved by stochastic.

5. Constraint Satisfaction Problems

- Formulation of CSPs:
    - State Representation (s_i): Variables ($X = \{x_1, x_2, \dots, x_n\}$) and their domains ($D = \{d_1, d_2, \dots, d_n\}$).
    - Initial State (s_0): All variables unassigned.
    - Action
    - Transition Model
    - Goal Test: Whether all constraints $C = \{c_1, c_2, \dots, c_m\}$ is satisfied. Each constraint corresponds to a subset of X. Each constraint contains a scope and a relation (e.g. $\text{scope} = (x_1, x_2); \text{relation} = x_1 < x_2$).
- Constraint Graph:
    - One binary constraint is two arcs.

		
Unary	Binary	Global

- Backtracking Algorithm:

```
assignment = {}
function backtrack(csp, assignment):
     if assignment is complete:
         return assignment
     var = select unassigned variable
           from assignment
     for each value in domain of var:
         if value consistent with csp:
         assignment.add {var=value}
         inference = infer(assignment)
         if inference not failure:
             csp.add(inference)
            result = backtrack(csp,
                     assignment)
            if result not failure:
                return result
            csp.remove(inference)
        assignment.remove {var=value}
    return failure
```

    - Total number of leaves: d^n, where d is number of values and n is number of variables.
    - Solution is found at depth n.
    - Variable order: Minimum-remaining-value heuristic + degree heuristic
    - Value order: Least-constraining-value heuristic
    - Maximum backtrack times: $O(2^n)$ (whole tree)
- AC-3 Algorithm

```
function AC3(csp):
     queue = a queue of all arcs
     while queue is not empty do:
         (Xi, Xj) = queue.pop()
         if REVISE(csp, Xi, Xj):
             if size of Di == 0:
                 return false
             for Xk in Xi.neighbours -
                      {Xj}:
                 add (Xk, Xi) to queue
     return true

function REVISE(csp, Xi, Xj):
     revised = false
     for each x in Di do:
         if no y in Dj satisfies
                      constraint:
             delete x from Di
             revised = true
     return revised
```

    - Time complexity: $O(n^2 d^3)$

6. Adversarial Search

- Formulation of games:
    - State representation
    - $\text{TO}-\text{MOVE}(s)$: The player to move in state s.
    - $\text{ACTIONS}(s)$: The legal moves in state s.
    - $\text{RESULT}(s, a)$: The transition model.
    - $\text{IS}-\text{TERMINAL}(s)$: Whether game is over.
    - $\text{UTILITY}(s, p)$: Defines the final numerical value to player p when the game ends in state s.
- Winning Strategy: A winning strategy for player 1, s_1^*, implies that for any strategy s_2 for player 2, the game ends in a win for player 1.
- Minimax Algorithm:

$$Minimax(s) = \begin{cases} \text{UTILITY}(s, MAX) \\ \max_{a \in \text{ACTIONS}(s)} Minimax(\text{RESULT}(s, a)) \\ \min_{a \in \text{ACTIONS}(s)} Minimax(\text{RESULT}(s, a)) \end{cases}$$

    - Minimax Algorithm is complete if game tree is finite, is optimal.
    - Time: $O(b^m)$; Space: $O(bm)$.
- $\alpha - \beta$ Pruning:
    - At MAX node n, $\alpha(n)$ is highest observed value found on path from n. Initially $\alpha(n) = -\infty$.
    - At MIN node n, $\beta(n)$ is lowest observed value found on path from n. Initially $\beta(n) = \infty$.
    - Given a MIN node n, stop searching below n if some MAX ancestor i with $\alpha(i) \geq \beta(n)$.
    - Given a MAX node n, stop searching below n if some MIN ancestor i with $\alpha(n) \geq \beta(i)$.

```
function AB-PRUNING(node, alpha, beta):
  if node is leaf:
     return node.val
  if isMaxTurn:
     bestVal = -INFINITY
     for each child:
       value = AB_PRUNING(child)
       bestVal = max(bestVal, value)
       alpha = max(bestVal, alpha)
       if beta <= alpha:
         break
     return bestVal
  if isMinTurn:
     bestVal = +INFINITY
     for each child:
       value = AB_PRUNING(child, alpha, beta)
       bestVal = min(bestVal, value)
       beta = min(bestVal, beta)
       if beta <= alpha:
         break
     return bestVal
```

7. Knowledge Representation

- Knowledge:
    - Knowledge Base: Set of sentences in a formal language prepopulated with domain knowledge. It is domain-specific content, while inference engine is domain-independent algorithm.
    - Entailment: $\alpha \vDash \beta \rightarrow M(\alpha) \subseteq M(\beta)$
- Algorithm Criteria:
    - $\text{KB} \vdash_{\mathcal{A}} \alpha$: Sentence α is derived from KB by inference algorithm $\mathcal{A}$.
    - Soundness: $\mathcal{A}$ is sound if $\text{KB} \vdash_{\mathcal{A}} \alpha$ implies $\text{KB} \vDash \alpha$. $\mathcal{A}$ does not infer nonsense.
    - Completeness: $\mathcal{A}$ is complete if $\text{KB} \vDash \alpha$ implies $\text{KB} \vdash_{\mathcal{A}} \alpha$. $\mathcal{A}$ can infer any sentence KB entails.
- Truth Table Enumeration: Checks all 2^n truth assignments to verify KB entails α (DFS).
    - Time: $O(2^n)$; Space: $O(n)$.
    - Sound and complete.
- Resolution:
    - Conjunctive Normal Form (CNF): Conjunction of disjunctive sentences, $R_1 \land R_2 \land \dots \land R_n$.
    - Sound and complete.
    - Rules:
        1. Convert $\alpha \Leftrightarrow \beta$ to $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$.
        2. Convert $\alpha \Rightarrow \beta$ to $\neg\alpha \lor \beta$.
        3. Convert $\neg(\alpha \lor \beta)$ to $\neg\alpha \land \neg\beta$
        4. Convert $\neg(\alpha \land \beta)$ to $\neg\alpha \lor \neg\beta$.
        5. Convert $\neg(\neg\alpha)$ to α.
        6. Convert $(\alpha \lor (\beta \land \gamma))$ to $(\alpha \lor \beta) \land (\alpha \lor \gamma)$.
    - Show that $KB \land \neg\alpha$ is unsatisfiable.
    - $R_1: \alpha \lor \gamma$ and $R_2: \beta \lor \neg\gamma$ are resolved to $\alpha \lor \beta$.

8. Uncertainty

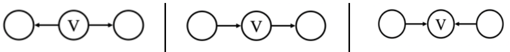
- Bayes Rule:

$$\Pr[A|B] = \frac{\Pr[B|A] \cdot \Pr[A]}{\Pr[B]}$$

- Chain Rule:

$$\Pr[R_1, R_2, \dots, R_k] = \prod_{j=1,2,\dots,k} \Pr[R_j | R_1, R_2, \dots, R_{j-1}]$$

- Path Blocking Scenario:



v is given | v is given | v is not given

- Suppose given S, all $T_1, T_2, \dots, T_{n-1}$ are independent, then:

$$\Pr[T_1, T_2, \dots, T_{n-1}, S] = \Pr[T_1|S] \cdot \dots \cdot \Pr[T_{n-1}|S] \cdot \Pr[S]$$

    - A joint distribution for n Boolean random variables results in at least $2^n - 1$ entries.
    - A joint distribution for n Boolean random variables with conditional independence results in $2n - 1$ entries.