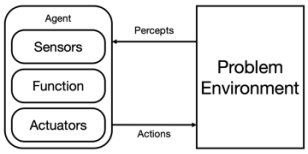
CS3243 Introduction to Artificial Intelligence

AY2021/22 Semester 2

1. Introduction

- Agent Function: $f: P \to a_t$, 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
```

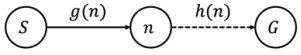
  - BFS: `frontier = queue`
  - DFS: `frontier = stack`
  - UCS: `frontier = priority queue`
- State vs Node:
  - State: A representation of the environment at some timestamp.
  - Node: Includes state, parent node, action, path cost (for UCS), depth.
- Algorithm Criteria:
  - Time Complexity
  - Space Complexity
  - Completeness: An algorithm is complete if it will find a solution when one exists and correctly report failure if it does not.
  - Optimality: An algorithm is 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:

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

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. Same as 1.
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: $(d+1)O(b^0) + dO(b^1) + \cdots + O(b^d)$.

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 is 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
```

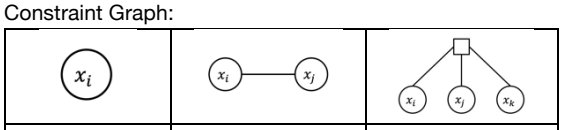
  - May get stuck at (1) local maxima, (2) shoulder or plateau and (3) ridge (sequence of local maxima).
  - Sideways Move: Replaces <= with <. This allows the algorithm to traverse shoulders.
  - Stochastic Hill Climbing: Chooses randomly among states with values better than `current`. May take longer 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:



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
```