CS3243 Cheatsheet (Midterm)
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 Algorithm

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

```
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
```

For **correctness** of search algorithms, we need to ensure 1) **Completeness** - whether an algorithm will find a solution when one exists **and** correct 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 overestimates.

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.
Consistent heuristics are always admissible.

For 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

The two Informed Search algorithms are 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 the state "seemingly" closest to the goal.

- **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 version of A* that adds nodes to `reached` table on pop instead of pushing.

- 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: It is not equivalent to k-parallel Hill Climbing.
Note: Local Beam requires problems with different possible starting states, otherwise it cannot be run.

Constraint Satisfication Problem (CSP)

CSP Formulation

- **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.
- **State**, where the initial / intermediate / goal state(s) have variables unassigned / partially assigned / all assigned.
- **Goal Test**, with constrains $C = \{c_1, ..., c_m\}$ to satisfy
- **Actions** and **Transitions**

Costs are not utilised in CSP.

Constraint Graph

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.

Results from Tutorials

(Quiz1Qn9) Testing goal upon pushing (than poppig) to frontier can save at most $(b^{d+1} - b)$ nodes in BFS.
(Quiz2Qn12) For Uninformed Search problems with the goal node near the root, the branching factor finite, and all action costs equal, **BFS** is preferred.
(Quiz2Qn13 For Uninformed Search problems with all nodes at a certain depth being goal nodes and all action costs equal, **DFS** is preferred.
Note the difference between pushing order and popping order in DFS.

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