COMP3702/7702 Artificial Intelligence
Semester 2, 2019
Tutorial 1 - Sample Solutions

Recall that the following components are required to solve an agent design problem:

- **Action Space** (A)
  The set of all possible actions the agent can perform.
- **Percept Space** (P)
  The set of all possible things the agent can perceive.
- **State Space** (S)
  The set of all possible configurations of the world the agent is operating in.
- **World Dynamics/Transition Function** ($T : S \times A \rightarrow S'$)
  A function that specifies how the configuration of the world changes when the agent performs actions in it.
- **Perception Function** ($Z : S \rightarrow P$)
  A function that maps a state to a perception.
- **Utility Function** ($U : S \rightarrow \mathbb{R}$)
  A function that maps a state (or a sequence of states) to a real number, indicating how desirable it is for the agent to occupy that state/sequence of states.

Question 1

With the above components in mind, an agent to play Tic-Tac-Toe can be designed.

First, we notice that Tic-Tac-Toe is a *fully observable* game, meaning that the agent always knows the exact state of the game from observing the board; there is no hidden information that the agent cannot perceive. This means that the **percept space** is the same as the state space; $P = S$. Following from this, the **perception function** Z simply maps each state to itself. Thus, P and Z are not required, and we only need to define A, S, T, and U.

Let us first consider the **action space**, A. The game board is a 3×3 grid, meaning there are 9 possible cells at which the agent can make a mark. If we think about ordering these cells from the top left to the bottom right, they can each be assigned a number from 1 to 9 to indicate their position on the board. Using this representation, $A = \{1, 2, 3, \ldots, 9\}$, where, for example, action 1 indicates making a mark at position 1 (the top left corner) of the board.

Using some similar logic, we can formulate a concise representation of the state of the game board. The state can be represented by a vector (or string) of length 9, with each possible element of the vector having 3 legal values, O (for a nought), X (for a cross), or e (for an empty cell). An example state in this form is $(X, e, X, e, O, X, O, X, O)$. The **state space**

is then just all possible legal vectors of this form. Note that a legal vector must contain at most 5 noughts or 5 crosses because of the turn-based nature of the game. Legal vectors also cannot contain a winning configuration of both noughts and crosses, as if a winning configuration is obtained by either player, the game ends.

You can visualise the **transition function** T as a lookup table with a row for each $(state, action)$ pair that stores the state that is the result of performing *action* in *state*. This function encapsulates both our move, and our opponent's move. Because the game is non-deterministic (we don't know what our opponent is going to do), the result of performing *action* in *state* could lead to multiple possible new states.

There are many possible utility functions for this game, and the choice of **utility function**, U, will determine how the agent behaves. One simple utility function is to assign a value of 10 to any state where the agent has a winning configuration of noughts/crosses, -10 to any state where the opponent has a winning configuration, and 0 for any other state. This utility function will reward the agent for winning the game, however it does not take into account the number of actions used to reach a winning state. Since the goal of the agent is to win in the least number of actions possible, one could assign a utility of -1 to any state that does not contain a winning configuration for the agent or the opponent. This way, the more actions the agent takes to win, the less utility the sequence of states it passes through has.

Question 2

There are many ways to answer this question, as it depends on the type of navigation app and what kinds of factors we want to be able to handle. Here we consider the case of a navigation app specifically for finding the shortest (not fastest) path to take in a network of roads. We ignore other 'adversarial' agents, such as people and cars in the environment, and don't include these entities as part of the states of the environment, mainly because the potentially random nature of these entities is far too difficult to model. To add a little bit of extra realism, we will also model one key feature of a road network, which is the fact that at an intersection some turns might be illegal! For this reason, we can't simply model states as intersections between roads, as the road we take on the way into the intersection determines which roads we can take on the way out.

Looking at the effects of the 3 assumptions listed: The first assumption will influence the formulation of the utility function for the agent, as the utility function dictates how the agent behaves. The second assumption causes the environment the agent is operating in to be *deterministic*, as the agent knows exactly where everything is in the environment. The third assumption causes the environment to be *fully observable* because the GPS reading tells the agent its exact location in the world.

a) Agent Design

As the environment is fully observable, defining the percept space and the perception function is not required. So, like Question 1, only A, S, T, and U need to be defined.

State space: The states of this problem describe locations of the user's vehicle in the world; in principle, these locations might be described by GPS coordinates. To simplify the problem, we consider states as corresponding to segments of road between junctions, where a "junction" is simply a point at which roads meet and thus it is possible to go in at least two different directions (e.g. go forward, turn left, take an exit off a highway, or take a specific exit out of a roundabout).

Additionally, since the dynamics of the world are very different depending on which direction one is travelling along a road, the direction of travel must also be included. Thus the state space can be formally described as

$$S = \{r : r \text{ is a road} \times \{1, 2, 3, \dots\} \times \{\textit{increasing}, \textit{decreasing}\}\},$$

where *increasing* and *decreasing* refer to whether house numbers increase or decrease in the direction of movement. This definition means that each state is a 3-element tuple consisting of a road, segment number, and the direction of travel such as

$$S = (\text{Gympie Rd}, 1, \textit{increasing}).$$

Note that not all states in the state space will be valid; some will be illegal or unreachable, since some roads are one-way roads, and each road only has finitely many segments.

Action space: Note that the app doesn't control the car, the app's user does! Thus the app's actions are not to drive the vehicle, but to give instructions to the user; the actions look like "tell the user to turn left" and not "turn left". This could be done in many ways, but the essential question is this: at a junction, which way out should the user take?

One way to represent this information within our agent is to number the possible ways out of a junction in clockwise order, so the action space is:

$$A = \{1, 2, 3, \dots\}$$

To make the navigation app more user friendly, hopefully a translation layer would be added so that "1" would be translated into, say, "turn left" or "take the first exit" depending on what kind of junction is coming up.

World dynamics/Transition function: The world dynamics describe how the environment reacts in response to the agent's actions. To make the problem easier, we assume that the user of the app will do what they are instructed to do. The world dynamics tell us what the next state will be, given the current state and the action the agent takes; this is a function[1]:

$$T : S \times A \to S',$$

which can be represented as a table with entries like this:

[1]Note: since not all states in the state space are valid, and not all actions are available in each state, it would be more accurate to say the domain of T is a subset of $S \times A$ which represents all legal state-action pairs.

State (S)	Action (A)	Next state (S')
(Gympie Rd, 40, increasing)	1	(Albany Creek Rd, 1, increasing)
(Gympie Rd, 40, increasing)	2	(Gympie Rd, 41, increasing)
(Gympie Rd, 40, increasing)	3	(Robinson Rd W, 20 decreasing)

The table above would imply that, for example,

$$T\big((\text{Gympie Rd}, 40, \textit{increasing}), 1\big) = (\text{Albany Creek Rd}, 1, \textit{increasing})$$

when represented in function form.

Percept space: Based on assumption (3), the problem is fully observable, and thus we do not need to formally specify the percept space. Equivalently, we could say that

$$P = S$$

Percept function: Since the problem is fully observable, a percept function is not required. Equivalently, it can be be described by the identity map, i.e.,

$$Z = S \to P$$

Utility function: Since the agent is supposed to find shortest paths, we can represent this in the form of utility by assigning the agent disutility (i.e. negative utility) proportional to the lengths of the road segments that it travels. This can be represented by a function

$$c : S \to \mathbb{R},$$

which specifies the length of each segment of road as a real number, e.g. in metres. The agent can then associate its total utility with the total cost of a sequence of states, i.e.

$$U\big((s_1, s_2, s_3, \ldots, s_n)\big) = -\sum_{i=1}^{n} c(s_i).$$

b) Defining environment

As already discussed, the assumptions given cause the agent to operate in a deterministic and fully observable environment.

If representing the state as GPS locations consisting of longitude and latitude (which can be expressed as arbitrary precision real numbers), the environment would be continuous. However, we have chosen to represent the state as a triplet of road, segment number and direction, so by our design, the state is discrete.

Because adversarial agents in the environment are not being modelled as part of the state, the environment is also static; the state cannot change while the agent is selecting an action.

c) Defining search problem

The translation to a search problem is fairly straightforward. The map can be represented as a graph, where vertices are states and edges are actions, and following an edge from one vertex to another models the transition function.

Similarly to how we thought about the utility function in the agent design problem, we could assign each edge a weight or cost to indicate how 'expensive' it is to transition along that edge. This cost could be based on the physical distance between the vertices' locations on the map. The search problem is then, given a starting vertex and a goal vertex, find the shortest path (path of least cost) between the two vertices in the graph.

Question 3

State space: The state of the agent is described by the last web page it retrieved, which we can consider to be the web page it is currently "on". Since each web page can be identified by a URL, the state space is therefore the space of possible URLs; or, alternatively, the subset of URLs corresponding to valid web pages. The key reason why a web crawler is useful is that the former space is extremely large, while the latter subset is far, far smaller, and it isn't known a priori.

Under assumption (b), i.e. considering the conditions of the real world, we might also choose to include additional information in a description of the state of the world, such as whether specific servers are up or down, or whether the agent's internet connection is up.

Action space: The web crawler functions by following links from the web page it is currently "on" to reach new web pages that it has not seen before. At the lowest-levels this consists of sending out IP packets, but a higher-level representation is more useful; we take the action space to be the space of possible URLs. In its simplest form, the actions available on a web page would consist only of the links from that web page, but this results in a major problem: what if the crawler reaches a dead end? For this reason, the agent should probably be allowed to go to URLs that are not linked to from the current page, at least via some form of backtracking.

World dynamics/Transition function: In the ideal world of assumption (a), a valid URL always leads to a valid webpage. If the agent only ever tries to visit URLs it has seen in links (which must be valid) and those web pages are always available, its world dynamics are represented by a transition function for which taking action $destURL$ results in the agent being in the state $destURL$, e.g. $T(\cdot, destURL) = destURL$).

Under the more realistic conditions of (b), attempting to visit a URL might give us an error message, or the request might time out before there is any reply from the server. This outcome can be represented by the agent remaining at its previous URL when attempting to visit the new one, while the details of the error can be represented via the percept space.

Percept space: For this problem, the observations the agent collects correspond to the information it receives back from the server when it tries to visit a URL; at the lowest

level this would come in the form of IP packets, but a higher-level representation is more useful.

In the ideal world of (a) the percepts might consist only of valid HTML pages, but for (b) we need to consider the types of failure that might happen. Thus (b) would have a space of observations that includes valid HTML pages, but also possible failure results such as failure to resolve DNS, Error 404, or a timeout.

Percept function: Since the agent doesn't know the contents of the web pages before it visits them (or there wouldn't be any point), the agent's perceptions are not very predictable, and cannot be described simply by a function.

However, there are some constraints on the perceptions, in that if *currentURL* and *destURL* are different, then the agent will perceive a full HTML page if and only if it transitions to *destURL*, but otherwise it will perceive an error message and remain on *currentURL*.

Utility function: Since the actual purpose of the web crawler is unclear, it is difficult to define a meaningful utility function. For this task, we assume the web crawler agent's goal is to visit as many unique web pages as possible, so it is rewarded every time it sees a new web page it hasn't seen before. Thus the agent's utility for a sequence of states, which are URLs that it has actually visited, is equal to the number of unique URLs in that sequence.