Artificial Intelligence: Week 2

1 What is an AI?

There are various definitions of an AI, ranging from thinking humanly and rationally to acting humanly and rationally. The *turing test*, is test in which a human interrogator interacts with a machine, sending it messages back and forth, and a machine passes if it fools the human into thinking that the messages are being sent to them by a human. For this a computer needs: **natural language processing, knowledge representation, automated reasoning and machine learning**. To pass the *total turing test* a computer would additionally need **computer vision and robotics.**

1.1 Intelligent Agents

An **agent** is just something that acts and a **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. **Percept** means the agent's perceptual inputs at any given time, and a **percept sequence** is the complete history of everything the agent has ever perceived. The **agent function** is an abstract mathematical description that maps a given percept to an action; an **agent program** is a concrete implementation of the agent function, running within some physical system. *It is better to design a performance measure according to what one wants in an environment, then how one wants an agent to behave.*

The proper definition of a rational agent is *for each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.* An **omniscient** agent knows the actual outcomes of its actions.

1.2 The Nature of Environments

An environment is defined as **PEAS**: performance measure, environment, actuators and sensors. There are different types of environments, namely:

- **Observable vs Partially-Observable**: it is observable when the agents' sensors have complete access to the environment's state at all times

- **Single agent vs Multi agent**: there could be multiple agents in an environment. There is also a question of what must be considered an agent. This gives way to the concept of **competitive** vs **cooperative** environments.

- **Deterministic vs Stochastic**: If the next state can be completely determined by the current state and the action executed by the agent, then it is deterministic; and stochastic otherwise. An environment is **uncertain** if it is not fully observable or not deterministic. Note that a **nondeterministic** environment is one where each action is characterized by its possible outcomes, but no probabilities are attached to them.

- **Episodic vs Sequential**: In an episodic environment the agent's experience is divided into atomic episodes. The next episode doesn't depend on the action taken in the previous episode.

- **Static vs Dynamic**: If an environment can change when an agent is deliberating, then it's dynamic, and is static otherwise. If the environment doesn't change when deliberating but the performance score does, then we call it **semi-dynamic.**

- **Discrete vs Continuous**: The distinction here applies to the state of the environment, the way time is handled and the percepts and actions of the agent. For example, chess having a discrete set of states; the same doesn't apply for taxi driving.

- **Known vs Unknown**: This applies to the agent's state of knowledge about the "laws of physics" of the environment. Note that it's possible that a known environment is partially observable like solitaire. Conversely, an environment can also be unknown and fully observable, like in a video game, one can see the state but one doesn't know the control until one tries to play.

1.3 The Structure of Agents

There are four basic types of agent programs:

- **Simple Reflex Agents**: Agents that select the current action based on the current precepts and ignoring the rest of the precept history. It is also important to note that these types of agents are usually implemented in a fully-observable environment.

- **Model-Based Reflex Agents**: The best way to handle a partially observable environment is to keep some sort of an internal representation of the aspects of the environment not currently observable. Therefore, an agent should have some sort of knowledge about how the world works and the agents who have said knowledge are called model-based agents.

- **Goal-Based Agents**: These types of agents consider how close they get to a goal, in addition to have a model of how their environment works. These types of agents are also quite flexible as they can update their actions on-the-fly depending on their goals and the feedback they get from the environment.

- **Utility-Based Agents**: Since the previous model does not differentiate between how it gets to its goal, and which state would make it more happy, it is not efficient. Therefore a utility function is needed to determine just that i.e. it is an internalization of its performance measure. The previous model will also fail when there are conflicting goals or when there are several goals the agent can aim for, none of which can be achieved with certainty; in both cases, a utility function can dictate which action to take to maximize expected utility.

There are different ways an agent can represent the world around it:

- **Atomic**: Each state is of the world is indivisible: it has no internal structure.

- **Factored**: Each state is split up into a fixed set of variables and attributes, each of which can have a value. Uncertainty can also be represented in this representation.

- **Structured**: This type of representation has objects and their relationship with each other.

2 Problem Solving by Searching

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. **Problem formulation** is the process of deciding what actions and states to consider, given a goal. In general, *an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.* The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase. Note while the agent is executing, it *ignores its percepts* when choosing its actions because it knows in advance what they will be.

Together, the **initial state, actions and transition model** implicitly define the **state space** of the problem - the set of all states reachable from the initial state by any sequence of actions. A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest

path cost among all solutions. The process of removing detail from a representation is called **abstraction**.

It is quite important to distinguish between a node and a state: a node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Furthermore, two different states can contain the same world state if that state is generated via two different search paths. We can evaluate the performance of an algorithm in four ways:

- **Completeness**: Is the algorithms guaranteed to find a solution if there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

In AI, the graph is often represented implicitly by the initial state, actions and transition model and is frequently infinite. Complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node; d, the **depth** of the shallowest goal node; and m, the maximum length of any path in the state space.

2.1 Uninformed Search Strategies

2.1.1 Breadth-First Search

This is a simple strategy in which all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The algorithm is given in figure 1.

BFS is *complete* - if the shallowest goal node is at some finite depth BFS will eventually find it after generating all the shallower nodes. Note that the *shallowest* is not always the *optimal* one. BFS is only optimal if the path cost is a non-decreasing function of the depth of the node. The time complexity of BFS is:

$$b + b^2 + b^3 + ... + b^d = O(b^d) \tag{1}$$

For the space complexity there will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, giving space complexity as $O(b^d)$. *Memory requirements are a bigger problem for BFS than execution time and time is still a major factor. Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

2.1.2 Uniform-Cost Search

This is similar to BFS, but it expands the node with the lowest path cost, the goal test is applied to a node *not when it's generated, but when it's chosen for expansion* and a test is added in case a better path is found to a node currently

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 1: Breadth-First Search.

on the frontier. The algorithm is given in figure 2. First, we note that whenever this algorithm selects a node for expansion, the optimal path to that node is found and second, paths never get shorter as nodes are added. Thus, *uniform-cost search expands nodes in order of their optimal path cost.* This search is complete given that the cost of every step exceeds some small positive constant ϵ. The space and time complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the optimal solution.

2.1.3 Depth-First Search

Depth-First Search always expands upon the deepest node in the frontier, this is accomplished using a LIFO queue. The graph-search version of the algorithm is complete in finite spaces but the tree-search version could potentially fall into a infinite loop. Although, it must be noted that both versions fail if there is an infinite state space, with an infinite non-goal path e.g. Knuth's 4 problem. Both versions are also not optimal. The time complexity of DFS is $O(b^m)$ and space complexity is $O(bm)$.

2.1.4 Depth-Limited Search

The problem of DFS failing in infinite spaces can be solved if we apply a limit l to the depth we expand up till. Although, it introduces incompleteness if we choose $l < d$ and non-optimality if $l > d$. Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$. Its pseudocode is shown in figure 4.

2.1.5 Iterative Deepening Search

The algorithm of IDS is shown in figure 5. Its space complexity is $O(bd)$. Like BFS, IDS is complete if the branching factor is finite and it is optimal when the

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 2: Uniform-Cost Search.

A recursive implementation of DFS:[5]

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

A non-recursive implementation of DFS:[6]

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Figure 3: Depth-First Search.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 4: Depth-Limited Search.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Figure 5: Iterative Deepening Search.

path cost is a non-decreasing function of the depth. The time complexity is:

$$(d)b + (d-1)b^2 + ... + (d-d+1)b^d = O(b^d) \tag{2}$$

In general, IDS is preferred if the search space is large and the depth of the solution is not known.

2.1.6 Bidirectional Search

Bidirectional search applies the idea of starting two searches: one from the initial state and one from the goal state, hoping that they meet in the middle; with the motivation that $b^{d/2} + b^{d/2} < b^d$. Thus the space and time complexity for this algorithm is $O(b^{d/2})$.

2.1.7 Comparison of Uninformed Search Strategies

Figure 6 shows comparisons for tree search versions of the algorithms discussed. For graph searches, the main difference is that DFS is complete for finite state spaces and that the space and the time complexities are bounded by size of the state space.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 6: Comparison of uninformed search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; b complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; if both directions use breadth-first search.