**Artificial Intelligence**

**Session 3**

1. **Problem Solving**:
   1. Reflex agent: act based on immediate observation / memory; often optimizes immediate reward.
   2. Planning agent: looks further into the future and “try out” different sequences of actions before taking an action; optimizes long-term reward.
   3. By this, the agents view goal-attainment as problem solving, and viewing that as a search through a state space.
2. **State-space model**:
   1. State space
   2. Initial state
   3. Goal state
   4. Operators/Actions
   5. Goal test: a boolean function over states, true if input is the goal state
      1. We need it to determine when the goal state is reached
   6. Path cost function: assign a cost to a path/action
3. **Branching factors**:
   1. If there are b possible choices at each state, the branching factor is b.
   2. If it take d steps (state transitions) to get to the goal state, then it may be the case that O(bd) states have to be checked.
4. **Search algorithm**:
   1. A search algorithm takes a problem as input, formulated as
      1. a description of the world (the initial state)
      2. a set of possible actions (or operators) to be applied
      3. a set of goals to be achieved (can be expressed as goal tests)
   2. It returns a solution in the form of an action sequence or plan
   3. Once a solution is found, the chosen actions can be carried out
      1. the execution phase
   4. A solution is a sequence of actions leading from start to goal state
   5. Given a problem formulation, we must map out the state space
      1. while we are searching the state space it’s sometimes called the search space
   6. Generate action sequences, using a search strategy (agenda):
      1. start at initial state
      2. check whether in goal state
         1. if yes, we’re done
      3. otherwise, expand a state
         1. Applying the actions to the current state and thereby generating a new set of successor states
   7. Conceptually, the search process builds up a search tree that is superimposed over the state space
      1. Root node of the tree ↔ Initial state
      2. Arcs ↔ Actions
      3. Leaves of the tree ↔ States to be expanded (or expanded to null)
      4. At each step, the search algorithm chooses a leaf to expand
   8. The state space and the search tree are not the same thing!
      1. A state represents a (possibly physical) configuration
      2. A search tree node is a data structure which includes:
         1. constitutes part of a search tree
         2. { parent, children, depth, path cost }
      3. States do not formally have parents, actions, children, depth, or path cost!
      4. Number of states is not equal to number of nodes
   9. *Uninformed (blind) search* 
      1. Can only distinguish goal state from non-goal state
   10. *Informed (heuristic) search* 
       1. Can evaluate states
5. **Breadth First Search**:
   1. All nodes at depth *d* in the search tree are expanded before any nodes at depth *d+1*
      1. First consider all paths of length *N*, then all paths of length *N+1*, etc.
   2. Doesn’t consider path cost – finds the solution with the shortest path
   3. Uses First In First Out (FIFO) queue  
        
      Example,  
        
      An undirected graph with 6 nodes ranging from A to F where,
      1. A points to B and C
      2. B points to D
      3. C points to B, D and E
      4. D has no edge
      5. E points to F
      6. F has no edge

The search tree becomes a directed and ordered search tree where,

1. The root is A
2. The left node B of the root is connected to a instance of D
3. The right node C is connected to separate instance of B, D and E, of which B is connected to D, D does not have an edge on both levels and E is connected to F which has no edge.

The Queue representation becomes:

(A) --> (B C) --> (C D) --> (D B D E) --> (B D E) --> (D E D) --> (E D) --> (D F) --> (F) -->

( )

1. **Uniform Cost Search**: Like breadth-first search, but always expands the lowest-cost node, as measured by the path cost function, g(n)
   1. g(n) is (actual) cost of getting to node n
   2. Breadth-first search is a special case of uniform cost search, where g(n) = DEPTH(n)
   3. If the path cost is monotonically increasing, uniform cost search will find the optimal solution
2. **Depth First Search**: Always expands one of the nodes at the deepest level of the tree
   1. Low memory requirements
   2. Problem: depth could be infinite
   3. Uses a stack (LIFO)

The Stack representation becomes:

(A) --> (B C) --> (D C) --> (C) --> (B D E) --> (D D E) --> (D E) --> (E) --> (F)

* 1. Pros: For problems with many solutions, DFS may be faster than BFS because it has a good chance of finding a solution after exploring only a small portion of the whole space
  2. Cons: can get stuck down a wrong search path

1. **Depth Limited Search**:
   1. Like depth-first search, but uses a depth cut-off to avoid long (possibly infinite), unfruitful paths
      1. Do depth-first search up to depth limit l
      2. Depth-first is special case with limit = infinity
2. **Iterative Deepening Search (IDS)**:
   1. Since the depth limit is difficult to choose in depth-limited search, use depth limits of l = 0, 1, 2, 3, …
   2. Do depth-limited search at each level
   3. IDS has advantages of
      1. Breadth-first search – similar optimality and completeness guarantees
      2. Depth-first search – Modest memory requirements
   4. This is the preferred blind search method when the search space is large, and the solution depth is unknown
3. **Bidirectional Search**:
   1. Both actions and predecessors (inverse actions) must be defined
   2. Must test for intersection between the two searches
   3. Really a search strategy, not a specific search method
4. **Tree Search** 
   1. Repetition of some states
   2. Do not need to remember states
5. **Graph Search**
   1. Remember all the states that have been explored
   2. Do not repeat some states
6. **Informed (heuristic) search**:
   1. Heuristic:
      1. “Heuristic” means “serving to aid discovery”
      2. A rule of thumb to find answers
      3. A shortcut to make a decision
      4. It helps estimate the quality or potential of partial solutions
      5. It helps when no algorithmic solution is available
7. **Greedy Best-First Search**:
   1. Greedy search – always expand the node that appears to be the closest to the goal (i.e., with the smallest h)
      1. Instant gratification, hence “greedy”
      2. The simplest kind of informed search estimates the cost from the current node to the solution and uses that to sort its agenda
      3. Most greedy search has an agenda of length 1
         1. all but the best node is thrown away, at every cycle
         2. the shorter the agenda, the more likely the search is to get caught in local minima, where there is a temporary dip in the solution cost
      4. throwing away solutions means you can’t recover if you go astray
         1. but it can be implemented with an agenda of any length
      5. In the route-finding problem, a good heuristic is the straight-line distance to the goal
         1. “as the crow flies”
         2. this idea, that a geometrical distance estimates a distance along a graph, is the basis of many heuristics
      6. But because it doesn’t account for cost so far, it can lead us astray
8. **A\* Search**:
   1. Uniform cost search minimizes g(n) (“past” cost).
   2. Greedy search minimizes h(n) (“expected” or “future” cost).
   3. “A\* Search” combines the two:
      1. Minimize f(n) = g(n) + h(n)
      2. Accounts for the “past” and the “future”
      3. Estimates the cheapest solution (complete path) through node n
   4. Focus on optimality (finding the optimal solution).
   5. “A\* Search” is optimal if h is admissible.
      1. h is optimistic – it never overestimates the cost to the goal
      2. h(n) ≤ true cost to reach the goal
   6. So, f (n) never overestimates the actual cost of the best solution passing through node n.
   7. The more accurate h is, the fewer unnecessary nodes will be expanded.
9. **Cost of algorithms**:
   1. One way to characterise the quality of a heuristic is the effective branching factor (b\*)
      1. if
         1. the total number of nodes expanded by A\* for a particular problem is N
         2. the solution depth is d
      2. then
         1. b\* is the branching factor that a uniform tree of depth d would have to have to contain N nodes
      3. N =1+b\*+(b\*)2 +...+(b\*)d
         1. e.g., if A\* finds a goal at depth 5 using 52 nodes, the effective branching factor is 1.91
      4. Usually, the effective branching factor exhibited by a given heuristic is constant over a large range of problems
         1. This can be used for empirical evaluation of a heuristic’s usefulness