**Artificial Intelligence in Games**

**Session 1**

1. **Search**: Search is the process of navigating from a starting state to a goal state by transitioning through intermediate states. Therefore, Search/State Space can be defined as the set of possible configurations a problem can be in.  
     
   For example. The N-Queens problem. The goal is to place N queens in a N-by-N board so that none of them threatens any other. A possible representation of potential solutions could be an array of 8 numerical values that indicate the positions of the queens. The state space of the N-Queens problem is the possible configurations of N-queens on an N-by-N board. The size of the state space can be calculated mathematically by dividing the factorial of the number of queens N by the product of the factorial of the number of queens on the board K multiplied with the factorial of N – K.
   1. *Tree Search*: Tree Search is a category of search methods that uses a tree structure to navigate through possible solutions. In Tree Search, the root of the tree represents the state where the search starts. Each edge from this state is a decision (or, in a game, an action a player takes) that leads to another state in the search space.
   2. *Uninformed Search*: In Uninformed Search (also called Blind Search), the algorithm performs search without any information about the goal. They are brute force algorithms: they explore the whole of the search space in a predetermined way until the goal is found. The most known are:
      1. Breadth-first search (BFS): is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.   
           
         *Advantage*: Very simple to implement and a complete solution.  
         *Disadvantage*: Time consuming and heavy memory requirements.
      2. Depth-first search (DFS): The algorithm starts at the root node and explores as far as possible along each branch before backtracking.  
           
         *Advantage*: Light memory requirements.  
         *Disadvantage*: Not a complete solution, can enter infinite loops.
2. **Uninformed Search Algorithm**: Dijkstra's algorithm is a classical pathfinding algorithm that is very efficient (polynomial runtime). It finds the shortest path from a node to any other node in the graph. It is a variation of breadth first search, which orders the next nodes to be visited based on their distance to the start node.   
     
   Dijkstra is complete and always finds the shortest path. Dijkstra's algorithms make use of a priority queue: a list that holds items with an associated priority. The list is always ordered such that the highest priority items are on top. The algorithm’s efficiency depends crucially on the implementation of the priority queue used.  
     
   To improve Dijkstra knowledge about the goal can be added. Dijkstra uses the distance travelled so far and utilises a priority queue to consider nodes in the graph with the smallest distance from the start. We would expect to perform better if we could also guess how close a node is to the target (for point-to-point distances).
3. **Best-First Search**: Best-First Search uses knowledge about the goal to conduct search. This is implemented in the form of a heuristic, which provides a way of establishing a preference in the order in which actions are explored from a given state. One of the most common Best-First Search algorithms is A-star, which is commonly used in games to determine the shortest path between two locations in a level. A-star uses a heuristic to estimate the utility of a node with respect to the target. This allows the algorithm to explore the graph towards the more promising regions.   
     
   A-star keeps a list of “open” nodes, reachable from the already explored nodes, but which that have not yet been visited. For each one of these “open” nodes N, an estimate of the distance to the goal is made. The next node/state to visit is the one with the smaller aggregated cost, which is the sum of the cost from the origin g and the cost to the destination h. This estimation is the heuristic: we use information about the goal to guide the search.  
     
   However, pathfinding algorithms can't work directly with the geometry that makes up the game level.   
     
   A graph needs to be constructed that can then be searched:
   * 1. Nodes represent points in 2D, or 3D space.
     2. Nodes have neighbours (adjacent nodes).
     3. Neighbouring nodes may have different (positive) distances / costs.
     4. Nodes (i.e., their locations) can be constructed manually or automatically.
4. **Statistical Forward Planning (SFP)**: Plan ahead by sampling future states.
   1. **Monte Carlo (MC) Search**: Monte Carlo methods are a class of computational algorithms that, by using repeated random sampling, approximate a value.  
        
      Structure of MC problem:   
      1. Given a domain of inputs   
      2. Generate them randomly from a probability distribution over this domain.   
      3. Perform a computation over these inputs, aggregating results.  
        
      This type of sampling can be used to approximate the game-theoretic value of a move. In the context of search, we can refer to the Q-value of a move a in a state s as the quality of this move in the game. The Q-value of an action can be mathematically defined as the product of 1 divided by the number of actions that were selected multiplied with the sum of the reward at that state.
   2. **Flat MC Search**: The process of determining an action by randomly sampling them with Monte Carlo simulations. It is possible to make this method most reliable by biasing action selection using the information gathered from the simulations.
   3. **Reflexive MC Search**: Reflexive MC applies MC searches within other MC searches.
   4. **Nested MC Search**: A very similar variant to Reflexive Monte Carlo Search is Nested Monte Carlo Search (NMCS). In NMCS, every sequence of moves explored is compared to the best sequence of moves ever found and, if it happens to be better, it becomes the new the best sequence.

**Summary**:

1. Nesting Monte Carlo sampling enhances search at the expense of computational time
2. Keeping the best solution found so far helps! Random process, we don’t want to lose it.
3. Uniformly sampling works, biased sampling can help.
4. Generality: Search is general: no previously provided game-based knowledge guides the search.