**Artificial Intelligence**

**Session 5**

1. **Optimal decisions in adversarial situations**:
   1. Search with no adversary
      1. Solution is a set of actions for reaching goal
      2. Heuristics and constraint satisfaction techniques can find optimal solution
      3. Evaluation function: estimate of cost from start to goal through given node
      4. Examples: path planning, scheduling, solitaire games
   2. Games – with adversary
      1. Solution is a strategy (considering every possible opponent reply to a move)
      2. Sometimes time limits force an approximate solution
      3. Evaluation function: evaluate quality of game position
      4. Examples: chess, checkers, Othello, war games, simulations of competition for limited resources
2. **Minimax algorithm**:
   1. Game setup
      1. Two players: MAX and MIN
      2. MAX moves first (by convention) and they take turns until the game is over
      3. Perfect information: The State is known to everyone
      4. Deterministic: Nothing is random
      5. Winner gets prize, loser gets penalty (same size as prize) → zero-sum
   2. S0: Initial state: e.g. board configuration of game
   3. PLAYERS(s): Returns which player has the move
   4. ACTIONS(s): Returns the legal moves
   5. RESULT(s, a): Output the state we transition to
   6. TERMINAL-TEST(s): Returns True if the game is over
   7. UTILITY(s, p): The payoff of player p at terminal state s
   8. E.g. MAX wins (+1), loses (-1) or draws (0) in Tic-Tac-Toe
3. **Game trees:**
   1. A game tree is like a search tree in many ways …
      1. nodes are search states, with full details about a position
         1. characterize the arrangement of game pieces on the game board
      2. edges between nodes correspond to moves
      3. leaf nodes correspond to a set of goals
         1. { win, lose, draw }
         2. usually determined by a score for or against player
      4. at each node it is one or other player’s turn to move
   2. A game tree is not like a search tree because you have an opponent.
   3. Optimal strategies:
      1. Find the contingent strategy for MAX against opponent MIN
      2. Assumption: Both players play optimally
         1. do not assume player will miss good moves or make mistakes
      3. Given a game tree, the optimal strategy can be determined by using the minimax value of each node, n:
         1. if n is a terminal node, MINIMAX-VALUE(n) = UTILITY(n)
         2. if n is a MAX node, MINIMAX-VALUE(n) = MINIMAX-VALUE(s)
         3. if n is a MIN node, MINIMAX-VALUE(n) = MINIMAX-VALUE(s)
4. **Minimaxing**:  
     
   Suppose we have a tree with node as 3 and three choices (left node = 3), (right node = -8).  
   The left node has two choices attached 7 and 3; the right node also has two choices -8 and 50.  
   1. Algorithm maximises the worst-case outcome for MAX
   2. Your opponent will choose smaller numbers
   3. If you move left, your opponent will choose 3
   4. If you move right, your opponent will choose -8
   5. Thus, your choices are only 3 or -8
   6. You should move left
5. **Minimax search**:
   1. The minimax decision maximizes the utility under the assumption that the opponent seeks to minimize it (if it uses the same evaluation function)
   2. Generate the tree of minimax values
      1. Then choose best (maximum) move
      2. Don’t need to keep all values around
         1. Good memory property
   3. Depth-first search is used to implement minimax
      1. Expand all the way down to leaf nodes
      2. Recursive implementation
   4. Algorithm:
      1. function MINIMAX-MaxPlayer(state) returns an (action, utility) pair
         1. input: state - current state of game
      2. if terminal-Test(state) then return (null, utility(state))
      3. best := (null, -infinity)
      4. for a in actions(state) do // actions() returns all legal moves from a state
         1. value := MINIMAX-MinPlayer(makeMove(state, a)).utility
         2. if value > best.utility then best := (a, value) •
      5. return best
   5. Definition of optimal play for MAX assumes MIN plays optimally
      1. maximises worst-case outcome for MAX
   6. But what happens if MIN plays worse than optimally?
      1. MAX will do even better
   7. Complexity:
      1. Time: O(b^m) - exponential in depth of tree
      2. Space: O(bm) - linear in depth of tree
   8. Exact search is intractable
      1. Idea 1: Pruning
      2. Idea 2: Cut off early and use a heuristic function
6. **Alpha-beta pruning**:
   1. The idea: don’t expand “dead-end” nodes!
      1. “smarter,” more efficient search •
   2. Pruning – eliminating a branch of the search tree from consideration
   3. Alpha-beta pruning, applied to a minimax tree, returns the same “best” move, while pruning away unnecessary branches
      1. Many fewer nodes might be expanded
      2. Hence, smaller effective branching factor
      3. …and deeper search
      4. …and better performance
         1. Remember, minimax is depth-first search
   4. Alpha pruning:
      1. alpha is the best minimax value that MAX currently can guarantee at that level or above.
   5. Beta pruning:
      1. Beta is the best minimax value that MIN currently can guarantee at that level or above.
   6. Use the same algorithm as before, but consider ranges instead of values
      1. range is expressed as a pair [alpha, beta]
   7. Alpha-beta algorithm:
      1. Initial call to the algorithm is: ALPHA-BETA-MaxPlayer (state, -infinity, infinity)
      2. function ALPHA-BETA-MaxPlayer(state, alpha, beta) returns an (action, utility) pair
         1. input: state - current state of game
         2. if terminal-Test(state) then return (null, utility(state))
         3. best := (null, -infinity )
         4. for a in actions(state) do
            1. value := ALPHA-BETA-MinPlayer(makeMove(state, a), alpha, beta).utility
            2. if value > best.utility then best := (a, value)
            3. if value >= beta then return (null, value)
            4. if value > alpha then alpha := value
         5. return best
      3. function ALPHA-BETA-MinPlayer(state, alpha, beta) returns an (action, utility) pair
         1. input: state - current state of game
         2. if terminal-Test(state) then return (null, utility(state))
         3. best := (null, infinity )
         4. for a in actions(state) do
            1. value := ALPHA-BETA-MinPlayer(makeMove(state, a), alpha, beta).utility
            2. if value < best.utility then best := (a, value)
            3. if value <= alpha then return (null, value)
            4. if value < beta then beta := value
         5. return best
      4. General alpha-beta pruning:
         1. Consider a node n somewhere in the tree
         2. If player has a better choice at
            1. Parent node of n
            2. Or any choice point further up
         3. n will never be reached in actual play
         4. Hence when enough is known about n, it can be pruned.
      5. Properties:
         1. Pruning does not affect final results (directly)
         2. Entire subtrees can be pruned
         3. Good move ordering improves effectiveness of pruning
         4. With “perfect ordering”, time complexity is O(b^{m/2})=O((b^{1/2})^{m})
            1. Branching factor of sqrt(b)
            2. Best case alpha-beta pruning can look twice as far ahead as plain MINIMAX in a similar amount of time
         5. With “random ordering”, time complexity is O(b^{3m/4})
         6. Repeated states are again possible
            1. Store them in memory; make large gain in memory from pruning
7. **Incomplete game trees**:
   1. For many games, MINIMAX and alpha-beta pruning require too many leaf-node evaluations
   2. Searching to terminal states will often be impossible within a reasonable amount of time
   3. Shannon (1950): ‣
      1. Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
      2. Apply heuristic (or static) evaluation function EVAL (replacing UTILITY function of MINIMAX)
8. **Heuristic evaluation**:
   1. Idea:
      1. produce an estimate of the expected utility of the game from a given position
   2. Algorithm performance depends on quality of EVAL
   3. Requirements:
      1. Computation may not take too long
      2. EVAL should value terminal states in the same order as UTILITY
      3. For non-terminal states, EVAL should be strongly correlated with the actual chance of winning
      4. Ideally should only be used for quiescent states (=no wild swings in value in near future)
   4. A simple evaluation function for Tic-Tac-Toe
      1. Count the number of rows where X can win
      2. Subtract the number of rows where O can win •
   5. Value of evaluation function at start of game is zero
      1. on an empty game board there are 8 possible winning rows for both X and O
   6. evalX = (number of rows where X can win) – (number of rows where O can win)
      1. After X moves in centre, score for X is +4
      2. After O moves, score for X is +2
      3. After X’s next move, score for X is +4
   7. evalO = (number of rows where O can win) – (number of rows where X can win)
      1. After X moves in centre, score for O is -4
      2. After O moves, score for O is -2
      3. After X’s next move, score for O is -4
9. **Cutting-off search**:
   1. Change the termination condition of MINIMAX:
      1. if TERMINAL-TEST(state) then return UTILITY(state)
   2. into
      1. if CUTOFF-TEST(state, depth) then return EVAL(state)
   3. Can introduce a fixed or dynamic depth limit
      1. selected so (e.g.) time will not exceed what the rules of the game allow
   4. When cut-off occurs, and we are not at a terminal state, heuristic evaluation is performed ‣
      1. in other words, if you don’t know, then take your best guess
10. **Multiplayer games:**
    1. Replace single zero-sum utility function with a function for each player
    2. Use a vector of values, one for each player
    3. A tree with root node as (1, 2, 6) and left and right nodes with similar vector syntax.
    4. In backgammon, you roll 2 dice, and then use both numbers in either order
       1. so a double (e.g. [1,1]) has probability 1/36; all other rolls have probability 1/18
    5. In this tree, we calculate the expected value of MINIMAX
11. **Expected Minimax value**:
    1. An expected value is a weighted average, where weights are probabilities
    2. Given a game tree, the optimal strategy can be determined by using the minimax value of each node, n:
       1. if n is a terminal node,
          1. EXPECTED-MINIMAX-VALUE(n) = UTILITY(n)
       2. if n is a MAX node,
          1. EXPECTED-MINIMAX-VALUE(n) = EXPECTED-MINIMAX-VALUE(s)
       3. if n is a MIN node,
          1. EXPECTED-MINIMAX-VALUE(n) = EXPECTED-MINIMAX-VALUE(s)
       4. if n is a CHANCE node,
          1. EXPECTED-MINIMAX-VALUE(n) = P(s) EXPECTED-MINIMAX-VALUE(s)
          2. where P(s) is the probability of outcome s
12. **Past paper question**:
    1. Does alpha-beta pruning result in a better game-playing agent? Explain your answer carefully. [4 marks]
       1. (A) Mostly yes since it reduces the number of tree’s search improving computation time and prunes trees, comparing the alpha and beta (alpha min, beta max) values if alpha is bigger than beta then the tree is pruned, however it does not search all branches if they are pruned the pruned branch could possess a better move for the max or min value however they are not taken into account.
       2. (B) No, alpha-beta pruning merely reduces the number of nodes we must visit. It produces the same outcome as minimax. It is therefore not a better game-playing agent but one that should perform better with regards to time taken and memory required.
       3. (C) Alpha-beta pruning results in a better game-playing agent because it improves the speed of the agent’s decision-making. This is important in adversarial search, as there may be a time element involved in the game. Without alpha-beta pruning, the minimax algorithm naively searches the entire game tree, possibly exploring whole subtrees that will never be used because they will return a utility value larger than their parent’s upper bound or lower than their parent’s lower bound. Alpha-beta pruning detects if the algorithm has encountered such a subtree, and saves the algorithm a significant amount of time by pruning the branch and moving on.