Juan Barrera

Professor Mark Voortman

Introduction to AI

Module 2 Assignment (Part B)

Sep 29 2025

Questions:

Discuss how well the standard approach to game playing would apply to games such as tennis, pool, and croquet, which take place in a continuous physical state space.

The standard "AI" approach to game playing, which views the problem as one of constructing an optimal search tree, works well for board games such as chess or checkers because states and actions are discrete and the state space is not excessively large. The state of the game is defined in an enormously broad physical space for games like tennis, pool and croquet. Similarly, actions are likewise continuous: how one hits the pool ball or swings the racket in tennis. As a result, it is impossible to enumerate all possible actions and states because their number (called branching factor) becomes infinite. What methods are employed to find an alternative solution? One of the methods is method is discretization, this is where the continuous parameters (like shot angle and force) are limited to a finite set of values. However, this simplification can either make the problem trivial or result in too many combinations to be feasible. Simulation-based approaches, where a range of candidate actions is evaluated by using a physics model to predict the consequences of each action. More sophisticated methods implement progressive widening in Monte-Carlo tree search, which begins with a limited subset of actions and methodically expands it as the search progresses. In practice, an instrument can be set up at high levels for strategy (such as playing styles) discretely and at low levels for the actual implementation of an action through some form of continuous-space optimization or reinforcement learning method. Therefore, techniques borrowed from or inspired by the minimax method (such as discretization of action spaces) are applicable to games with continuous actions, even though minimax cannot.

Describe how the minimax and alpha beta algorithms change for twoplayer, non-zero-sum games in which each player has a distinct utility function and both utility functions are known to both players. If there are no constraints on the two terminal utilities, is it possible for any node to be pruned by alpha–beta?

Minimax works in zero-sum games because maximizing one player's utility is equivalent to minimizing the other's. Alpha–beta pruning derives tight upper and lower bounds from this equivalence, allowing vast parts of the search tree to be pruned. In a non zero sum game, their utility functions differ; hence, maximizing the payoff of one does not minimize that of another. The game tree is solved by backward induction, wherein the mechanism of action at one node is for the player to select the option that maximizes his or her own utility, with the assumption that the opponent will do the same in subsequent steps. This can generate a subgame perfect equilibrium but not minimax values (a single scalar). For pruning, alpha–beta cannot be directly applied because there are no scalar α and β values shared between players; rather, a node has a vector of utilities (u1,u2,…). There is no general monotonic relationship that would allow discarding a branch of the tree based on the information obtained from another branch (e.g., if one position is "bad" for player 1 then some other position can be considered irrelevant). Pruning can be possible in certain situations. For example, if utilities can be represented as a single known weighted sum (a common utility function) or if one branch is Pareto-dominated by another (both players' utilities are less or equal and at least one strictly less), it is possible to eliminate branches. But in the fully general case (arbitrary independent utilities), no pruning is possible. Thus, although backward induction generalizes in a straightforward manner, alpha–beta pruning does not unless very strong assumptions on the utility structure are imposed.

Suppose you have a chess program that can evaluate 5 million nodes per second. Decide on a compact representation of a game state for storage in a transposition table. How many entries can you fit in a 1-gigabyte inmemory table? Will that be enough for the three minutes of search allocated for one move?

A chess engine that searches 5 million positions per second for three minutes will search approximately 900 million nodes. To avoid searching duplicated positions once more, engines make use of transposition tables, which store previously computed results. A small entry typically holds a Zobrist hash key (64 bits, 8 bytes) to uniquely identify the position, and some additional fields: the best move found from that position (typically 2 bytes), the evaluation score (2 bytes), the search depth to which the node was searched (1 byte), a node

type flag indicating whether the value is exact, a lower bound, or an upper bound (1 byte), and typically an age or generation counter to assist in replacement policies (1 byte). With alignment padding, this is usually 16 bytes per entry, a common layout for real-world engines. A 1 GB table (1,073,741,824 bytes) can therefore hold approximately 67 million entries. This is to be contrasted with the 900 million nodes generated in three minutes, so the table cannot hold all visited nodes; in fact, only on average about 1 in 13 nodes could be held at any one time. This does not make the table useless, however: a great many nodes in a chess search tree are transpositions (positions resulting from different move orders), so storing even 67 million distinct states includes a large proportion of useful positions and eliminates a vast amount of redundant calculation. Engines also use replacement strategies (e.g., displacing shallow nodes with deeper ones, or older entries with more recent ones) to maximize the use of finite table space. Practically speaking, a 1 GB transposition table is very big and very good for a 3-minute search, even if it cannot store all 900 million nodes.