Study Notes for AI (CS 440/520)
Lecture 4: Adversarial Search

Corresponding Book Chapters: 5.1-5.2-5.3-5.4-5.5
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes in or any other issues with the notes to the instructor.

Review of Previous Material

In the previous lectures we explored uninformed search techniques, i.e., algorithms that search the state space of a problem for a solution and have available only the successor function and the cost of actions. Examples of these techniques include breadth-first search and depth-first search.

Then we moved on to informed search, a type of search that uses heuristic information to improve performance. A heuristic is a function $h(n)$ that attempts to estimate for every state n the cost of the path from that state to the goal.

The basic strategy for informed/heuristic search is **best-first search**. In best-first search, an evaluation function $f(n)$ is defined. Best-first search selects as the next node to expand the node which has the optimum value according to $f(n)$. There are two popular ways to implement best-first search:

1. The simplest form of best-first search is **greedy best-first search**, where the evaluation function is equal to the heuristic one: $f(n) = h(n)$. If the heuristic is a very good estimate of the true path cost then greedy best-first search will work well. In the general case, however, greedy best-first search is an incomplete and suboptimal algorithm.
2. An alternative form of best-first search is the **A* algorithm**, where the evaluation function is defined as: $f(n) = g(n) + h(n)$, where $g(n)$ is the true path cost from the root of the tree to the current node n. A* search is complete, optimal and optimally efficient for a given heuristic, as long as the heuristic satisfies certain requirements (i.e., admissibility in the case of `TREE-SEARCH` and consistency in the case of `GRAPH-SEARCH`.

1 Adversarial Search

Consider agents with conflicting objectives, where one agent is working against the other. This introduces adversarial search problems, or games. A "game" in A.I. is a multi-agent environment where agents are working against one another, and they significantly impact each other. Here, we will focus on a popular specialized setup:

* deterministic, turn-taking, two-player problems where the utility values of the two agents at the end of the game are always equal and opposite.

Game theorists call them "zero-sum games of perfect information". Games are different than previous search problems because we no longer look for the sequence of actions that will lead to a goal state. Now we have to take into account that between our actions the opponent is also taking actions that effect our evaluation function. In adversarial search we focus on finding an **optimal strategy** that will allow our agent to maximize the evaluation function. An optimal strategy in a game leads to outcomes at least as good as any other strategy when playing against an infallible opponent.

Examples of Related Games:

- Checkers
- Othello
- Chess
- Backgammon
- Go

It is infeasible to compute an optimal strategy for most realistic games because the search trees are very large and often too hard to solve exactly. In this lecture we will study ways to efficiently approximate the optimal strategy.

1.1 Minimax Algorithm

Consider a game with two players called MAX and MIN, where MAX moves first and then the players alternate their moves. At the end of the game the winner receives a positive utility and the loser a negative one. The Minimax algorithm assumes that the opponents plays optimally and finds the optimal response given this assumption.

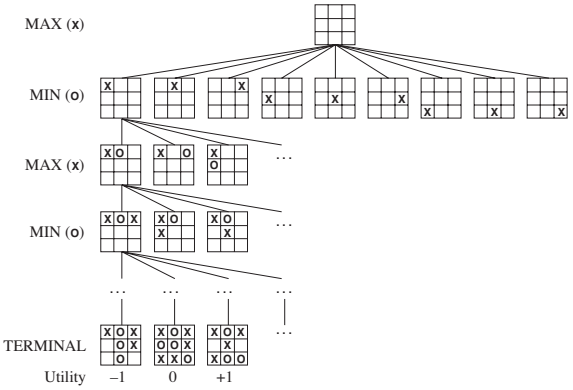

Figure 1: A partial search tree for tic-tac-toe

To evaluate the optimal strategy in the minimax algorithm a game tree is created. Figure 1 shows the search tree for the game of tic-tac-toe. Every node in this tree is a assigned a minimax value.

The most important difference between a traditional search tree and a game tree is that in the second case we need additional levels in order to express the actions of the opponent. Figure 2 provides an example. A level of the tree that corresponds to the actions of our agent is called a `MAX` level, because the agent tries to maximize the evaluation function. A level that corresponds to the actions of the opponent is called a `MIN` level, since the opponent will try to minimize the evaluation function. The `MAX` and `MIN` levels alternate, as the actions of the two players are also alternating. These separate levels are referred to as "plies".

The leaf nodes of a game tree are called the terminal nodes and correspond to states where we can assign a value. For example, if our agent is winning at a particular terminal state then the value of that state could be 1, while it is -1 if our agent is loosing at that terminal state. In order to be able to compute the optimal strategy, we should compute the value of the intermediate states as well. These values are called "minimax" values and are computed using the assumption that the opponent is an intelligent agent and plays to win. The minimax value of a node is computed as follows:

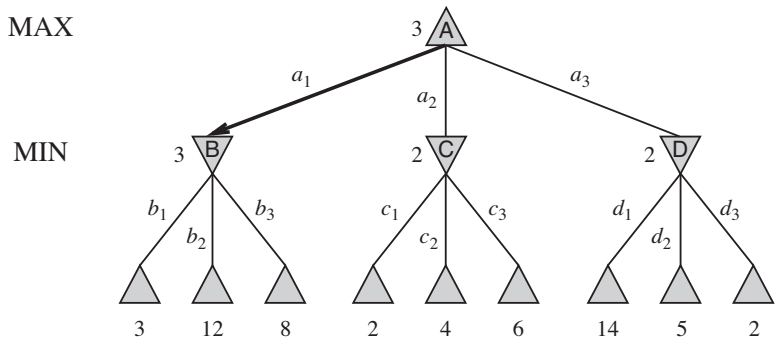

Figure 2: A game search tree.

$$Minimax - Value(n) = \begin{cases} Utility(n) & \text{n: terminal state} \\ \max_{s \in succ(n)} MV(s) & \text{n: max node} \\ \min_{s \in succ(n)} MV(s) & \text{n: min node.} \end{cases}$$

The minimax algorithm selects the sequence of actions that maximizes the minimax-value of the root node of the search tree. The minimax values can be computed by a depth-first traversal of the search tree. Figure 2 provides an example of its operation. Starting at the first node of the `MIN` level, node B, we check the three terminal nodes below it: 3, 12, 8. The minimum of those is 3, which is returned to node B. The same is done for nodes C and D, which return 2 and 2. Now at node A, the `MAX` level, we pick the maximum of our returned `MIN` values, which is 3. This represents our best choice after our opponent picks the action which is worse for us.

It must be noted that if the opponent does not play optimally, then our agent can potentially do better than the minimax value of the root node. In this case, suboptimal strategies for the `MAX` player may do even better than the minimax algorithm, but are guaranteed to do worse against optimal opponents.

Because the minimax search is a depth first search, it has the following properties:

Time complexity: $O(b^m)$
Space complexity: $O(bm)$

where b is the possible number of moves at each iteration of the game (branching factor) and m is the maximum depth in the tree or the maximum number of moves in the game.

1.2 Alpha-beta Pruning

The time complexity of minimax is exponential, so this search technique would not be practical for a game like chess with a large number of moves. Chess has an average branching factor of 35, and games usually average around 50 moves per player. This means the tree would have 35^{100} or 10^{154} nodes!

We can improve this algorithm by using alpha-beta pruning. Alpha-beta is a way to not consider nodes that are not worth considering. Looking at Figure 2, if we are at the second step (`MIN` level of the tree) and are evaluating node C, we first check node c_1 and get a value of 2 returned. This value is already less than the returned value of 3 from node B, so we know that the max step for node A will never pick an action that leads to a lower value (value 2). At this point we can move on to node D, effectively skipping the remaining children of node C (nodes c_2 and c_3).

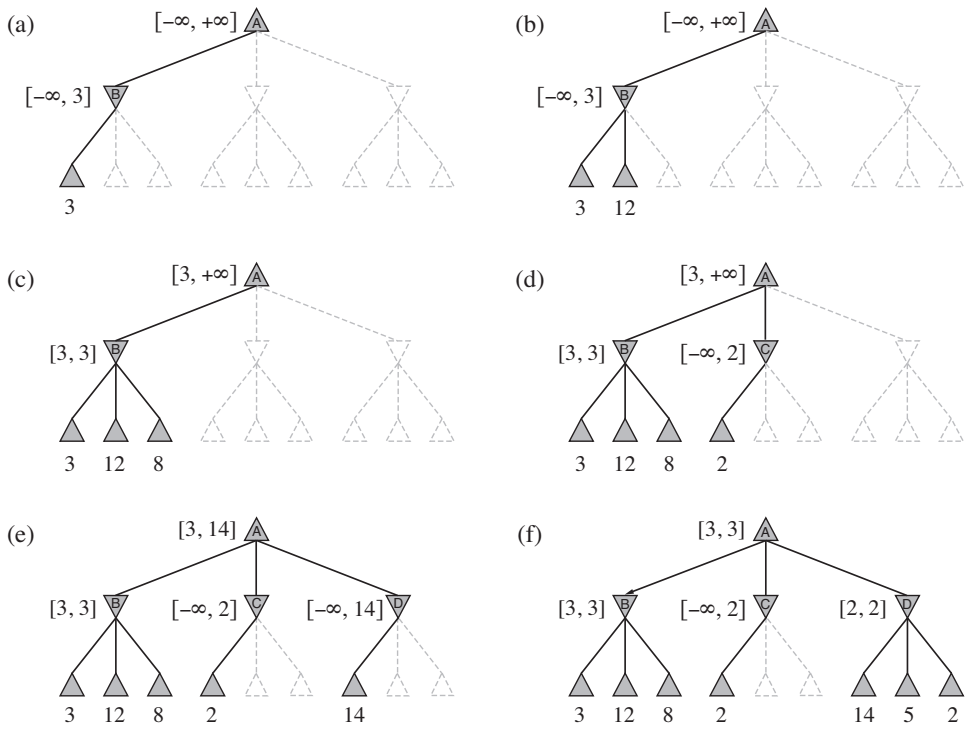

Figure 3: Alpha-beta pruning.

Figure 3 shows an example of the operation of alpha-beta pruning. This time the nodes of the tree store two values, the alpha and beta values, which correspond to the maximum and minimum bounds for the minimax value for the node. These bounds are used to distinguish when to skip nodes. For example, once the maximum value for node C is lower than minimum value of node B, we skip C's remaining children.

Note that alpha-beta pruning is only effective when the nodes are in "good order". In the above example, the terminal nodes below node C are arranged so that the minimum value, 2, is returned first. If the order was 6, 4, 2, then 6 would still be a possible candidate for the minimax choice of node A. This means we would still have to check all three terminal nodes.

The computation below shows that even if the values 4 and 6 of nodes c_2 and c_3 are unknown (values x and y) the algorithm can still safely compute the minimax value:

$$\begin{aligned} minimax(root) &= max(min(3,12,8), min(2,x,y), min(14,5,2)) \\ &= max(3, min(2,x,y), 2) \\ &= max(3,z,2) \qquad \text{(and since } z \leq 2) \quad = 3 \end{aligned}$$

If the nodes are in a good order, we can cut the exponential growth of the time complexity in half.

Time (perfect pruning): $O(b^{m/2})$ Time (average): $O(b^{3m/4})$

For games such as chess, there is an established priority of moves:

- try captures first
- try threatening moves
- forward moves
- backward moves

Since chess has a good order of moves, alpha-beta pruning is effective in decreasing the time complexity. Still, however, with alpha-beta pruning we have to search down to the terminal nodes of the search tree.

1.3 Heuristic Adversarial Search

To use minimax effectively in realistic games, the search must be cut off early and a heuristic evaluation function should be applied on non-terminal nodes. This evaluation function replaces the utility function and returns an estimate of the distance to the goal.

The evaluation function should:

- be fast to compute
- order terminal states the same way that the utility function would
- correlate to the actual chances of winning for non-terminal states

1.3.1 Computing Heuristics for Minimax - Test Case: Chess

In chess, game pieces are given a value to calculate the advantage for each player. By adding up these values for the pieces left on the board, a player can get a score to evaluate the current state. A larger score indicates a better chance at winning the game.

- pawn: 1
- knight/bishop: 3
- rook: 5
- queen: 9
- good pawn structure: 0.5
- king safety: 0.5

The number of pieces on the board can be considered as "features" f_i and their points can be viewed as weights f_i in our evaluation function. Then the evaluation function for this problem would look like this:

$$EVAL(s) = \sum_i w_i f_i(s)$$

But how deep should we search before cutting off?

1. We could pick a predefined threshold
2. A better solution is to apply iterative deepening.

Unfortunately because the evaluation function is only an estimation of the true minimax value, the cut off might lead to a problematic situation. For example, this can occur when the next move after the cutoff state has the potential to considerably change the minimax value.

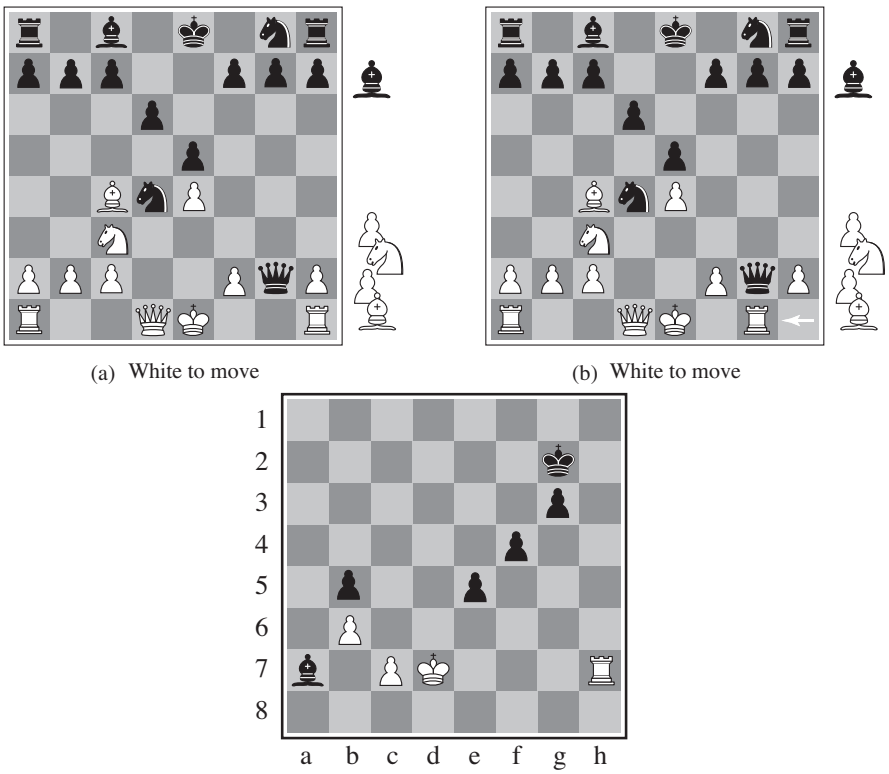

Figure 4: States that can cause problems when cutting off the evaluation of the minimax value.

Such a situation is shown in Figure 4. The two displayed states to the left are the same in terms of the available pieces on the board. The difference is that the rook in Figure b) is threatening the queen. The state a) is a state where the black has a considerable advantage and the b) state is a state where the white has the advantage. Nevertheless, the evaluation function defined above would score both states the same way. The feature of the b) state is that the next action is going to considerably change the value of the state (removing the black queen from the board). The above example implies that the cutoff procedure should only be applied on **quiescent** states, states where large changes in value, are not likely to happen.

There is an issue, however, with the **horizon** of the search process. Drastic changes may be unavoidable, but are hidden deep in the search tree. This is the case in the right side of Figure **??**. The black has the advantage according to the evaluation function and is the next one moving. The optimum move is to threaten the white king. Nevertheless, if both players play optimally after 12 moves the white will be able to get a new queen from the white pawn at the top. This will considerably alter the evaluation function in favor of the white player. This change in the evaluation function is inevitable, nevertheless it is hidden deeply in the search tree. Heuristics are also needed to address such horizon problems.

2 Games with Chance

In a game like backgammon (Figure 5) there is also the element of chance. The tree for games with chance must include an additional ply with `CHANCE` nodes as well as `MIN` and `MAX` nodes. Since there is no way to know the outcome of the dice rolled, only the expected value can be calculated where the expectation is taken for all possible outcomes of the dice. This is reflected in the evaluation function, which depends on the probabilities of rolling a specific outcome:

$$Expectiminimax(n) = \begin{cases} Utility(n) & \text{n: terminal state} \\ \max_{s \in succ(n)} EMMV(n) & \text{n: max} \\ \min_{s \in succ(n)} EMMV(n) & \text{n: min} \\ \sum_{s \in succ(n)} P(s) * EMMV(s) & \text{n: change.} \end{cases}$$

$P(s)$ is the probability that the event s occurs. Due to the presence of probabilities in the computation, the name of the algorithm for computing the optimal strategy in games with chance is "Expectiminimax". Nevertheless, it still corresponds to a depth-first traversal of the tree.

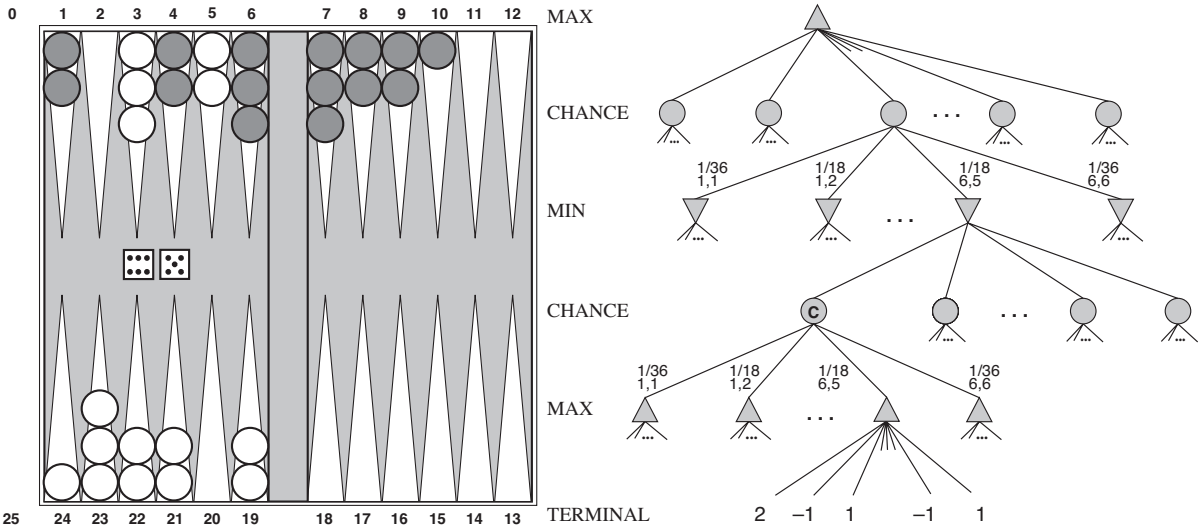

Figure 5: The game of backgammon and its game search tree.

For example, Figure 5 shows the different probabilities associated with the various `CHANCE` nodes. The event of bringing [1,1] has a probability of 1/36, while the event of bringing [1,2] has a probability 1/18.

In classic minimax if we scale the values of terminal nodes but we retain the same order, the algorithm will return the same optimal strategy. On the other hand, in games with chance, the scale of evaluation values at terminal nodes can have a large affect on the best choice for a move. Consider the tree in Figure 6 with leaves 1, 2, 3, and 4; A_1 is the best move. For the tree with leaves 1, 20, 30, 400, however, A_2 is the best move.

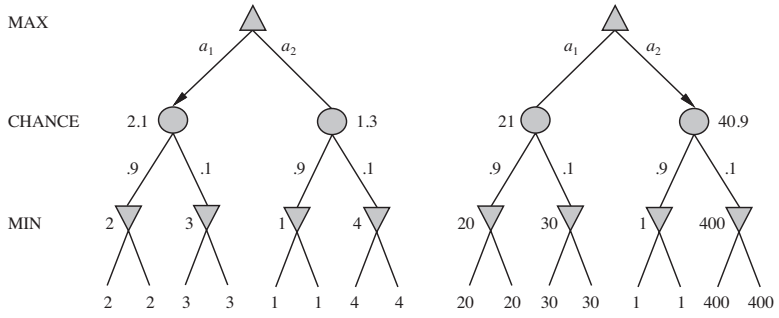

Figure 6: The scale of the utility at the terminal nodes is important for games with chance.

Complexity of Expectiminimax: $O(b^m n^m)$, where n is the branching factor of change nodes (e.g., the various alternative outcomes of the dice). This complexity is worse than regular minimax, meaning that it is more difficult to find optimal strategies for games with chance.

3 More than 2 Players

For games with additional players, a new ply is added to the tree for each player. Figure 7 provides an example of a search tree for a game with three players. Notice an additional difference compared to the two-player version of a search tree. On each node we must store more than one value. In the general case, we have to store a vector of size equal to the number of players. Each value in this vector represents the value of the node for each player. Beyond these differences, however, the basic principles of minimax and alpha-beta pruning still hold for multiplayer games.

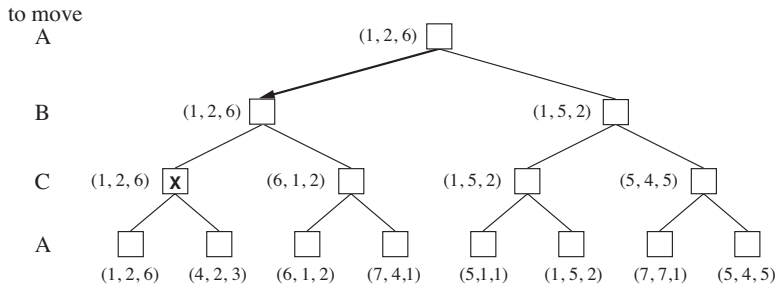

Figure 7: A multiplayer game search tree.