Implementing a Backgammon Player

Vinícius Miranda*
College of Computational Sciences, Minerva Schools at KGI
CS152: Harnessing Artificial Intelligence Algorithms

May 8, 2019

*The implementation is available at `https://nbviewer.jupyter.org/gist/viniciusmss/eb6a7903d49533a5b1ce009b49e6d6a8`

1 Problem Definition

The goal of this project is to design and implement a backgammon player. Backgammon is a stochastic, zero-sum game of perfect information where two players take turns. In the beginning of every turn, players roll two dice that determine how far they can move their pieces. Backgammon is not, however, solely about luck. Players can capture and have their pieces captured according to the disposition of checkers on the board. Hence, over several turns the effectiveness of one's strategy dominates the outcome of the game since the effect of chance is often averaged out.

Backgammon has several rules and exposing all the mechanics of the game is beyond the scope of this text. The reader is referred to Wikipedia (n.d.) for a full treatment of the game. For now, it suffices to address its core aspects. Players commence by rolling one die each. The higher numbered die determines who starts the game. Each player has 15 pieces which are distributed in a standard initial position along the 24 cones, or points, of the board (Figure 1). Players move in different directions and their pieces can move by the amount determined by the two dice they roll at the beginning of each turn. Players cannot move to cones that are taken by another player, with the exception of when there is only one checker of the opponent. In that case, moving to that cone entails capturing the opponent's piece. The opponent is impaired from moving any other checkers until they reintroduce to the game the captured piece in their next turn. Once all the pieces of a player are within their home board, they can start taking their pieces off the game. The player whose pieces have all been removed from the game wins.

The problem of implementing a backgammon player is challenging since the game's stochastic nature adds a non-trivial complication to common AI approaches. In deterministic games, algorithms can derive the game tree based on the utility of certain movements for all players, such as the well-known max and min in two-player games. In games involving chance, the game tree becomes substantially more complex since it needs to take account of all the possible outcomes of the chance aspect of the game. In backgammon, it entails accounting for all possible outcomes of rolling two dice. Not only does the branching factor of the tree increase quite dramatically, but standard algorithms needs to be modified to be able to handle these *chance nodes* in the tree. The next section will motivate these modifications.

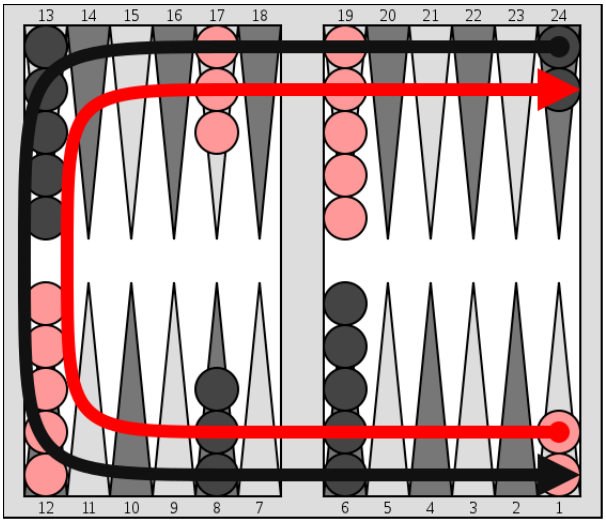

Figure 1: The initial position of backgammon displaying the direction of movement of the players. Source: Wikipedia (n.d.). *Public domain.*

2 Solution Specification

My first step was to design a working implementation of a backgammon game in Python. Since I considered the coding of AI methods instead of the game itself the focus of the assignment, I tried to find an available and working version of the game online. I thus chose to use the code provided in Aimacode (2018) as a starting point. The implementation, however, relaxes the game considerably by not implementing several of its aspects (e.g., by not allowing consecutive movements of the same piece in one turn). These relaxations are all acceptable since they do not change the fundamental dynamics of the game, expect for the lack of captures in this implementation. Without captures, the pieces can only move forward and the outcome of the game is almost completely determined by chance.

Therefore, I extended on Aimacode (2018) to account for the possibility of captures. The change is non-trivial since it affects many parts of the game, necessitating an extension to the state representation of the board, other checks for the legality of moves, handling of the reintroduction of pieces to the game, among other small modifications. The final version of the backgammon game can be succinctly described as follows.

1. **State representation.** A named tuple of five elements, consisting of (a) whose turn it is, (b) whether the state is a goal state, (c) the disposition of the board coded as a list of 25

size-2 dictionaries (corresponding to the 24 points and the number of captured pieces for each player), (d) all possible moves for the next player, and (e) the outcome of the dice roll.

2. **Initial state.** The disposition shown in Figure 1 coded according to the state representation above.

3. **Actions.** A tuple of two pieces which move according to the outcomes of the two dice, pairwise.

4. **Transition model.** Updates the state representation by executing the move, changing the player turn, and handling captures.

5. **Goal test.** Whether one of the players has removed all of their pieces from the game.

Now, we must address the question of how to enable an AI agent to play the game. Both the *minimax* and *alpha-beta search* algorithms provide useful stepping stones. As mentioned before, we must extended on these algorithms to enable them to handle the dice rolls in the game. These chance nodes modify the interpretation of a player node in the game tree. Since we are not certain of the outcome of the dice, we calculate the **expected** utility of a move by averaging across all possible scenarios (Russell & Norvig, 2016, p. 177). The algorithm which extends on *minimax* to handle chance nodes is called *expectiminimax* and its pseudocode is shown in Figure 2.

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

Figure 2: Pseudocode of the *expectiminimax* algorithm. Source: Russell and Norvig (2016, p. 178)

Both Aimacode (2018) and chanddu (2017) implement a version of the algorithm with a few shortcomings, the main of which is that neither implementation enables the AI agent to play as *min*. I thus start off from these implementations to build an extended version of the *expectiminimax*. The extensions include:

1. **Search cut off.** The high branching factor makes an exhaustive search of the game tree impossible. I thus implemented a cut off test based on the depth of the tree. Such modification necessitates an evaluation function to compare provide a comparative utility to different states, which I also implement.

2. **Alpha beta pruning of *max* and *min* nodes.** I allow *min* and *max* nodes to be pruned to reduce the cost of the search. Pruning of chance nodes is a non-trivial task and I do not pursue it.

3. **Playing symmetry.** I design the algorithm to work regardless of whether the AI is playing as *max* or *min*.

4. **Transposition table.** I implement a simple transposition table to cache the results of the evaluation function.

3 Analysis of Solution

Since we cannot explore the entire game tree, we are interested in finding an appropriate depth at which the player can return a move in reasonable time. It turns out that due to the high branching factor of backgammon, the reasonable depth is three. The search time becomes impractical for depths of four or larger. Table 1 shows the average game run time of *expectiminimax* players without and with pruning competing against a random player.

Depth	No Pruning	Pruning
1	0.49	0.34
2	46.71	28.74
3	108.27	28.71
4*	-	2413.50

Table 1: Average of 10-game running times (s) for games played between a *expectiminimax* player against a random player. *At depth 4, only one game is played.

Although only five games were played and hence the results are highly uncertain, we can

already see that pruning decreases the run time of the algorithm and that search at depth 4 is impractical.

The final analysis pitches different AI players against one another to test their effectiveness. The three different players are: (a) Random player, (b) depth-1 *expectiminimax* with pruning, called AI1, and (c) depth-3 *expectiminimax* with pruning, called AI3. Only five games are played since they can take significantly long for AI3. Results are shown in Table 2.

Player 1	Player 2	Results
Random	Random	3 x 2
Random	AI1	2 x 3
AI1	AI3	0 x 5

Table 2: Results for three different player pairs over 5 matches.

We can see that at depth 1, the algorithm is unable to dominate a random player since it is socially maximizing over chance events over which it has no control. At depth 3, the player seems to dominate its depth 1 version, showing that indeed it is able to strategize and find better moves. Finally, an interface is provided so that the AI can play with a human player as well.

References

Aimacode. (2018). games.py [Python code]. *GitHub Repository.* Retrieved from `https://github.com/aimacode/aima-python/blob/master/games.py`

chanddu. (2017). backgammon.py [Python code]. *GitHub Repository.* Retrieved from `https://github.com/chanddu/Backgammon-python-numpy-/blob/master/backgammon.py#L273`

Russell, S. J., & Norvig, P. (2016). backgammon.py [Python code]. *Artificial intelligence: a modern approach* (3rd ed.). Boston: Pearson.

Wikipedia. (n.d.). Backgammon. Retrieved from `https://en.wikipedia.org/wiki/Backgammon`

Appendix

Original Proposal

Problem Definition: I want to design a backgammon player. Backgammon is a game that has a chance component, so part of my challenge is incorporating the chance element into the player's strategy.

Proposed Solution: Russel & Norvig mention to algorithms to prune a game tree with chance nodes, referred to as expectiminimax and *-alphabeta. My goal is to implement at least one of these algorithms to build a functional player. The main LO I will be applying is #search.

Deliverables: The code which implements the strategy and the player.