Using Markov Decision Theory to Provide a Fair Challenge in a Roll-and-Move Board Game

Éric Beaudry, Francis Bisson, Simon Chamberland, Froduald Kabanza

***Abstract*— Board games are often taken as examples to teach decision-making algorithms in artificial intelligence (AI). These algorithms are generally presented with a strong focus on winning the game. Unfortunately, a few important aspects, such as the gaming experience of human players, are often missing from the equation. This paper presents a simple board game we use in an introductory course in AI to initiate students to the gaming experience issue. The Snakes and Ladders game has been modified to provide different levels of challenges for students. The game with such modifications offers theoretical, algorithmic and programming challenges. One of the most complex is the generation of an optimal policy to provide a fair challenge to an opponent. A solution based on Markov Decision Processes (MDPs) is presented. This approach relies on a simple model of the opponent's playing behaviour.**

I. Introduction

It is well known that computer science students are often avid video game players. Thus, using games in computer sciences classes is a good teaching strategy to get students interested. It is not a surprise that most textbooks [1], [2] in the field of artificial intelligence (AI) use games to introduce formal algorithms. Simple board games such as the n-puzzle, tic-tac-toe, Gomoku and chess are often used in AI courses because they are usually well-known by most people. These games also offer the advantage of being easy to implement since they generally have a discrete representation as well as simple rules.

In classical AI courses, the main focus of homework assignments usually is the design of an AI that optimizes its decisions to win the game. Unfortunately, a few important aspects are disregarded—one of them is user experience.

Today, an important trend in the design of video games is to provide a fair challenge to human players. Properly balancing the difficulty level of an intelligent adversary in a video game can prove to be quite a laborious task. The agent must be able to provide an interesting challenge to the human player in order not to bore him. On the other hand, an opponent that exhibits an optimal behaviour will result in a player that will either be discouraged or accuse his opponent of cheating. Different strategies can be adopted to create an AI with various difficulty levels.

A *rubber band* (i.e., cheating) AI [3] describes an artificial player that is given an advantage over human players through various means. For example, artificial players may have a perfect visibility of the state of the world, obtain better items, or attain greater speeds. Rubber banding is especially common in racing video games (for instance, the Mario Kart series [4]); human players are led to believe they are winning the race, only to see their opponents get speed boosts for dragging behind too much, and zoom right past them at the last moment. A human player experiencing such a situation is likely to be frustrated and stop playing the game.

In this paper, we show how it is possible to use a Markov Decision Process (MDP) [5] solving algorithm to compute a policy for an autonomous intelligent agent that adjusts its difficulty level according to its opponent's skill level. The resulting policy will ensure that the artificial opponent plays suboptimally against an inexperienced player, but also plays optimally when its adversary is leading the game. This allows the opponent to offer a challenge to the human player while not exhibiting a cheating behaviour.

A modified version of the well-known Snakes and Ladders board game is used as a testbed and as the game framework for a homework assignment in our introductory AI course for undergraduate students. As opposed to the usual rules of the game where chance totally determines the final state of the game, our modified game allows the players to decide of an action to take at each turn. Three actions are allowed: advancing by one square on the board, throwing one die, or throwing two dice. Since each action has a different probabilistic outcome, the player has to carefully think about which action is the best on each square of the board. The board configuration, i.e., the snakes and ladders, strongly influences the actions to be taken. Since this game is non-deterministic and involves a sequence of decision-making, the Markov Decision Process (MDP) formalism comes as a natural approach to compute optimal policies.

Although the game seems trivial at first glance, it nevertheless offers different types of interesting challenges. The simplest problem in Snakes and Ladders is to decide which actions to take in order to reach the end of the board as quickly as possible. This problem is easily solved using an MDP to compute an optimal policy which assigns an action to each board position. However, in a multiplayer context, adopting this policy is not always the best strategy for winning the game. In many situations, players may have to attempt desperate or riskier moves in order to have a chance to win the game. Consequently, the position of the opponent has to be considered to act optimally. The MDP framework can be used to solve this problem optimally.

A more interesting issue arises when trying to provide a

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds québécois de la recherche sur la nature et les technologies (FQRNT). We would like to thank the referees for their comments, which helped improve this paper. The authors are with the Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada. URL: http://planiart.usherbrooke.ca/. E-mail: {firstname.lastname}@USherbrooke.ca.

fair challenge to the adversary. One possible solution is to model the gaming experience of the opponent. Instead of generating a policy that exclusively optimizes the winning probability, the MDP could generate a policy which optimizes the opponent's gaming experience.

There are also other types of issues. For instance, very large virtual boards can be quite hard to solve optimally. Fortunately, many strategies can be used to speed up MDP solving: heuristics to initialize the values of states, an improved value iteration algorithm like Real-Time Dynamic Programming (RTDP) [6] or Labeled RTDP (LRTDP) [7], and other *ad hoc* programming tricks.

The rest of this paper is organized as follows. Section II introduces the MDP framework. Sections III and IV describe, respectively, how to compute an optimal policy to win the game and an optimal policy to optimize user experience. A conclusion follows in Section V.

II. Background

Markov Decision Processes (MDPs) are a well-established mathematical framework for solving sequential decision problems with probabilities [5]. They have been adopted in a variety of fields, such as economic sciences, operational research and artificial intelligence. An MDP models a decision-making system where an action has to be taken in each state. Each action may have different probabilistic outcomes which change the system's state. The goal of an MDP solving algorithm is to find a policy that dictates the best action to take in each state. There exist two main formulations for MPDs: one strives to minimize costs and the other aims to maximize rewards.

A. Minimizing Costs

Some problems, like path-finding, are easier to model using a cost model for the actions. The objective is to compute a policy which minimizes the expected cost to reach a goal state. Formally, an MDP is defined as a 7-tuple $(S, A, Pr, C, s_0, G, \gamma)$, where:

- S is a finite set of world states;
- A is a finite set of actions that the agent could execute;
- $Pr : S \times S \times A \to [0, 1]$ is the state probability transition function. $Pr(s', s, a)$ denotes the probability of reaching state s' when executing action a in state s;
- $C : A \to \mathbb{R}^+$ is the system's cost model;
- $s_0 \in S$ is the initial world state;
- $G \subseteq S$ is the set of goal states to be reached;
- $\gamma \in]0, 1]$ is the discount factor.

A *decision* is the choice to execute an action $a \in A$ in a state $s \in S$. A *policy* is a strategy (a plan), that is, the set of decisions for every state $s \in S$. An *optimal policy* is a policy which assigns the action which minimizes the expected cost to reach the goal in every state.

Several algorithms exist to compute an optimal policy, given a cost model. The value iteration algorithm [5] uses the Bellman equation to compute the best action for each state in a dynamic programming fashion. It starts by computing a value $V(s)$ for each state $s \in S$ by making several iterations of Equation 1.

$$V(s) = \begin{cases} 0, \text{if } s \in G \text{ else} \\ \min\limits_{a \in A} \left(C(a) + \gamma \sum\limits_{s' \in S} Pr(s', s, a) \cdot V(s') \right) \end{cases} \quad (1)$$

Once the values of states have converged, an optimal policy can be extracted using Equation 2. There may exist several optimal policies since, given a state, it is possible for two or more different actions to have the same minimal expected cost.

$$\pi(s) = \arg\min_{a \in A} \left(C(a) + \gamma \sum_{s' \in S} Pr(s', s, a) \cdot V(s') \right) \quad (2)$$

In other words, each state $s \in S$ is associated with the action $a \in A$ that has the best compromise between cost ($C(a)$) and the expected remaining cost of actions' outcomes. When $\gamma = 1$, this problem is also known as the *stochastic shortest path problem*.

B. Maximizing Rewards

Other problems are not naturally expressed with a cost model. Consider the robot motion planning domain in Figure 1. The map is represented as an occupancy grid where black and grey squares are obstacles, the blue triangle is the robot's initial position and the green circle is the goal. Computing a policy which avoids black and grey squares as much as possible could be done by attributing a positive reward to the goal state and a negative reward to undesirable states (e.g., obstacles). Thus, the objective with this formulation of MDPs is to compute a policy which maximizes the expected reward in each state.

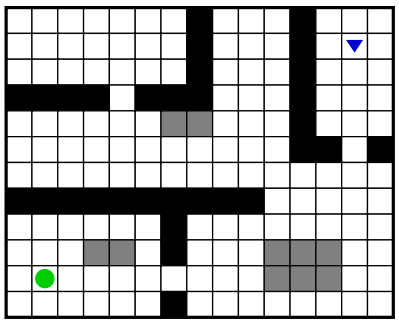

Fig. 1. Occupancy grid in a robot motion planning domain. Different colours represent the different initial reward values: $Black = -1$, $Grey = -0.4$, $White = 0$ and $Green$ (goal) $= 1$. The robot's initial position is denoted by the blue triangle.

The formal definition of a rewards-maximizing MDP is identical to that of cost-minimizing MDPs, except for the cost model ($C : A \to \mathbb{R}^+$) and the goal G, which are replaced by a rewards model ($R : S \to \mathbb{R}$). This rewards model associates each state with a desirability degree. The Bellman equation for this formulation is given in Equation 3.

$$V(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} Pr(s', s, a) \cdot V(s') \quad (3)$$

In this formulation, the best action maximizes the expected reward in every state instead of minimizing the cost to reach the goal. An optimal policy is then extracted using Equation 4.

$$\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} Pr(s', s, a) \cdot V(s') \quad (4)$$

C. Algorithms for Solving MDPs

There exist several algorithms for solving MDP problems, such as value iteration and policy iteration. The value iteration algorithm iteratively evaluates the Bellman equations (Equations 1 and 3) for each state until they all converge. After convergence, the policy is extracted using Equation 2 or Equation 4, depending on the chosen MDP formulation.

As its name suggests, the policy iteration algorithm iterates on policies rather than on state values. It starts with an arbitrary policy that is iteratively refined. During an iteration, the $\max$ operator in the Bellman equation may be removed, since the policy is fixed; this results in a linear equation system. This linear equation system is solved to compute the state values $V(s_i)$. At the end of each iteration, a new policy is extracted. The optimal policy is obtained when there is no change in two successive iterations.

These algorithms converge in $\mathcal{O}(||S||^3)$, where $||S||$ is the size of the state space. Since the state space can be very large, it is not always possible in practice to extract optimal policies.

Some advanced techniques have been proposed for solving MDPs. The Real-Time Dynamic Programming (RTDP) algorithm [6] is a popular technique to rapidly generate good, near-optimal policies. The key idea is that some states have a higher probability than others to be reached during execution. Instead of iterating on the entire state space, the RTDP algorithm begins trials from the initial state, makes a greedy selection over the best action, and then stochastically simulates the successor state according to the state probability transition function. When a goal state is reached, a new trial is started. This process is repeated until convergence of the greedy policy.

To be efficient, the RTDP algorithm requires a heuristic function to initialize the state values. The algorithm is guaranteed to converge to an optimal policy when the heuristic is admissible [7]. The main advantage of the RTDP algorithm is that, when given a good heuristic function, it can produce a near-optimal policy much faster than any value iteration or policy iteration algorithm.

Although RTDP gives good results fast, its convergence is very slow, due to the greedy nature of the selection of states to explore. States with high probabilities of being reached will be visited (and their values computed) over and over again, to the detriment of other, less likely states.

Labeled RTDP [7] (or LRTDP for short) is an improved version of RTDP that consists in labelling states which have already converged to their optimal values. Solved states (i.e., states that have reached convergence) are avoided during the stochastic choice of successor states in the trials, thus allowing the algorithm to visit more states and converge faster towards an optimal policy.

III. Optimal Policy for Winning the Game

A. The Modified Snakes and Ladders Game with Decisions

The Snakes and Ladders game is a roll-and-move game which is played on a grid board. The winner is the player who first reaches the end of the board. In the classic game, players throw two dice and advance their position by the sum of the dice's values. Thus, the game is totally determined by chance. Snakes and ladders linking board squares are spread across the grid. When players reach a square with a ladder, they automatically advance to the square located at the top of the ladder. When players reach a square with a snake's head, they must go back to the square pointed by the tip of the snake's tail. It is also possible for a player to have several successive instantaneous moves (e.g., if a snake starts at the top of a ladder).

The Snakes and Ladders board game has been modified in order to introduce decisions. Each turn, players have to decide of an action from the set of actions $A = \{a_1, a_D, a_T\}$ where:

- a_1 is the action to advance by a single square;
- a_D is to throw one die;
- a_T is to throw two dice.

Each action has a set of possible outcomes which are defined by the function $N : A \to 2^{\{1,\ldots,12\}}$. Outcomes define the number of squares by which the player could advance on the game board. For instance, $N(a_D) = \{1, 2, 3, 4, 5, 6\}$. The probability of outcomes of an action $a \in A$ is denoted by $Pr(n \in \{1, \ldots, 12\}|a)$. For instance, $Pr(6|a_T) = \frac{5}{36}$.

Figure 2 presents a simple board for the game with $n = 20$ squares. The function $T : \{0, \ldots, n-1\} \times \{1, \ldots, 12\} \to \{0, \ldots, n-1\}$ defines the square the player will be in after considering the snakes and ladders on the board. For instance in the example board, $T(0, 2) = 2$, $T(0, 1) = T(0, 4) = 4$ and $T(10, 1) = 18$. Moves that would bring the player beyond the last board square are prohibited and result in the player not moving. The last position has to be reached with an exact move. Thus, $T(18, 1) = 19$ but $T(18, 2) = 18$ because position 20 does not exist.

B. Single-Player

The simplest problem in the modified game of Snakes and Ladders is to decide which actions to take in order to reach the end of the board as quickly as possible without considering the opponent. This problem is easily solved using an MDP to compute an optimal policy which attributes an action to each board position. Since the goal is to minimize the expected number of turns, the cost formulation of MDPs is the most appropriate one. The state space S is defined as the

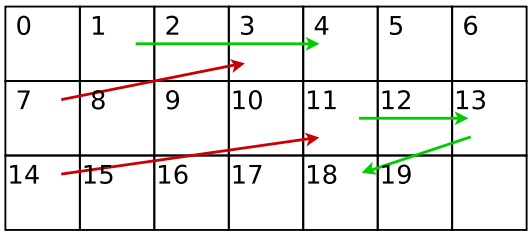

Fig. 2. Simple board for the Snakes and Ladders game. Red arrows represent "Snakes", and green arrows represent "Ladders".

set of states for each board position $S = \{s_0, s_1, ..., s_{n-1}\}$, where s_0 is the initial state and $s_{n-1} \in G$ is the final (goal) state. The probabilistic state transition model is defined by Equation 5.

$$Pr(s_j, s_i, a) = \sum_{k \in \{x \in N(a) | T(i,x) = j\}} Pr(x|a) \qquad (5)$$

Since the state horizon is finite (the game ends once the last square is reached) and the goal is to find the shortest stochastic path, the discount factor $\gamma = 1$. All actions cost one turn; thus, the $C(a)$ term could simply be replaced by 1 and removed from the summation. By merging Equations 1 and 5, we obtain Equation 6.

$$V(s_i) = \begin{cases} 0, \text{if } i = n-1 \text{ else} \\ 1 + \min\limits_{a \in A} \sum\limits_{x \in N(a)} Pr(x|a) \cdot V(s_{T(i,x)}) \end{cases} \qquad (6)$$

The value iteration algorithm can be implemented in a straightforward manner simply by programming Equation 6. For small boards, the policy generation is very fast. However, on larger board (thousands or millions of states) the convergence could be very slow if implemented in a naïve way. To speed up convergence, many strategies can be used.

The value iteration algorithm updates the state values V during several iterations until convergence. Most AI textbooks present this algorithm as the process of updating a V vector using the values V' from the previous iteration. A faster implementation may be achieved by updating a unique vector of V values instead. Updated values are thus used sooner.

Another strategy that could be added to this one is the use of a particular ordering for iterating on states. Iterating in the state order will be very slow because several iterations will be necessary before cost values propagate from the goal state to the initial state. Thus, starting each iteration from the final state results in much faster convergence.

Table I shows empirical results for a few iterations of the algorithm on the board from Figure 2. The first column indicates the states. The next columns presents the current $V(s_i)$ value after the n^{th} iteration. After nine iterations, the algorithm has converged and the last column shows the extracted policy. Values in the last iteration column represent the expected number of remaining turns to reach the end of the board.

State	Iter#1	Iter#2	Iter#3	...	Iter#9	π
s_0	4.03	3.70	3.67	...	3.66	a_T
s_1	3.75	3.50	3.48	...	3.47	a_T
s_2	3.44	3.28	3.26	...	3.26	a_T
s_3	4.18	3.17	3.08	...	3.07	a_T
s_4	3.67	3.00	2.94	...	2.93	a_T
s_5	3.26	2.91	2.88	...	2.87	a_T
s_6	2.84	2.82	2.81	...	2.80	a_T
s_7	2.82	2.78	2.77	...	2.77	a_T
s_8	2.62	2.61	2.61	...	2.61	a_D
s_9	2.72	2.69	2.67	...	2.67	a_D
s_{10}	2.00	2.00	2.00	...	2.00	a_1
s_{11}	2.42	2.40	2.39	...	2.39	a_T
s_{12}	2.00	2.00	2.00	...	2.00	a_1
s_{13}	2.00	2.00	2.00	...	2.00	a_1
s_{14}	2.00	2.00	2.00	...	2.00	a_T
s_{15}	3.33	3.11	3.04	...	3.00	a_D
s_{16}	3.00	3.00	3.00	...	3.00	a_1
s_{17}	2.00	2.00	2.00	...	2.00	a_1
s_{18}	1.00	1.00	1.00	...	1.00	a_1
s_{19}	0.00	0.00	0.00	...	0.00	a_T

TABLE I
EMPIRICAL RESULTS FOR THE VALUE ITERATION ALGORITHM ON THE BOARD FROM FIGURE 2.

Figure 3 compares the running time of a standard MDP implementation with an optimized one on various board sizes.

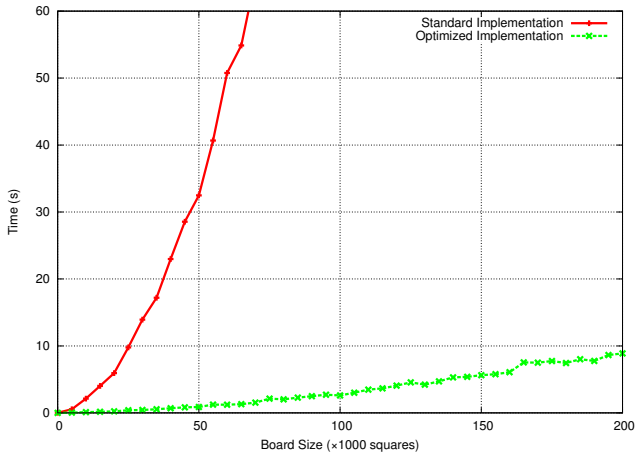

Fig. 3. Performance improvement of an optimized policy generator.

C. Two Players

Playing with the previous policy which minimizes the number of moves to reach the end of the board as quickly as possible is unfortunately not always the best strategy to win against an opponent. The state space in now defined by considering the position of both players on the board. Consider the board and the generated policy in Figure 4. Let the current game state be such that player P_A is at position 17 and player P_B is at position 18. What is the best action for player P_A? Using the optimal single-player policy, P_A will choose action a_1 and then reach position 18. However, player P_B will win the game at his next turn. The probability of winning the game in this state is thus zero if we

adopt the strategy to reach the end of the board as quickly as possible. In this situation, the best move is a desperate one: by selecting action a_D, player P_A can hope to move by exactly 2 squares and then win the game. Thus, in this game state, player P_A still has a probability of $\frac{1}{6}$ to win the game using action a_D.

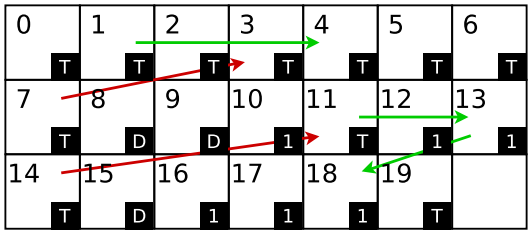

Fig. 4. Simple board from Figure 2 with an optimal single-player policy. Actions a_1, a_D and a_T have been abbreviated to 1, D and T, respectively, for conciseness.

The computation of the best decision to take in a two players context is more challenging than in a single-player context. Indeed, decisions in a multiplayer context do not only depend on the player's position, but also on the opponent's position. A two-players policy associates an action to a pair of positions, and is defined as $\pi : S \rightarrow A$, where $S = \{s_{i,j} \ \forall \ (i,j) \in \{0,\dots,n-1\}^2\}$.

Provided that the opponent's strategy can be modelled using one such policy π_{sp} (which does not evolve over time), we can calculate a policy $\pi'(\pi_{sp})$ maximizing the chances of winning the game against said opponent. Several algorithms can be used to compute $\pi'(\pi_{sp})$.

1) Alpha-Beta Pruning: Since this is a zero-sum game with two players, one could suggest using Alpha-Beta Pruning-based algorithms. Algorithm 1 presents an adaptation of the classic algorithm to consider chance nodes [8]. A limitation of Alpha-Beta Pruning is that it requires to reach leaf nodes of the search tree to make an optimal decision. Even if a very small board is used, leaf nodes could be very deep in the search tree. Another problem is that the outcome of the actions are probabilistic, which may produce infinite loops with a non-zero probability. A common strategy is to cut the search tree by setting a maximum depth. Nodes at this depth are evaluated using a heuristic function. This evaluation function is generally approximate and cannot guarantee optimality.

2) MDP-based Solution: To guarantee optimality, an MDP can be used. Since MDPs are designed for sequential decision-making problems, one may question this choice because it does not naturally fit games with adversaries. However, since an assumption was made on the behaviour of the opponent (by using a fixed policy), we could integrate the opponent's choices in the player's decisions.

The cost formulation of MDPs is not exactly appropriate anymore since the goal is not to minimize the expected number of turns, but rather to reach the end of the board before the opponent. We thus adopt the rewards formulation: a reward is simply put on states where the player wins the game. Since all winning states are considered equivalent

Algorithm 1 Alpha-Beta Pruning with Chance Nodes

```
1.  AlphaBetaSearch(s_{i,j})
2.    (value, action) ← MaxNodeSearch(s_{i,j}, −∞, +∞)
3.    return action

5.  MaxNodeSearch(s_{i,j}, α, β)
6.    if i = n − 1 return +1
7.    if j = n − 1 return −1
8.    value ← −∞
9.    for each a_i ∈ A
10.     v ← 0
11.     for each x ∈ N(a_i)
12.       i' ← T(i, x)
13.       (v_n, a_j) ← MinNodeSearch(s_{i',j}, α, β)
14.       v ← v + Pr(x|a_i) · v_n
15.     if v > value
16.       value ← v
17.       action ← a_i
18.     if value ≥ β break
19.   return (value, action)

21. MinNodeSearch(s_{i,j}, α, β)
22.   if i = n − 1 return +1
23.   if j = n − 1 return −1
24.   value ← +∞
25.   for each a_j ∈ A
26.     v ← 0
27.     for each y ∈ N(a_j)
28.       j' ← T(j, y)
29.       (v_n, a_i) ← MaxNodeSearch(s_{i,j'}, α, β)
30.       v ← v + Pr(y|a_j) · v_n
31.     if v < value
32.       value ← v
33.       action ← a_j
34.     if value ≤ α break
35.   return (value, action)
```

(winning by a distance of 2 or 200 positions is equivalent) the reward is set uniformly as given by Equation 7.

$$R(s_{i,j}) = \begin{cases} 1, \text{if } i = n-1 \wedge j < n-1 \text{ else} \\ 0 \end{cases} \tag{7}$$

The transition probability function is defined in such a way as to consider that both players move simultaneously, as described in Equation 8.

$$\begin{aligned} M_{i,i'} &= \{x \in N(a) | T(i,x) = i'\} \\ M_{j,j'} &= \{y \in N\left(\pi_{sp}(s_j)\right) | T(j,y) = j'\} \\ Pr(s_{i',j'}, a, s_{i,j}) &= \sum_{x \in M_{i,i'}} Pr(x|a) \sum_{y \in M_{j,j'}} Pr(y|\pi_{sp}(s_j)) \end{aligned} \tag{8}$$

Integrating Equations 3 and 8 results in Equation 9.

$$\begin{aligned} V(s_{i,j}) = \ & R(s_{i,j}) \\ & + \max_{a \in A} \sum_{x \in N(a)} Pr(x|a) \\ & \cdot \sum_{y \in N(\pi_{sp})} Pr(y|\pi_{sp}) \cdot V(s_{T(i,x),T(j,y)}) \end{aligned} \tag{9}$$

Since we consider simultaneous moves in state transition, an important question is what happens when both players reach the end of the board during the same turn. Since we

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
2	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
3	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
4	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
5	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
7	D	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
8	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T	T
9	1	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T	T
10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T	T
11	1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T	T
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D	T
14	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	D	T
15	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T
16	1	1	1	1	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D	T
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T
19	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Fig. 5. Optimal policy to beat an opponent playing with an optimal policy to reach the end of the board as quickly as possible. The row index i gives the position of the AI player and the column index j gives the opponent's position.

play before the opponent, reaching the state $s_{n-1,n-1}$ from an another state $s_{i,j}$ such that $i, j < n-1$ means that the first player reaches the end before the opponent. Thus, if a draw is not allowed, a reward should be put on this particular square: $R(s_{n-1,n-1}) = 1$.

Figure 5 shows an optimal policy to beat an opponent playing an optimal policy to reach the end of the board as quickly as possible. Note that the optimal policy is not necessarily unique. At each turn, the player in square i, which plays against an opponent in square j, looks up the cell (i, j) to take the best action which maximizes his chance of winning.

Table II presents results which compare the percentage of games won by the single-player optimal policy (computed as presented in Section III-B) against the two-players optimal policy (computed as presented here) on a board of size $n = 1000$ squares. Results shows an improvement of 3 % of winning chances when using the two-players optimal policy against a single-player optimal one.

	Single-Player Policy	Two-Players Policy
Single-Player Policy	50 %	47 %
Two-Players Policy	53 %	50 %

TABLE II
IMPROVEMENT WHEN CONSIDERING THE OPPONENT.

Figure 6 presents the required time to generate single-player and two-players policies. Optimal decision-making which considers the opponent comes at a cost: the state space grows quadratically as the board size increases.

D. Optimality Against an Arbitrary Opponent

It is important to note, however, that as $\pi'(\pi_{sp})$ is conditioned on π_{sp}, it maximizes the player's chances of winning

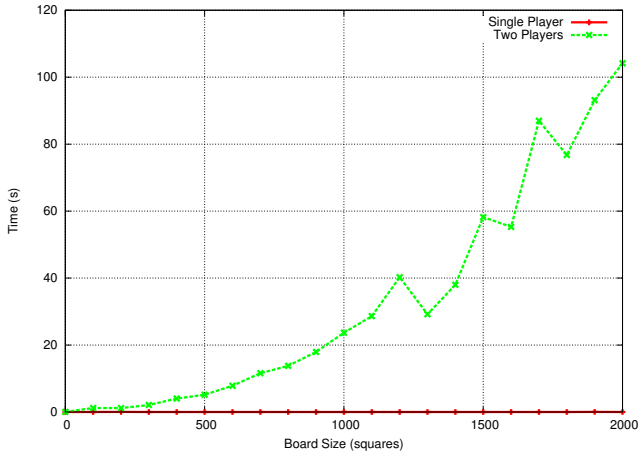

Fig. 6. Required time to generate single- and two-players policies.

exclusively against this particular policy. This leads to the fundamental question: can we find a policy maximizing the chances of winning against an arbitrary opponent?

Maximizing one's chances of winning against a given opponent involves taking advantage of the shortcomings of this particular opponent. Since two different opponents do not play the same way, we need two different policies to play optimally against each one of them.

Proof: Consider a 3-square board in which $s_{0,1}$ denotes the current state, i.e., the player whose turn it is to play is in square 0 while his opponent is in square 1, one square away from victory. Given this particular state, the optimal action is the one maximizing the probability of winning the game. Let an arbitrary policy π_1 associate the action a_1 to the state $s_{1,x} \forall x$. This policy instructs the player to move one square

forward to victory when he's on the penultimate square. An optimal policy $\pi'(\pi_1)$ therefore maps the state $s_{0,1}$ to the action a_D, preserving a probability of winning of $\frac{1}{6}$.

Let another arbitrary (and naïve) policy π_2 associate the action a_D to the state $S_{1,x} \forall x$. A player following this policy throws a die on the penultimate square. Contrarily to $\pi'(\pi_1)$, the optimal policy $\pi'(\pi_2)$ maps the state $s_{0,1}$ to the action a_1, indicating that the best winning chances are obtained by advancing one square forward, hoping that the opponent does not get 1 on his die roll, and then claiming victory (winning with probability $\frac{5}{6}$). Using $\pi'(\pi_1)$ against the weaker π_2 policy would greatly reduce the winning chances. Indeed, $\pi'(\pi_1)$ instructs the player to roll a die in the state $s_{0,1}$, yielding winning chances of $\frac{11}{16}$ (see Equation 10) $< \frac{5}{6}$, which corresponds to the probability of winning by rolling a 2 added to the probability of winning by rolling a 1 instead and advancing one square during the next turn (providing the opponent did not already win).

$$\frac{11}{36} \cdot \lim_{n \to \infty} \sum_{i=1}^{n} (\frac{4}{6} \cdot \frac{5}{6})^i = \frac{11}{16} \tag{10}$$

■

Hence there does not exist an optimal policy against an arbitrary opponent; optimal policies are rather conditioned upon a particular opponent's strategy.

However, instead of extracting the opponents' actions from fixed policies, one could allow these actions to vary over time. In an MDP setting, this amounts to optimizing the expected reward with respect to a different opponent policy during each new iteration. If we set the opponent's policy to be at all times the exact same policy that is currently being computed (which possibly changes at each iteration), solving the MDP amounts to finding a policy which plays optimally against itself. We postulate that this process eventually converges to an optimal policy that we denote π_{opt}, such that $\pi'(\pi_{opt}) = \pi_{opt}$. Empirical experiments have yet to show an example where this is not the case, but such convergence has not been theoretically proven. Note that π_{opt} is not necessarily unique, as many actions may have the same expected reward. This last policy can be considered as a form of globally-optimal policy, in the sense that it offers at least a 50 % victory percentage against any arbitrary opponent. It does not, however, exploit the particular weaknesses of each individual opponent.

E. Generalization to Multiplayer

Considering more than two players offers a new challenge. Similarly to the single-player to two-players generalization, a simple avenue is to add a new dimension to the state space. Thus, a state could be defined by a m-tuple $s = (p_1, p_2, \ldots, p_m)$ which gives the positions of all m players. A problem with this approach is that the size of the state space grows with the size of the board and the number of players, i.e, $||S|| = n^m$. Solving this problem in an optimal way becomes quickly intractable. A viable approximation for multiplayer is to model the game as a two players game and only consider the opponent that is closest to the goal state.

IV. Optimal Policy for Gaming Experience

A more complex aspect of the game involves considering the gaming experience of a human opponent. Playing against an opponent with a similar skill level is generally more fun than playing against someone who is too strong or too weak. Properly balancing the difficulty level of an intelligent adversary in a video game can prove to be quite a laborious task.

Before proposing a solution to this problem, there is an important question we need to answer: what exactly is gaming experience? Without a clear definition, it is difficult to try to maximize it. Thus, a formal gaming experience mathematical model is required. Elaborating such a model is an orthogonal problem to that of optimizing an AI player's decisions to maximize the opponent's gaming experience. Once a gaming experience model is given, it is simply integrated into the equation.

As was the case with the policy developed in Section III-C, an assumption has to be made about the opponent's playing strategy. For the rest of the paper, consider that the opponent does not play optimally but rather only follows his intuition. Since throwing two dice generally allows the player to move closer to the final state, the opponent player selects an action using the strategy π_{opp} presented by Equation 11. Note that the opponent only considers his own position j and does not consider the AI player's position i.

$$\pi_{opp}(s_{i,j}) = \begin{cases} a_T, & \text{if } j < n-6 \\ a_D, & \text{if } j < n-3 \\ a_1, & \text{otherwise} \end{cases} \tag{11}$$

A. Simple Opponent Abandonment Model

As a simple model of gaming experience, an abandonment rule is defined as follows. The opponent player abandons the game if he is too far from the AI player's position on the board, i.e., the opponent believes he has no chance of winning. Another source of abandonment is when the opponent has no challenge, i.e., when the opponent believes that the game is too easy. More precisely, the opponent abandons the game when the distance between the players' positions is greater than half the size of the board.

Thus, the goal of the AI player is to maximize his chance of winning the game before the opponent abandons. This problem can be solved using an MDP in a similar way of the one which maximizes the chance of winning against a given opponent.

To solve this problem, we use the same strategy as the one presented for a two opponents game (Section III-C.2), i.e., a state is defined as a pair of positions on the board. The main difference is that there is a set of abandonment states $S_{ab} \subset S$ which is defined as $S_{ab} = \{s_{i,j} \ \forall \ (i,j) \in \{0, ..., n-1\}^2 \ : \ |i-j| \geq \frac{n}{2}\}$, where n is the number of board squares. By definition, states $s \in S_{ab}$ are states where the opponent abandons the game because of the lack

of enjoyment (the opponent being too weak or too good). Thus, these states are terminal where no action is applicable. An action applicability function, defined as $App : S \rightarrow A$, is used to find out which actions are applicable in a given state.

Table III presents empirical results (1 000 000 simulations on a board of size $n = 1000$) that demonstrate how using an optimal policy to win the game results in the abandonment of most games (first column). Instead, a policy computed by taking account of the opponent's model greatly reduces the number of abandonments, while still exposing an optimal behaviour, in the sense the the AI player wins the majority of games. Note that this could also discourage the opponent; the policy could be improved to try to balance the number of wins and losses.

	Optimal Policy	Considering Opponent Model
Wins	33.9 %	97.1 %
Losses	0.6 %	1.6 %
Abandonments	65.5 %	1.3 %

TABLE III

IMPROVEMENT WHEN CONSIDERING THE ABANDONMENT MODEL.

Solving MDPs for large state spaces takes a very long time to converge to an optimal policy. In most situations, a near-optimal policy, generated with an algorithm such as RTDP, is very much acceptable since these techniques produce good results in a short amount of time. Figure 7 empirically compares the performance of value iteration and RTDP on a large game board ($n = 1500$) over 1 000 000 simulations. The quality of the policies, measured in terms of the percentage of victories without the opponent abandoning, is displayed as a function of the allotted planning time. The difference, while not astounding, would still be welcome in a time-critical context.

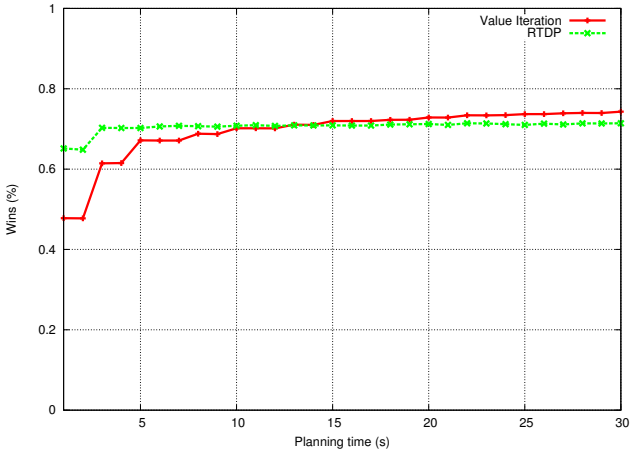

Fig. 7. Quality of plans as a function of the allotted planning time.

This simple abandonment model could be further improved. For instance, the abandonment could be probabilistic instead of deterministic. We could define a probability density function which specifies the probability of abandonment given the position of both players.

B. Distance-Based Gaming Experience Model

Another user experience model that could be used is a distance-based one. Rather than setting rewards on states which correspond to the end of the game, rewards can be attributed on all states identified by Equation 12. The maximum reward (0) is obtained when both players are in the same board position and it decreases quadratically as the distance between both players increases.

$$R(s_{i,j}) = -(i - j)^2 \tag{12}$$

Since the goal is to maximize the user gaming experience, the finite time horizon assumption may be not valid anymore. An infinite loop is now possible. For this reason, the discount factor γ has to be set to a value $\gamma < 1$. This value is set to weight the AI player's preference between short-term and long-term rewards.

V. CONCLUSION

In this paper, we presented a modified version of the classic Snakes and Ladders board game. This game is used in our introductory course in artificial intelligence to teach Markov Decision Theory to undergraduate computer science students. Although the game is simple, it contains many interesting aspects also present in more complex computer games. This game offers several perspectives which require different MDP formulations and strategies to generate optimal policies. The most complex challenge is the generation of a policy which optimizes the gaming experience of a human player.

As future work, we consider a few possible avenues to add new challenges for the development of AI algorithms in this game framework. One of them is to use machine learning techniques to automatically learn the opponent's gaming experience model. For instance, we could provide a database of previously-played games where each game is associated with a score (win, loss, draw, abandonment, etc.) A model could be learned from this history and then be integrated into the MDP policy generation algorithm.

REFERENCES

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Mordern Approach*, 3rd ed. Prentice Hall, 2010.

[2] G. F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 6th ed. Addison Wesley, 2009.

[3] B. Schwab, *AI Game Engine Programming*, 2nd ed. Rockland, MA, USA: Charles River Media, Inc., 2008.

[4] B. Medler, "Views from atop the fence: neutrality in games," in *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, New York, NY, USA, 2008, pp. 81–88.

[5] R. E. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, pp. 679–684, 1957.

[6] A. Barto, S. Bradtke, and S. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, no. 1–2, pp. 81–138, January 1995.

[7] B. Bonet and H. Geffner, "Labeled RTDP: Improving the convergence of real-time dynamic programming," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2003, pp. 12–21.

[8] B. W. Ballard, "The *-minimax search procedure for trees containing chance nodes," *Artificial Intelligence*, vol. 21, no. 3, pp. 327–350, 1983.