A Risky Proposal: Designing a Risk Game Playing Agent

Juan Lozano, jlozano@stanford.edu
Dane Bratz, dbratz@stanford.edu

Abstract

Monte Carlo Tree Search methods provide a general framework for modeling decision problems by randomly sampling the decision space and constructing a search tree according to the sampling results. Artificial Intelligences employing these methods in games with massive decision spaces such as Go and Settlers of Cataan have recently demonstrated far superior results compared to the previous classic game theory approaches [Browne et al. 1]. We apply Monte Carlo Tree Search methods, particularly the UCT variant, to an online version of the popular board game Risk.

1 Introduction

1.1 Risk and Lux

Risk is a popular game where players compete to control and conquer countries using armies that their countries generate. The game is classically played using a map of the globe, and players sequentially act individually on a turn. A turn consists of four phases: the cards phase, during which the player can exchange a set of cards that he holds for additional armies to add to this turn's army income; the place armies phase, during which the player decides how to distribute his income armies amongst his countries; the attack phase, during which the player conducts any desired attacks (and subsequent army moves into conquered countries); and the fortify phase, during which the player can choose to redistribute armies between adjacent countries. Players can obtain cards either by conquering at least one country a turn or by eliminating an opponent and taking the cards they held. The initial countries for each player are chosen randomly or through a draft among the players.

Lux by Sillysoft is a popular online platform for Risk. The platform notably offers a development package with which one can develop both maps and AI agents to be used by the platform for gameplay. The development package offers example AI agents that are deterministic in nature as well as the helper methods that the AI agents use to quantify higher level aspects of the game state. Several of the agents provided, from our personal experience playing against them, offer a respectable level of competition.

1.2 Motivation

The game of Risk presents an interesting opportunity to study the boundary between classic games and modern video games. The former games usually have a discrete state space and deterministic outcomes; while the latter normally have a continuous state space and indeterministic outcomes. The game of Risk contains an interesting blend of the two, on one hand the board state is discrete and at any given turn is easily represented on a modern computer, but the number of possible board states that could result from the current board state is massive. In fact it can be shown that the state space complexity of Risk is infinite and the game tree of Risk has an exponential branching factor [Wolf 34]. Additionally, the current reward for a given board state is easily calculated, and deterministic. Unfortunately, the future reward for a current board state is uncertain, due to the chance elements involved in attacks; even calculating the expected outcome of a battle is computationally quite complex [Wolf 49]. These conclusions indicate that the classic approaches to modeling and learning games, such as minimax, would be computationally infeasible for the game of Risk.

The above concerns led us to a general class of algorithms which has performed exceptionally well in infinite search spaces: Monte Carlo Methods and in particular Monte Carlo Tree Search. These methods have been successful in other computationally complex board games such as Go and Settlers of Cataan, where traditional game theoretic methods had previously struggled [Chaslot et al 2]. Additionally, Monte Carlo Methods have recently been used in video game development for generating an evaluation function to accurately rate the quality of a video game AI [Chaslot et al 2]. The common thread between these diverse applications is that the state space to explore is very large and the reward function is usually uncertain and highly variable.

1.3 Data Set

We modified 10 of the existing AI's to write to a file various aspects of the board state (i.e. number of countries owned, number of opponent armies in a given country, etc.) before and following the phases of the agent's turn. Overall, we wanted to be able to correlate certain features of gameplay with winning for a particular agent. We then played four of the better agents against each other for 500 game simulations. We gathered data in this way for two general reasons: we wanted a robust data set while also ensuring that our data would be relevant. By playing the harder AI's against each other, we hoped to improve our data's quality by forcing the winning agent to have played at least somewhat well to have won. Playing a game between only AI's can also be done quite quickly - within a matter of seconds, which allowed us to gather information for a large number of games.

2 Approach

2.1 Risk Strategy

Risk strategy varies heavily with the specific rule set that is being used (which is especially apparent in the Lux platform). It depends on how what the value progression is for turning in cards, the map structure, and whether continents increase in bonus value over time. Because of this, we decided to focus on a particular variant of Risk: in particular, we used the classic map, with the initial countries for each player chosen randomly, with a card bonus that increased by 2 armies per cash in (starting at four), and with a 5% continent increase per round (these are the default Lux settings).

On these settings, we noticed a few general things. First, that maximizing our troop income for the next turn seemed to be an incredibly effective strategy. This was best done by conquering and holding continents, conquering and holding more countries, and getting cards by either taking at least one country per turn or eliminating an opponent and taking their cards.

2.2 Monte Carlo Tree Search and UCT Algorithm

Monte Carlo Tree search methods are characterized by two key assumptions: random simulation may be used to approximate the true value of an action, and the subsequent approximate value can be efficiently incorporated to update the search of the game tree toward a best first strategy [Browne et al. 5]. Based on these two assumptions, a Monte Carlo Tree Search method iteratively constructs a partial game tree using the results of previous searches of the tree to guide the current construction of the tree. When searching a game tree, the algorithm is parameterized by a game simulator, a default strategy, a tree policy, a game state evaluation function, and an input game state [Browne et al. 5]. The input game state provides the root of the partial game. The tree policy dictates the starting node for new searches and the construction of new game tree nodes based on the results of previous searches. Searches are conducted from a given starting node through simulation of the default strategy for a specified number of turns, ideally until completion of the game. The evaluation function is then used to evaluate the final simulated game state, the results of this evaluation are then back propagated up the game tree.

The UCT algorithm is a specific variant of Monte Carlo Tree Search that has been applied to other games like Go [Browne et al. 8]. The UCT algorithm specifies a tree policy that constructs and searches the partial game tree in such a way that there is always a nonzero probability that we will explore every possible strategy [Browne et al. 7]. But, the strategies that yield higher rewards have a higher probability of being explored which implies that on average they will be explored earlier in the search [Browne et al. 7].

2.3 Application to Risk

In order to apply a Monte Carlo Tree search method to the game of Risk we made some simplifying assumptions. First, we decided to limit the search space by employing heuristic strategies provided by the Lux API to choose our initial troop placement, our turn by turn troop placement, our fortification moves, and our card turn in strategy. These are drastic simplifying assumptions, since the strategy employed in each phase is dependent on the strategy employed in the other phases. However, each phase also deserves independent treatment, since the structure of the rewards, and the strategies employed to optimize those rewards, vary drastically from phase to phase. Gibson et al, for example, applied Monte Carlo methods to the specific strategy of choosing which continents to occupy. However, we observed that for average to relatively skilled humans, the troop placement and fortification strategies all consisted of a combination of heuristics provided by the Lux API. Since these heuristics already seemed to be working well for the provided agents, and providing a comprehensive strategy that incorporates all of the game phases is outside the scope of a single paper, we decided to focus solely on the attack phase.

The attack phase presents an interesting challenge because it drives the dynamics of the entire game. Unfortunately, even though the number of game states that can result from a single attack is only two, the number of possible game states that can result from a series of attacks, a battle, grows very quickly; since we can choose to stop attacking after each roll of the dice. A more simplistic model assumes all battles are carried out until completion, meaning that a series of attacks on a single country ends when the attacking country no longer has enough troops to attack, or the defending country has been conquered. We made this assumption in order to ease the computational burden and we also observed that in Risk, the most important factor is the final outcome of the battle [Wolf 49]. Also, if we assume all our attacks are carried out to completion, then losing an attack corresponds to the worst possible scenario, and we would like our agent's general strategy to be robust enough to endure these scenarios. Our application of the UCT algorithm to a Risk Game with m players and w countries consisted of the following components:

i. A game tree node: $g \equiv \{S, L, n, v\}$, where:
   - $S \in \aleph^{m \times w}$, the game state matrix, where $s_{i,j}$ is the number of armies for player i on country j, from the rules of Risk all countries must have at least one army on them during every turn.
   - $L \equiv \{g_1, g_2, \ldots\}$, are the child game tree nodes of g.

- $n = 1,2,3...$, is the number of times the game tree node g has been visited by a search.
- $v \in \Re$, is the total value of all the game states that resulted from searches that passed through node g.

ii. A tree policy $F(g)$, which takes as input a game tree node g and returns the child node g_l that maximizes the following function [Browne et al. 7]:

- $\frac{v_l}{n_l} + 2A\sqrt{\frac{2ln(n)}{n_l}}$, where n is the number of times the current parent node has been visited, and the subscripted values refer to the child node, and A is a parameter. The first term incentives exploitation of high reward choices, while the second term ensures that we explore choices that have fewer visits, and the exploration constant A dictates how likely we are to explore other options [Browne et al. 5]. We tried various values of A and settled on $A = .29$.
- In the context of the attack phase of Risk, the child nodes of a game state node are all the possible game states that could result from any of the possible attacks the agent could make in the game state described by g.

iii. A game state evaluation function $Q(S)$, which takes as input a game state matrix and outputs a value v for the game state. The value, for a given game state, corresponds to our agent's expected troop income after subtracting off the expected troop income of the opponent with the largest troop income. Where the troop income for a player is a function of the number of countries held by the player, the continents controlled by the player, and the number of cards the player has.

iv. A default strategy, $D(S)$, which takes as input a game state matrix and outputs an attack $a_{D(s)}$ to simulate.

- We actually had several default strategies that corresponded to high level goals such as taking a continent, eliminating an opponent to gain their cards, or ensuring opponents do not secure a continent.

v. A battle simulator, $\Psi(S, a_{D(s)})$, which takes as input a game state matrix and an initial attack to simulate and outputs the game state matrix which results from simulating the series of attacks, beginning with the input attack and going until completion, as explained above.

A typical attack phase for our agent consisted of using the UCT version of Monte Carlo Tree Search to build separate partial game trees for each possible default strategy in parallel, each default strategy space was searched for the same amount of time, after which time the agent executed the strategy specified by the game tree with the highest value at its root. The strategy chosen was executed until the agent lost a battle, at which point the above process was repeated with the current game state serving as the root node for the partial trees. The agent stopped attacking when none of the strategies yielded an expected positive value.

3.0 Results and Future Considerations

Using the above strategy, our agent was able to defeat the best provided AIs 73 percent of the time. To put this number in perspective, the provided AI with the highest win percentage against the same competition was only able to win 38 percent of the time. In the future we would like to expand our implementation to include the other game phases in the decision process. Once this has been completed we will be able to reverse train our agent in order to determine better forms for the game state evaluation function as well as a better value for the exploration constant. Additionally, we plan on implementing a basic form of pattern recognition which will allow the agent to begin implementing combinations of strategies based on general families of board types.

References

- Gibson, R.; Desai, N.; and Zhao, R. 2010. An Automated Technique for Drafting Territories in the Board Game Risk. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2110
- Wolf, M. (2005). *An Intelligent Artificial Player for the Game of Risk*. (Unpublished doctoral dissertation). TU Darmstadt, Knowledge Engineering Group, Darmstadt Germany. http://www.ke.tu-darmstadt.de/bibtex/topics/single/33
- Guillaume Chaslot, Sander Bakkes, István Szita, and Pieter Spronck (2008). Monte-Carlo Tree Search: A New Framework for Game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference* (eds. Michael Mateas and Chris Darken), pp. 216-217. AAAI Press, Menlo Park, CA. (presented at the AIIDE08). http://sander.landofsand.com/index.php?research
- Browne et al. A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games, Vol. 4, No. 1, March 2012. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6145622&tag=1
- Sillysoft. Lux Delux - The best Risk game there is. http://sillysoft.net/lux/.