

# Solving Coup as an MDP/POMDP

Semir Shafi  
Dept. of Computer Science  
Stanford University  
Stanford, USA  
semir@stanford.edu

Adrien Truong  
Dept. of Computer Science  
Stanford University  
Stanford, USA  
aqrtruong@stanford.edu

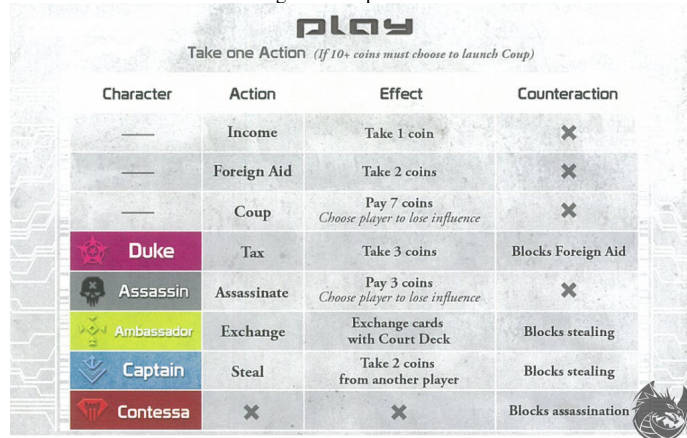
David Lee-Heidenreich  
Dept. of Computer Science  
Stanford University  
Stanford, USA  
dleeheid@stanford.edu

**Abstract**—We modeled the card game Coup as a Markov Decision Process and attempted to solve it using various methods learned in CS238. Due to our large state space, we focused on online methods. Since Coup is a multi-agent game we generated optimal policies against players with specific strategies. We first modeled the game as an MDP where we knew everything about the game state and developed policies against a player doing random actions. We used forward search, sparse sampling, and monte carlo tree search. We then modeled the game as a POMDP with state uncertainty where we did not know our opponents cards. We implemented Monte Carlo Tree Search, sparse sampling and forward search with both incomplete and complete information. Finally, to try and beat our Monte Carlo Tree Search player, we implemented Forward Search with Discrete State Filtering for updating our belief.

**Index Terms**—MDP, POMDP, Coup, multi-agent

## I. INTRODUCTION

Fig. 1. Coup Rules



The table is titled 'Coup Rules' and includes a sub-header 'Take one Action (If 10+ coins must choose to launch Coup)'. It lists actions for various characters and their effects and counteractions.

Character	Action	Effect	Counteraction
—	Income	Take 1 coin	×
—	Foreign Aid	Take 2 coins	×
—	Coup	Pay 7 coins Choose player to lose influence	×
Duke	Tax	Take 3 coins	Blocks Foreign Aid
Assassin	Assassinate	Pay 3 coins Choose player to lose influence	×
Ambassador	Exchange	Exchange cards with Court Deck	Blocks stealing
Captain	Steal	Take 2 coins from another player	Blocks stealing
Contessa	×	×	Blocks assassination

### A. Coup Overview

Coup is a popular deception and strategy board game that contains a lot of uncertainty. There are five different card roles and three of each type in the deck. Each player is dealt two of these cards at random, and each player can observe only their own cards. Each card has its own unique actions and counteractions (refer to Fig. 1). The objective of the game is to remain alive and eliminate all other players. A player is eliminated if both of their cards are face up and are observable to all the players, rendering their cards useless. The strategy of

the game is to deceive the other players by lying and claiming you have whatever card suits you best. Because lying can give a significant advantage to the player, the other players try to determine when a player is lying and call bluff. If they call bluff and you were not lying, the player who called bluff must flip over one of their cards (and cannot use it anymore). If the other players catch you lying, you must flip over one of your cards.

### B. Sources of Uncertainty

There are several sources of uncertainty in the game:

- Players are uncertain what roles (cards) other players have until they are eliminated
- Players are uncertain what actions/claims their opponent will make

### C. Related Work

To the best of our knowledge, there isn't any previous work that has tried to compute the optimal policy or computed online planning strategies to play the board game Coup. We review work done on related games here. There's another similar game called One Night Werewolf where the objective of the game is to try and discern which players are lying. It was a topic at the Computer Games Workshop at IJCAI, and they discussed Monte Carlo Tree Search (MCTS), reinforcement learning, alpha-beta, and nested rollout policy adaptation [1]. Yet, they note that the most popular method was MCTS and Deep Learning. Similarly, in our project we try out using MCTS to decide the best action from a given game state.

## II. MODELING COUP

We can represent Coup as an MDP with the following states, actions, transitions, and rewards.

### Actions

- 1) Income
- 2) ForeignAid
- 3) Coup1 (target opponent 1)
- 4) Coup2 (target opponent 2)
- 5) Tax
- 6) Assassinate1 (target opponent 1)
- 7) Assassinate2 (target opponent 2)
- 8) StealFrom1 (target opponent 1)
- 9) StealFrom2 (target opponent 2)

- 10) ChallengeOrBlock (Challenge and block are never simultaneously valid so we represent these 2 actions as 1 action)
- 11) NoOp

Note: Depending on the state of the game, not all actions are valid.

### State

- 1) PlayerState 1
  - a) Number of coins ( $0 \leq i \leq 12$ )
  - b) Card 1 ( $0 \leq i \leq 3$ )
  - c) Whether card 1 is alive or not (True/False)
  - d) Card 2 ( $0 \leq i \leq 3$ )
  - e) Whether card 2 is alive or not (True/False)
- 2) PlayerState 2 - Same format as player state 1
- 3) PlayerState 3 - Same format as player state 1
- 4) Current player index ( $0 \leq i \leq 2$ )
- 5) Current player's action ( $0 \leq i \leq 10$ )
- 6) Blocking player index ( $0 \leq i \leq 2$ )
- 7) Turn Phase ( $0 \leq i \leq 3$ )
  - a) Begin turn - The current player needs to submit an action while their opponents submit a NoOp action.
  - b) Challenge current player's right to play their action - The current player must submit a NoOp action. Their opponents submit either a Challenge action or a NoOp action.
  - c) Block current player's action - The current player must submit a NoOp action. Their opponents submit either a Block action or a NoOp action.
  - d) Challenge right to block - If a player decides to submit a Block action in the previous phase, their opponents must submit a Challenge action or a NoOp action. The blocking player may only submit a NoOp action.

Note: Some values of the state are only relevant for specific values of Turn Phase (e.g. if we are in the Begin Turn phase, the "blocking player's index" value is ignored). Thus, it is possible for two states to have different values but be considered an equivalent state.

We also removed the Ambassador card due to the difficulty of modeling its interactions.

### Rewards

Gaining  $x$  coins:  $+x$

Losing/spending  $x$  coins:  $-x$

Causing an opponent to lose a card (1 remaining): +30

Losing your first card (1 remaining): -30

Causing an opponent to lose a card (0 remaining): +100

Losing your last card (0 remaining): -100

Winning the game: +300

Note: We really only care about winning the game. However, to help our policies navigate the state space, we shape the rewards and provide rewards for actions that lead to winning the game (as determined by domain knowledge).

The reward values chosen here are human crafted rather than inherently a part of the game.

### Transitions

The transition function is a function of 3 actions from all 3 players. State transitions are deterministic given the 3 actions.

$$T(s'|s, a_1, a_2, a_3) = \delta_{s'}(\text{StepGame}(s, a_1, a_2, a_3))$$

where  $\delta$  is the Kronecker delta function and *StepGame* is a function that outputs a new state deterministically given a current state and all 3 players' actions.

From the perspective of a single agent, we treat  $a_2$  and  $a_3$  as unknown parameters that transform the transition function into a probabilistic function.

$$T(s'|s, a_1) = \sum_{a_2, a_3} T(s'|s, a_1, a_2, a_3) * P(a_2|s) * P(a_3|s)$$

where  $P(a_2|s)$  and  $P(a_3|s)$  are unknown to the agent.

## III. PROBLEM STATEMENT

We present two versions of this game. First, as a warm up, we consider a version where the full state is exposed to all agents, meaning each agent knows their opponents' cards. In this environment, agents are only uncertain about the actions that their opponents will take. Formally, they are uncertain about  $P(a_2|s)$  and  $P(a_3|s)$ . This is the model uncertainty case.

Next, we consider a version where agents are also unable to see their opponents' cards (as in a real game). In this version of the problem, we have both model uncertainty and state uncertainty. To learn about their current state, agents are able to observe a vector of 2 integers that represent the actions their opponents took.

In both cases, we wish to develop algorithms that can learn optimal policies that perform well under uncertainty and are able to win the game.

## IV. BUILDING COUP SIMULATOR

We created a robust coup simulator where we can input different game parameters such as the number of players, the type of players, set the initial cards for each of the players or choose randomly, and recreate instances of the game that we want to explore more. Our simulator follows the MDP as described above. In a single instance of a game, we can step through and choose each action we want the player to make. For each possible state, the simulator outputs tuples containing our possible actions, the possible states we can enter, and the rewards associated with that transition. We can extend this simulator to run several games where the action is determined by the type of player and the simulator then outputs the winner at each round.

## V. ONLINE METHODS VS OFFLINE METHODS

There are two approaches to solving MDPs. We can either precompute an entire policy for all possible states or compute optimal actions in real time as we are playing the game. We first considered offline methods such as value iteration. We began by computing the upper bound for our state space. For a single player state, there are  $12 * 4 * 2 * 2 = 768$  possible values. For the entire state, there are upwards of  $768^3 * 3 * 11 * 3 * 4 = 179, 381, 993, 472$  possible values. (Note that this is an upper bound because some states may have different values but are considered equivalent states as explained earlier). With a state space this large, clearly, it is computationally infeasible to apply offline methods and compute an optimal action for every possible state. Thus, we are forced to only consider online methods. The number of possible next states is much less than the overall number of states which makes the problem more tractable.

## VI. METHODS TO SOLVE MODEL UNCERTAINTY CASE

As a warm up, we started by making a simplifying assumption that agents had knowledge of their opponents' cards. In other words, when a player is about to select an action given a state, the state provided contains not only their two cards and the number of coins that each player has but also the two cards of each of their opponents.

We tried three different methods:

- 1) Depth limited forward search / lookahead
- 2) Monte Carlo Tree Search
- 3) Sparse Sampling

In each method, to resolve the model uncertainty, we assume

$$P(a_1|s) = \frac{1}{|A|} \text{ and } P(a_2|s) = \frac{1}{|A|}$$

In other words, we assume our opponents are random.

### A. Depth Limited Forward Search / Lookahead

Forward Search simply looks ahead from some initial state to some depth,  $d$ . The algorithm iterates over all possible actions and next state pairings until the desired depth is reached. We assume our players are random and thus all possible outcomes are equally likely. We choose the action that yields the highest utility in expectation. We do not use a discount factor and set  $\gamma = 1$ .

### B. Sparse sampling

Sparse sampling avoids the worst case exponential complexity of forward search by using a generative model to produce samples of the next state and reward given an initial state. Instead of exploring all possible outcomes by iterating through all possible opponent actions, we randomly sampled a subset of all possible opponent actions.

## C. Monte Carlo Tree Search

Unlike sparse sampling, the complexity of MCTS does not grow exponentially with the horizon. Instead of exploring all possible actions, we only explore a random subset of possible outcomes that can result from a given action. Essentially, we do not iterate through all possible opponent actions but rather randomly sample a few of them. In addition, instead of calculating the expected utility to be the utility after 4 steps, we run simulations using a random policy as a heuristic for the expected utility with an infinite horizon (i.e. until the game ends).

## VII. METHODS TO SOLVE STATE UNCERTAINTY CASE

We now move on to a model that more closely resembles a real game where players do not have knowledge of their opponents' cards. We can extend the three methods we've previously discussed to the state uncertainty case. Again to resolve the model uncertainty, we simply assume our opponents are random. To deal with the state uncertainty over opponents' cards, we adopt a uniform belief over all the card combinations our opponents' may have. In essence, this simply means we explore more possible next states instead of those restricted by cards our opponent's actually hold (since we no longer have that information). The core of the algorithms remain the same.

### A. Forward Search With Discrete State Filtering

We now discuss a method that can do something smarter than just assume a uniform belief over our opponents' cards. Previously, we assumed we did not know our opponents' strategies. Thus, we simply assumed they were random players and as a result were forced to assume a uniform belief over all possible cards they may have since no information can be gleaned from our observations of our opponents' actions (since they are random and by definition independent of the underlying state). However, if we are given our opponents' strategies, we can do better. With this extra knowledge, we are able to construct an informed belief over the cards they may have from the actions they choose. Essentially, we assume we know  $P(a_1|s)$ ,  $P(a_2|s)$  where  $P(a_1|s) \neq P(a_1)$  and  $P(a_2|s) \neq P(a_2)$ . We can then use Bayes theorem to find  $P(s|a_1)$ ,  $P(s|a_2)$  and form a belief over the underlying state (i.e. our opponents' cards)

With the goal of beating an opponent that uses Monte-Carlo Tree Search, we implemented a player that uses Forward Search with Discrete State Filtering to update belief. We followed the Forward Search algorithm outlined in section 6.5.2 of the textbook [2] and we used Discrete State Filtering from section 6.2.1 to update our belief.

1) *Modeling Belief:* We held a belief over all possible permutations of cards that our opponents could have. In our case, there were four total types of cards and our opponents could each have two cards, so our belief space was of size  $4^4 = 256$ . To save space and computation, we structured our belief to be comprised of two sub-beliefs which were beliefs over just one opponent's possible cards. In other words we kept track of 2 beliefs, each of size  $4^2 = 16$ . We made

the assumption that one player’s cards were independent of another player’s cards. This allowed us to simply multiply our two sub-beliefs to compute our total belief over the state space. While this assumption is not completely accurate to the game in real-life, for our purposes, it is okay.

2) *Modeling Observations*: Our model of Coup involves three players. Every turn, we as the player submit an action and our two opponents also each submit an action. We modeled our observation as the actions that our opponents each took. Consider a game of Coup between our Forward Search player and two opponents, which we will call *Opponent1* and *Opponent2*. If after 1 round of the game, *Opponent1* took action  $a_1$  and *Opponent2* took action  $a_2$ , then our observation for that round would be the tuple  $(a_1, a_2)$ . To calculate  $O(o|s)$ , we simulated our opponents taking an action given the state  $s$ .

$$\begin{aligned} a_1 &= Action_1(s) \\ a_2 &= Action_2(s) \end{aligned}$$

Since our opponents were using MCTS to implement their policy, our opponents’ actions were (almost) always deterministic given a state. In other words, given a state  $s$ , our opponents would always do the same action. We could therefore model our  $O(o|s)$  function using Kronecker delta functions  $\delta$  as follows:

$$\begin{aligned} O(o|s) &= O(o_1, o_2|s) \\ &= P(o_1|s)P(o_2|s) \\ &= \delta_{o_1}(Action_1(s))\delta_{o_2}(Action_2(s)) \end{aligned}$$

where  $o_1$  and  $o_2$  are the actions of *Opponent1* and *Opponent2* respectively and where  $a_1$  and  $a_2$  are the actions we got from calling  $Action(s)$  on our opponents above.

3) *Optimizations*: During Forward Search, we need to calculate the term  $P(o|b, a)$ . From the way we modeled observations, our observation is not dependent on our action  $a$ . Therefore, we were able to save time by pre-computing  $P(o|b, a)$  outside the for loop iterating through our action space. Additionally, to save computation, as discussed earlier we are able to split up our observation into two smaller observations.  $o_1$  represents opponent1’s action and  $o_2$  represents opponent2’s action. Together, our complete observation is  $o = (o_1, o_2)$ . We calculate  $P(o|b, a)$  as follows:

$$\begin{aligned} P(o|b, a) &= P(o|b) \\ &= P(o_1, o_2|b) \\ &= P(o_1|b)P(o_2|b) \\ &= \sum_{s \in S_s} P(o_1|s)b_1(s) \sum_{s \in S_s} P(o_2|s)b_2(s) \end{aligned}$$

$b_1$  and  $b_2$  are the 2 sub-belief states we discussed earlier, which each hold beliefs over one players hand. Thus, in the above equation  $b_1$  and  $b_2$  are beliefs over two-card state spaces (16 total states) rather than the full 4 card-state space (256 states). Likewise  $S_s$  above represents the state space made up of two cards (16 total states).

## VIII. RESULTS

Forward search with a fully observable state beats random agents all the time for a depth greater than or equal to 4. A depth of 4 corresponds to the state where the player is given reward for selecting an action. A depth of 2 corresponds to the state where the player is given reward for either challenging or blocking another player and therefore, while random may sometimes beat Lookahead with a fully observable state for a depth of 2, Lookahead will always win the challenges and blocks.

TABLE I  
FORWARD SEARCH INCOMPLETE VS 2 RANDOM PLAYERS

Depth	Win %	Time(sec, 1 action)
1	19	.17
2	76	.42
3	80	1.23
4	93	5.74

In Table 1, we have Forward Search agent with a partially observable state competing against two random agents.

TABLE II  
SPARSE SAMPLING COMPLETE STATE VS. RANDOM PLAYERS

Depth	# samples	Win %	Time(s/1000 games)	Time(s/action)
1	1	70.9	10.98	.00045
1	2	69.9	18.49	.00090
1	5	70	46.42	.0021
2	1	94	15.93	.0015
2	2	97.9	44.32	.0050
4	1	92.8	42.66	.0043
4	5	100	7889	1.1015

TABLE III  
SPARSE SAMPLING INCOMPLETE STATE VS RANDOM PLAYERS

Depth	# samples	Win %	Time(s/1000 games)	Time(s/action)
1	1	69.7	9.4	.00046
1	2	67.6	16.38	.00086
1	5	70	36.29	.0020
2	1	93.5	11.96	.0009
2	2	97.5	31.41	.0029
4	1	77.6	30.26	.0020
4	5	90	4777	.530

In Table 2, we have a Sparse Sampling agent with access to the full state competing against Random agents. In Table 3, we have a Sparse Sampling agent with access to the partially observable state against two Random agents. We can see that in both figures the larger the depth and the more samples that we take, the better our agent performs. Yet, the greater the depth and number of generative samples, the more time it takes. The fully observable state agent always outperforms the agent with the partially observable state, but it takes less time for the partially observable state to choose its action. Since this is a board game and a small percentage increase in winning is not very significant, then it might be better to prefer speed.

In Table 4, we have a MCTS agent with access to the fully observable state competing against two Random agents. In

TABLE IV  
MCTSINCOMPLETE VS. RANDOM PLAYERS

Depth	# sims	Win %	Time(s/1000 games)	Time(s/action)
1	4	29.8	25.1	.0007
1	10	25.5	53.7	.0017
4	1	3.5	41.81	.00072
4	4	32.4	98.41	.0025
10	10	60	355.58	.018
10	20	73.9	721	.031
20	10	55.2	541	.027

TABLE V  
MCTSCOMPLETE VS. RANDOM PLAYERS

Depth	# sims	Win %	Time(s/1000 games)	Time(s/action)
1	4	70.4	19.97	.0007
1	10	69.6	42.2	.0011
4	1	3.5	29.43	.00044
4	4	64.2	68.33	.0017
10	10	82	230.26	.010
10	20	90.3	486	.019
20	10	73.4	393	.024

Table 5, we have a MCTS agent with access to the partially observable state competing against two random agents. The same reasoning for variation in speed and win percentage hold in Table 4 and Table 5 as in Table 2 and Table 3. MCTS agents take less time than Sparse Sampling to choose an action but have a reduced win percentage.

TABLE VI  
SPARSESAMPLINGCOMPLETE VS. 2 SPARSESAMPLINGINCOMPLETE PLAYERS (10 GAMES)

Depth	# samples	Win %
1	1	0
1	5	0
2	1	20
2	2	40
4	1	60
4	5	70

TABLE VII  
SPARSESAMPLINGINCOMPLETE VS. 2 SPARSESAMPLINGCOMPLETE PLAYERS (10 GAMES)

Depth	# samples	Win %
1	1	0
1	5	0
2	1	20
2	2	30
4	1	10
4	5	10

In Table 6, we have a Sparse Sampling agent with complete state competing against two Sparse Sampling agents with incomplete state. In Table 7, we have the opposite: a Sparse Sampling agent with incomplete state competing against two Sparse Sampling agents with complete state. Notice that if the depth is less than four, then our agent never wins. This is

because all the agents believe that they are playing against random agents and assume that the other players will lie uniformly. However, none of the agents are willing to lie themselves since the expected reward for lying is  $-30 * 0.5 = -15$  if we assume our opponents are random and will challenge us 50% of the time. Since none of the agents are willing to lie, the third agent always wins by taking income and coupling. Once the depth is greater than four, then the agents are looking past just the first action and therefore, they all should have an equal probability of winning.

TABLE VIII  
MCTSCOMPLETE VS 2 MCTSINCOMPLETE PLAYERS

Depth	# sims	Win %
1	4	0
1	10	0
4	1	0
4	4	0
10	10	30
10	20	0
20	10	50

TABLE IX  
MCTSINCOMPLETE VS 2 MCTSCOMPLETE PLAYERS

Depth	# sims	Win %
1	4	0
1	10	0
4	1	0
4	4	0
10	10	10
10	20	0
20	10	40

In Table 8, we have an MCTS agent with complete state information competing against two MCTS agents with incomplete state. In Table 9, we have the opposite: an MCTS agent with incomplete state competing against two MCTS agents with complete state. The same logic can be held for the pair of figures as in Table 6 and Table 7.

For Forward Search with Discrete State Filtering, we were able to implement an initial version but it ran too slow for us to gather meaningful results. However, we do believe that if we refactored our code to avoid redundant computation, Forward Search with Discrete State Filtering would be a feasible method that would work in practice. We could also explore different approximation methods such as particle filtering and others.

## IX. DISCUSSION

Because this a friendly board game where there isn't any monetary incentive to win, it is better to sacrifice a bit on the chance of winning it for a quicker game. Forward Search with a partially observable state is more accurate than any of the methods that use a generative function but costs a lot more time. Therefore, it might be more valuable to use either Sparse Sampling or MCTS to compute the best action to take.

We believed, initially, that we could use offline methods to generate an optimal policy to play the game. We realized

that the state space is too large to feasibly iterate through and update all state action combinations. We then decided to try and reduce our state space by removing an entire role, creating qualitative classes to represent the quantitative number of coins and removing the opponents cards from the state. It turns out, that still the order of the magnitude of the state space is still huge and unfeasible to run value iteration or Q-learning.

To extend this project, it would be interesting to consider how we can create players to challenge our player that uses forward search with particle filtering. Inspired by the level-K model, if we know that all our opponents are implementing forward search, can we leverage that knowledge?

## X. WORK BREAKDOWN

Adrien is taking this class for 3 units and tackled representing Coup as an MDP. He also implemented Forward Search, contributed to debugging Sparse Sampling and MCTS and helped analyze the results. David is also taking this class for 3 units and was integral in the development of optimizations for Forward Search with discrete state filtering and transforming our results into a presentable manner. Semir is taking the class for 4 units and combed through the literature. He also contributed to the initial design of Coup as an MDP and creating the simulator. He implemented Sparse Sampling, MCTS, and contributed to Forward Search with discrete state filtering. He helped outline the report and ran the simulations to generate the results and interpreted the findings.

## REFERENCES

- [1] Srivastava, Biplav, and Gita Sukthankar. "Reports on the 2016 IJCAI workshop series." *AI Magazine* 37.4 (2016): 94.
- [2] Mykel J. Kochenderfer; Christopher Amato; Girish Chowdhary; Jonathan P. How; Hayley J. Davison Reynolds; Jason R. Thornton; Pedro A. Torres-Carrasquillo; N. Kemal Ure; John Vian, "State Uncertainty," in *Decision Making Under Uncertainty: Theory and Application*, MITP, 2015, pp.