Julius Cheng, Rebecca Krauthamer, Ido Ofir, Sabeeka Siddiqui

CS227B: Final Paper

Team: Always Play Scissors

Description of General Game Playing Agent

The overall strategy of our final game playing agent AlwaysPlayScissors combines of alpha-beta pruning, UCT, and an efficient propnet to enable those searching strategies.

Move selection

The final version of our game player performs the following every turn; if there is only one move available, use the entire duration to build and search the UCT tree. If there is more than one move, use the first half of the allowed time to do UCT, and the second half to do an alpha-beta search. Thus, UCT forms the "core" of the strategy, while alpha-beta is a measure to avoid guaranteed losses and pursue guaranteed wins.

The following will be a description of the key components of the agent, roughly in descending order of importance.

UCT

We use the UCT algorithm as described by Kocsis and Szepesvűari with opponent modeling modifications. Basic UCT builds a search tree and iteratively searches the path that maximizes $\bar{X} + c\sqrt{\frac{2\ln(t)}{s}}$, where $\bar{X}$ is the average reward from simulations pursuing that path, c some constant, t the total number of trials performed, and s the total number of visits down that path. We use a minimax scheme to model the opponent; since the next state can only be generated from the joint move of all players, we pursue the joint move where our agent maximizes $\bar{X} + c\sqrt{\frac{2\ln(t)}{s}}$ and the opponents collective choose the joint move that minimizes $\bar{X} - c\sqrt{\frac{2\ln(t)}{s}}$. We see that this models simultaneous turn games, but also models turn based games, since players whose turn it is not will have a single action that the UCT algorithm must always choose.

This addition confers distinct advantages over vanilla UCT or pure Monte Carlo. In turn-based games, we achieve minimax behavior so as to avoid incorrect optimism, since the expected value of an action approaches to the minimax as t approaches infinity. Also, it gives us a predictive model for simultaneous turn games that converges to an equilibrium.

However, this model is imperfect primarily due to our over-general minimax assumptions, which we will discuss in a later section. Of course, it also suffers when too few trials are performed.

Alpha-beta pruning

We use the standard alpha-beta pruning algorithm with iterative deepening, plus a few generalizations: we virtually split a turn into two turns where our agent moves first, then all opponents jointly produce a response. This models an n-player game with any turn-taking style as a 2-player turn-taking game.

Our inclusion of alpha-beta is in some ways redundant with UCT, which has minimaxing and pruning features, but we kept it because empirically it often able to find correct moves faster and deeper than UCT. We felt that in most cases, $\frac{1}{2n}$ as many trials for n-player non-simultaneous games (since we don't alpha-beta on turns where there is no choice), does not sacrifice as much correctness as alpha-beta earns.

Our choice to split the time equally between UCT and alpha-beta was arbitrary; generating a heuristic to determine an optimal split in any game seems hard, impossible, or simply not worthwhile.

Propnet

We use the Java propnet factory provided to generate a propnet, and we use the generated Java object for all of our computations. Upon startup, we initialize the propnet and immediately produce the topological ordering used for propagation. We use a topological ordering to forward propagate rather than recursively backwards from a set of target nodes, since the latter does not guarantee that each proposition will be calculated exactly once.

While we have not optimized the structure of the propnet itself in any way, we compute subsets of the topological ordering for particular purposes. The motivation is that if we are interested in the value of a particular subset of propositions, then it only makes sense to compute the values that feed into those propositions. Specifically, we compute separate orderings for the terminal node, for each player's legal actions, for all the goal nodes combined (not for each goal, as we'll discuss later), and for transitions.

Our algorithm to do this works as follows: for the special ordering for some set of nodes S, recursively get all of the inputs to S and put them in a set T until reaching a base, input, or initial proposition. Then iterate through the original total ordering, checking if each node is in T. If so, then append it to the special order. We do this for all of the special orders in one pass, and the time to compute is almost always negligible.

As an example, we checked the relative ordering sizes for various games. For Checkers (Tiny), the total ordering contains 4223 nodes, the terminal ordering contains 575, and the goal ordering a mere 50. In Connect Four, the same orderings contain 229, 20, and 25 nodes respectively. So we achieve an order of magnitude of efficiency for these and many other games over a naive propagation strategy.

Memory considerations

There are two places where we store data: in the UCT tree and in a state machine cache.

UCT, of course, requires persistent data in the form a tree that grows as it is searched. But the tree can get quite large, and it is always the case that after moving to the next state, we can discard all but one branch of the original tree. Therefore, we leverage native Java garbage collection by maintaining only a pointer to the root of the UCT tree, and when reaching another state, we set the pointer to the approach child state. The other children are automatically discarded.

To cut down on redundant computations, we implemented caches that save the output of some state machine computations. Most importantly, when alpha-beta solves the minimax value for a state, we save the result such that we pretend that that state is terminal and the goal value is the minimax score. This way, alpha-beta never recomputes that state. We also have tried saving the output of state machine methods such as getting the goal, getting legal moves, etc. in a cache to avoid recomputation. This yielded some benefits when we were still using the prover state machine, but upon switching to propnet, the additional overhead of caching and checking the cache was higher than the cost of simply recomputing in most cases. It still yielded some time savings when the propnet was very large, such as in Minefield, so ultimately, we added a metagaming check to test propagation speed versus caching. If caching was slower, then it was disabled it for the game.

Our cache implementation used dual HashMaps. Putting and getting was done on a primary cache. Whenever something was accessed from the primary cache, we put it in the secondary cache. Whenever total memory usage exceeded a certain level, we set the primary cache to the secondary one, and create a new empty secondary. Again, this leverages native garbage collection.

In all cases where we evacuate stored data, we do it at the beginning of a turn to avoid stop-the-world garbage collection. We do this as a precaution; we had not previously observed any case of garbage collection causing the agent to miss deadlines.

Latch detection

We successfully implemented detection of latches and used them when appropriate to determine outcomes. To do so, we generated an *implication graph*, where each literal was represented by a vertex. Edges represented implication. We compute edges by iterating through each literal and searching its closest output propositions, and creating an edge when we find that a literal implies another, using boolean logic rules to pass through logic gates when appropriate. For instance, for the express $(\neg a \vee b) \rightarrow c$, it is true that $\neg a \rightarrow c$ and $b \rightarrow c$.

Once we have an implication graph, we use Tarjan's linear-time algorithm for retrieving strongly connected components. If any strongly connected component contains exactly one base proposition, then we consider it a latch. (We could not solve the case where there is more than one base proposition because of implications across transitions; for instance, in Connect Four, the *control* propositions for each player implied each other through transitions, and clearly neither are latches.)

If any latch implies a goal proposition, then we consider any state containing that latch to be terminal, and the score of the state the score of the implied goal. The case implied false goals was trickier, since a goal is only guaranteed if the falseness of all other goals are guaranteed.

Our latch detection scheme successfully detected a latch and used it to win Minefield 2p Easy as well as the game in the final tournament that included latches.

To date, we have not seen any "naturally occurring" latches that force goals, latches that exist by an accidental feature of the game and not by design.

Factoring

We did not successfully implement factoring for general disjunctively factorable games. We only successfully factored Lights On given its special case. We hindered by the difficulty of correctly flooding the propnet and excluding special nodes, designing a way to give information about factors to the player, and having the player making intelligent decisions about factors.

Nonetheless, we found our player to be effective in the disjunctively factorable games surveyed by the class; our agent computed enough to make reasonably intelligent decisions in each of those cases.

Metagaming

Our metagaming tasks were covered in the above sections: 1) computing the propnet, 2) computing optimized orderings, 3) detecting latches, and 4) deciding whether or not to cache. We used the remaining time to start the UCT search early.

Goal coercion

In many games, score is given as 100 in a win, 50 in a tie, and 0 in a loss. This is not the case for many games such as Free-For-All, where the score is 10 times the number of captures. In all competitions in the class, only winning or losing is significant, not the actual score. Since our core strategies use minimax and choose based on expected value, we needed a solution.

Our final player "coerces" the goal values to be 0, 50, or 100. It does this by checking all goals that are true in a state. If a player's score is higher than the max of all other players' scores, then the score is 100. If it is tied then 50, and 0 if lower. This addition adds a miniscule computational burden while enabling reasonable minimaxing in games such as Free-For-All.

This assumption is actually harmful in certain cases, specifically if the game has cooperative elements, or if performance is measured by an aggregate of actual score over many games. But, it was very helpful given the evaluation technique used in the class.

Perils of minimax

Minimax is a largely successful strategy, but in the general game-playing setting, and especially in our implementation, it leads to several pitfalls: 1) games are not necessary zero-sum, so the agent did not assume cooperative behavior when appropriate 2) it assumes that the other player is symmetrically minimaxing, which is not always true 3) it assumes that multiple opponents are conspiring against it, which leads to "hopelessness" in situations that are not at all hopeless 4) in simultaneous turn games, it assumes that opponents play best responses to the move chosen, which contains the false assumption that opponents can predict the agent's moves.

To rectify this, one could use a "maximax" model, where each opponent chooses to maximize their individual score. Simultaneous turn games are trickier, since there are game theoretical considerations and are in principle not optimizable in the absence of dominant strategies.

Development history

The final core strategy of UCT was implemented late in the quarter. Along the way, there were several schemes that we tried. We implemented one-step mobility and focus heuristics, and performed a linear regression of random samples during metagame to estimate weights. These were discarded quickly in favor of pure Monte Carlo, which were a great improvement. But due to the slowness of the prover state machine and flaws of pure Monte Carlo, the agent still played in a way that, from a human perspective, did not feel intelligent.

We consider propnets and UCT to be categorically superior to those previous methods. Propnets are nearly always faster and more space-efficient. UCT with minimaxing adds opponent modeling to traditional Monte Carlo methods, and is generally more efficient by pursuing "interesting" paths. Also, UCT is a direct estimate of expectation, while the heuristics we surveyed had a weak correlation at best in all but the most contrived of settings.

Final performance

We are proud (and relieved) to report that we are at the top of the leaderboard for this offering of the class. Certainly, Tiltyard is not the best measure of overall performance, and Agon skill, Elo, and average score are all imperfect measurements of game-playing agents. But AlwaysPlayScissors currently sits at the top of each of these categories among the agents in this class, which we believe to be a strong indicator of skill. Additionally, out of 66 games, our agent has only one reported error, which was due to human error (accidentally shutting down the host machine mid-game).

We do believe that our agent was particularly positioned to perform well in the types of games on Tiltyard. Generally the games were 2-player, turn-based, and small in the number of propositions needed to represent the game. In these settings, our UCT/alpha-beta minimax model, combined with our fast-enough propnet yielded tens of thousands of trials per turn and a deep alpha-beta search (depth 9 in Connect Four in 8 seconds). Given the set of games on Tiltyard, it is no wonder that our agent performed so well.

An interesting example of a case where we do not perform well occurred on Tiltyard, where there was a game of Three Player Connect Four in which our agent determined through UCT that it was in a forced loss situation, and the estimated scores ranged from 2-4 (out of 100). It needlessly picked an action that allowed the next player to win immediately. It turned out that, in most cases of Three Player Connect Four, although our agent is pessimistic due to its opponent model, the perceived best move still most often corresponded to legitimately good moves. In this special case, it perceived the opponents to perfectly conspire to force a loss, and the error rate was high enough compared to the estimation such that it essentially played randomly. Generally, our win rate in Three Player Connect Four covered the majority of games, but we were nonetheless susceptible to these "mistakes".

In the in-class tournament, we reached 2nd place, losing at the final match. The games played often ran counter to the settings we excelled in, but we were pleasantly surprised in some aspects of our performance, and disappointed in others. All the games before the final one were simultaneous turn, so we were not quite certain how UCT would fare. After reading through the logs of those games, we realized that because UCT expected perfect opponent responses, it weighted all actions close to 50 for most of the starting stages of games. So because of minimax, it would play moves that were not harmful, but not necessarily advantageous. It only made progress when the opponent made a suboptimal move, which it would then capitalize on.

Another interesting case was in Burning Sheep. Because of our goal coercion scheme, winning by herding sheep was weighted as equally as ending the game by destroying the opponent's field. Our player earned 10 points from herding sheep, then immediately ended the game.

Our flaws were revealed in the final match, where we played Cephalopod against PGGGPPG. We performed poorly most likely due to a combination of the game's depth and the turn time allowed (15 seconds). Recalling that on the agent's turn, it splits its time between UCT and alpha-beta, we only spent 7 seconds on UCT and 7 seconds on alpha-beta. The 7 seconds of UCT yielded 40-50 trials on average, and the alpha-beta results were mostly useless since there were no shallow terminal conditions. The other team mentioned that they achieved roughly 300 trials per turn, having compiled their propnet to binary. From a human perspective, PGGGPPG made reasonable choices, while our agent made moves that were immediately recognizable as mistakes.

We attribute the loss primarily due to inefficient use of time, which were excused in easier games and longer turn lengths. Therefore, to address these issues, we would focus on 1) determining an intelligent allocation of time between UCT and alpha-beta, probably eliminating alpha-beta in deep games, and 2) propnet optimization and compilation. Since our loss was due to failure to execute UCT, and not perceivably due to an irreparable flaw in UCT itself, our solution, at least within the spirit of our stated strategy, would be to power UCT as much as possible.