Our final player uses Monte Carlo Tree Search to explore and expand the state tree. To determine whether to exploit or explore, our implementation uses UCT with a fixed constant balancing the observed value of a state and its number of visits. The player does not combine any other heuristics with MCTS; however, if we approach a timeout in the middle of a depth charge, then we use the goal value of the state currently being examined as the assumed goal value of the charge (a kind of goal-proximity heuristic). While we have implemented the propositional network state machine, we found better performance with the prover state machine. During the metagaming phase, the player begins building the state tree from the initial state using MCTS. This gives our player a better estimation of the value of future states.

We implement MCTS using a max-max strategy. Rather than assume our opponents will work to minimize our score, we assume our opponents will work to maximize their own. More specifically, we treat the game as taking place in sequence; that is, rather than having joint moves, we operate as if our player moves, then the next player moves with that prior knowledge, then the next, and so on until every player has a move for the state. This interpretation leads to a state tree where each layer represents a particular player. When we backpropagate the results of our depth charge, we update the utility of each node according to the player whose move is represented by it.

This leads to an interesting data structure for our nodes. Each node bears a single utility value, a player ID, an associated state, and a list of actions entered by previous players for that state. The associated player ID is used in backpropagation to apply the correct utility score. The list of actions is equal to the parent’s list of actions, in addition to whatever move the given node represents. If the list of actions has an action for every player, then it is emptied and the associated state is updated based on those actions.

Our code is structured around the general Heuristic interface. When running minimax or compulsive deliberation search, the player passes in an implementation of the Heuristic object. This object’s getValue function (defined by the particular implementation) is called when a heuristic value of a state is required. This can be triggered after achieving a certain depth or when close to timing out. Different player pass different heuristics to their search function. Our MCTS player does no search, so it immediately runs the MCTS algorithm and picks a move according to the values produced.

Our player’s max-max strategy allows it to perform well against other players. Against players with the same level of optimization as ours, we tend to win, since we do not assume the game is zero-sum. Our player, however, does poorly on games requiring factoring or where high numbers of fast depth charges are needed. In the final competition, we did very well on games such as Chinese Checkers and Free-for-All, but poorly on games like Eightpuzzle and Big Hunter.

We launch 260 depth charges on turn 1 and 276 depth charges on turn 2.

We also implemented several heuristics-based players, including mobility, goal proximity, and opponent mobility. These players used minimax or compulsive deliberation until reaching a timing threshold, at which point they used heuristics to analyze state value. We also created a weighted heuristic player, which uses some weighted combination of these heuristics.

We implemented a propositional network with backward propagation. For this task, the data structures we needed were already provided, all we needed to do was use the mappings from GDL sentences to propositions that were provided in order to build a set of them that defined the current state. Propagating states was definitely the most challenging aspect of the propnet code, but there were many other difficulties that we encountered in smaller tasks, such as in initializing the propnet and clearing it correctly. One of the failed extensions that we attempted was defining a type for each node when the propnet was being built, but this failed when we realized nodes labeled “anon” were hard to capture. This was largely the result of a codebase difficult to understand for propnets- we were typecasting nodes within the functions that built the sets of input/base propositions/etc but there was no such thing for anon nodes and we were unsure of how to achieve this. It would be greatly appreciated if there were better documentation for propnets, of all the notes it was the section we found most lacking of adequate guidance and led to a lot of frustrating debugging. Had we spent less time learning the propnet codebase from the ground up we could have implemented factoring and other propnet extensions.

Source Code:

<https://github.com/vishalsubbiah/General-Game-Playing>

During the course, we found that implementing the algorithms themselves was not hard, but understanding and improving the underlying architecture through propnets took significantly longer to build. The earlier assignments were quite straightforward and did not take as long to implement as we thought it would. This surprised us later as propnets took us four weeks to build rather than the two we assumed it would take. Due to this we were unable to complete the last two assignments factoring and optimizing the propnets.