IEOR8100: Economics, AI, and Optimization
Lecture Note 7: Search in Extensive-Form Games

Christian Kroer*

February 27, 2020

1 Introduction

We previously saw how to compute a Nash equilibrium of a two-player zero-sum extensive-form game (EFG) by using dilated distance-generating functions or the CFR framework. We also saw that even if computing gradients $g_t = Ay_t$ is too time-consuming we can still run algorithms using gradient estimates constructed via sampling. However, for some real-world games such as two-player no-limit Texas hold'em, this is still not enough. The game tree in this game has roughly 10^{170} nodes, and the strategy space is much too large to even write down strategy iterates. Faced with this situation, we need to make even coarser approximations to our problem.

One major innovation for solving large-scale poker games was the use of *real-time search*. Traditionally, poker AIs were created by precomputing an approximate Nash equilibrium for some extremely coarsened representation of the full game using e.g. CFR$^+$. Then, that offline strategy was simply employed during play. In real-time search, the precomputed Nash equilibrium approximation is refined in real time for subgames encountered during live play. This allows the AI to reason in much more detail, especially towards the end of the game, where the encountered subtree is manageable in size. In order to understand how search works in EFGs, we will first show how it works in the simpler setting of perfect-information EFGs, where there are no information sets, and so players know exactly which node they are currently at. Search in perfect-information EFGs has historically been extremely successful, it was used in AI milestones on Chess and Go.

2 Backward Induction

Perfect-information EFGs (meaning that all information sets consist of a single node) can be solved via *backward induction*. Since the game is played on a tree, and a player always knows exactly where in the tree they are, we can reason about the optimal strategy at a given node purely by considering the subgame rooted at the node. We do not need to worry about what happens in any other parts of the game tree. Backward induction exploits this fact by recursively solving every subgame. It starts at leaf nodes, and then at any internal node, the algorithm pick the action that leads to the best subgame for the player acting at the node (breaking ties arbitrarily). An example is shown in Figure 1.

*Department of Industrial Engineering and Operations Research, Columbia University. Email: christian.kroer@columbia.edu.

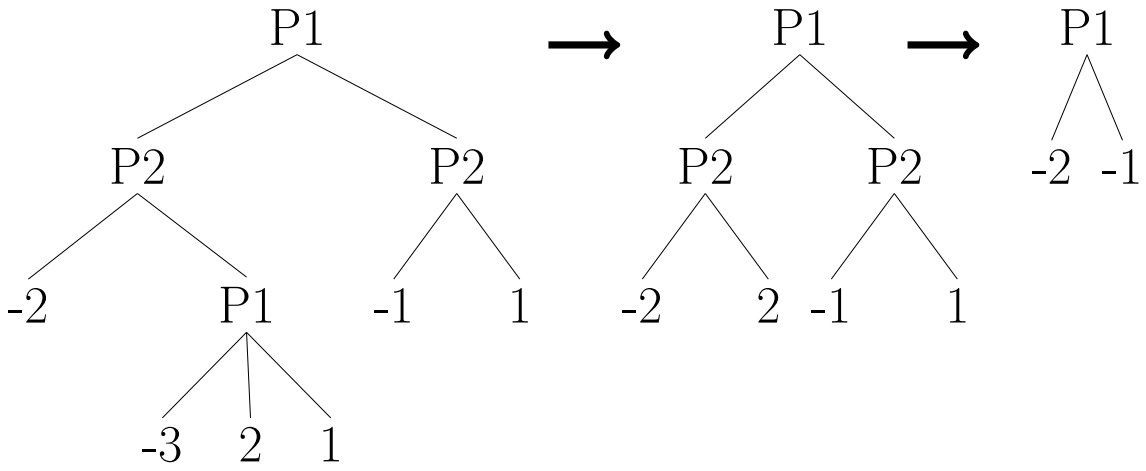
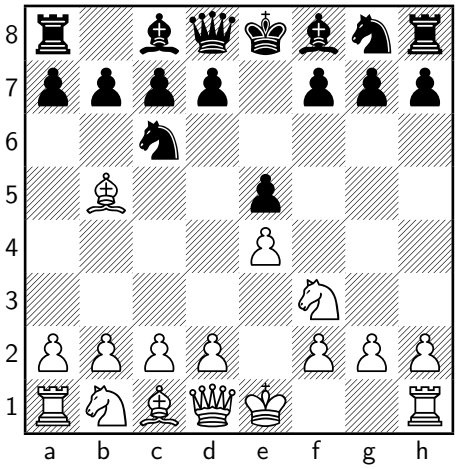


Figure 1: A perfect-information EFG solved via backward induction. P1 maximizes leaf node values and P2 minimizes. First P1 picks the best action at the bottom node. Then we replace that P1 node with its value 2. Then P2 picks the best action at each of their bottom nodes, and we replace those nodes with their value. Finally, P1 chooses an optimal action at the root.

3 Search in Games

In search, we search for a solution in real-time during play. Say that we are playing chess, which is a perfect-information EFG. Say that some set of moves already happened, resulting in the board state shown below:

1 e4 e5 2 ♘f3 ♘c6 3 ♗b5



In order to decide a next move for black, we can now perform real-time search. We perform backward induction starting at the subgame rooted at the current board state. What this means is that we try all sequences of legal moves starting with the current state, and then we pick the best action based on having solved the subgame via backward induction. However, unless we are close to the end of the game, the size of the subgame usually makes backward induction much too slow. Instead, the search is performed only up to a certain depth, say 10 moves ahead. This generally won't get us to a leaf node of the game, and so instead we replace the nodes at depth 10 with fake leaf nodes that we assign some heuristic estimate of the unique value that would have resulted from backward induction (we will call these fake leaf nodes *subgame leaf nodes*). In order to do that,

we need to construct an estimate of what value an internal node would have in the solution. A visualization is shown in Figure 2.

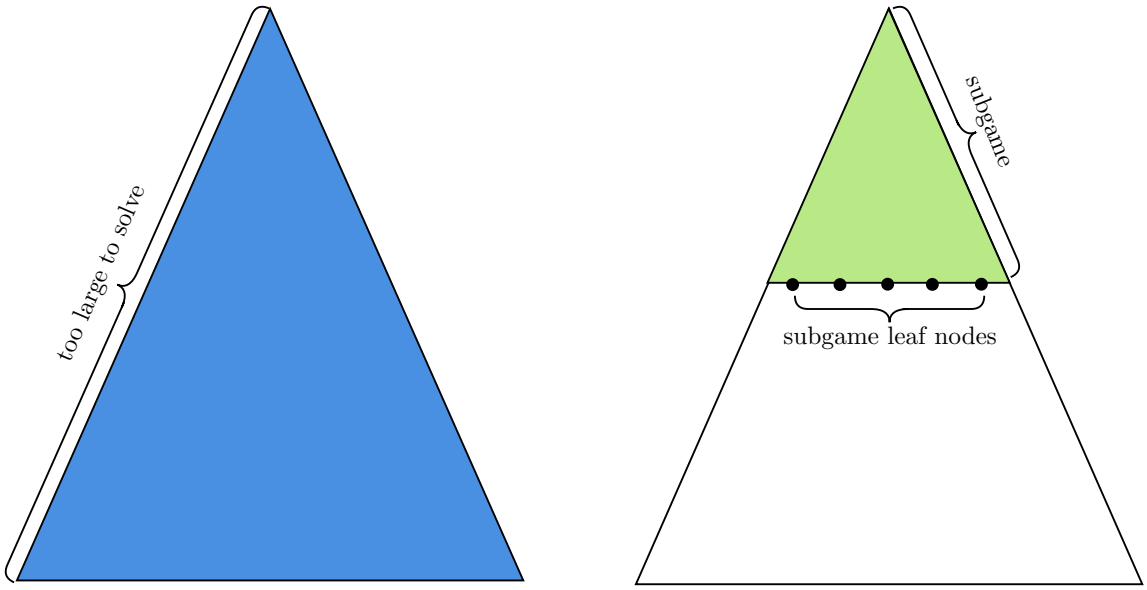


Figure 2: A large game truncated to a depth-limited subgame starting at the root.

In order to estimate the value of some internal node h in the game tree, we assume that we have some *value estimator* $v : H' \to \mathbb{R}$, where H' is the set of nodes in the game tree that are leaf nodes in the subgame. Each subgame leaf node h is then assigned the value $v(h)$ in the subgame. In perfect-information games each node h has some unique value associated to the solution arising from backward induction. In that case, our goal is simply to have $v(h)$ be a good approximation to this unique value. If $v(h)$ provides perfect estimates then backward induction in the subgame recovers the solution to the original game.

So how do you get a value estimator? It can be handcrafted based on domain knowledge (this was done for *Deep Blue*, a chess AI which beat Garry Kasparov, at the time considered the best chess player in the world); it can be learned by training on expert human games (this was done by *AlphaGo*, a Go AI which beat Lee Sidol, a top-tier professional Go player); or finally it can be done via self-play (this was done by *AlphaZero*, a generalization of AlphaGo, and *Pluribus*, a poker AI that beat humans at 6-player poker).

For imperfect-information games such as poker, things are more complicated. The primary issue is that backward induction no longer works: The value of a given node cannot be understood purely in terms of the subtree rooted at the node. Instead, we must take into account the rest of the game tree. Further complicating matters is the fact that a node does not have a single well-defined value; the value of a node may change depending on which Nash equilibrium we are considering. Finally, even if we manage to estimate the value of a node in equilibrium, we may end up choosing a strategy where the opponent can best respond in order to exploit us in the truncated part of the game tree. This is easily seen by considering the EFG representation of rock-paper-scissors: At the root node player 1 chooses rock, paper, or scissors. Then, player 2 has a single information set containing all three nodes corresponding to each choice for player 1, and they choose rock, paper, or scissors at that information set. If we truncate the game at depth 1 and assign each player 2 node its value in equilibrium (which is 0), then player 1 ends up with 3 actions, all leading to a payoff of 0. Thus, for the subgame player 1 can choose any pure strategy, e.g. always play rock,

and based on the subgame think that they achieve a value of zero. However, once we play in real time, if our opponent knows that we truncated the game and picked rock, they may exploit us by playing paper.

We can resolve this issue as follows: instead of having a function $v(h)$, we can have a function $v(h|p)$ that estimates the value of h, conditional on p. Here p is a probability distribution over the subgame leaf nodes. For the rock-paper-scissors example, $v(h_r|[p_r, p_p, p_s])$ would estimate the value of, say, the rock node h_r conditional on the distribution $[p_r, p_p, p_s]$ over the 3 possible nodes. This is obviously a much more complicated value estimator, since we are now trying to construct a mapping from $H' \times \Delta^{|H'|}$ to $\mathbb{R}$. This is the approach taken in the *DeepStack* poker AI, which beat a group of professional poker players in two-player no-limit Texas hold'em. Values are estimated using a deep neural network that was pretrained by generating random distributions over subgame leaf nodes, and then solving each of the subgames defined by truncating the *top* of the game, and having a chance root node that randomizes over the subgame leaf nodes using the randomly generated distribution.

Once an estimator v has been constructed, real-time search with this setup looks as follows:

1. Define a subgame by looking k moves ahead
2. Solve the subgame using a regret-minimization algorithm for EFGs (e.g. CFR or OMD with dilated DGF)
3. What are the leaf node values that we should use?
4. On each iteration t of the regret-minimization algorithm:
   (a) Set strategies x_t, y_t based on RM algorithms
   (b) x_t, y_t defines a probability distr. $p(Z)$ over subgame leaf nodes
   (c) For each leaf node z, ask value network for estimates $v(z, p(Z))$
   (d) Set loss for the x player to $g_t = A_t y_t$, where A_t is the payoff matrix associated to the subgame with subgame leaf estimates $v(z, p(Z))$. Define the loss for the y player analogously.

If the value network is perfect, then this setup computes a strategy for the subgame that is part of some Nash equilibrium in the full game.

To summarize the approach described here: we train our value network offline, e.g. by generating random distributions over nodes, and solving those subgames. This generates training data. Then, during play we use the already-trained value network to solve subgames as we encounter them.

This still leaves open the question of how to solve the subgames needed to create the value network, since those subgames could be very large themselves (e.g. subgames starting near the root of the game tree). One way to do it is to start by randomly generating shallow games near the bottom of the game, say depth d_1. Once we have a good value network for predicting the value of nodes at deptth d_1, we can move up one level. Next, we randomly generate distributions over nodes at depth d_2, and truncate those games at depth d_1 using our value network that we already constructed for depth d_1. We can then apply this recursively.

So far we have described our methodology and examples as if we are solving a depth-limited subgame starting at the root node of the game tree. However, in practice we would like to solve subgames starting at arbitrary decision points in the game. In perfect-information games this is easily done. We may treat it exactly the same as solving from the root, since every node provides

a well-defined subgame with that node as the root. However, in imperfect-information games this is not so.

To construct an imperfect-information EFG subgame, we assume that we have so far been playing according to some *blueprint* strategy which we computed ahead of time (our opponent need not follow the blueprint strategy in practice). Typically this blueprint strategy would be computed using CFR^+ on a very coarsened abstraction of the game.

When constructing a non-root subgame in an imperfect-information game, we will in general not know exactly which node we are at, and so instead we would have to start the subgame at the information set that we are currently at. But even taking all of the nodes in the current information set as the root (and applying Bayes' rule to derive a chance node that selects among them), will not be enough. In particular, nodes in subtrees rooted at the information set may be in information sets that contain nodes that are *not* in any of the subtrees. To remedy this, we construct our subgame by starting with all nodes in subtrees rooted at the current information set. Then, we add to our subgame every node that shares an information set with at least one node currently in our subgame. We then repeatedly add nodes in this fashion, until we reach the point where there are *no* nodes outside the subgame which share an information set with any node in our (now much larger) subgame. Finally, in order to finish our construction we need a probability distribution over all the nodes that are at the top level (i.e. same level as the information set we wanted to create a subgame for) of this new subgame. The most naive approach would be to make a single chance node as the root, and use the conditional distribution over the set of top-level nodes given our blueprint strategy. This approach is typically called *unsafe subgame solving*. The reason it is called unsafe is that we are generally not guaranteed that we will be weakly better off by applying subgame solving, as compared to our blueprint strategy. By not considering the rest of the game, it turns out that we might open ourselves up to exploitation. Nonetheless, unsafe subgame solving is often used in practice.

There are various methods for performing "safe" subgame solving. These typically require adding additional gadgets to the unsafe subgame construction, either by enforcing that the opponent achieves a certain level of utility in the subgame (this prevents us from overfitting to the subgame), or replacing the initial chance node with a number of opponent nodes, where they can reject the subgame unless they achieve a certain utility level.

4 Historical Notes

Search was used in several poker AIs that beat human poker players of various degrees of expertise, both in two-player poker [9, 2] and 6-player poker [3].

Endgame solving, where we solve the remainder of the game, was studied in the unsafe version by Ganzfried and Sandholm [7]. Safe endgame solving was studied by Burch et al. [6], Moravcik et al. [8] and Brown and Sandholm [1]. The more general version of subgame solving where we do not have to solve to the end of the game was studied by Moravčík et al. [9], Brown et al. [5, 4].

References

[1] Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games. In *Advances in neural information processing systems*, pages 689–699, 2017.

[2] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[3] Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, 365 (6456):885–890, 2019.

[4] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. *arXiv preprint arXiv:1811.00164*, 2018.

[5] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. In *Advances in Neural Information Processing Systems*, pages 7663–7674, 2018.

[6] Neil Burch, Michael Johanson, and Michael Bowling. Solving imperfect information games using decomposition. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[7] Sam Ganzfried and Tuomas Sandholm. Endgame solving in large imperfect-information games. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 37–45, 2015.

[8] Matej Moravcik, Martin Schmid, Karel Ha, Milan Hladik, and Stephen J Gaukrodger. Refining subgames in large imperfect information games. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.