Accelerating Information Set Tree Searches with CountSketch and MinHash

Sketchy Algorithms for Sketchy Games

William Bernardoni

December 19, 2019

Contents

1 Motivating Problem

Starting last summer, I competed in a tournament for a game called Reconnaissance Blind Chess[4] hosted by the APL. This tournament was in the goal of developing algorithms for managing uncertainty in an imperfect information game where you were allowed to make directed observations.

The game worked like this: the set up and the piece movements and features (i.e. castling, en-passant, ect.) were identical to chess, except with an additional option to pass your turn, and to attempt "illegal" moves like having a pawn attack a tile where there is no piece. Any illegal move would be identical to a pass.

However you were not told your opponents move, you were only told if they took one of your pieces and on which tile if they did. Similarly on your moves you were told what move ended up being played, and if it took a piece, but not which piece.

The main source of information was that at the start of each turn you could "scan" a 3x3 square, and see all of the pieces on those tiles.

A game was won by capturing the opponent's king. No checkmate was needed, and they were not told if they were in check.

After the tournament the top ranked players were invited to NeurIPS, and there we discussed how each of our bots worked. Surprisingly to me, with the exception of one (the LeSalle bot), they were nearly identical. Each bot kept track of all of the potential states, and looked at the moves that a regular chess engine generated for each state. They then selected the best move based off of a heuristic. Each bot used different heuristics, and each one had its share of different design quirks and little features, but in essence each of them were driven by an engine that was playing Chess, not Reconnaissance Blind Chess. At most every bot looked at the effect of their chosen move on a single board state, but did not consider the different sets of information that could rise out of that move further in the game tree.

This had bothered me during my own development of my bot, as this problem seemed insurmountable. I had policies on how to scan, and different methods for sketching the state space and the like, but when my bot was considering a move it was not at all considering the information gained by that move, or if that move would be good when it has little grasp on the board state. It was playing Chess with one eye closed and the other squinting, and not really playing the right game.

I made a version of the bot that did a tree search based off of the sets of states it could be at, but what I found is that the game tree of that imperfect information version of Chess was exponentially larger than that of Chess – and as Chess is already a huge game tree, it became a completely intractable problem. To compute a single move using an Alpha-Beta search with a depth of only 3 moves took two full days – which is not acceptable when the time limit for a whole game was fifteen minutes.

The question that kept bothering me was this: how do we make that game tree approachable?

How could we accelerate that tree search, and exploit any symmetries that arise out of the set structure of the information sets?

While the approach below was developed too late for the tournament, I believe there is worth in the question past the specific game of Reconnaissance Blind Chess, as many different decision tasks in the real world can be modeled as games of imperfect information.

2 Terminology and Background

When analyzing games of imperfect information we can think of our state in terms of an "Information Set". How we will use the term in this paper is defined as below:

Definition. *An **Information Set** is a set of game states I reachable in an imperfect information game such that:*

- *The player P whose action it currently is is identical between each state $x \in I$*
- *When the game reaches a state within I, the player P has no knowledge that would distinguish between element of the set I.*

This definition is similar to the one found in "Game Theory: A very short introduction" [1]

We will be focusing in on games similar to the ones discussed before, where it is based off of an "underlying" perfect information game. We will call that the **Base Game** and the imperfect information version of the game the **Induced Game**. While it may be possible to create an imperfect information game that does not have a base perfect information variant, the vast majority of encountered imperfect information games seem to have such a variant.

For instance in Poker, the Base Game is the game where all cards are dealt face up, and all players are aware of each card. The Induced Game is the one in which we are not aware of the cards. The game tree is the same between the two games, but rather than knowing the true game state, each player instead has a set (an Information Set) of potential hands the other players could have been dealt.

A key distinction with Poker is that the players have little control over the state of the board. They are dealt hands, but have no method of choosing them. We will be focusing on games where the state of the board is entirely controlled by player actions, such as the ones discussed before. We will be focusing on games with no random element. The only "deciding factor" in the movement along the game tree will be that of the player's decisions. This however does not mean that our proposed algorithm cannot be used for games with randomness inherent in them, but modifications past the scope of this project would need to be made.

We will also refer to a **Value Function**. In our model of games, each player has such a function, and the goal of the player is to end the game in a terminal state with a maximal value function. We will be focusing on zero-sum, two player games, meaning that there are two participants in the game, and at the end of the game the sum of their two value functions is zero. This means that a player can only win if their opponent loses.

This value function can be extended to the non-terminal nodes in the game tree recursively via "min-max". On each parent node of a leaf of the game tree you look at all of the children. If the node is a state of the other player's turn the value of that node becomes the value of its minimally valued child, if it is a state of your turn its value is the maximum value of its children. By repeating this process you value the entire game tree in a way that at each node you can guarantee that you can do no worse than the value function, even if your opponent is playing optimally.

3 Analysis of Information Sets

We will begin with the following motivated definition (under the prior assumptions on our game – i.e. zero sum, non random, two player):

Definition. *Let I be an information set of a player, and $v : G \to \mathbb{R}$ the value function for that player in the base game, where G is the set of all game states. Then we can extend v to the induced game, and:*

$$v(I) = \min_{x \in I} v(x)$$

Each element in successive information sets is generated by a single element from I – meaning that if $x \in I$ it doesn't matter if there is some other element $y \in I$ to determine the successor nodes of x. Assuming optimal play on the opponents side, they can only ever put you in your "worst" state, and there will always be a path to a terminal node with that state's value. Which means that regardless of what states are in the information set, there is a strategy that will guarantee you a valuation of at least $v(I)$.

This gives rise to several useful inequalities.

Let X, Y be information sets, and $X \subseteq Y$, then $v(X) \geq v(Y)$, as the element which has minimal value in X is also in Y, thus the value of Y has to be less than or equal to the value of X.

This gives rise to the following:

$$X \subseteq Y \subseteq Z \implies v(X) \geq v(Y) \geq v(Z)$$

It's important to note that actually computing $v(I)$ is extremely expensive. To do that we have to traverse almost the entire tree succeeding the set I. There are various methods used, such as Alpha-Beta pruning, which can on average cut down the amount of nodes that are needed to be traversed, but the worst case cost is always that it will take exploring the entire tree – which in the case of induced games, the tree is exponentially larger than the base game.

That is why the above inequality is extremely useful. If we don't care about the exact value of $v(Y)$ – which we often don't, as we are usually looking for the maximal or minimal one – then we can determine if we need to actually compute $v(Y)$ before expending the large time cost to do so. All we need to do is find the smallest superset and the largest subset and we get bounds on the potential value of Y.

This then brings in the question, "how often do you actually see $X \subseteq Y$ where X and Y are the information sets that you encounter in a tree search?" The answer to this is highly dependent on the game, but in most imperfect information games it happens quite a bit. There are many ways to partition a set of potential states, and many of those will be subsets or supersets of others.

4 Difficulties

The found inequality doesn't fully solve our problem however. As we said earlier, the amount of information sets you can encounter is usually exponential with regards to the size of the game tree, and the game trees are usually quite large themselves. If we tried to store the value of every information set we have evaluated then we would quickly run out of space.

In addition, these information sets can be extremely large. In Reconnaissance Blind Chess for instance, most players were encountering information sets with sizes in the millions. It is quick to compute if X is a subset of Y when X and Y are small, but when X and Y have sizes in the millions that can be an extremely costly operation.

We then have two problems we need to solve:
Problem 1: We need to determine which information sets occur most often, so that we are most likely to find a subset or a superset.
Problem 2: We need a quick way to determine if $X \subseteq Y$ when X and Y can be extremely large.

5 Solutions

We will loosen the requirements of each problem slightly, and attack each individually.

5.1 Most Frequent Sets

We don't necessarily care about finding the exact most common elements, we only really care about finding some of the more common items. Problem 1 is just APPROX-TOP in a different form. We want to find k elements such that the frequency of each of those elements is greater than $(1-\epsilon)f_{[k]}$ where $f_{[k]}$ is the k-th highest frequency.

We will define our frequency as not just being an element that we see, but being an element which sets a bound on another. What I mean by this is that say in our search we encounter elements $[I_1, I_2, I_3]$ and we find that $I_1 \subseteq I_3$ then we would say $f(I_1) = 2$, $f(I_2) = 1$, $f(I_3) = 1$, as I_1 both occurs on its own, and as a found bound to I_3.

We can apply Count Sketch[3] to our problem. Count Sketch can find all heavy hitters in polylogarithmic space, and can solve APPROX-TOP in an efficient amount of space and time.

The reason I mention heavy hitters in particular, is that trees of the induced game tend to be rather "bushy", and when using a directed search like Alpha-Beta pruning you tend to have many elements that occur quite a bit, and few elements that occur not very often, so the distribution of frequencies would likely be heavily skewed.

5.2 Quickly Determining $X \subseteq Y$

We will loosen the requirement, as we don't fully need that $X \subseteq Y$ for $v(X) \geq v(Y)$, we just need that among the elements $m(X) = \{x \in X | v(x) = v(X)\}$, $m(X) \bigcap Y \neq \emptyset$.

This allows us to take a powerful step back. Assuming that the probability that a specific element is the minimal one is uniformly distributed – which is a strong assumption, but not an unreasonable one as the partitions of sets that form the information sets tend to be quite even so the probability that a specific element is in two sets tends to be quite uniform in the games which I have encountered – we can compute a probability of that relation based off of the intersection of two sets.

$$p(v(X) \geq v(Y)) \geq 1 - \frac{|X| - |X \bigcap Y|}{|X|}$$

as:

$$\frac{|X| - |X \bigcap Y|}{|X|}$$

Is the probability that the minimal element of X is not in Y.

So if we can quickly determine $|X \bigcap Y|$ we can bound our probability that we can use a set as a bound, and if it is within an acceptable threshold then we can use it as a bound.

It becomes useful to introduce a metric called the Jaccard index, given two sets A, B their Jaccard index is:

$$J(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

As $|A \bigcup B| = |A| + |B| - |A \bigcap B|$, we can rewrite the above to solve for $|A \bigcap B|$:

$$|A \bigcap B| = \frac{J(A, B)(|A| + |B|)}{1 - J(A, B)}$$

So it is sufficient to find an efficient way to find $J(X, Y)$ and store $|X|, |Y|$. $|X|, |Y|$ are integers so storing them takes little space, so we just need to find an efficient way to find $J(X, Y)$.

Luckily, there's an algorithm for that!

MinHash[2] allows you to estimate the Jaccard index with low error. Given a single hash function, and two sets A, B you compute a signature for each S_A, S_B where S_X are the minimal k values given by computing the hash of every element in X. To compare two sets A and B you look at $S_A \bigcap S_B$, the size of this intersection is approximately $J(A, B)$. This estimator has expected error in $O(\frac{1}{\sqrt{k}})$.

This turns a problem of computing set intersections of large sets, into set intersections of much much smaller sets.

Even better, given the nature of the problem, we don't even need a particularly low error. Preliminary trials suggest the application is robust to error in this estimator.

6 Proposed Algorithm

The algorithm follows from the above two analyses. For each set we compute a signature using MinHash, and we create a hash identifying the set itself (for this we can just use any hash function on each individual element and a commutative function to combine them).

We then have a queue of the k most seen elements (storing in the queue just the above signature, identifier, size of the set, and the value found for that element), where that queue is updated using the estimations given by running Count Sketch on the identifiers of each set.

When we encounter a set X in our search, we do the following:
For each element q in the queue, we estimate the Jaccard Index $J(X, q)$.
We then compute the two equations below (as derived earlier):

$$P(q \text{ is a lower bound}) \geq 1 - \frac{|q| - \frac{J(q,X)(|q|+|X|)}{1-J(q,X)}}{|q|}$$

$$P(q \text{ is an upper bound}) \geq 1 - \frac{|X| - \frac{J(q,X)(|q|+|X|)}{1-J(q,X)}}{|X|}$$

If we can lower bound them above a certain threshold then we can create a bound on the set X using q. We keep track of the smallest upper bound in value and the largest lower bound in value given by iterating across the queue, and use those final values.

If we fully evaluate X, we update its frequency in Count Sketch, and modify its position in the queue if necessary.

Similarly, if we use q in our queue as an upper or lower bound we also update its frequency.

7 Analysis of the Proposed Algorithm

The space complexity of this algorithm is polylogarithmic with respect to our error rate, as Count Sketch is.

The time complexity on each step is similarly polylogarithmic, as we just need to run Count Sketch's update at most three times per element that we encounter (One for each bound, and one for it itself). Computing the similarity per element q is takes $O(\frac{1}{\epsilon^2})$ steps where ϵ is our error rate for our Jaccard Index. We do that at most k times.

While this adds complexity at each timestep of our search, with each element we are able to bound we cut off a potentially exponential number of successor nodes that we may not need to evaluate. The efficiency of number of steps no longer needed due to the bounds versus the number of steps added total highly depends on the chosen game, so a closed form solution is not possible. However we do have some experimental results to back that there is an efficiency.

8 Experimental Results

I decided to use Reconnaissance Blind Chess as a testing platform for the algorithm, as it inspired this pursuit in the first place.

Unfortunately, on both games, running a single turn at even just a search depth of 3 turns took over two days in the baseline (without the above algorithm applied). In addition a critical bug was found in the code just two days ago, so I did not have time to run many trials.

What I did find however was this: when initialized in the same position evaluating the moves for the white player on the first turn in Reconnaissance Blind Chess with a search depth of 3:

Version	Runtime (Hours)
Baseline	55.7
Accelerated	19.4

That is, we got a speedup factor of 2.87. Unfortunately due to time constraints that was the only fixed datapoint that we were able to get.

We also care about the move valuation, it doesn't matter if we are running faster if we are not getting similar results.

The average distance between the outputted valuation on a given move from the Accelerated version of the bot and the Baseline version of the bot was less than 100 centipawns. With the engine used there was a certain amount of stochasticity, and the same board state would be valued with a variance of around 200 to 300 centipawns.

Not a single move's evaluation in the completed trial ran or the several ongoing trials has differed more than 300 centipawns between the accelerated and the baseline versions.

While I do not have any more fixed numbers in regard to speedup than that current trial, on the two other ongoing trials in Reconnaissance Blind Chess, in the time that it took the baseline version to evaluate 8 moves, the accelerated has computed more than 15, and in the time that it took the baseline to evaluate ten moves, the accelerated has computed more than 22.

It is key to note, that the accelerated version gains performance as time goes on. The amount it accelerates grows with each move evaluated. Preliminary trials also suggest that the performance gain increases with the depth of the tree increasing, however a depth of 3 is the minimal amount that we can use the algorithm on, and so testing on a larger depth is unfortunately not in the scope of this paper.

9 Further Areas to Explore

A similar method could likely be applied to speed up probabalistic searches like Monte-Carlo Tree Searches. Unfortunately Alpha-Beta searches and other deterministic searches are only used in certain games (such as Chess), but games that are even larger can be unapproachable with such deterministic searches.

If a similar method can be used for Monte-Carlo Tree Searches then a much wider class of game, and much larger games can be tackled.

In addition, generalizing this to work on n-player or non-zero sum games could also be beneficial. I suspect that in n-player games the same result would hold, however this result does currently rely on the game being zero sum. The inequalities would function differently in a non-zero sum game.

References

[1] Binmore, K. *Game Theory: A Very Short Introduction.* Very Short Introductions. OUP Oxford, 2007.

[2] Broder, A. On the resemblance and containment of documents.

[3] Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (Berlin, Heidelberg, 2002), ICALP '02, Springer-Verlag, pp. 693–703.

[4] Ryan Gardner, Casey Richardson, C. L. Reconaissance blind chess.