

Board Game Tree Evaluation

Lecturer: Russell Impagliazzo
Scribes: Will Chang, Buhm Han

January 31, 2013

When at all do randomized algorithms work better than deterministic algorithms? In this lecture we'll see that for board games, if we treat the rules of the game as a black box, randomized algorithms can provably do better than deterministic ones.

Figure 1: Game tree

1 Complete version

Say that two players are competing in a board game. Two players take turns with each other. With a tree, we can represent all possibilities of moves until the game ends. (Figure 1) Each node corresponds to a state, and every edge spreading down from a node corresponds to various choices of moves. Given that player 1 starts first, even depths of tree are player 1's turns and odd depths are player 2's turns. On leaf nodes, we have no more possible moves, meaning that the game ended. We don't consider draw games, so we assign a value for the two possibilities: 1 if player 1 wins, and 0 if player 2 wins.

Can we assign a value to the non-leaf node similarly? Yes, we can. See figure 2 showing a parent node with

Figure 2: Updating labels based on winning strategy

several leaf children nodes. If this parent node is player 1's turn, this node is assigned 1, since player 1 can force a win by choosing a path to 1. Similarly, if it's player 2's turn, the node is assigned 0 since we have at least one 0 which can be taken by player 2 as winning move. We can easily see that if a node is player 1's turn, it is assigned 0 only when all the children are 0's. Thus we can assign the value of interior nodes as a boolean formula of values of its children.

Let $v(x)$ be the label of x . Then,

$$v(x) = \begin{cases} \bigvee_{\text{child } c \text{ of } x} v(c) & \text{if it's player 1's turn} \\ \bigwedge_{\text{child } c \text{ of } x} v(c) & \text{if it's player 2's turn} \end{cases}$$

Our objective is to get $v(\text{root})$: (which determines who wins this game when 2 experts play). We have two simplifying assumptions:

1. All games are won or lost in exactly d moves.

2. For every position, there are exactly 2 possible moves.

Figure 3: Complete and Incomplete Game Tree; Evaluating p

With assumptions above, we have a complete d -depth binary tree with alternating $\vee$'s and $\wedge$'s. We call it a complete game tree. Let $N = 2^d$ be the number of leaf nodes. Now let's assume that every rules of the game are in a black box; in other words, there's no way to logically use those rules in our algorithm. Then, the following theorem shows that any deterministic algorithm to evaluate $v(\text{root})$ will have worst case time at least N .

Theorem 1. *No deterministic algorithm for complete game tree evaluation has worst case time less than N .*

Proof. As a corollary of lemma below. □

Definition 1. A monotone read-once formula F is a binary tree whose leaves are labelled by variables and interior nodes by $\vee, \wedge$.

Lemma 1. *Evaluating any monotone read-once formula F on N leaves requires N queries to variables in the worst case.*

Proof. We'd like to evaluate node p , whose parent is node g . (Figure 3) p has two children. In the deterministic algorithm one of the siblings is determined to be evaluated first; we call it x_i , and the other s . For the worst case analysis, we can simply set an adversary who knows our deterministic algorithm and tries to maximize our evaluation time. Let's see this for the case that the operation at p is $\vee$. Three possible sets of siblings are $\{0,0\}$, $\{0,1\}$ and $\{1,1\}$. Now $\{1,1\}$ is the best case, since we only have to evaluate one sibling. So in the worst case, we have at least one 0 among the two children, and the adversary can set $x_i = 0$ so that we examine 0 first. This will force the algorithm to evaluate both of the siblings, and the problem is equivalent to a new formula F' where p is replaced by s . F' has $N - 1$ nodes, and recursively it takes $N - 1$ queries to evaluate F' . (Case of $\wedge$ is similar.) □

Until now we have shown that any deterministic algorithm can't help us. Can randomness help us in this situation? Recall that we said randomness can replace expertise; then let's see how much an expert help us here. Say that a game novice is playing against an expert. The expert says "If I move first, I will win," but the novice is skeptical. How much time would it take the expert to convince the skeptical novice? Figure 4 assumes the expert is player 1 and the novice is player 2. The expert plays first, so the root node is $\vee$. Expert

Figure 4: Expert to convince skeptical novice

is trying to convince that $v(\text{root}) = 1$. In depth 0(root), expert only need to show one child is 1. In depth 1, novice have both possible moves to depth 2. Expert should convince to novice that both nodes in depth 1

are labeled 1s. To show this, expert only need to show that one node in depth 3 per each node in depth 2 that novice chose is labeled 0, meaning “If you move that, I will move this.” Thus in expert’s turn, he only show one of his winning choice; no need to evaluate both children. Then the tree width doubles in even depths and doesn’t change in odd depths. Therefore the number of leaves is reduced to $2^{(d/2)} = N^{1/2} = \sqrt{N}$. The expert can help us!

Now randomness can give us similar effect as expertise. Say that our algorithm pick one of two siblings randomly and evaluate them first. Then the adversary won’t have any knowledge of which one we will pick first.

procedure $Eval(r, d)$

- 1: Pick child c of r at random
- 2: $\alpha \leftarrow Eval(c, d - 1)$
- 3: If $\alpha = 1$ and r is $\vee$, return 1
- 4: If $\alpha = 0$ and r is $\wedge$, return 0
- 5: $\beta \leftarrow Eval(c', d - 1)$ for other child
- 6: Return β

This algorithm doesn’t help us when we have to evaluate both children regardless of our choice. If r is $\vee$ and both children were to 0’s, then we are to evaluate both of them anyway. We call it a bad case. On the other hand, if we have a 1 labelled child and a 0 labelled child, the randomness will help us as an expert with half probability. We call it a good case. Then what if we have bad cases for all the nodes in tree? Note that there’s no completely bad tree here. For example, say that one $\vee$ node is a bad case; both of children are 0’s. Then those children are $\wedge$ ’s and are their values are determined to be 0’s. Given a such child, the grandchildren will have at least one 0, which means it is a good case. In fact, the good cases and bad cases appears alternatively. So randomness will help us.

How much does randomness help? Let $T(\wedge, 0, d)$ be the expected time if $v(r) = 0$, $op(r) = \wedge$ and depth is d . We know that the good cases are $T(\wedge, 0, d)$ and $T(\vee, 1, d)$, and the bad cases are $T(\vee, 0, d)$ and $T(\wedge, 1, d)$. The good cases are symmetric, so we call them G_d . Similarly bad cases are B_d .

In bad case, we have to evaluate two good subcases. So always

Figure 5: Analyzing the Complete Version

$$B_d = 2G_{d-1}$$

In good case, see Figure 5. we will only see the case $v(r) = \vee$ without loss of generality. If the children are $\{1, 1\}$, we will end up evaluating only one randomly chosen sibling, which is a bad subcase. If children are $\{0, 1\}$, we will evaluate only one bad subcase with probability half, or we will evaluate both good subcase and bad subcase with another half probability.

$$G_d = \max \left\{ \begin{array}{l} \frac{1}{2}B_{d-1} + \frac{1}{2}(G_{d-1} + B_{d-1}) \\ B_{d-1} \end{array} \right. = B_{d-1} + \frac{1}{2}G_{d-1}$$

Substituting for B_{d-1} and solving the recurrence, we get:

$$\begin{aligned}
Gd &= \frac{1}{2}G_{d-1} + 2G_{d-2} \\
w^d &= \frac{1}{2}w^{d-1} + 2w^{d-2} \quad (\text{Guess } G_d = cw^d) \\
w^2 &- \frac{1}{2}w - 2 = 0 \\
w &= \frac{1}{4} + \frac{1}{4}\sqrt{33} \\
\Theta(w^d) &= \Theta(N^{\log(\frac{1}{4} + \frac{1}{4}\sqrt{33})}) = \Theta(N^{0.7537}) < \Theta(N).
\end{aligned}$$

2 Incomplete version

In reality, $\Theta(N^{0.75})$ is still too large to search. Recent chess game programs do this: they evaluate the game tree up to some fixed depth from the current state. Then, using a heuristic, it evaluates whether if player 1 is advantageous, scoring the state with a real number $\gamma \in [0, 1]$. Now we can replace $\vee$ with *max* and $\wedge$ with *min*. We call this the incomplete version of game tree. We can easily see that the complete version is nothing more than a special case of this incomplete version, where we restrict every node to either the real number 0 or the real number 1.

Now, the evaluation algorithm has to be changed. We add two more parameters α and β so that $Eval_{incomplete}(r, d, \alpha, \beta)$ returns

$$\begin{cases} \alpha & \text{if } v(r) < \alpha \\ v(r) & \text{if } \alpha \leq v(r) \leq \beta \\ \beta & \text{if } v(r) > \beta \end{cases}$$

The parameters α and β specify a range in which player 1 is advantageous enough to win. The algorithm to evaluate at each node is

```

procedure  $Eval_{incomplete}(r, d, \alpha, \beta)$ 
1: Pick child  $c$  of  $r$  at random
2:  $\gamma \leftarrow Eval_{incomplete}(c, d-1, \alpha, \beta)$ 
3: if  $r$  is a max then
4:   if  $\gamma \geq \beta$  then
5:     return  $\beta$ 
6:   else
7:      $\delta \leftarrow Eval_{incomplete}(c', d-1, \max(\alpha, \gamma), \beta)$ 
8:     return  $\delta$ 
9:   end if
10: end if
11: if  $r$  is a min then
12:   if  $\gamma \leq \alpha$  then
13:     return  $\alpha$ 
14:   else
15:      $\delta \leftarrow Eval_{incomplete}(c', d-1, \alpha, \min(\gamma, \beta))$ 
16:     return  $\delta$ 
17:   end if

```

18: **end if**

To analyze the running time of the incomplete version, we divide $v(r)$ into three cases:

$$\begin{cases} v(r) \leq \alpha & (\text{"lo" case}) \\ \alpha \leq v(r) \leq \beta & (\text{"in" case}) \\ \beta \leq v(r) & (\text{"hi" case}). \end{cases}$$

Let $T(\max, \text{hi}, d)$ be the expected time if $op(r) = \max$, $v(r)$ the hi case, and depth is d . By symmetry, we can group the six combinations at depth d into three cases:

$$\begin{aligned} H_d &= T(\max, \text{hi}, d) = T(\min, \text{lo}, d) \\ I_d &= T(\max, \text{in}, d) = T(\min, \text{in}, d) \\ L_d &= T(\max, \text{lo}, d) = T(\min, \text{hi}, d) \end{aligned}$$

Because of this symmetry we can focus on the case where $op(r) = \max$.

Figure 6: Analyzing the Incomplete Version

For the L_d case, we need to check that the values of both children are $\leq \alpha$. Since the operation is min for the children, each child has a H_{d-1} case, so

$$L_d = 2H_{d-1}.$$

For I_d case, there are two sub-cases: both $v(c), v(c') \in (\alpha, \beta)$ (i.e. {"in", "in"}), or it could be $v(c) \in (\alpha, \beta)$ but $v(c') \leq \alpha$ (i.e. {"lo", "in"}).

- {in, in}: There are two possibilities: we either pick the child that has $v(r)$ first, or we pick the other child that has a value $\leq v(r)$.
 1. The first sibling is an I_{d-1} case; the second sibling we have to verify that $v(c') \leq \gamma$, so a H_{d-1} case.
 2. The first sibling is an I_{d-1} case; the second sibling $\gamma \leq v(c') \leq \beta$, so another I_{d-1} case.
- {lo, in}:
 1. An I_{d-1} case and then a H_{d-1} case,
 2. A H_{d-1} case and then an I_{d-1} case.

We save more time on H_d than in I_d (as we will see), so we get

$$I_d = \max \left\{ \begin{aligned} &\frac{1}{2}(I_{d-1} + H_{d-1}) + \frac{1}{2}(2I_{d-1}) \\ &\frac{1}{2}(I_{d-1} + H_{d-1}) + \frac{1}{2}(H_{d-1} + I_{d-1}) \end{aligned} \right\} = \frac{1}{2}(I_{d-1} + H_{d-1}) + \frac{1}{2}(2I_{d-1}).$$

For the H_d case, there are three sub-cases: the value of the children are {"lo", "hi"}, {"in", "hi"}, {"hi", "hi"}.

- {lo, hi}:

- hi picked first: L_{d-1} once
- otherwise: $H_{d-1} + L_{d-1}$
- {in,hi}:
 - hi picked first: L_{d-1} once
 - otherwise: $I_{d-1} + L_{d-1}$
- {hi,hi}: L_{d-1} once

Therefore,

$$H_d = \max \begin{cases} \frac{1}{2}(L_{d-1}) + \frac{1}{2}(H_{d-1} + L_{d-1}) \\ \frac{1}{2}(L_{d-1}) + \frac{1}{2}(I_{d-1} + L_{d-1}) \\ L_{d-1} \end{cases} = \frac{1}{2}(L_{d-1}) + \frac{1}{2}(I_{d-1} + L_{d-1}).$$

Here's an intuitive reason why the running time is $\leq N = 2^d$. If the L case is $\leq 2^d$, then the I case is $\leq 2^d$, and if the H case is $\leq 2^d$, then the I case is $\leq 2^d$, and every time we get to an H case, we save some time (factor of 1.5 instead of 2).

A way to arrive at the closed form is to write the recurrence in matrix form:

$$\begin{aligned} \begin{pmatrix} I \\ H \\ L \end{pmatrix}_d &= \begin{pmatrix} 3/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 2 & 0 \end{pmatrix} \begin{pmatrix} I \\ H \\ L \end{pmatrix}_{d-1} \\ &= M^d \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \end{aligned}$$

The M^d term is dominated by the d^{th} power of the largest eigenvalue of M , which is $\approx 1.84^d \ll 2^d$.