CS 388R: Randomized Algorithms, Fall 2021 September 7, 2021

Lecture 4: Game Tree and Complexity Classes

Prof. Eric Price *Scribe: Ethan Lao, Tongrui Li*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Deterministic Game Tree

1.1 Problem Formulation

Suppose we have a game where two player, white and black, makes move by turn. At each turn, a player can make one of two moves, both leading to a subsequent deterministic state.

The state of the game can be represented by a binary tree. The k th layer in the tree represnets the number of turn. The value in the leaf node is 1 if there is a subsequent move that enable the current player to force the game into a win state.

Question Given all possible end states of game (in other words, the leaf nodes at k th layer), how many nodes does do we need to look at before knowing the state of the game at the first $k = 1$th turn?

1.1.1 Simple Case

Consider a game which ends in two turns. There are 4 possible scenarios:

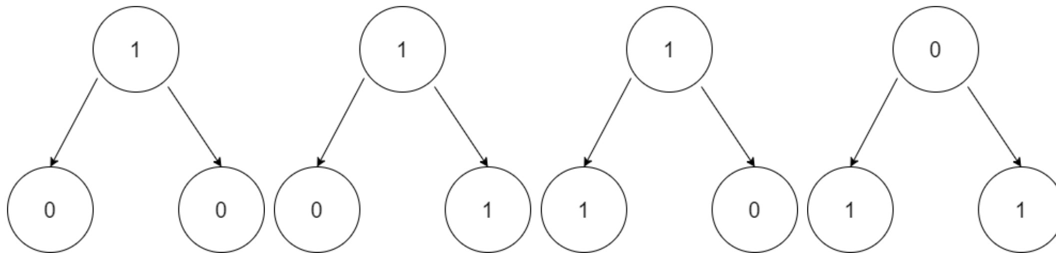

Figure 1: All possible state of a simple 2 turn deterministic game tree

The below explanation assumes that black starts out the first turn.

1. If both children contain the state of 0, that implies that that black at first turn has the ability to win the game as all subsequent state will result in white loosing, hence the root node will have a value of 1

2. If only one children nodes contain the state of 0, that implies that black at first turn can force the game in to the favorable branch (left child in the second example and right child in the third example in figure 1), thus resulting 1 in the root node.

3. If both children contain the state of 1, then that means that at the current turn, there is no way for black to win as all subsequent state will be a loose. Hence, the value at root node will be 0.

Observation The parent is exactly NAND of the children.

1.2 Deterministic Algorithm

Algorithm 1 Deterministic

Require: $n \geq 0$
  F(x):
  if $len(x) == 1$ **then**
    return $x[0]$
  else if $f(x[:\frac{n}{2}]) == 0$ **then**
    return 1 ▷ NAND Short Circuit
  else
    return 1 - $f(x[\frac{n}{2}:])$
  end if

Algorithm 1 performs a DFS search across the binary tree and essentially NAND its child together from the last leaf layer. The algorithm performs a short circuit when the **left child's value is 0**, as $NAND(0, *) = 1$.

1.2.1 Run time under adversarial environment

At the worst case scenario, we have a tree whose left value is always 1 at every level, which effectively disable the short circuit in algorithm 1. Hence, the algorithm will have to look at effectively the entire tree, resulting in $O(n)$ nodes traversed.

1.3 (Optimal) Non Deterministic Algorithm

Algorithm 2 Non Deterministic with Oracle

Require: $n \geq 0$
  F(x):
  $x_{first} = g(x)$
  $x_{second} = \not g(x)$
  if $len(x) == 1$ **then**
    return $x[0]$
  else if $f(x_{first}) == 0$ **then**
    return 1 ▷ NAND Short Circuit
  else
    return 1 - $f(x_{second})$
  end if

Lets assume an oracle g that outputs the optimal node to look at first. Incorporating this into the algorithm will **maximize the occurrence of short circuit** and greatly improve the runtime.

1.3.1 Runtime Analysis

Let:

- k = depth of tree = number of turn
- $L(k)$ = maximum number of nodes to look at if first player loses (root of entire tree = 0)
- $W(k)$ = maximum number of nodes to look at if first player wins (root of entire tree = 1)

We can analyze the runtime by induction

Base Case Trivially, $W(0) = 1$ and $L(0) = 1$

Inductive Case $L(k)$ will require both child's value to be reviewed (and equal 1) according to section 1.1.1, hence we would need 2x the computation from the previous layer, which will result in $L(k) = 2W(k-1)$.

$W(k)$ will require only one child's value to be 0. Under the assumption of an oracle, this implies that we at most need to look at one child in this instance. Therefore, $W(k) = L(k-1)$.

We can then make the following statements:

$$W(k) = L(k-1) = 2W(k-2) \tag{1}$$

$$W(k) \simeq 2^{(\frac{k}{2})} \simeq 2^{\log_2(n)/2} \simeq \sqrt{n} \tag{2}$$

Hence, the runtime is roughly $O(\sqrt{n})$

1.4 Randomized Algorithm

Algorithm 3 Randomized

```
Require: n ≥ 0
  F(x):
  x_first, x_second = rand_shuffle(x[n/2 :], x[: n/2])
  if len(x) == 1 then
      return x[0]
  else if f(x_first) == 0 then
      return 1                                   ▷ NAND Short Circuit
  else
      return 1 - f(x_second)
  end if
```

It is unrealistic to assume that an oracle described in section 1.3 exists. Hence, instead of optimally picking the child, we **randomly** pick the child first to traverse through.

1.4.1 Runtime Analysis

We mostly follow the same format as in section 1.3.1.

Base Case Trivially, $W(0) = 1$ and $L(0) = 1$

Inductive Case $L(k)$ will **still require both child's value to be reviewed** (and equal 1) according to section 1.1.1, hence we would need 2x the computation from the previous layer, which will result in $L(k) = 2W(k-1)$.

$W(k)$ will require only one child's value to be 0. It is possible to have 3 senerios:

1. Under assumption of children = $\{0,1\}$ or $\{1,0\}$, we can look at 0 or 1 first, which means that about **half of the time we can short circuit** $L(k-1)$**, and the other half of time we will be forced to look at both child** $(W(k-1)+L(k-1))$. This will result in a runtime of $\frac{1}{2}L(k-1)+\frac{1}{2}(W(k-1)+L(k-1))$
2. Under assumption of children = $\{0,0\}$, we can guarantee a short circuit, which will result in runtime of $L(k-1)$

In practice we take the maximum of the two cases. Note that the first case = $\frac{1}{2}W(k-1)+L(k-1) > L(k-1)$ = second case. Hence, we can then make the following derivation

$$W(k) = max(\frac{1}{2}L(k-1)+\frac{1}{2}(W(k-1)+L(k-1)), L(k-1)) \tag{3}$$

$$W(k) = \frac{1}{2}W(k-1)+L(k-1) \tag{4}$$

$$W(k) = \frac{1}{2}W(k-1)+2W(k-2) \tag{5}$$

$$\tag{6}$$

The above can be viewed as a modified Fibonacci sequence, whose runtime is roughly $O(2^k)$. Hence, lets assume that $W(k) = X^k$. We can then do the following

$$X^k = \frac{1}{2}X^{k-1}+2X^{k-2} \tag{7}$$

$$X^2-\frac{1}{2}X-2 = 0 \tag{8}$$

$$X = \frac{\frac{1}{2}\pm\sqrt{\frac{1}{4}+8}}{2} \tag{9}$$

$$X = \frac{1\pm\sqrt{33}}{4} \tag{10}$$

At this point, we take only the positive root. Hence:

$$X = \frac{1+\sqrt{33}}{4} \simeq 1.68 \tag{11}$$

We can then go back to deriving the complexity:

$$w(k) = \frac{1+\sqrt{33}}{4}^k = \frac{1+\sqrt{33}}{4}^{\log_2(n)} \tag{12}$$

$$w(k) = n^{\log_2(\frac{1+\sqrt{33}}{4})} \simeq n^{0.753} \tag{13}$$

Hence, the **runtime of this randomized algorithm is** $O(n^{0.753})$ It turns out that this bound is tight - this is the best we can do in this scenario.

2 Complexity Classes

The following may refer to a language L.

2.1 Review of P and NP

1. P (polynomial time): problems that can be solved deterministically in polynomial time
   - Outputs 1 $\forall x \in L$
   - Outputs 0 $\forall x \notin L$
2. NP (nondeterministic polynomial time): problems where a correct answer can be verified in polynomial time, or alternatively, problems that can be solved in nondeterministic polynomial time. We can also check an advice string (an additional input that depends on the length of the input but not the input itself) in PTIME. If $A(x) = \begin{cases} 1 \text{ if } x \in L \\ 0 \text{ otherwise} \end{cases}$,
   - $\forall x \in L, \exists$ output y s.t. $A(x, y) = 1$
   - $\forall x \notin L, \nexists$ output y s.t. $A(x, y) = 1$

2.2 ZPP, RP, and coRP

1. ZPP (zero-error probabilistic polynomial time): problems where an algorithm exists such that
   - It always returns the correct answer
   - The expected running time is polynomial for every input
2. RP (randomized polynomial time): problems where an algorithm exists such that
   - It always runs in polynomial time
   - If $x \notin L$, it will always reject
   - If $x \in L$, it will accept with probability $\geq \frac{1}{2}$
3. coRP: the complement of RP; problems where an algorithm exists such that
   - It always runs in polynomial time
   - If $x \in L$, it will always accept

- If $x \notin L$, it will reject with probability $\geq \frac{1}{2}$

To show that ZPP $\subseteq$ RP, we can simply run the ZPP algorithm for at least double its expected running time, and return the answer if the algorithm finds it. If it doesn't give an answer, reject. The chance of finding an answer before stopping is at least $\frac{1}{2}$, so ZPP $\subseteq$ RP. ZPP $\subseteq$ coRP for the same reason.

To show that RP $\cap$ coRP $\subseteq$ ZPP, suppose that $A(x)$ is a RP algorithm and $B(x)$ is a coRP algorithm. Construct the new algorithm as follows:

- Run A on the input. If it accepts, accept.
- Run B on the input. If it rejects, reject.
- Repeat until the algorithm either accepts or rejects.

Since A and B give the correct answer with probability $\geq \frac{1}{2}$, the probability of this algorithm reaching the next iteration of running A and B shrinks exponentially. Therefore, the algorithm will run A and B 2 times each in expectation, resulting in a polynomial running time. Thus, RP $\cap$ coRP $\subseteq$ ZPP, hence ZPP = RP $\cap$ coRP.

2.3 PP and BPP

1. PP (probabilistic polynomial time): problems where an algorithm exists such that

   - If $x \in L$, it will accept with probability $> \frac{1}{2}$
   - If $x \notin L$, it will accept with probability $\leq \frac{1}{2}$

   Note that NP $\subseteq$ PP. To show this, begin with an algorithm A and guess an answer y at random. If $A(x, y) = 1$ return yes, otherwise accept with a 50% probability. So if the correct answer is yes, the algorithm will output yes with probability of about $\frac{1}{2} + \frac{1}{2^n}$, showing that the algorithm is in PP.

2. BPP (bounded-error probabilistic polynomial time): problems where an algorithm exists such that

   - If $x \in L$, it will accept with probability $\geq \frac{2}{3}$
   - If $x \notin L$, it will accept with probability $\leq \frac{1}{3}$

We know that P $\subseteq$ RP $\subseteq$ NP. But, the following questions are still unknown:

- Is RP = coRP = ZPP?
- Is RP = P?
- Is BPP $\subseteq$ NP?
- Is P = BPP?

2.4 Adelman's Theorem

Let P/poly be the class of languages that have a polynomial time algorithm with a polynomial-bounded advice function (advice depends on n and the problem statement, but not on the input itself). Adelman's theorem states that BPP $\subseteq$ P/poly.

To show this, suppose there are 2^n inputs of size n and we are given a BPP algorithm A that runs in T time and succeeds with probability $\frac{2}{3}$. This implies that there exists an algorithm A' that runs in $O(Tn)$ time and succeeds with probability $1 - \frac{1}{2^{n+1}}$. Since 50% of A's random seeds work on all size-n inputs, we can pick one and fix that seed for a deterministic algorithm.