

1 Recap of Previous Lecture

1.1 Summary of Yao's Minimax Principle from Previous Lecture

The minimax principle tells us that for query problems

$$\min_{\text{rand. algs. } A} \left(\max_{\text{inputs } x} \mathbb{E}[\# \text{ queries}] \right) = \max_{\text{dist. of inputs } x} \left(\min_{\text{deterministic algs. } A} \mathbb{E}[\# \text{ queries}] \right)$$

Or simply, that the performance of the best possible ZPP (zero-error) randomized algorithm on its worst-case (e.g. adversarial) input is equal to that of the best deterministic algorithm for its worst case input distribution.

This is helpful when deriving lower bounds for randomized algorithms' complexity, using difficult input distributions to deterministic algorithms.

1.2 Revisiting the Randomized Routing Example from Previous Lecture

We need to strengthen the lemma we used in the previous lecture's proof that randomized routing of a permutation on 2^n nodes arranged in a hypercube can be done in $O(n)$ time, with high probability.

Revised lemma: [time to deliver packet i to $f(i)$] \leq [# steps in i 's path] + [# of other packets whose path intersects i 's]

Proof: Note the fact that if two paths intersect, it will be in a single interval (since we are 'fixing' the bits from left-to-right).

Say that a packet on path e_1, e_2, \dots, e_m has delay l if it is at the end of e_t at time $t + l$.

We can see that a packet p only incurs an increase in its delay (from l to $l + 1$) if it is forced to wait at an edge that another packet q is currently traversing, and that in such case q will have delay l during and after traversing that edge. Then, for any delay value l , there will always be at least one other packet with delay l traversing p 's path, until a packet with delay l exits the intersecting path segment. Because each other packet can only incur at most one unit of delay for p 's traversal, the total delay p incurs must be at most the number of other packets intersecting its path. Therefore, if packet p ends with a delay of L , there must have been at least L distinct other packets that intersected its path and left their intersection with delays $1, 2, \dots, L$.

Combining this upper bound on delay with the fact that each step in the path takes one timestep to traverse gives an upper bound on the total traversal time of packet i to its random, intermediate node $f(i)$.

2 Game Tree Evaluation / NAND Tree Evaluation

2.1 Game Tree

Consider a 2-player game, in which the players take turns making binary decisions for n turns. Each leaf is a winning state for either player 1 or player 2, and at each step the current player will attempt to take a move that guarantees the path will end at a leaf in which they win. Since one player's winning state is another's losing state, this is a zero-sum game, which means Yao's Minimax Principle is applicable.

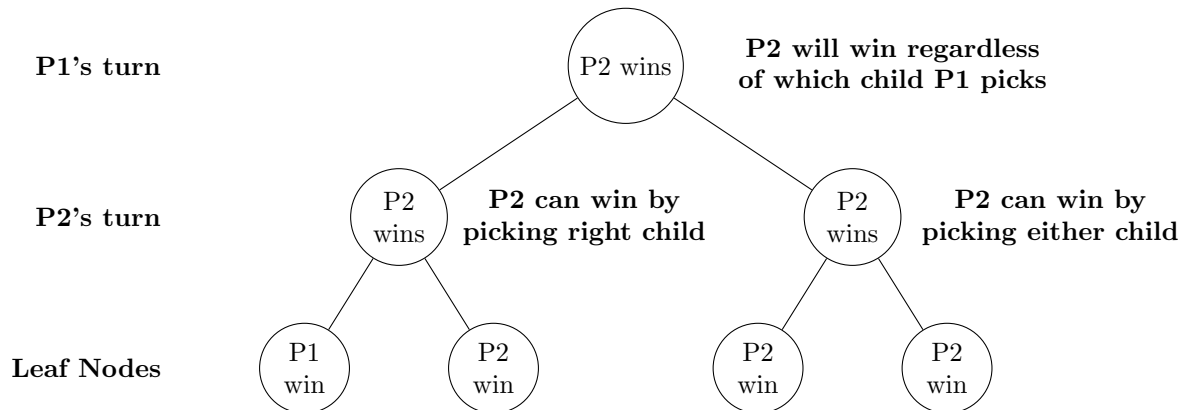


Fig. 1: Example game tree of height $n = 2$

2.2 NAND Tree

A game tree can be equivalently represented as a NAND tree, in which we label each node with a 1 if the current player is winning, and a 0 otherwise. This leads to the label of a node equalling the NAND of its children (the only way player 1 will lose from a node is if both of their moves allow player 2 to win). The NAND tree representation is generally easier to use for analyzing the complexity of algorithms on a game tree.

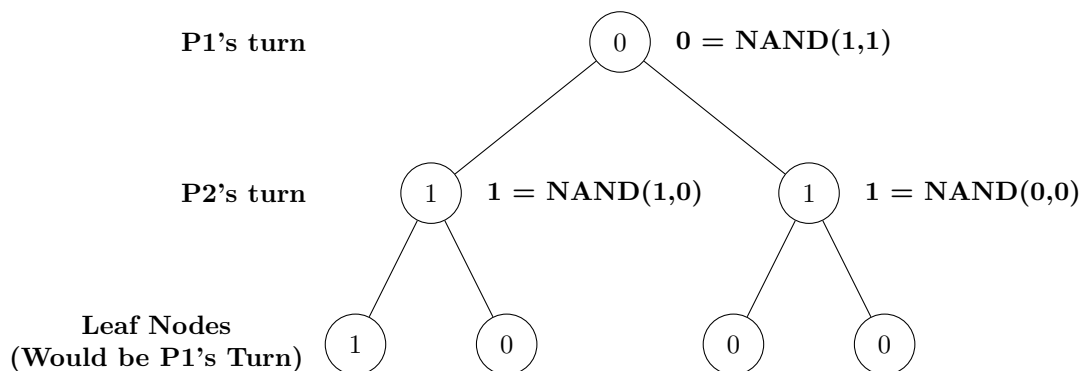


Fig. 2: Equivalent NAND tree representation of the game tree shown in Fig. 1.

2.3 Complexity of Deterministic Algorithm

Theorem: A deterministic algorithm to evaluate a game tree of height n will have complexity $O(2^n)$

Proof: For any deterministic algorithm, we can adversarially pick the inputs (leaf nodes) such that no internal node's value is determined until all of its descendants have been evaluated, requiring our deterministic algorithm to consider all 2^n leaves.

2.4 Complexity of NP Algorithms

For NP algorithms, we assume that we can magically determine the best possible set of queries needed to definitively determine the outcome of the game.

Theorem: The complexity of NP algorithms for game tree evaluation is $O(2^{n/2})$.

Proof: To prove that player X wins, we simply need to provide their winning strategy, and show that it is a win regardless of Player Y's actions. Since each player moves only on every other depth, there are only $O(2^{n/2})$ possibilities which we must consider for how player Y can respond to any move.

2.5 Complexity of Randomized Algorithm

Consider the following randomized algorithm:

```

1 eval(node):
2   if node is leaf:
3     return the value of the leaf
4   otherwise:
5     let c1 be a random child of the node, and c2 be the other child
6     if eval(c1) is 0:
7       return 1
8     otherwise:
9       if eval(c2) is 0:
10        return 1
11      otherwise:
12        return 0

```

That is, evaluate one of the children at random. If that child has value 0, the current player can win by taking the corresponding action, so we do not need to evaluate the second child. Otherwise, the child's value is 1, and we must also evaluate the second child to determine if the current player can win.

Theorem: The above randomized algorithm has upper bound $\mathbb{E}[\# \text{ of queries}] \leq 3^{n/2} = (\sqrt{3})^n \approx 1.732^n$.

Proof: Let us define:

$$A_n : \mathbb{E}[\# \text{ of queries needed to evaluate node at height } n, \text{ where node's value is } 0]$$

$$B_n : \mathbb{E}[\# \text{ of queries needed to evaluate node at height } n, \text{ where node's value is } 1]$$

For this algorithm, if a non-leaf node's value is 0, we must evaluate its two child nodes of value 1, thus

$$A_n = 2B_{n-1}$$

If a non-leaf node's value is 1, we must evaluate one child node with value 0, and with probability at most $\frac{1}{2}$ will have first evaluated a child with value 1 (either the children both have value 0, in which case we will never need to evaluate the second child; or the children have different values, in which case we have $P = \frac{1}{2}$ of picking the one with value 1 first, necessitating evaluating the second). Therefore,

$$B_n \leq A_{n-1} + \frac{1}{2}B_{n-1}$$

We know that the leaf nodes will always require a single query to evaluate, so

$$A_0 = B_0 = 1$$

Substituting the expressions for A_n and B_n into themselves, we get

$$\begin{aligned} A_n &\leq 2A_{n-2} + B_{n-2} \\ B_n &\leq 2B_{n-2} + \frac{1}{4}A_n \leq \frac{1}{2}A_{n-2} + \frac{9}{4}B_{n-2} \end{aligned}$$

In either case, to evaluate a node of height n , we need (in expectation) to evaluate at most 3 nodes of height $n - 2$. Combining this with the base case of $A_0 = B_0 = 1$, we can determine that the expected total queries is bounded by $\mathbb{E}[\# \text{ of queries}] \leq 3^{n/2} = (\sqrt{3})^n$.

Theorem: The above randomized algorithm has lower bound $\mathbb{E}[\# \text{ of queries}] \geq \phi^n = \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 1.618^n$.

Proof: We will use Yao's Minimax Principle to find a lower bound for $\mathbb{E}[\# \text{ of queries}]$ of our randomized algorithm. Recall that this entails choosing an input distribution, and then proving the performance of the optimal deterministic algorithm for that distribution.

We consider the input distribution where each leaf is independently 1 with probability p . For each node, if its children independently have probability p of having value 1, then that node has probability $1 - p^2$ of having value 1. With the goal of making the input distribution difficult, let us choose the value of p such that all nodes have an equal probability of having value 1. This requires that $p = 1 - p^2 \rightarrow p^2 + p - 1 = 0 \rightarrow p = \frac{-1+\sqrt{5}}{2}$

We use the lemma that for a NAND tree where the values of siblings are i.i.d for all sets of siblings in the tree, the best deterministic algorithm is depth-first search – essentially our randomized algorithm, but let us arbitrarily choose to always evaluate the left child first. This lemma was not proved in class, but intuitively makes sense given that the sibling pairs in our chosen distribution are all i.i.d.

For this tree, define

$$C_n : \mathbb{E}[\# \text{ of queries needed to evaluate node of height } n]$$

If the left node has value 1 (happens with probability p), we need to also evaluate the right node, meaning $C_n = (1 - p)C_{n-1} + p(2C_{n-1}) = (p + 1)C_{n-1}$. Since the leaf nodes have $C_0 = 1$, that means $C_n = (1 + p)^n$. Substituting the value our chosen value of $p = \frac{-1+\sqrt{5}}{2}$ that defines this input distribution, we see that

$$\mathbb{E}[\# \text{ of queries}] = C_n = \left(\frac{1 + \sqrt{5}}{2}\right)^n = \phi^n \approx 1.618^n > 1.414^n \approx (\sqrt{2})^n$$

By Yao's Minimax Principle, it is not possible to have a randomized algorithm that outperforms this.

Note that for the bounds of this algorithm that we proved here, the lower bound is less than the upper bound. This mismatch results from the case where a node has value 1. In the upper bound analysis, our input distribution is not fixed, so it is possible that there is no probability that such a node's children both have value 0, in which case with probability $\frac{1}{2}$ we will evaluate the child with value 1. However for the lower bound analysis since our input distribution is known, we know that there is a nonzero chance of a 1-valued node having two 0-valued children, meaning that for that node the probability we must evaluate a 1-valued child is strictly less than $\frac{1}{2}$.

It is possible to prove better bounds (with approaches involving constructing the tree top-down), although the best known lower and upper bounds are still not equal.

3 Hashing / Dictionary Data Structures

Assume that we have $S \subset U$, for universe U , and we want to store the values of S (possibly with associated data), such that we have very fast query of whether $x \in S$.

We can accomplish this with a hashing function $f : U \rightarrow \{1, \dots, n\}$ for array A of size n , where we set the values of A such that if $x \in S$, $A[f(x)] = x$. However, this requires that there are no collisions ($\forall x, y \in S, f(x) \neq f(y)$). This prevents us from using a deterministic hash function unless $n \geq |U|$ (by the pidgeonhole principle), or unless $n \geq |S|$ where S is fixed. The universe U will generally be extremely large making deterministic hashing impractical. Next lecture will explore the use of random hashing functions that with high probability do not have too many collisions.