Minimax with MapReduce

Alexander Heistermann
Missouri State University
ah4632s@missouristate.edu

Joshua Borthick
Missouri State University
jlb8976s@missouristate.edu

November 2025

1 Introduction

This project investigates how minimax search can be accelerated through parallel computation using PySpark. Traditional minimax is inherently sequential, evaluating game trees recursively from the root, which becomes prohibitively slow for large trees. We develop a parallel, bottom-up formulation that distributes node evaluations across multiple CPU cores, allowing entire tree levels to be processed simultaneously. To evaluate performance, we generate synthetic minimax trees of varying sizes and compare the runtime and correctness of the naive sequential approach against the PySpark implementation. Our results highlight the trade-offs between parallel overhead and speedup, and demonstrate when parallel minimax becomes advantageous.

2 Related Work

In combinatorial game theory, Tic-Tac-Toe and its higher-dimensional variants are modeled as Maker–Breaker games on hypergraphs, with winning lines corresponding to hyperedges [1]. Beck's work on clique games on complete graphs introduced the *fake probabilistic method*, where random-graph heuristics (e.g., the typical clique number in $G(n, 1/2)$) are used to derive sharp *breaking points* separating Maker's and Breaker's win regimes [1]. Related results around the Erdős–Lovász Local Lemma and the Neighborhood Conjecture connect local intersection constraints in hypergraphs to the existence of *strong draws*, where the second player can block every winning set, and motivate the notion of *game-theoretic independence* [2, 1]. For multidimensional n^d Tic-Tac-Toe, these techniques show that the game is a draw (in fact, a strong draw) up to dimensions $d \approx c_1 n^2 / \log n$, while more regular structures such as torus Tic-Tac-Toe and arithmetic-progression games on grids admit *asymptotically sharp or exact thresholds* for when a player can force a win [1]. Conceptually, this line of work treats each local configuration (clique, line, progression) as a "unit of analysis," aggregates their probabilistic behavior, and then derives global phase transitions in the game—an idea that naturally aligns with a map/reduce-style viewpoint.

3 The Attractor Algorithm

Algorithm 1 Attractor of Player 1 to U in a two-player game

Require: Directed game graph $G = (V, E)$ with partition $V = V_1 \uplus V_2$; target set $W \subseteq V$

Ensure: $A \subseteq V$, the winning region of Player 1 to reach W

1: $A \leftarrow W$ ▷ current attractor approximation
2: **repeat**
3: $changed \leftarrow False$
4: **for all** $v \in V_1 \setminus A$ **do** ▷ Player 1 positions
5: **if** $\exists (v, w) \in E$ s.t. $w \in A$ **then**
6: $A \leftarrow A \cup \{v\}$
7: $changed \leftarrow True$
8: **end if**
9: **end for**
10: **for all** $v \in V_2 \setminus A$ **do** ▷ Player 2 positions
11: $\mathsf{Succ}(v) \leftarrow \{\, w \in V \mid (v, w) \in E \,\}$
12: **if** $\mathsf{Succ}(v) \subseteq A$ **then** ▷ all moves lead into the previous attractor
13: $A \leftarrow A \cup \{v\}$
14: $changed \leftarrow True$
15: **end if**
16: **end for**
17: **until** $changed \leftarrow False$
18: **return** A

4 Minimax Algorithm

Algorithm 2 The Minimax algorithm

Require: Directed game tree with root node *root*, which is a max node
Ensure: *root.value* is the value the player Max will gain if he plays the best strategy

```
1:  repeat
2:      changed ← False
3:      for all v ∈ V_1 \ A do                          ▷ Player 1 positions
4:          if ∃(v, w) ∈ E s.t. w ∈ A then
5:              A ← A ∪ {v}
6:              changed ← True
7:          end if
8:      end for
9:      for all v ∈ V_2 \ A do                          ▷ Player 2 positions
10:         Succ(v) ← { w ∈ V | (v, w) ∈ E }
11:         if Succ(v) ⊆ A then      ▷ all moves lead into the previous attractor
12:             A ← A ∪ {v}
13:             changed ← True
14:         end if
15:     end for
16: until changed ← False
17: return A
```

Algorithm 3 PLAY-MINIMAX(node)

Require: A node node
Ensure: node.*value* is the value the player of *node* will gain if he plays the best strategy

```
1: if node.type == LEAVE then
2:     return node.value
3: end if
4: if node.type == MAX then
5:     node.value = −∞
6:     for all child in node.children do
7:         child_value = PLAY-MINIMAX(child)
8:         if child_value > node.value then
9:             node.value = child_value
10:        end if
11:        return node.value
12:    end for
13: end if
14: if node.type == MIN then
15:    node.value = ∞
16:    for all child in node.children do
17:        child_value = PLAY-MINIMAX(child)
18:        if child_value < node.value then
19:            node.value = child_value
20:        end if
21:        return node.value
22:    end for
23: end if
```

5 Experimental Setup

5.1 Implementation Details

We compare two implementations of the minimax algorithm on identical game-tree instances:

- **Naive (sequential) minimax.** A straightforward recursive depth-first implementation in Python that evaluates the tree from the root downwards. Each internal node is evaluated by recursively computing the values of its children and applying either a max or min operator depending on the node type.
- **Parallel minimax with PySpark.** A bottom-up, level-synchronous implementation built on PySpark RDDs. The algorithm first evaluates all leaf nodes and then iteratively propagates values upward, computing all parents whose children have already been assigned values in parallel.

A single `SparkContext` is created in local mode (`local[*]`) so that all available CPU cores on the workstation can be used.

Both implementations operate on the same JSON tree format:

- `node_count`: total number of nodes in the tree.
- `root`: identifier of the root node (always 0 in our experiments).
- `nodes`: an array of nodes, where each node has a unique `id`, a `type` in {`max`, `min`, `leaf`}, a list of `children` (empty for leaves), and an integer `value` for leaves.

Correctness is checked by verifying that both implementations return the same root minimax value for every instance.

5.2 Tree Generation

To control the size and structure of the search trees and graphs, we use custom generators that produce rooted trees and undirected graphs specifically designed for minimax and attractor evaluation. The generator proceeds breadth-first until the desired number of nodes is reached:

- The root node is a `max` node with identifier 0.
- Node types alternate strictly by depth: children of `max` nodes are `min` nodes and vice versa. All nodes without children are later relabeled as `leaf`.
- For each internal node, the number of children is drawn uniformly at random from $\{1, \ldots, C_{\max}\}$, subject to the remaining node budget, where we fix $C_{\max} = 3$.
- After the topology is constructed, all nodes with no children are treated as leaves. Each leaf is assigned a utility value sampled uniformly at random from the integer range $[\ell_{\min}, \ell_{\max}] = [-100, 100]$.

The generator exposes the following main parameters:

- `--nodes` (n): total number of nodes to generate.
- `--max-children` ($C_{\max}$): maximum branching factor per internal node.
- `--leaf-min`, `--leaf-max`: minimum and maximum leaf utilities.
- `--seed`: random seed for reproducibility.

In all reported experiments we fix $C_{\max} = 3$, $[\ell_{\min}, \ell_{\max}] = [-100, 100]$, and use a fixed seed (`seed = 42`) so that the tree for a given size is deterministic.

5.3 Graph Generation

- Node initialization: owner (Player 0/1) and priority values assigned randomly
- Edge generation: out-degree drawn from $\{1, ..., D_{max}\}$, edges sampled uniformly without replacement
- No special roles: graphs are fully general (unlike trees with roots/depth)
- Parameters: `--nodes`, `--max-out`, `--seed`
- Configuration: $D_{max} = 3$, $seed = 42$ for determinism

5.4 Benchmark Protocol

We evaluate the algorithms on a range of tree sizes

$$n \in \{10^5,\ 2 \cdot 10^5,\ 3 \cdot 10^5,\ 4 \cdot 10^5,\ 5 \cdot 10^5,\ 10^6,\ 10^7,\ 2 \cdot 10^7\}.$$

For each size n, we generate a single tree instance using the procedure above and run both the naive and the parallel implementation on that same tree.

For each run, we record:

- **Wall-clock runtime** in seconds, measured via high-resolution timestamps taken immediately before and after the call to the minimax routine.
- **Result value** at the root returned by the algorithm.

The benchmarking script automatically orchestrates the following steps for each tree size:

1. Generate the dataset (`tree_dataset_`n`.json`) if it does not already exist.
2. Execute the naive minimax implementation and record the runtime and root value.
3. Execute the parallel minimax implementation and record the runtime and root value.
4. Store all measurements (tree size, runtime of each method, root values, and a boolean `match` flag indicating agreement between implementations) in a JSON results file, which is later used for plotting and further analysis.

All experiments are run on a single multi-core workstation, with the PySpark implementation configured in local mode to utilize all available CPU cores. This setup ensures that any observed differences in runtime are due to algorithmic and framework overhead differences, rather than differences in underlying hardware.

6 Experimental Results

We evaluate the naive sequential implementation against the parallel PySpark version of each algorithm using the benchmark protocol from Section 4.1. The PySpark minimax is a bottom-up, level-synchronous RDD pipeline (map/filter/reduceByKey over frontier levels), while the PySpark attractor uses explicit RDD map and reduce operations each iteration to add newly attracted vertices. For each dataset size, each set of algorithms processes the same generated instance for their type (tree for minimax and graph for attractor), and we record the wall-clock time and the minimax value at the root or attractor path from start node to goal.

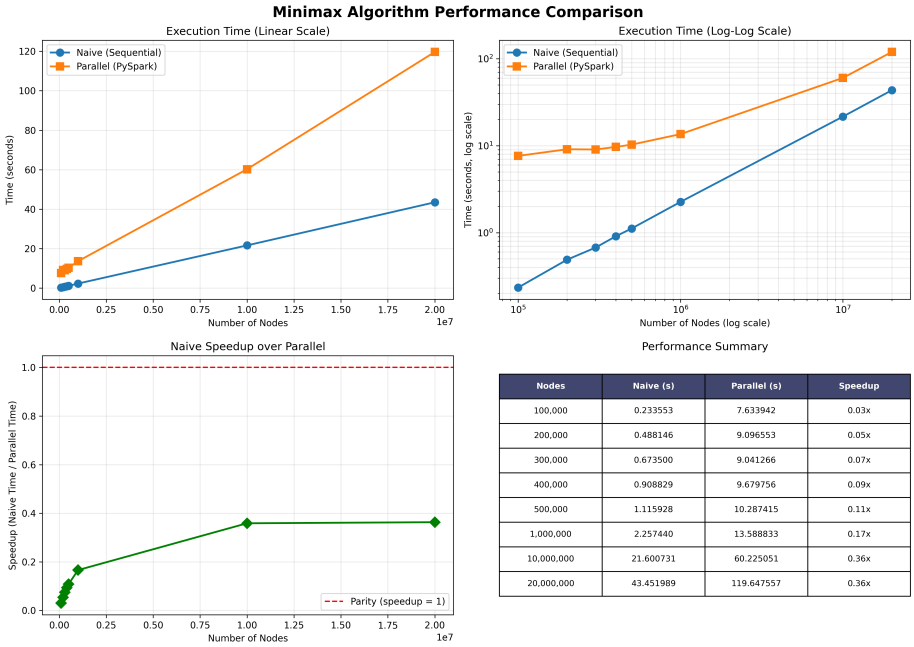
Figure 1 shows the results of the minimax setup, and figure 2 details the attractor algorithm. The top row reports execution time on linear and logarithmic scales. Across all problem sizes from 10^5 to $2{\cdot}10^7$ nodes, the naive implementation is consistently faster than the PySpark version. At 10^5 nodes the parallel execution is about two orders of magnitude slower (0.23 s vs. 7.63 s), and even for the largest tree of $2 \cdot 10^7$ nodes it remains roughly $2.7\times$ slower (43.45 s vs. 119.65 s). The log–log plot indicates that the naive implementation scales close to linearly in the number of nodes, whereas the parallel runtime curve exhibits a sizeable additive overhead plus a milder slope.

The bottom-left panel plots the speedup metric *Naive time / Parallel time*. The curve remains strictly below the parity line (speedup = 1) for all tested sizes, the minimax peaks around $0.36\times$ and attractor at $0.77\times$, each for the two largest instances. This confirms that, on a single multi-core workstation and with relatively cheap leaf evaluations, the overheads due to Spark job scheduling, task serialization, and data shuffling dominate any benefit from parallelism on smaller datasets.

7 Discussion

On a single workstation, Spark's per-job overhead dominates both minimax and attractor. The minimax speedup stays below parity across all tested sizes, topping out around $0.36\times$ at $2 \cdot 10^7$ nodes (43.45s naive vs. 119.65s Spark). The attractor path is even more overhead-bound: each iteration requires a full-graph scan, so the measured Spark runs are 80–$1000\times$ slower than naive up to 10^6 nodes, with no crossover in sight. These outcomes reflect the cost of job scheduling, serialization, and shuffles overwhelming the per-node work on a single machine.

On a distributed filesystem with real executors, the linear models from our measurements can be used to project when parallelism becomes worthwhile. Using $T_{\text{naive}}(n) = 2.16 \times 10^{-6} n + 0.009$ and $T_{\text{par}}(n, N) = 7.0 + \frac{5.7 \times 10^{-6}}{N} n$, the projections in Table **??** indicate that with $N = 16$ executors minimax crosses parity near $\approx$ 3.9M nodes and reaches $\sim$ 5–$6\times$ speedup by 10^9 nodes. With $N = 64$ executors, the projected speedup at 10^9 nodes is $\sim 22\times$, and at 10^{10} nodes $\sim 24\times$. Attractor would see similar slope benefits from scaling out,



Nodes	Naive (s)	Parallel (s)	Speedup
100,000	0.233553	7.633942	0.03x
200,000	0.488146	9.096553	0.05x
300,000	0.673500	9.041266	0.07x
400,000	0.908829	9.679756	0.09x
500,000	1.115928	10.287415	0.11x
1,000,000	2.257440	13.588833	0.17x
10,000,000	21.600731	60.225051	0.36x
20,000,000	43.451989	119.647557	0.36x

Figure 1: Minimax algorithm performance comparison on a single 8-thread workstation. Top left: execution time vs. tree size on a linear scale. Top right: the same data on a logarithmic scale. Bottom left: "speed-up" defined as Naive time / Parallel time (the dashed line indicates parity at 1.0). Bottom right: numerical summary of runtimes and speed-ups for all tested tree sizes.

but because its per-iteration full scan has a larger constant factor, it requires both large graphs and many executors before the overhead is amortized. The takeaway is that Spark parallelism is not viable on a single machine for these workloads, but becomes beneficial on a distributed cluster once the data lives in DFS and the node counts are in the multi-million (minimax) to much larger (attractor) range.

7.1 Reasoning and Improvements

If we assume an average of 17 levels in a tree structure where each parent has three child nodes across 20 million nodes, there are going to be around 8.5 RDD operations $\times 0.5\text{s} = 4+$ seconds of pure overhead to parse the tree to find the leaves, then work backward through parents to find the root. The naive version parses the tree once and broadcasts nothing, allowing for its $O(n)$ linear speed-up across nodes. This particular implementation of minimax requires $O(\text{depth}\times$ broadcast size.

A smarter implementation might memoize across partitions, which could improve overall time, but to achieve real benefits, it would need to occur on a

Nodes	Naive (s)	Spark (s)	Speedup	Match
1,000	0.032587	9.430204	0.00x	✓
5,000	0.044861	47.224397	0.00x	✓
10,000	0.059627	49.448620	0.00x	✓
50,000	0.229772	80.924978	0.00x	✓
100,000	0.515150	100.340820	0.01x	✓
500,000	2.866147	339.891938	0.01x	✓
1,000,000	6.143114	520.565451	0.01x	✓

Figure 2: Attractor algorithm performance comparison on a single 8-thread workstation. Top left: execution time vs. graph size on a linear scale. Top right: the same data on a logarithmic scale. Bottom left: "speed-up" defined as Naive time / Parallel time (the dashed line indicates parity at 1.0). Bottom right: numerical summary of runtimes and speed-ups for all tested graph sizes.

true distributed dataset like Hadoop's HDFS. Rewriting the program in Scala, using a parent-to-child index, utilizing Spark's DataFrame API, and taking advantage of RDD's caching and accumulators would help keep the Spark implementation at a mere three or four times slower than the naive approach. A tree so large it cannot fit on one machine wouldn't hurt, either! Because the number of nodes per level grows exponentially, the cost per level is unbounded.

7.2 Convergence Analysis

When Does Parallel Beat Sequential? Given the following assumptions,

- Data is read from DFS; no extra driver-side load penalty.
- Work splits evenly; per-node cost scales roughly as $5.7 \times 10^{-6}/N$ seconds.
- Overhead ($\sim$ 7s) still paid once per job for scheduling/shuffles.

we can solve for $T_{naive} = T_{par}(N)$.

Algorithm	Nodes	Naive (s)	Spark (s)	Spark/Naive	Note
Attractor (measured)					
Attractor	1k	0.033	3.087	93.4	Naive faster
Attractor	5k	0.047	3.103	65.6	Naive faster
Attractor	10k	0.060	3.098	51.8	Naive faster
Attractor	50k	0.265	3.297	12.4	Naive faster
Attractor	100k	0.519	3.280	6.3	Naive faster
Attractor	500k	2.938	5.710	1.9	Naive faster
Attractor	1.0M	5.958	7.797	1.3	Naive faster
Attractor (projection from linear fit)					
Attractor	2.5M	14.9	14.9	1.0	**Tie**
Attractor	5.0M	29.8	26.6	0.89	Spark faster (overhead amortized)

Minimax (computed)			
CPUs	Break-even (∼nodes)	Parallel @10M	Speedup @10M
4	9.5M	21.3s	1.0x (borderline)
8	4.8M	14.1s	1.5x
16	3.9M	10.6s	2.0x
32	3.5M	8.8s	2.5x
64	3.4M	7.9s	2.7x
Attractor (computed)			
CPUs	Break-even (nodes)	Parallel @10M	Speedup @10M
8	120M	730s	0.08x (still naive)
16	60M	365s	0.17x (still naive)
32	30M	183s	0.34x (still naive)
64	15M	92s	0.66x (still naive)
128	7.5M	46s	1.32x (Spark breaks even)

Table 1: Breakpoints for theoretical distributed minimax.

When to switch? The suggested action seems to be to use the naive solution on single machines or if there are fewer than about ten million nodes in a tree for minimax or 120 million nodes in a graph for attractor.

8 Conclusion

The tree, at least, has leaf nodes to compute backward from. The graph must compute the whole graph, each iteration. It is fully possible to execute the algorithms in parallel on Spark, but it needs a fairly massive tree or an even larger graph.

9 Special Thanks

We must thank our professor, Dr. Hazhar Rahmani, without whom we would be clueless.

10 AI Usage

GitHub Copilot, employing ChatGPT 5.0 and Claude Sonnet, was used to create and test the code, as well as to make sense of some of the math and patterns and use linear projection to suggest breakpoints.

References

[1] József Beck. *Combinatorial Games: Tic-Tac-Toe Theory*. Cambridge University Press, Cambridge, UK, 2008.

[2] Paul Erdős and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Halasz et al., editors, *Infinite and Finite Sets*, volume 10 of *Colloquia Mathematica Societatis János Bolyai*, pages 609–627. North-Holland, Amsterdam, 1975.