Chapter 1

Dilemmas in Extensive Form Games

Contributed by Justus Hibshman

1.1 Introduction

Ever since the introduction of classic games such as The Prisoner's Dilemma and the Centipede Game [4], game theorists have realized that rational play by both agents can lead to extremely un-intuitive outcomes.

The Prisoner's Dilemma is storied as follows: Two fellow criminals are caught and held separately. They must independently decide whether or not to testify against the other. If they both keep quiet, they make off with a light sentence. If one testifies and the other doesn't, the testifier goes free and the other is in prison an extremely long time. If both testify, they receive a moderately long sentence. What do they do?

This game has the interesting property that no matter what one player does, the other will fare better (get a lighter sentence) by testifying against them. Then, according to classic game theory reasoning, they both testify against each other and get the outcome that is net worst.

In the Centipede Game (CG) (Fig: 1.1), players take turns deciding whether or not to stop or continue the game. When the game is stopped, both players arrive at some "outcome." The longer the game goes, the more the players prefer the outcomes, but the game is set up in such a way that players always prefer the outcome they could stop and cause over the next one. The game always has a final round, in which the last player may choose between two outcomes. We represent preferences with numbers. When shown in a figure, the numbers are ordered in player order. The more an outcome is preferred the higher the number.

These conclusions follow directly from the following assumptions:

1. All players are rational.

2. All players act so as to reach outcomes they prefer.

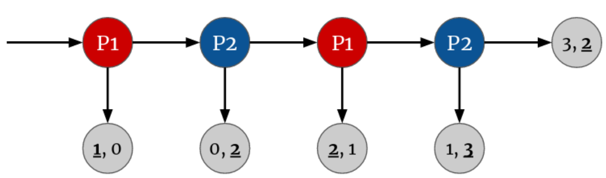

Figure 1.1: A Centipede Game (preference numbers are for players 1 and 2 respectively)

3. Items 1 and 2 are "Common Knowledge." Informally, something is common knowledge if "All know X. All know all know X. All know all know all know X. etc."

When these assumptions are applied to the CG, a strange but very straightforward "backward induction" applies: If the game gets to the last round, the player whose turn it is, operating solely according to their preferences and having complete reign over the outcome, will choose the outcome more favorable for themselves. Thus, the other player can conclude this, and if the game ever reaches the second-to-last round, they will know to end it there. This chain of reasoning follows all the way until the first round, telling the starting player to end the game right away.

In 1995 Aumann proved that given the assumptions of rationality and common knowledge backward induction is the inevitable result for all extensive form games [1]. His work uses an intuitive and widely-accepted definition of "rational." In another work he proved that backward induction holds in the CG even with a weaker definition of rationality [2].

These conclusions are unavoidable (and perhaps fascinating) facts. Furthermore, they present a potentially serious concern to anyone who intends to act or create agents which act strictly according to a set of preferences.

Indeed, some have proposed new "solution concepts," such as "Iterated Regret Minimization" [3] wherein players no longer act strictly according to their basic preferences but rather attempt to minimize a mathematical notion of possible regret. (Note: This particular solution concept leads to more intuitively "correct" behavior but has an "epistemic" flaw. The authors note this flaw but then rebutt it with an argument which fails to be compatible with the idea behind the "iterated" component of their solution concept. In short: the "Regret Minimization" component assumes uncertainty about the other player's strategy; the "Iterated" component assumes the other player is also using Iterated Regret Minimization.)

As far as this author is aware, no studies have been done on how likely a dilemma-like situation such as the centipede game is to occur "in the wild." This paper contains some simple simulations as a step towards answering that question.

Define a Dilemma to be a game in which, if all players act strictly according to their preferences, are rational, and these facts are common knowledge, then an outcome is reached which is strictly less preferred by all players than some other outcome in the game.

1.2 The Problem as a Graph

While the notions of dilemmas can be applied to all games, I will be looking at how they apply to extensive form games. Extensive form games are those where players take turns making decisions until an outcome is reached. They can be represented with tree structures.

In the simplest version of extensive form games, there are two kinds of nodes - decision (internal) and terminal (leaf). Gameplay consists of players selecting a path down the tree to a leaf node. Decision nodes are labeled with a player; that player gets to choose which of that node's children to travel to. Terminal nodes represent final outcomes and are labeled with preferences of all players.

1.3 Some Realistic Data Sets

Currently I am not aware of any data sets. Analysis of these games is generally of the theory, or of gameplay by AI or humans on choice games. Thus, I wrote code to generate games randomly.

The generator can generate two kinds of games: "balanced" and "chain". Balanced games are simply balanced trees. The code can vary the trees in number of players, number of nodes, and the degree of the nodes. Chain games have a structure just like the Centipede Game; the only difference is the arrangement of preferences. In both kinds of games, the generator creates a distinct random ordering of preferences over the outcomes for each player.

1.4 DIL-A Key Graph Kernel

To check to see whether or not a generated game is a dilemma, I will perform the backward induction computation to obtain the game's result and then compare this result to other outcomes to make sure there's no other outcome which all players prefer.

Depth first search is a good vehicle for the backward induction logic, because it explores all subgraphs rooted at a node before completely finishing with that node.

```
Def DepthFirstSearch(node)
    If self.visited
        // Perform desired computation
        Return
    self.visited = True
    // Perform some other desired computation
    For neighbor in self.neighbors
        DepthFirstSearch(neighbor)
```

Depth first search is $O(|V|+|E|)$. In the case of trees, this is simply $O(|V|)$ since $|E| = |V|-1$.

1.5 Prior and Related Work

Much work has been done to run depth first search efficiently. I simply piggy-back on this research to analyze large games. Specifically, I make use of the open source Boost Graph Library for C++ [5].

As mentioned before, I am not aware of any work to explore how often dilemmas surface in random games.

1.6 A Sequential Algorithm

When depth-first search is run on trees, there is no need to actually check for whether or not a node has been visited before, since all nodes can be reached by exactly one path. This can potentially simplify an algorithm, though I shall still build my backward induction on an existing depth first search system. Some pseudocode for backward induction is shown below:

```
Def BackwardInductionOutcome(root)
    If(root.children.size() == 0)
        Return root.preferences
    best_outcome = -inf
    result = Null
    For child in root.children
        child_result = BackwardInductionOutcome(child)
        If child_result[root.player_id] > best_outcome
            best_outcome = child_result[root.player_id]
```

```
            result = child_result
    Return result
```

1.7 A Reference Sequential Implementation

For backward induction, the main difference between my actual code and the algorithm outlined in the pseudocode in 1.6 is that the actual code is not recursive. Instead, it uses the Boost Graph Library's iterative depth first search implementation. Boost allows the user to register functions which get called at various events, such as the first time a vertex is visited or the last time an edge is crossed.

Some C++ which corresponds to my event handler is shown below. In English, it does the following: Whenever a vertex has the results (values) from the backward induction, it compares those with the results the parent currently has registered. If the parent prefers the results from the finishing child vertex, set the parent's result to be the same as the child's.

```
void finish_vertex (const Vertex &v, const basicGameGraph &g) {
    gameGraphView view(&g);
    if (view.properties(v).id != 0) {
        const auto &selfProperties = view.properties(v);
        const auto &parentProperties =
            view.properties(selfProperties.parentId);
        const auto &parentPreferences =
            (*values)[parentProperties.playerId];
        if (parentPreferences[selfProperties.id] >
                parentPreferences[parentProperties.id]) {
            for (unsigned int i = 0; i < values->size(); i++) {
                (*values)[i][parentProperties.id] =
                    (*values)[i][selfProperties.id];
            }
        }
    }
}
```

[Note: I realized when writing this that my code has a bug in it effectively causing every non-leaf vertex to have one extra child leaf. I will re-run the computation for my final results.]

The creation of "random" games worked as follows: First, generate a tree structure - either balanced trees or "chain" trees ("chain" trees have the same structure as the centipede game.). Then, for each player, assign a strict ordering of preferences over the leaf nodes. The ordering is random (uses C++'s std::shuffle()) so players' preferences are uncorrelated.

1.8 Sequential Scaling Results

All code was run on the author's personal machine.

Many different parametrizations were tested:
$|V|$ = 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 | 10,000,000
|Players| = 2 | 3 | 4 | 5
"Chain" Tree or Balanced Tree

If balanced, vary degree: 2 | 4 | 8 | 16 | 32. In balanced trees, the outcomes are the leaves of the tree. Thus the game doesn't end until it gets to the bottom.
For chain trees with more than 2 players, the same structure exists as a 2 player graph. Nodes are assigned to players cyclically (i.e. First player 1 gets to decide whether to end the game. Then player 2. Then 3... Then player 1 again.). Thus frequency with which a player gets to decide whether to end the game or not becomes 1 over the number of players.

For every one of these parameter combinations I ran 100 tests and counted the fraction of games where a non-optimal outcome was reached (i.e. the fraction of games which were dilemmas).

Runtime scaled linearly in both the number of nodes in a tree and the number of players. See figure 1.2 for runtime scaling with game size and figure 1.3 for runtime scaling with number of players.

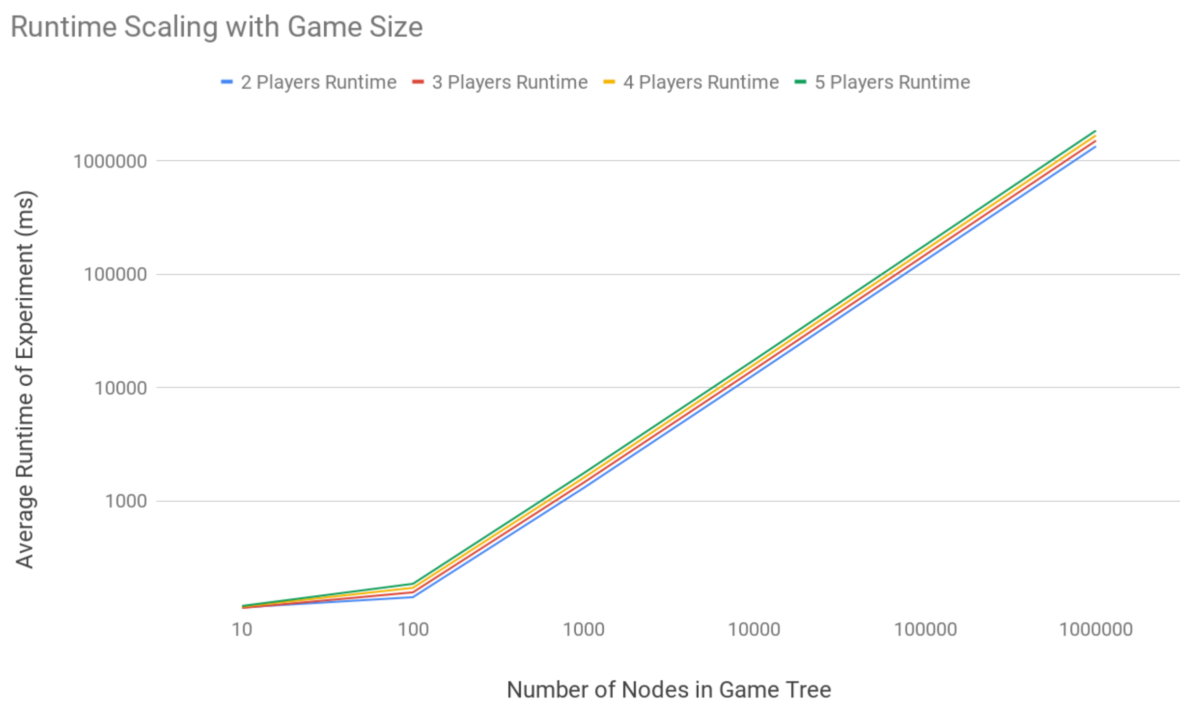

Figure 1.2:

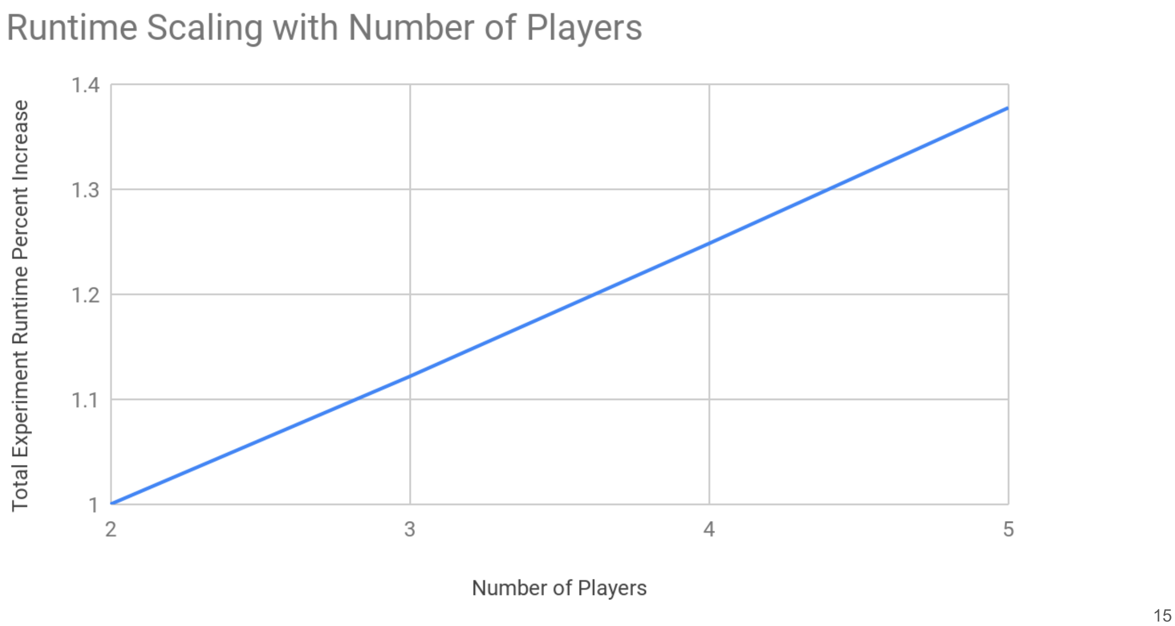

Figure 1.3:

Varying the degree of the balanced trees made little difference to the average fraction of the dilemmas, though it did change what game sizes had fractions above or below the overall trend. Varying the number of players did little to change the overall results as well. Game size and balanced vs. chain were the two main contributors to differences.

Figure 1.4 shows the fraction of games that are dilemmas as a function of game size for balanced trees. Note that as the game size increases, the fraction of dilemmas comes to be around $\frac{3}{10}$ games.

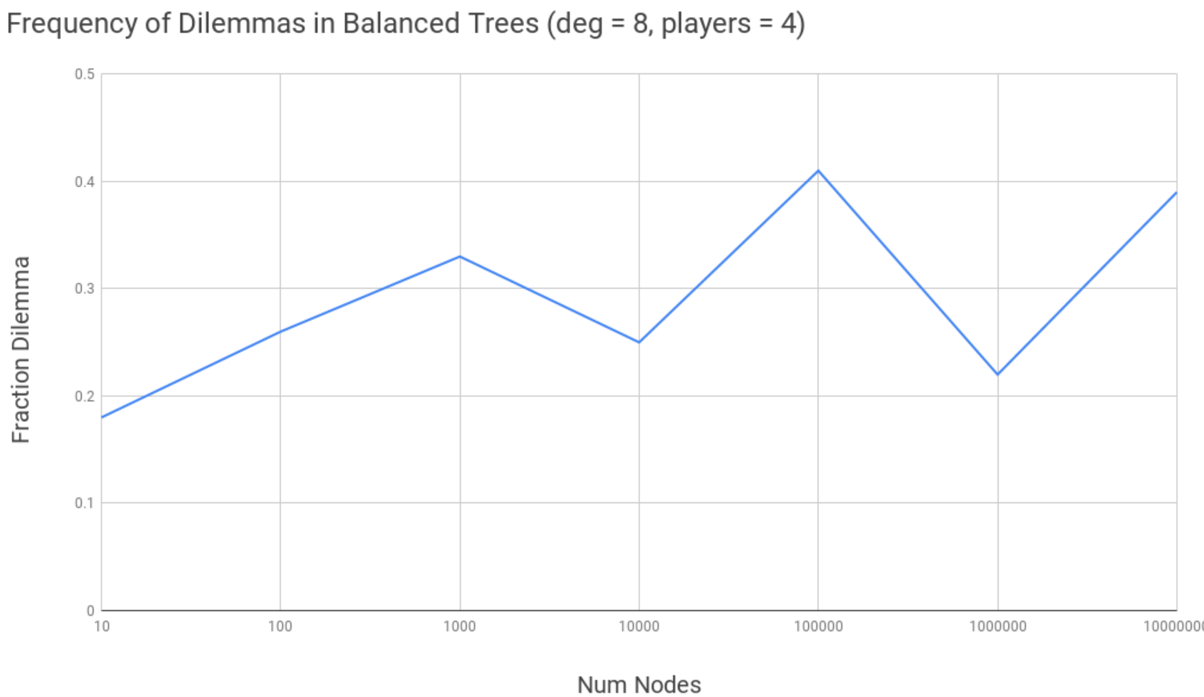

Figure 1.4:

Lastly, figure 1.5 shows the fraction of non-dilemma **chain tree** games as a function of game size and player number. Note that as the game size increases the fraction of games that are dilemmas approaches 1.

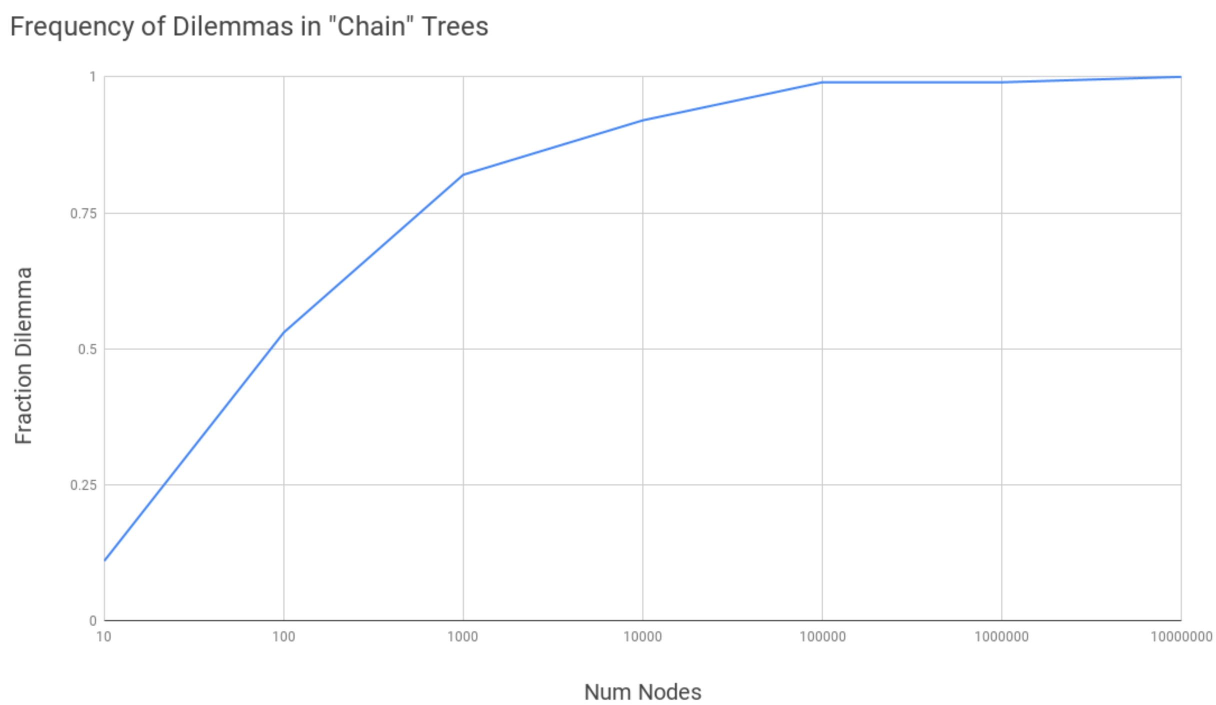

Figure 1.5:

1.9 An Enhanced Algorithm

A pool of worker threads is established. Rather than actually exploring the subtree rooted at a node, a worker can tell another available worker to explore that subtree. Workers which have called others do not go any deeper in the tree themselves. Rather, they wait for all the workers they've called to finish and/or for more workers to become available to assign to as-of-yet unexplored subtrees.

As an example: Let us say that a worker W_1 is exploring a tree and is currently at a node with 3 children. There are two workers in the available pool, W_2 and W_3. W_1 would assign W_2 and W_3 to explore 2 of the children of the node it's currently at. Then it would wait for them to finish. If W_2 finishes before W_3, W_1 would re-assign W_2 to its third child.

The computational complexity is still linear in the number of tree nodes, but is now divisible by the number of workers expected to be running at once. How many workers can run at once depends on both the total number of workers, the number of machine cores, *and the tree structure of the game being explored.* If the tree essentially requires a sequential computation (e.g. a chain tree) then more workers will offer little benefit.

1.10 A Reference Enhanced Implementation

Due to issues getting the Boost Graph Library to run in parallel, I stopped using Boost entirely for my enhanced implementation. The new version was faster running single-threaded than my original Boost code. Thus, there were two enhancements: The Boost-less implementation plus the subsequent parallelization. The code for this constitutes several hundred lines of C++ code. All paralellization was done with pthreads.

1.11 Enhanced Scaling Results

The basic sequential implementation gained about a 10x speedup over the Boost-based implementation. This may be because the code was designed with a specific application in mind whereas Boost is designed to be extremely general. On top of this, the parallelization added additional speedup, particularly on large graphs. All tests were run on a 4 core machine.

As expected, no speedup was gained from parallelization on the chain trees. The sequential code performed the fastest. Figure 1.6 shows this result.

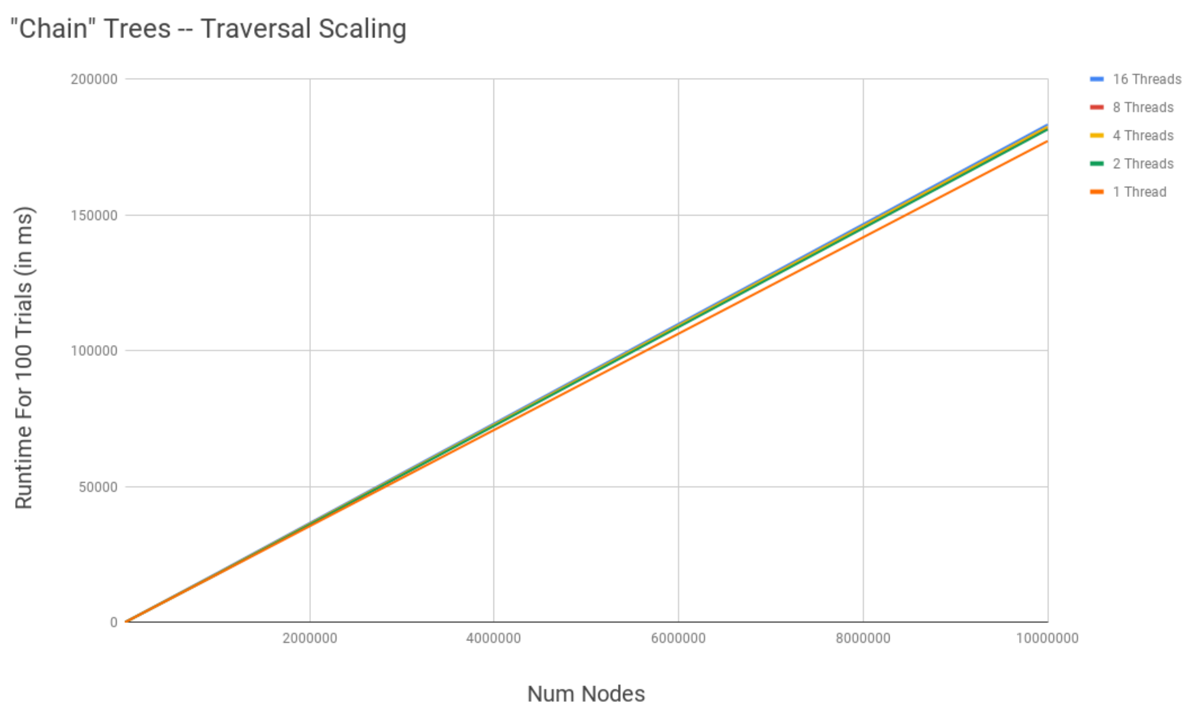

Figure 1.6:

Figure 1.7 shows the same data as 1.6 but shows it on a log-log scale and adds in a line for the Boost sequential implementation. In this figure we can see that on smaller graphs the overhead added by extra threads damages performance by orders of magnitude but that this factor becomes irrelevant once graphs reach about 10,000 nodes. Here we also see that everything runs about 10x faster than the Boost depth-first-search implementation.

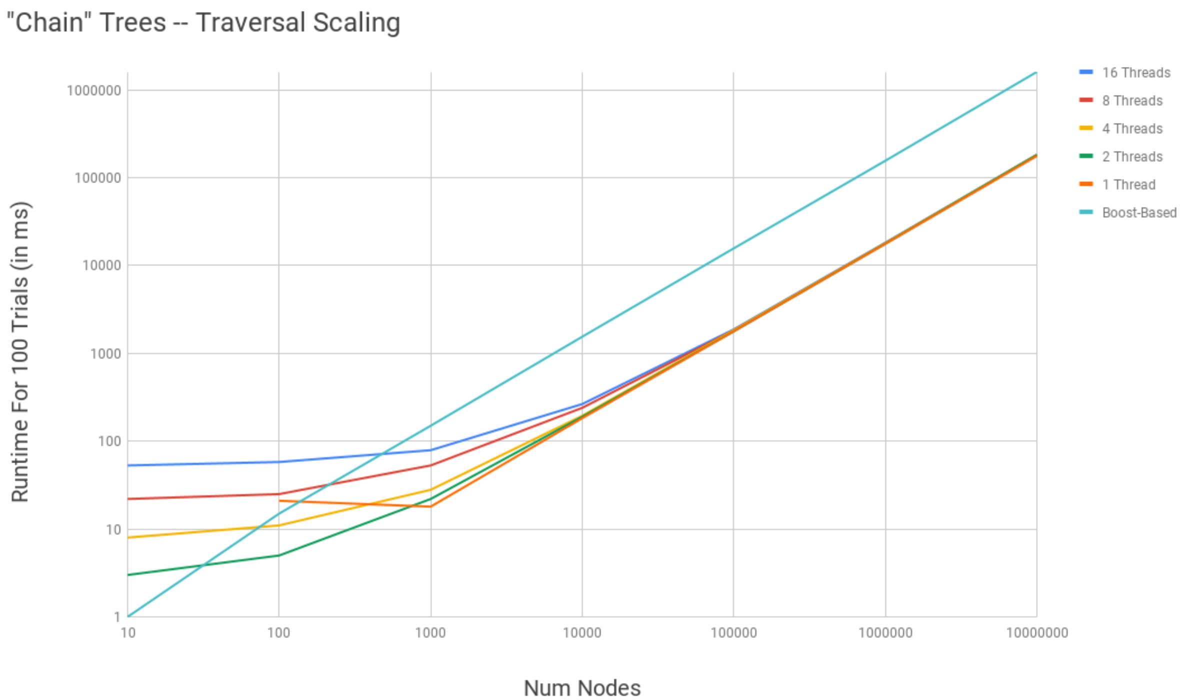

Figure 1.7:

On balanced trees, parallization becomes useful. The best results were obtained using the same number of worker threads as cores (4). Figure 1.8 illustrates this.

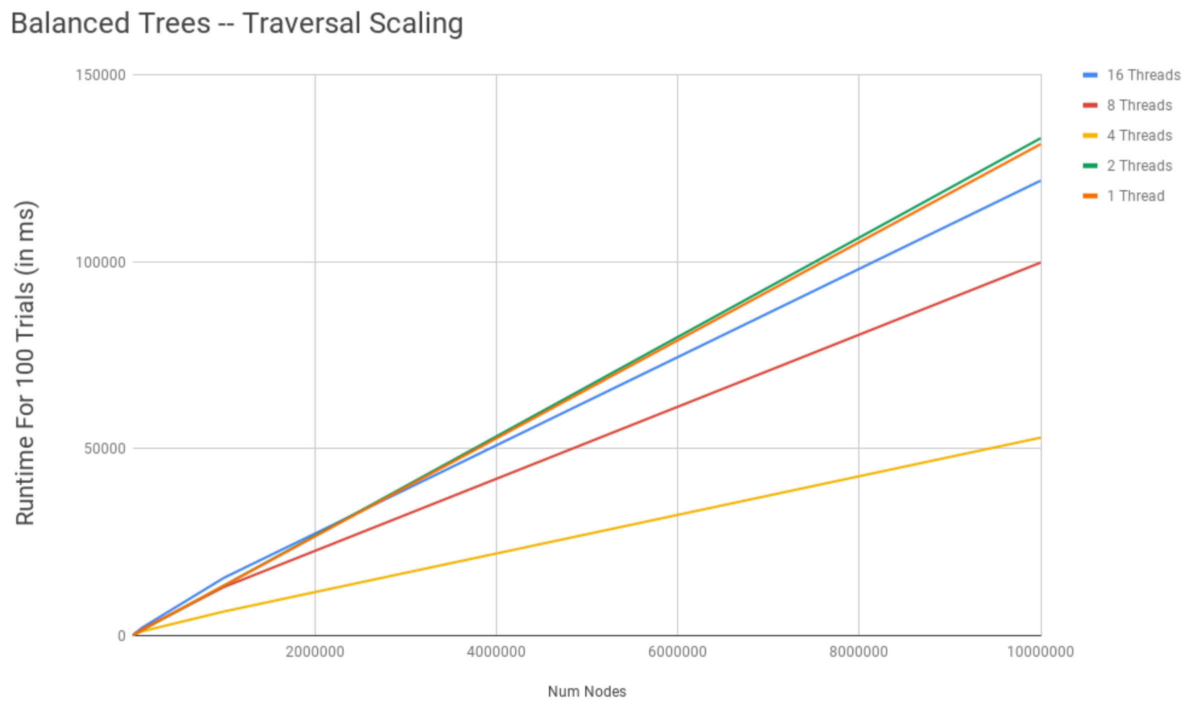

Figure 1.8:

And again, when shown on a log-log plot (Figure 1.9), we can see that at first overhead from parallization dominates but eventually it outperforms the sequential implementation. Boost is again about 10x slower than the new sequential.

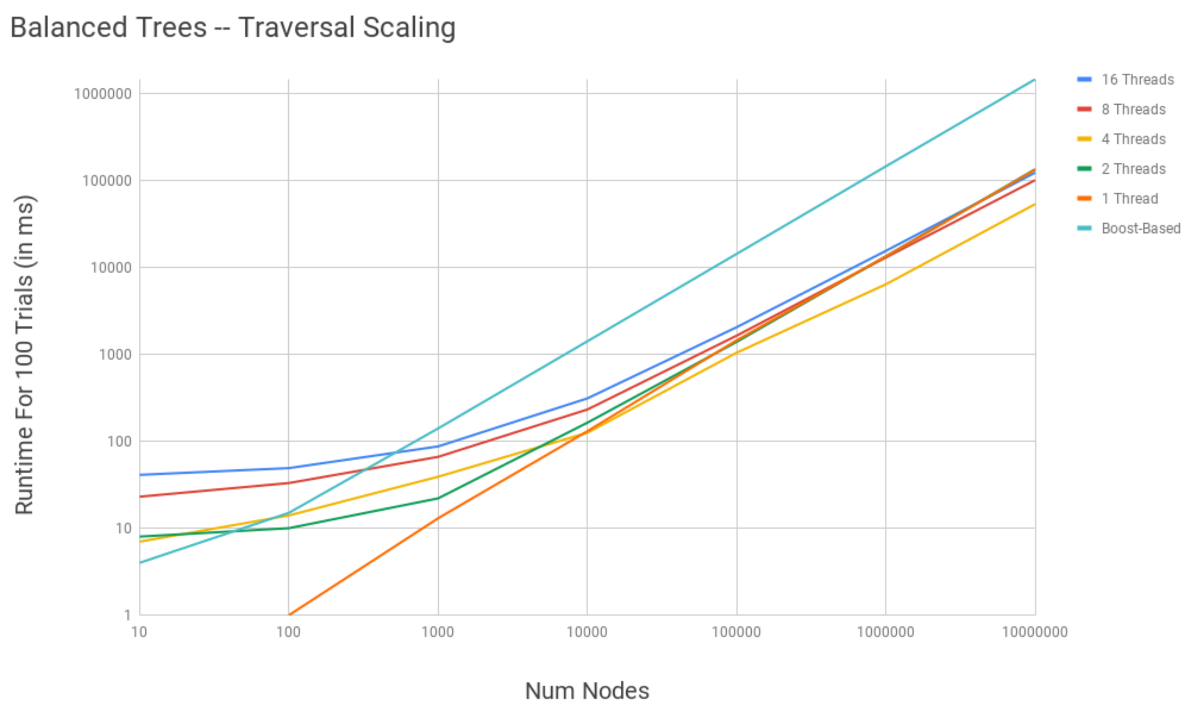

Figure 1.9:

1.12 Conclusion

Dilemmas occur quite frequently in large games if (as in this paper) players' preferences are uncorrelated. If simulating these scenarios, parallel implementations are of value depending on the structure of the game trees being examined. Problem-specific code outperforms even general libraries written in a high-performing language.

Future work would include looking at games where players' preferences have some amount of correlation. It would be wonderful if real-life scenarios could be turned into game trees, but the authors are currently unaware of such data.

From an implementation perspective better code for permuting player's preferences in parallel would be a valuable addition.

1.13 Response to Reviews

Some things I changed from the second round of reviews:

There were some small grammar issues I fixed.

Someone commented that they were unclear how chain games worked with more than 2 players. I added stuff to explain that.

I also explained that balanced tree games don't ever end until you get to the bottom.

I made a change to the pseudocode based on a suggestion.

I figured out what the units were for my chart with previously "unknown units."

Bibliography

[1] Robert J Aumann. Backward induction and common knowledge of rationality. *Games and Economic Behavior*, 8(1):6–19, 1995.

[2] Robert J Aumann. On the centipede game. *Games and economic Behavior*, 23(1):97–105, 1998.

[3] Joseph Y Halpern, Rafael Pass, et al. Iterated regret minimization: A new solution concept. *Games and Economic Behavior*, 74(1):184–207, 2012.

[4] Robert W Rosenthal. Games of perfect information, predatory pricing and the chain-store paradox. *Journal of Economic theory*, 25(1):92–100, 1981.

[5] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost graph library. https://www.boost.org/doc/libs/1_68_0/libs/graph/doc. Accessed: 2018-11-15.