An Embedded Domain-Specific Language for Algebraic Tournament Structures

Construction and Evaluation of Sorting Structures in Haskell

Michael Ledger, Australian National University*

November 20 2023

Abstract

An Embedded Domain-Specific Language (commonly known as eDSL) is developed for easy expression and analysis of arbitrary tournament structures. The core tournament structure type is inspired by work on algebraic graph representations in Haskell, and sorting networks, and enables composition in terms of either interleaving or sequencing. The core design balances between allowing arbitrary logic within a tournament structure and what static analysis can be achieved given such a structure. A software library is developed that, provides the foundations for future work in measuring tournament characteristics such as reliability and fairness. Analogies from tournaments to sorting networks are observed and common tournament structures are contrasted with sorting networks to see how viable they can be as tournament structures. A tool is also developed to perform and visualise tournaments described within the eDSL.

*u5582972@anu.edu.au

Contents

Introduction

Tournament structures are used to determine a ranking of players in a game, such as a real-life sport or an eSport. Many different tournament structures are employed in real games, for a variety of reasons; for example, what resources are available, or how many participants there are, or how much time is available. Moreover, the choice of tournament structure can have a significant effect on the outcomes of the tournament. The choice of what tournament structure is most appropriate for a given set of circumstance is mostly an open one, and tournaments are ran as a combination of other tournament types.

There is possible contribution to be made in helping to determine what tournament structures are most effective under the constraints of a real-world tournament. This project aims to make such a contribution, in the form of a software library that helps with the design and evaluation of tournament structures.

More specifically, this project encapsulates my effort to do that over the course of my undergraduate Advanced Computing Project (COMP4560), by creating a Embedded Domain-Specific Language for tournament design and evaluation, as a Haskell library. The software library developed for this project is available online on GitHub[1], which also hosts its API documentation[2]. This thesis then serves as an overview of existing literature into relevant topics, the software I have developed, and the results of using this project to perform analysis on a variety of tournament structures.

Acknowledgements

Much thanks is given to Ranald Clouston, who supervised this project through the year.

[1] https://github.com/mikeplus64/journeyman
[2] https://mikeplus64.github.io/journeyman

Background and Motivation

The economics of sports and eSports tournaments are staggering; in 2021 nearly $200bn USD was spent on sports betting [1] alone. At least $303m USD has been allocated in prize pools for *DOTA 2* tournaments since its release [2]. Quoting from an article The Economist wrote in July 2020 about growth of eSports spurred on by the COVID-19 pandemic [3]:

> *Take "League of Legends", perhaps the biggest e-sport in the world. It was launched in 2009 by Riot Games, an American firm now owned by Tencent, China's biggest tech firm. It is a complex strategy game, in which teams of five players command "heroes" in a battle to defeat each other. As many people play it regularly as play tennis; at any one time, 8m people may be online. It also supports a professional game that is, at least in terms of the number of players earning a living from it, also larger than tennis. The final of the League of Legends World Championship last year was watched live by 44m people. By comparison the Super Bowl, America's biggest live sporting event, was watched by roughly twice that.*
>
> *Twelve professional leagues now span all regions of the globe except Africa, with 120 franchised teams and perhaps 1,000 professional players. Whereas tennis stars in the world's top 200 often struggle to make a living, "League of Legends" players in America are guaranteed a minimum salary of $75,000. There, players are entitled to the same visas that other foreign athletes can get. The average salary is closer to $400,000, says Chris Greeley of Riot Games. Lee Sang-hyeok, a Korean star, known by his tag "Faker", may be the highest-paid sportsman in his country.*

With this in mind, it is worth asking how it is that players are ranked and evaluated against each-other. That is: Are the tournaments reliable - are the results repeatable? Are they fair? How could we determine this? Indeed, examples exist of large real-world tournaments that were possibly unfair [4][5] or where players perceived that there would be a benefit to purposefully losing (throwing) matches [6][3] due to the structure of the tournament.

The initial motivation for this project was my experience in helping to run tournaments for the games *Diabotical* and *Quake Champions* through a website[4] I built for that purpose. Since its inception in 2020, 79 tournaments have been completed through it. Frequently questions of fairness in tournaments would arise due to some players seeming to have "unfairly" hard or easy tournament brackets. This project then arises from a desire to be able to quantify the fairness of the different tournament structures that were employed – if a software tool existed that could aide in the design and evaluation of tournament structures (by, for instance, analysis and/or simulation), that could have been employed to determine what tournament structures were appropriate. Instead, the logic that describes tournaments

[3] This example is quite poignant, including a scene where both teams in a match of an Olympic badminton group stage endeavoured to lose against eachother. A total of 4 teams were disqualified in this incident. [7]

[4] `https://kuachi.gg`

on `kuachi.gg` are hard-coded into the system, and while some composition options are supported, it is a fairly inconvenient process to add new ones.

A key contribution this project seeks to make is to recommend tournament structures that are resistant to "false" seeding (initial ranking). If a tournament has a number of rank inversions in its seeding, does that number of rank inversions reduce or increase after the tournament is ran? That is, to what extent does the tournament really sort the players? This is a common problem with real-world tournaments as often the relative skill level of players is simply an unknown factor, and so guesses must be made.

This project contributes a small set of primitives that can be used to describe arbitrary tournaments, an Embedded Domain-Specific Language (eDSL) for tournament description that employs those primitives, an interpreter capable of running and analysing tournament structures built using the eDSL. Additionally, discussion of analysis of common tournament structures by the software library is included in this thesis.

Scope and aims of this project

With this background in mind, these are the aims of the project:

1. To create a computer language that enables the convenient description of common tournament structures, and also to enable creation of new tournament structures as compositions of existing ones.
2. To create an interpreter for this language that can evaluate tournament descriptions; to be able to run the tournaments described by the language defined in Goal 1.
3. To systematically evaluate the efficacy of different tournament structures in terms of quantifiable properties such as fairness, length, repeat-match avoidance, and so on.

I call the software library developed Journeyman.[5]

[5] A Journeyman is a term for a consistently high-achieving career sportsperson that is not at the highest level of their sport.

Definitions

As an aide to readers, the following glossary is provided. I specialise certain terms to have exactly one specific definition in order to avoid confusion and limit the scope of this thesis.

Tournament A schedule of Matches by which a set of Players are ranked in order of most-skilled first to least-skilled last.

Player A participant in a Tournament.

Match A pairing of exactly two Players in a tournament that has an uncertain outcome to be determined by a game of some sort.

Bracket A schedule of games in an elimination tournament, commonly expressed as a binary-tree diagram of player matchups.

Single Elimination A kind of elimination tournament in which the loser of each match during a round is eliminated. Each round successively narrows the field of remaining players until just one winning player remains. Also known as a knockout or sudden-death tournament.

Double Elimination A kind of elimination tournament in which each player is allowed to lose up to 1 match before they are eliminated. Also known as a double knockout tournament. The tournament is divided into two brackets; an "upper" or "winner's" bracket, and a "lower" or "loser's" bracket; the upper bracket contains only players that have not lost a match up to that point, and the lower bracket contains players who have lost a match in the upper bracket prior to that point.

Seeding A ranking of the players of a tournament before the tournament begins. Seeding can have a significant impact on the outcome of elimination tournaments; consider the worst-case seeding of a Single Elimination tournament in which the two best players match against each-other immediately.

In a seeded tournament, Player 0 is the "highest" or "best" seed, denoting the player with the highest skill. Player $(n-1)$ is the "lowest" or "worst" seed, denoting the lowest skilled player.

Slaughter Seeding A method of seeding a tournament such that the initial match for each player is the mirror of that player; the best player shall play the worst, the second best shall play the second-worst, and so on.

Round Robin A kind of tournament where each player plays each other player once.

Group Stage A kind of tournament where players are split into a fixed number of groups, and a Round Robin tournament is played within each group. those groups.

Stage Where a tournament is split into 2 or more sub-tournaments that run in sequence, one after the other, those sub-tournaments are often coined *stages*.

Upset A match result is said to be an *upset* if the higher seeded player loses to the lower seed. (Regardless of how close their true skill levels may be.)

Literature Review

Before proceeding to the design and implementation of this project, I will provide an overview of some of the relevant literature that informed how the project proceeded and what gaps existed in knowledge that it could help narrow. I consider literature across the topics of Domain-Specific Language design, tournament design, and tournament fairness. Each sub-section here denotes a particular paper that was considered.

"Design Guidelines for Domain Specific Languages" Karsai *et al.*[8]

This paper provides a list of guidelines to follow for the design of DSLs. I followed it as a rough guide used in the process of designing the Journeyman eDSL. The design guidelines are codified into a list of suggestions; here I respond to each.

1. **"Identify language uses early"** See *Scope and aims of this project*.
2. **"Ask questions"**
   - **"Who is going to model in the DSL?"** Tournament designers who may find the software useful.
   - **"Who is going to review the models?"** Tournament designers who may find the software useful.
   - **"When?"** Prior to the design of a tournament, but when factors such as time and resource constraints are available.
   - **"Who is using the models for which purpose?"** Tournament designers may use the eDSL to help identify and analyse the efficacy of various tournament structures, and apply those results; for myself to the real-world use-case I identify in the section.
3. **"Make your language consistent."** I apply an Algebra of Graphs-esque approach to the design of the eDSL in order to keep its semantics simple.
4. **"Decide carefully whether to use graphical or textual realization"**

   A textual representation will be the primary format for this eDSL, as it is intended to be a programming library with the ability to describe arbitrary tournament structures. That said, the software includes the ability to visually interpret a tournament.
5. **"Compose existing languages where possible"**, and,
6. **"Reuse existing language definitions"** One of the key factors that decided the choice of eDSL vs DSL was the ability for an eDSL to leverage its host language to the full extent. Indeed, it does become apparent that many tournament structures can be expressed quite conveniently with Haskell's list functionality alone.
7. **"Reuse existing type systems"** As will be seen in later sections, the type-system is used effectively here to restrict certain kinds of tournament composition and to simplify the logic required to evaluate a given tournament.

"Algebraic graphs with class (functional pearl)" Mokhov[9]

This paper and the accompanying Haskell library `algebraic-graphs` largely largely inspired the design of the core `Tournament` data type that the Journeyman eDSL manipulates. The key observation is that graphs can be described using two operators `Connect` and `Overlay` which each have desirable properties.

In the Journeyman eDSL, the `Overlay` operator is reproduced to mean the interleaving or parallel existence of two sub-tournaments, and can be intuited as an algebraic sum operator. The `Connect` operator is not reproduced, as it does not have an intuitive analogue in tournament structures; instead, a `Sequence` operator connects tournaments by running them one after the other, and can be viewed as an algebraic product operator. Type information is then used to guarantee that we can turn any arbitrary tournament, which can nest combinations of `Overlay` and `Connect` at will, into a "flattened" stream of rounds. For more information, see *Design*.

"The structure, efficacy, and manipulation of Double Elimination tournaments" Stanton *et al.*[7]

This paper provides information about Double Elimination tournaments in particular. Several important theorems are provided for their design - in particular a result about the optimal linking between the upper and lower brackets of a Double Elimination tournament - as well as statistical analysis performed on the efficacy of tournaments.

Statistical analysis is performed to compare the reliability of Single Elimination tournaments to Double Elimination ones, where Double Elimination is shown to be much more efficacious in allowing the most skilled player to win. Simulations are performed using chosen models for the probabilities of players winning against each other, rather than on real-world data.

Manipulation of Double Elimination tournaments is also considered, and an interesting case study provided to demonstrate the need for tournaments that are robust against manipulation: "in the 2012 Olympics, four of the top badminton teams were disqualified for trying to intentionally lose matches, causing an uproar and angering fans. While the tournament structure used there was not a DET, this demonstrates that players really will exploit poor tournament design when possible." The importance of seeding in the outcome of elimination tournaments is noted as well. Several theorems are provided on the complexity of manipulation of a tournament by players.

Double Elimination **Link Functions** are described in this paper: The Link Function is the algorithm that chooses where in the lower bracket a player from the upper bracket should go after a loss. The choice of Link Function is quite important in order to avoid re-matching players who already faced each other in the upper bracket as much as possible [6]. Two operations, named Swap and Reverse, are described in constructing a Link Function, and an optimal Link Function that avoids

[6] Avoiding rematches has intuitive benefits; players are able to gauge their skill against multiple opponents, and thus so does the tournament.

rematches as much as possible can be constructed using $\lceil \log(R) \rceil$ Link or Swap operations.

"Double Elimination Tournaments: Counting and Calculating" Edwards[10]

This paper provides broad information about the construction of Double Elimination tournaments. The efficacy of "unbalanced" Double Elimination tournaments is considered in detail. A system for uniquely numbering Single Elimination tournaments is also provided, with extension then to number Double Elimination tournaments by the structure of the lower bracket as well as the linking function used.

Statistical analysis is performed by using an assumed preference matrix, denoting the pairwise probabilities of one team winning a game against another. Using a preference matrix contrasts against methods used in "real" games to calculate the probability of a player winning a particular match against another, such as the Elo Rating System[11]; in Elo, all players are assumed to have an absolute quantifiable skill level, that satisfies transitivity; if player A is more skilled than player B, and player B is more skilled than player C, then A must be more skilled than player C. A preference matrix approach allows for the fact that some players may do particularly well or poorly against other players. It may be possible to calculate a preference matrix from existing public data from existing games, by assigning a secondary ranking to players by treating each possible pair as its own separate game.

The larger Double Elimination tournament shown in this paper does not to have a "balanced" lower bracket. Convention in modern Double Elimination tournaments is that, to maximise fairness and minimise the number of rounds required, one should alternate between rounds where players are from the lower bracket play against each other, and where "new" players are added in to the lower bracket from a round in the upper bracket. This is shown in "The structure, efficacy, and manipulation of Double Elimination tournaments" [7].

"Simulating competitiveness and precision in a tournament structure: a reaper tournament" Dinh *et al.*[12] and "Reaper Tournament System" Pham *et al.*[13]

This pair of papers describes a novel tournament structure coined "Reaper tournaments". [13] describes most of the results and [12] develops the knowledge of the Reaper tournament system further, and creates a similar (but new) tournament structure called *Reaper elimination*.

The structure of a Reaper tournament is that it operates initially as an inverted Single Elimination tournament, where only the losers of each match advance along the Single Elimination bracket. An algorithm is applied to successively select and eliminate the worst player successively. Indeed, Reaper tournaments seem to be analogous to common Selection Sort algorithms; and by only eliminating a single player in each round, it is able to achieve 100% ranking precision - or at least that no player shares the same result as another.

The number of matches required in a Reaper tournament is not given a general formula in the system, nor the number of rounds, which is a significant weakness to its adoption as a tournament structure in practice - tournaments need to happen usually within some known time constraints. Description of the Reaper tournament system as a sorting network, though the Journeyman eDSL, may help to elucidate its properties. For $n = 8$, the Reaper tournament requires $m \in [15, 17]$ matches compared to $m = 14$ for double elimination or $m = 28$ for a round-robin.

It is also shown that the *stability progression*, measuring whether winning a game is more desirable than losing, is preserved in the Reaper tournament structure. It is never a desirable outcome to lose a match in the Reaper tournament structure.

Additionally, two-stage tournament systems where a Group Stage precedes an Elimination are considered [12]. The group stage has multiple groups of players in each group, and a tournament structure such as round-robin (or Reaper), is conducted within that group. Such structures are quite common in practice.

Various metrics are created to measure the efficacy of tournaments in practise and in simulation. The key metrics are *ENM*, meaning "expected number of matches", *ARW*, the "average rank of the tournament winner" (ideally, 1), and RankCor $\in [0, 1]$ where a value of 0 means the tournament had a completely random result with respect to the player's "true" skills/rankings, and a value of 1 means that the tournament perfectly preserved those a priori rankings.

Theoretical experiments on 8 player tournaments are conducted that show the excellent RankCor of the original Reaper tournament structure. Double Elimination stages are also shown to have quite good RankCor (at this size of tournament). Real-world tournament data is used that demonstrates the robustness of Double Elimination tournaments in terms of RankCor, with Reaper tournaments also performing excellently, though doubling the number of matches required.

Description of the Reaper tournament structure algorithm

I reproduce in my own words the algorithm for the Reaper tournament structure here.

Information:

- Each player has a respect list of players who they have previously lost to. This is updated every time a game occurs.
- The tournament is assumed to be $n = 2^k$ in size; there must be a power-of-2 number of players.

Steps:

1. Reaper selection: In Round 1, pairs of players are matched together, so that every player is in a match. The losers in the round are then paired against each other, and again, until a round where only a single player loses a match (who lost all matches prior to this round), and they are eliminated from the tournament. Let the winner of the final game in this step be the Reaper.

This basically describes an "inverted" Single Elimination tournament - where to proceed to the next round, you must lose the current round. The "winner" (i.e., loser of all games) then of this inverted Single Elimination tournament is the one who is actually eliminated from the tournament.

The question of what matching algorithm is used is left open by the authors of the paper, but it is likely significant in determining the outcome of the Reaper selection stage.

2. Reaper candidates: A candidate list is created consisting of:
   - If there are players who are not in a respect list, those players.
   - Otherwise, the players who are in the respect list of the Reaper.

   The size of the candidate list then determines the next step:
   - If > 1, proceed to (3).
   - If $= 1$, proceed to (4).
   - Otherwise ($= 0$), the tournament ends.
3. Candidates match: The two best players play each other. Update the respect lists accordingly and go back to step (2).
4. Reaper match: The single player in the candidates list plays the Reaper. The loser here is eliminated and is ranked above the previously eliminated participant, while the winner is set to be the new Reaper.

The expression of this structure in an eDSL is challenging, as the tournament needs to be able to "respond" to the results of matches in order to maintain a respect list and candidates list.

Reaper Elimination

A follow-up structure is proposed in the second paper [13] that develops the Reaper tournament structure to give it an upper bound on the number of matches required, and a static tree structure. Thus, it is likely a tournament that could be expressed as a sorting network. It is shown that the number of matches required is $O(N \log_2 N)$.

"Quantifying the unfairness of the 2018 FIFO World Cup qualification" Csató[4] and "Risk of Collusion: Will Groups of 3 Ruin the FIFA World Cup?" Guyon[5]

These papers look at real-world sports tournaments, namely the FIFA series of soccer/football tournaments. As these are huge events with massive prize pools that carry great prestige for participating teams, nations, and hosts, examination of these events for fairness criteria is important. These papers demonstrate how real-world data can be used to examine and quantify fairness of tournament structures.

It is shown in [4] that the origin continent of a team had an out-sized effect on the likelihood of a team in qualifying into the FIFA World Cup in 2018. It is found that

a fixed draw rather than a random draw for qualification would reduce unfairness.

Unfairness is measured by "ranking the teams according to their Elo, and summing the differences of qualifying probabilities that do not fall into line with this ranking".

In [5], the conditions required to aggravate the risk of collusion between teams is examined. This can occur when two teams in a Group Stage are already guaranteed entry into the proceeding stage, but the result of their match can adversely affect whether or not another team in that group makes it through to the next stage or not. Examples of collusion are examined in real-world games. Games such as soccer where a draw is a possible outcome may be susceptible to colluding outcomes; teams can agree in advance to draw against each other, and neither will lose face nor prestige, while still possibly being able to gain the points required to proceed on to the next stage of the tournament.

"Handling fairness issues in time-relaxed tournaments with availability constraints" Van Bulck *et al.*[14]

This paper examines computational complexity of time-relaxed tournament game scheduling. That is, the problem of scheduling games where there is not a tight deadline to complete the games, but there may be sporadic player and venue availability. This situation frequently occurs during "long format" group stage formats which are ran over weeks or months, where the scheduling of each game is done by each player participating in that game together. However, this is out of scope to the research aims of this project. The fairness measures proposed by this paper also concern scheduling, which is outside of the scope of this project.

"The impossibility of a perfect tournament" Placek[15]

This paper provides an important result that shows that their constructed fairness and balance metrics trade off against one-another, and elimination tournaments cannot be constructed that maximise both metrics. The author concludes that a perfect tournament design cannot be made because of the inherent uncertainty of outcomes and player seeding; indeed, if perfect ranking was already available at the outset, there would be little point to running a tournament in the first place. The author also provides discussion on the tournament outcomes and spectator interest; where players who play optimally are perceived to be dull or unimaginative.

The *fairness* metric here is that the sum of the ranks of winners of each match must be maximised across the whole tournament. This is an interesting definition that intuitively works quite well when the tournament structure is also minimising the number of matches required - one could construct a degenerate-case tournament structure that maximises this sum, by, for example, matching 2 weak players repeatedly until the sum generated by the winners of those matches must be greater than the sum generated by the winners of the other matches in the tournament.

The *balance* metric here is to minimise the difference in ranks between players across all matches. By doing this, you create tournament structures that provide as

more information about players who are closely matched. In the single elimination case, it is clear that maximising balance minimises fairness. Maximising balance can have the effect of increasing spectator interest, as closer games are assumed to be more exciting to watch than "blow-out" games, which I can validate from my own anecdotal experience.

"A new knockout tournament seeding method and its axiomatic justification" Karpov[16]

This paper demonstrates the determining effect of seeding to Single Elimination tournament outcomes, and proposes an "equal gap" seeding method contrary to the traditional "slaughter seeding" method, that, under a deterministic domain assumption, satisfies the fairness, competitive integrity, and equal rank difference axioms that are introduced. Here, determinism refers to assuming that given any match, the player with the highest seed/skill shall win.

Applicability of the proposed seeding method outside of the domain assumption is an open question, and may be a useful application of the Journeyman library to examine its effects.

"The efficacy of tournament designs" Sziklai *et al.*[17]

Efficacy of tournaments is analysed in terms of ranking inversions exhibited at the end of the tournament. A valuable result is that triple-elimination does not greatly improve the efficacy of ranking players compared to Double Elimination, especially when accounting for the extra matches required.

Swiss-style tournaments are shown to be very effective at ranking players and generally exhibit fewer inversions than any other format considered, for the same number of matches - although Swiss-style tournaments use a matching algorithm each round to determine who plays who, they are ran to a fixed number of rounds, so they can be engineered to desired level of accuracy and matches. Swiss-style tournaments are shown to be superior to single/double/triple elimination and group stage tournaments. The choice of matching algorithm here likely has the greatest effect on result.

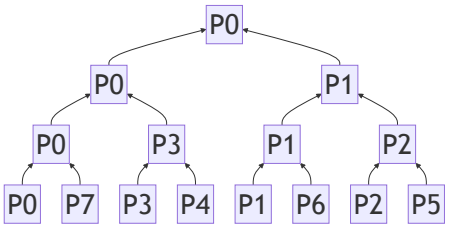
Design

In this chapter I will explain the design of the software artefact developed. To understand the final design that was implemented, it's first worth examining what information lead to it.

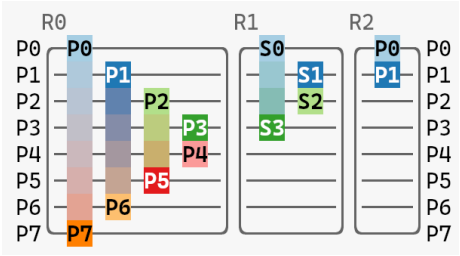
Analogies between sorting networks and tournaments

A key observation of made at the outset of this project is that there is an analogy between sorting networks and many tournament structures. Here, a sorting network refers to a fixed schedule or "network" of comparisons between a fixed set of objects. Each comparison has a fixed coordinate or *wire* in the network, and results in either the objects staying in the same position as they were (if they were already in order with respect to each-other), or they exchange places if not.

I call a sorting network "partial" if it does not determine a complete ordering among players; for instance if out of 16 elements it determines the greatest 8 in order, but leaves the remaining 8 in two buckets of 4 items where only the order of the buckets is known.



(a) Progression of a Single Elimination tournament under Slaughter Seeding



(b) Progression of the same Single Elimination tournament, expressed as a sorting network

A Single Elimination tournament has a partial sorting network construction. This is because at each round we can draw matches between the winners of the previous round only; that is, those players that now occupy the "high" position of the sorting network, after a single step of the network. At the same time, the losers of each round are simply not given any further comparisons (i.e., matches) from the point that they were eliminated, and so remain ranked at whatever point they were eliminated.

Similarly, we can also construct a Double-Elimination tournament by sorting network, by creating matches between those players that lost in first round of the "upper" bracket, and then alternating rounds that either accept new losers from the previous upper bracket round into new matches in the lower bracket, or that play off players who are in the lower bracket.

Swiss-style tournaments may fall under sorting networks if a pairing algorithm is chosen that does not enforce a no-rematch rule. Indeed, the sorting algorithm titled sorting algorithm *I Can't Believe It Can Sort* [18], a mistaken version of inser-

tion or bubble sort, can be viewed as a Swiss-style tournament with a matching algorithm that allows rematches, ran to $\Theta(n^2)$ rounds for n players; players that are closely ranked are repeatedly matched/compared. With the result of Sziklai *et al.* in mind, this sorting network with comically poor characteristics may indeed be an extremely effective tournament structure – if a quadratic number of rounds is permissible.

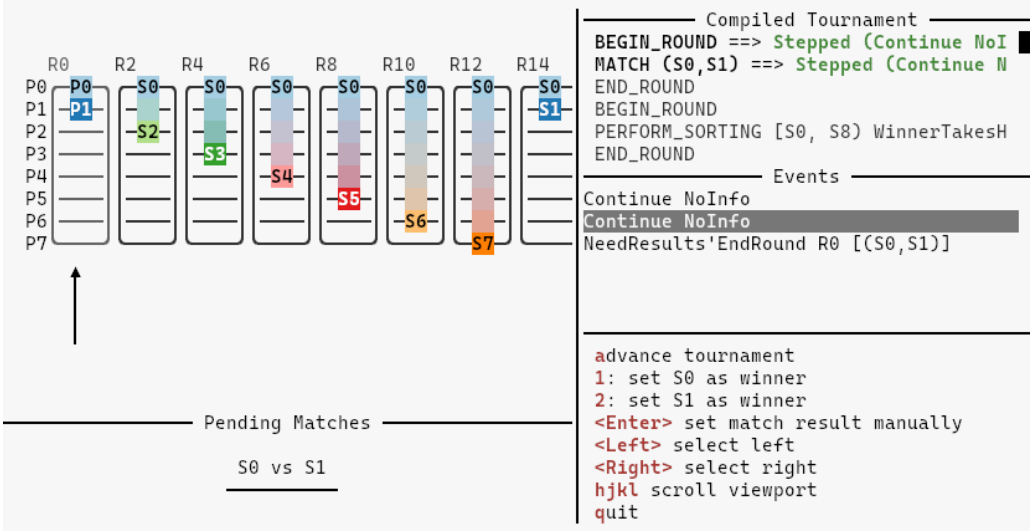


Figure 1: The first 8 rounds of an I Can't Believe It Can Sort tournament structure, visualised in the Journeyman-UI tool.

I call tournaments that have a direct sorting network analogy Sorting Network Tournaments (SNT). SNT does not encapsulate all tournaments that intuitively have a static structure. Namely, they cannot express round-robin tournaments. In a round-robin tournament, every player has exactly one match against every other player. But after a single round in a SNT, which players occupy what slots of the network are unknowable; hence we cannot proceed after a round of a SNT to pair "every other" player without the possibility of a rematch. Since in a SNT the players must exchange position upon a player from a numerically greater slot beats a player from a numerically lesser slot, a SNT Round-Robin requires knowledge of each prior round to avoid rematching.

We can work around this by defining another category of tournaments that intuitively have a static structure (given some n number of players), which I call Static Non-Sorting Network Tournaments (SNSNT), in which swap-exchange matches are removed, but a sorting mechanism is available based on points accumulated by players each match over a fixed number of rounds. A round-robin is such a tournament, since we can trivially construct a full round-robin schedule in advance using known algorithms such as the Circle Method.

A Type for Tournaments

The core type of the eDSL is described here. The primary feature of this type is its simplicity and ease of analysis. Tournaments in this library depend on two key operators inspired by the `algebraic-graphs` package, which are used to manipulate a small set of primitive tournament types. The most basic primitive chosen is `Match :: (Int, Int) -> Tournament`, which simply requests that match specified by the pair of sorting network slots. To justify the design of the core eDSL type, we shall build it from scratch:

Firstly, we need a way to, at a minimum, have two matches run at once. We can then generalise this operatorto allow any two tournaments to run in lock-step. Thus, the first operator defined is named `Overlay`, which overlays two tournaments together, running them in parallel. The `(+++)` operator provides an infix shorthand for this operator. For instance, the first round of a 4-player Single Elimination tournament can be encoded with just `(+++)` and `Match`: `(Match 0 3) +++ (Match 1 2)`, or 8-player: `(Match 0 7) +++ (Match 1 6) +++ (Match 2 5) +++ (Match 3 4)`. Note that unlike in a binary tree representation, the order of matches with respect to each-other has no effect on what tournament is described. To illustrate this, recall Figure 1(a); how would the expected outcome change if at the initial stage, the match $(0, 7)$ was swapped with $(2, 5)$?

The `(+++)` operator satisfies these algebraic properties:

1. Transitivity: `(x +++ y)` $\equiv$ `(y +++ x)`
2. Associativity: `(x +++ (y +++ z))` $\equiv$ `((y +++ z) +++ x)`
3. Identity: `(Empty +++ x)` $\equiv$ `(x +++ Empty)` $\equiv$ `x`

Secondly, we need a way to, at minimum, run two matches one after the other. We can then generalise this operator to allow any two tournaments to run in sequence. Thus, the second operator defined is `Sequence`, which connects two tournaments together by running them one after the other. It is given an infix shorthand `(***)`. It cannot satisfy transitivity, it does satisfy associativity. It does also *not* satisfy Identity; the reason for this is that creating empty rounds may be useful in the context of certain kinds of interleaving two tournaments together, such as allowing one tournament to run in even rounds, and another to run in odd ones. This could be achieved by sequencing an `Empty` round to every other round in the two input tournaments, and then overlaying them both – but only if the Identity property is not granted to `Sequence`.

We now have a simple tournament type `data Tournament0 = Overlay Tournament0 Tournament0 | Sequence Tournament0 Tournament0 | Match Int Int | Empty` which I claim is isomorphic to SNT, under an interpretation of `Match` always causing a swap-exchange to occur if there was an upset result. This structure does present some challenges in being able to generate a schedule of matches from it; since there can be an arbitrary, possibly unbalanced, mix of `Overlay` and `Sequences`; consider the following tournament:

```
unbalancedT0 = ((Match 0 1) +++ (Match 2 3))
           +++ ((Match 4 5) *** (Match 0 1))
```

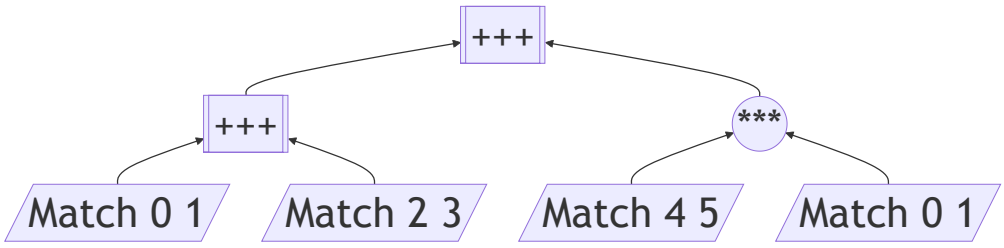
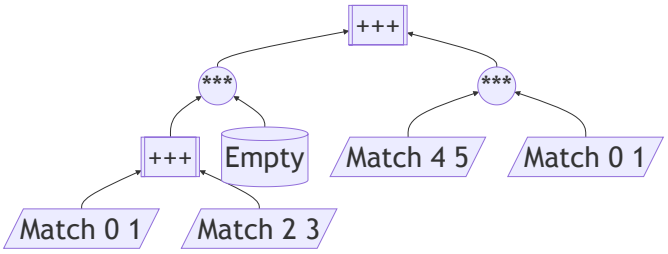
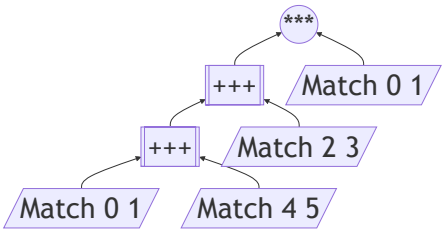


Figure 2: An unbalanced combination of sequences and overlays in a tournament

How to interpret this may not be immediately obvious. Obviously, on the second branch, match $(4, 5)$ and $(0, 1)$ are intended to run one after the other. But what does it mean to overlay on this structure? The interpretation taken by Journeyman is logically to insert an equivalent number of "empty sequences" that balance the tournament with the side that has more sequences than the other. That is, we *aligning* the two sides of an overlay operation.



(a) Step 1



(b) Step 2

A tournament that is expressed as a sequence of overlays of matches is said to be in Tournament Round Normal Form (TRNF); it can be thought of as a sequence of rounds of a tournament or sorting network, and it is now trivial to traverse this to create a schedule of rounds, of matches. Tournaments that satisfy this property are much easier to interpret than tournaments that don't.

By encoding a measure of depth (think, number of rounds) of a tournament into its type, we can greatly simplify functions that interpret tournaments; so long as we can transform any tournament into TRNF, we only need to express a function in terms of a single round of a tournament to be able to lift it to work on arbitrarily-deep tournaments.

Consider if we simply add a type-level natural number to the `Tournament` type:

```
data Tournament :: Nat → Type where
  -- | Connect two tournaments by having them occur
  -- simultaneously; analogous to the "Overlay" operator
  -- from algebraic-graphs
  Overlay :: Tournament a → Tournament a → Tournament a
  -- | Connect two tournaments by running one after the
  -- other. Note the change in depth
  Sequence :: Tournament a → Tournament b → Tournament (a + b)
  -- | Request a single match to be played
  One :: Match → Tournament 1
  -- | Do nothing
  Empty :: Tournament t

-- | Play the two players occupying the /slots/ specified by
-- this match together
data Match = Match Int Int
```

This is now getting quite close to the final type used by the Journeyman eDSL. Under this design, so long as we have a function that can lift a function from rounds to sequences of rounds as above, and a function that can transform an arbitrarily-deep/unbalanced tournament into TRNF, then interpreters over tournaments effectively only need to be concerned with singular rounds. As it turns out, adding this type parameter guarantees that our function `toTRNF :: Tournament a -> NonEmptyList (Tournament 1)` does indeed only return single rounds.

Since we only care about distinguishing tournaments of depth > 1 between tournaments of depth ≤ 1, we can swap out type-level arithmetic for a more lossy representation of $\mathbb{N}$: `data Depth = TOne | TMany`, and parameterise the tournament over that instead.

We also need an operation to allow a tournament to be parameterised over the number of players available. We would also like to be able to manipulate that player count, and moreover, to have a convenient mechanism for dividing tournaments along the "player" axis, to, for instance, represent a group stage round robin format.

```
  ByPlayerCount :: (Int -> Tournament t) -> Tournament t
  ByFocus :: (PlayerCount -> [Focus]) -> Tournament t

-- | A focus represents a slice over the current set of
-- players. Tournament interpreters must use the current
-- focus as an offset for matches generated within.
data Focus = Focus { start, length :: Int }
```

We also lack a way to tell a tournament interpreter what sorting strategy to use; should every upset match result in a swap-exchange? Or should no swaps occur, and only a final tally of points be used to finalise player standings? Additionally, we can not yet support tournaments that require past match results to decide what

to do (e.g., conventional sorting algorithms). This motivates the final three constructors:

```
BySwaps :: Tournament t -> Tournament t
ByPointAward :: Tournament t -> Tournament t
ByStandings :: (Standings -> Tournament t) -> Tournament t
```

For an API-centric view of the `Tournament` type, see the `Tourney.Algebra.Unified`[7] module. The main conceptual difference is that by adding a `ByStandings` constructor, tournaments can no longer necessarily be inspected completely in advance; they may have arbitrarily many rounds, conditional on certain match results. Therefore, an abstraction is built that allows tournaments to be inspected to their first "pure" subset – that is, the first (in terms of `Sequence` order) sub-tournament before a call to `ByStandings`.

The design of an abstract streaming type to support this follows from other open-source streaming libraries for Haskell, namely, `streaming`. The streams defined in Journeyman however support $O(1)$ concatenation and, as above, have the key difference that they do not necessarily require any effectful code to be ran to be able to inspect an element of the stream; a stream can purely yield values (i.e., matches, or rounds) that can also be popped off purely. Additionally, operators are defined for *aligning* two streams of values together, which is how the operation in Figure 3(a) is implemented.

With this representation, compilation of tournaments can ultimately reduces them to a single stream of a very small set of commands, which become trivial to interpret: `data Op = BEGIN_ROUND | END_ROUND | MATCH Match | PERFORM_SORTING Focus SortMethod`. The key detail is that an interpreter must not proceed past an `END_ROUND` or a `PERFORM_SORTING` until all the relevant matches have been completed[8].

Another key implementation detail is that the `Match` constructor is guaranteed to always refer to different slots in the sorting network, and that the first slot is always less than (in terms of index) the second. This greatly simplifies some logic such as keeping a sparse matrix of match results arranged `(Slot, Slot) → Maybe Result`; since the values are always ordered and unequal, only an upper triangle of such a matrix ever needs to be considered.

[7] `https://mikeplus64.github.io/journeyman/Tourney-Algebra-Unified.html`

[8] For `END_ROUND`, all matches are relevant. For `PERFORM_SORTING`, only those matches under the `Focus` specified are.

Final Definitions

I reproduce the final definition below:

```
data Tournament :: Depth -> Type where
  One :: Match -> Tournament TOne
  Empty :: Tournament t
  -- | Modify a tournament's focus; that is, the slice of slots of the
  -- sorting network that it concerns
  SetFocus :: (Focus -> [Focus]) -> Tournament t -> Tournament (TMod t)
  -- | Overlay two tournaments, to describe running two sub-tournaments
  -- in parallel. The depth of the tournaments must be the same
  Overlay :: Tournament a -> Tournament a -> Tournament a
  -- | Sequence two tournaments one after the other. The resulting
  -- tournament has a depth 'TMany' which restricts what functions
  -- are able to manipulate it.
  Sequence :: (KnownDepth a, KnownDepth b)
           => Tournament a -> Tournament b -> Tournament TMany
  -- | Sort the inner tournament by some sorting method
  Sort :: SortMethod -> Tournament t -> Tournament t
  -- | Depend on the player count to produce an inner tournament
  ByPlayerCount :: (PlayerCount -> Tournament t) -> Tournament t
  -- | Depend on the current standings, at the outset of the current round, to
  -- run the tournament.
  ByStandings :: (Standings -> Tournament t) -> Tournament t
  -- | Lift a single round of a tournament into having a depth 'TMany'
  LiftTOne :: Tournament TOne -> Tournament TMany
  -- | Lift a modified round of a tournament into having a depth 'TMany'
  LiftTMod :: KnownDepth t => Tournament (TMod t) -> Tournament TMany

-- | The depth of a tournament. Since we only care about distinguishing single
-- rounds from sequences of rounds, that is the only information stored here.
data Depth = TOne | TMany | TMod Depth

-- | Reflect a type-level 'Depth' into a term-level value.
class Typeable d => KnownDepth (d :: Depth) where
  depthVal :: proxy d -> Depth

instance KnownDepth d => KnownDepth ('TMod d) where
  depthVal _ = depthVal (Proxy :: Proxy d)

instance KnownDepth 'TOne where
  depthVal _ = TOne

instance KnownDepth 'TMany where
  depthVal _ = TMany
```

Since there are numerous types in the sorting network representation that are ultimately just `Int`, I introduce some `newtype` definitions to prevent the accidentally mixing of, for instance, a sorting network slot, and a player. These are named `Slot`, `Player`, and `RoundNo`. For convenience, each has the full breadth of `Num` and `Integral` operations available that `Int` has.

The Journeyman eDSL: An accumulating continuation over `Tournament`

The final eDSL is defined as a continuation-based accumulation monad over the original algebraic `Tournament` type. Since there are two main operations for merging tournaments together (overlays and sequences), the builder is parameterised by what depth it is at, which is used to provide the default merge operation at that depth. That is, a builder of just matches within a round is a `Round ~ Builder TOne`; a builder of rounds then is a `Steps ~ Builder TMany`.

The key convenience gained here is to be able to syntactically invert the `ByPlayerCount` and `ByStandings` constructors; thus:

```
-- | Retrieve the current player count
getPlayerCount :: Merge t => Builder t () PlayerCount

-- | Retrieve the current standings
--
-- Warning: this operation will have the effect of segmenting
-- the tournament into two sections, as a tournament runner cannot
-- continue past this point without first having the standings.
getStandings :: Merge t => Builder t () Standings
```

As the full API largely is a consequence of the primitives outlined above, they will not be reproduced here. For API-centric documentation of the Journeyman eDSL, see the online documentation[9]. Most functions here simply add something to the current `Tournament` accumulation, by whatever merging strategy (overlaying or sequencing) is most appropriate.

A virtual machine for Tournaments

To actually run a tournament, a virtual-machine, coined Tournament Virtual Machine (TVM), is constructed. The TVM connects a Tournament compiler, which has been described already, and an interpreter over that small set of tournament "opcodes".

The TVM is what powers the Journeyman-UI which has provided screenshots throughout this thesis, and is capable of evaluating tournaments, including displaying their pure subset ahead of actually arriving at those sections, as well as actually "peeking" into the future of an impure tournament (by supplying it with mock standings).

The TVM also has simulation capability. This is a fairly simple combination of the main TVM loop and a function that, given a match, creates or retrieves its result somehow. Using this, I implement a tournament interpreter for evaluating tournaments by an initial Elo distribution of players.

The key API relevant here, provided by the core VM module `Tourney.VM`, is:

[9] `https://mikeplus64.github.io/journeyman/Tourney-Algebra-Builder.html`

```
-- | Create a 'VM' from a tournament and a fixed player count.
setup :: Tournament t -> PlayerCount -> IO VM

-- | Get a history of standings updates, by round, from a VM.
getStandingsHistory :: VM -> IO (MapByRound Standings)

-- | Run a VM to completion, using the input Elo distribution
-- to compute the win probabilities of each player per match.
-- A final Elo distribution is returned which reflects changes
-- in Elo that occured throughout the tournament.
simulateByEloDistribution :: Vector Float -> VM -> IO (Vector Float)
```

Tournament Interaction and Display

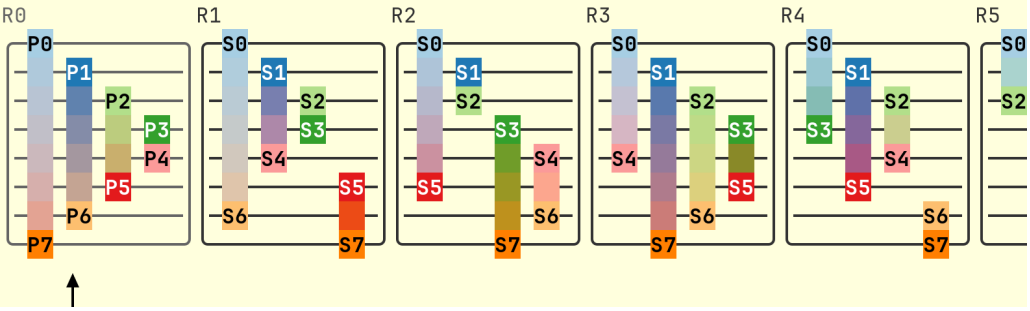
In order to inspect and interact with tournaments, a terminal-based UI is built. It enables the display and running of tournaments, and allows for inspection of the core "Virtual Machine" that compiles and interprets tournaments into a stream of commands, such as to easily identify the number of rounds/true depth of a tournament, and its parallelisation (if any). Some screenshots of this tool have already been included in this thesis.

Demonstration of the Journeyman eDSL

We shall now tour some tournament structure definitions I have created that use the Journeyman eDSL.

Round Robin and Group Stage Round Robins

The *Circle Method* for round-robin scheduling is used here. The key point to note here is the position of the `points` function, which requests sorting by point awards for the matches within.




```
roundRobin :: Steps () ()
roundRobin = points do
  count <- getPlayerCount
  let (!midpoint, !r) = count `quotRem` 2
  let !n = count + r
  mapM_
    (round_ . map match)
    [ foldAround midpoint (map Slot (0 : ((n - i) ..< n)
                                      ++ (1 ..< (n - i))))
    | i <- [0 .. n - 2]
    ]
```

More interestingly, we can use the `SetFocus`-based combinator `divideInto` to run multiple parallel round-robin tournaments.

```
groupRoundRobin :: Int -> Steps () ()
groupRoundRobin numGroups = divideInto numGroups roundRobin
```

Insertion Sort, and *I Can't Believe It Can Sort*

Three variations are provided here. The "naiive" variant of insertion sort here follows the folk imperative definition of the insertion sort algorithm; but this approach does not take advantage of the parallelisation available in a sorting network. While it should be possible for the Journeyman tool to simplify the graph of a tournament through an analysis of match dependencies, that is not within the scope of work undertaken in this project.

A parallelised sorting network implementation of insertion sort is thus also provided. Finally, the *I Can't Believe It Can Sort* algorithm is provided. Note that as expressed, the *I Can't Believe It Can Sort* algorithm cannot be parallelised; every match except for the initial one has a dependency on the prior match's result. Possibly, we can improve this by creating a fully saturated network that simply alternates between matching the odd-adjacent and even-adjacent slots, but that idea is not pursued here.

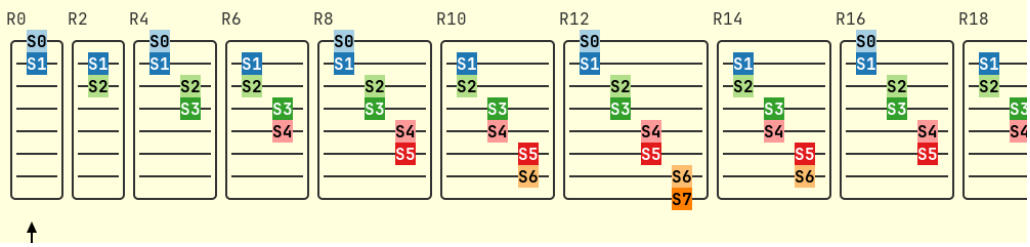


Figure 3: Insertion sort of 8 players represented in Journeyman-UI.

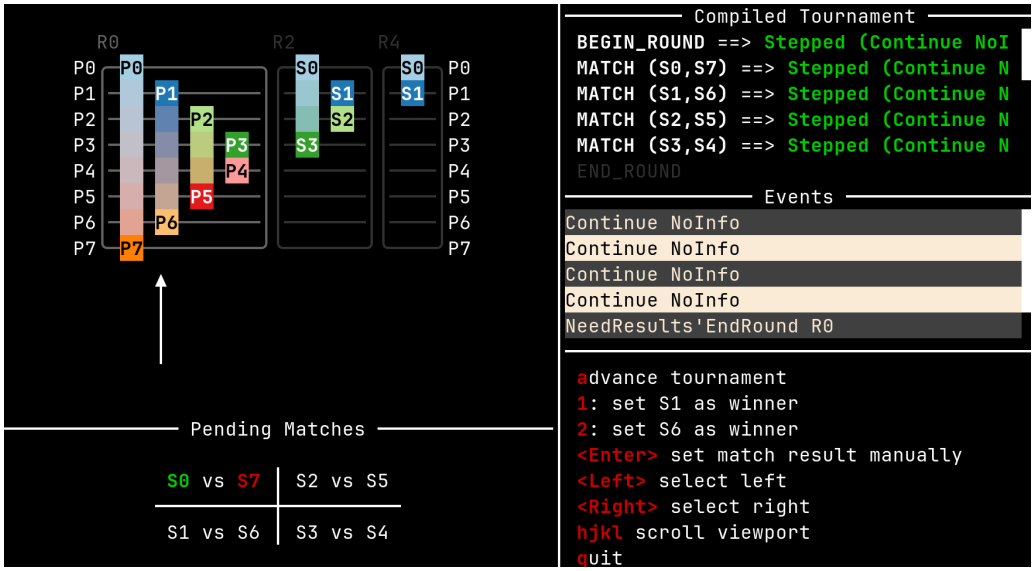
```
insertionSortNaiive :: Steps () ()
insertionSortNaiive = do
  n <- getPlayerCount
  i <- list (0 ..< Slot n)
  j <- list (i ..> 0)
  swaps (round_ (match (Match j (j - 1))))

insertionSortNetwork :: Steps () ()
insertionSortNetwork = do
  n <- Slot <$> getPlayerCount
  i <- list ([0 .. n - 2] ++ reverse [0 .. n - 3])
  swaps . round_ . asRound $ do
    let (m, r) = i `divMod` 2
    j <- list [0 .. m]
    match (Match (j * 2 + r) (j * 2 + 1 + r))

iCan'tBelieveItCanSort :: Steps () ()
iCan'tBelieveItCanSort = do
  n <- getPlayerCount
  i <- list (0 ..< Slot n)
  j <- list (0 ..< Slot n)
  when (i /= j) do
    round_ $ swaps $ match (Match i j)
```

Single Elimination

Single Elimination tournaments turn out to be quite easy to represent in Journeyman, especially when compared to traditional binary-tree approaches, where the exact order of nodes/leaves has a determining effect on outcomes. The key observation is that for the first round, with the simple arrangement of players from most skilled to least skilled, we can use list operations to split that list into two and fold the two sides together; thus making the most skilled player meet the least skilled, and the second-most skilled to the second-least skilled, and so on, until the two players at the middle level meet. Since this operation is quite common in tournament construction, it is given the name `foldAroundMidpoint`.

```
singleElimination :: Steps () ()
singleElimination = do
  count <- getPlayerCount
  let depth = log2 (2^ceil (log2 count))
  d <- list [depth, depth-1 .. 1]
  swaps (round_ (foldAroundMidpoint [0 .. 2^d - 1]))
```

Note that this tournament does output bye matches between non-existent slots if the player count is not a power of 2. These matches are treated as automatic wins for the valid player, which is a necessary common practice for elimination tournaments that have a binary-tree structure.



Double Elimination

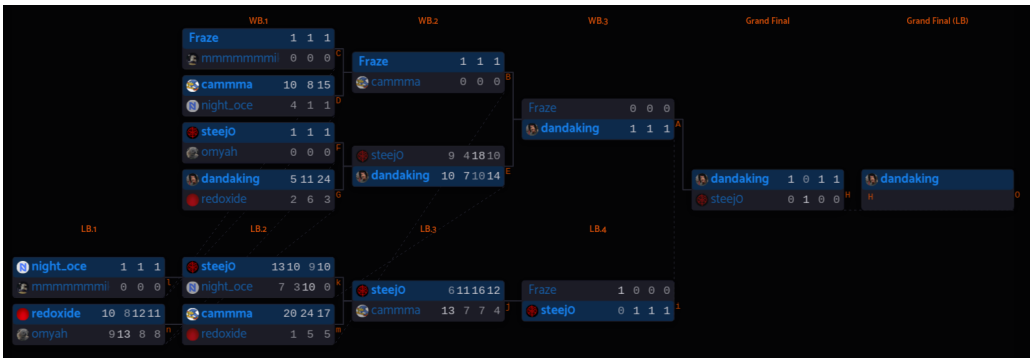


Figure 4: An example Double-Elimination bracket hosted on `kuachi.gg` [19]

For Double Elimination it is worth first considering how an analogous sorting network can be constructed. Obviously, the upper bracket can proceed identically to a single elimination tournament, so we can definitely express it as some overlay operation like `singleElimination +++ lowerBracket`. To elucidate what structure is appropriate, I diagram the structure required for an 8-player Double Elimination sorting network:

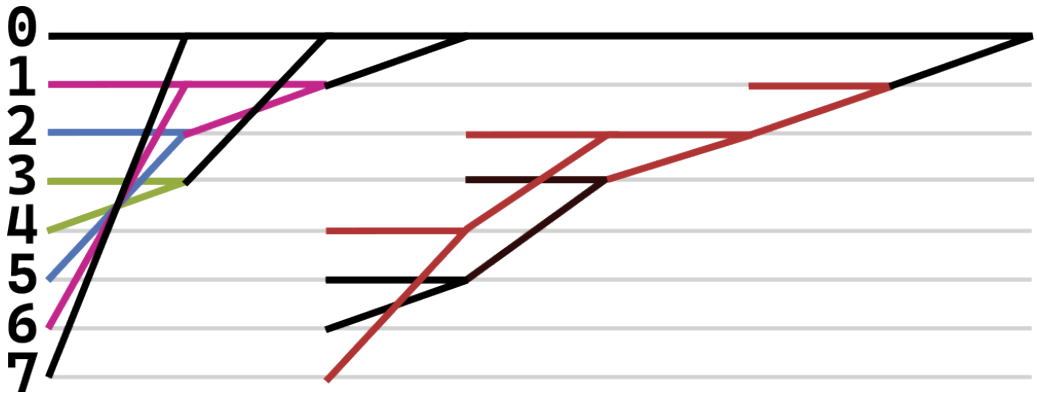


Figure 5: For contrast, the same Double Elimination bracket expressed as a sorting network.

I use the result provided by [7] to create a linking function.

```
-- A direct implementation of Double Elimination tournaments.
--
-- The approach here is to:
-- 1. Construct a single elimination upper bracket
-- 2. Create an initial lower bracket round, from the lowers of the first round of
--    the single elimination bracket
-- 3. For each other upper bracket round:
--    3.1. Create a lower bracket round that accepts the losers from that round
--    3.2. Create a lower bracket round that plays off only LB players against
--         eachother.
--
-- Since we only depend on the rounds generated in the upper bracket, we can
-- parameterise that; so our "doubleElimination" function is generalisable to be
-- able to add an extra loser's bracket to _any_ tournament, although it is
```

```
-- likely to have strange results when given tournaments that are not in the
-- same shape as single-elimination.
--
-- Thus we can use this to create quadruple elimination brackets, but not
-- triple.
doubleElimination :: Steps () ()
doubleElimination = addLosersBracket singleElimination

addLosersBracket :: Steps () () -> Steps () ()
addLosersBracket original = do
  Right (ub1 :< ubs) <- inspect (ByRound Flat) original
  let lowerRound1 = foldAroundMidpoint (ub1 ^.. each . likelyLoser)
  swaps (round_ ub1)
  swaps (round_ lowerRound1)
  evaluatingStateT (lowerRound1 ^.. each . likelyWinner) do
    (i, upper) <- lift (list (V.indexed ubs))
    lastWinners <- get
    let shuffledLosers = linkFun i (upper ^.. each . likelyLoser)
    -- Accept new losing players from the upper bracket
    let acceptRound = zipWith Match lastWinners shuffledLosers
    -- Then perform a round of just lower bracket players being eliminated
    -- I.e., match up the winners of the round_ we just wrote
    let losersRound = foldAroundMidpoint (acceptRound ^.. each . likelyWinner)
    -- Finally, add these rounds, and store the winning players in losersRound
    -- for the next iteration
    round_ do
      swaps (toRound upper)
      swaps (toRound acceptRound)
    round_ (swaps (toRound losersRound))
    put (losersRound ^.. each . likelyWinner)

linkFun :: Int -> [a] -> [a]
linkFun size = foldr (.) id (replicate size linkFunSwap)

linkFunSwap :: [a] -> [a]
linkFunSwap l = drop h l ++ take h l
  where
    h = length l `div` 2
```

Future work and limitations

There is little way to verify that the tournament structure generated is the one that is actually intended, other than to view it in the Journeyman-UI tool. Additionally, analysis is currently limited; the UI tool will tell you how many rounds a tournament has, but that is the extent that the information goes.

The original aims of statistical analysis of arbitrary tournaments has not been met. While Journeyman does have the facility to do simulations by an Elo rating distribution, the work of collating results from such simulations has not been undertaken yet. Concretely, the future work to complete this aspect of the project would be:

1. Run the existing Elo-driven simulator repeatedly for a particular tournament structure.
2. Taking into account changes in Elo that happened throughout the course of the tournament, tally the number of inversions over many runs. It may also be interesting to compare the results when assuming a fixed Elo (i.e., absolute certainty over players' relative skills) compared to allowing for the self-correction mechanism of Elo to occur within a simulation.
3. Do the same for other tournament structures, with the same initial Elo distribution, and compare results.

Unfortunately the related aim to compare common tournament structures with existing sorting algorithms is not completely met; it has proved to be too much work for me to complete over the 2 semesters. That said, the process for doing so would be identical to the above – which is to say that it is a tantalisingly small amount of remaining work.

Additionally, there are undoubtedly bugs in the implementation I have created. A more rigorous testing regime from the start may have been useful here, but due to the design of the eDSL being an invention created for under this project, it was not clear what the correct semantics *should* be to be tested for in the first place.

References

[1] *Topic: Sports betting worldwide*, [Online; accessed 19. Nov. 2023], Nov. 2023. [Online]. Available: https://www.statista.com/topics/1740/sports-betting.

[2] *Biggest eSports ever games by tournament prize pool 2023*, [Online; accessed 19. Nov. 2023], Nov. 2023. [Online]. Available: https://www.statista.com/statistics/532840/share-esports-prize-pool-global-by-game.

[3] *The pandemic has accelerated the growth of e-sports*, [Online; accessed 19. Nov. 2023], Nov. 2023. [Online]. Available: https://web.archive.org/web/20230727061236/https://www.economist.com/international/2020/06/27/the-pandemic-has-accelerated-the-growth-of-e-sports.

[4] L. Csató, "Quantifying the unfairness of the 2018 fifa world cup qualification," *International Journal of Sports Science & Coaching*, vol. 18, no. 1, pp. 183–196, Apr. 2022, ISSN: 2048-397X. DOI: 10.1177/17479541211073455. [Online]. Available: http://dx.doi.org/10.1177/17479541211073455.

[5] J. Guyon, "Risk of collusion: Will groups of 3 ruin the fifa world cup?" *Journal of Sports Analytics*, vol. 6, no. 4, pp. 259–279, Jan. 2021, ISSN: 2215-0218. DOI: 10.3233/jsa-200414. [Online]. Available: http://dx.doi.org/10.3233/JSA-200414.

[6] P. Walker and T. Branigan, "Badminton's world governing body apologises after players are disqualified," *the Guardian*, Dec. 2012. [Online]. Available: https://www.theguardian.com/sport/2012/aug/01/badminton-body-apologises-players-disqualified.

[7] I. Stanton and V. V. Williams, "The structure, efficacy, and manipulation of double-elimination tournaments," *Journal of Quantitative Analysis in Sports*, vol. 0, no. 0, pp. 1–17, Jan. 2013, ISSN: 2194-6388. DOI: 10.1515/jqas-2012-0055. [Online]. Available: http://dx.doi.org/10.1515/jqas-2012-0055.

[8] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, *Design guidelines for domain specific languages*, 2014. DOI: 10.48550/ARXIV.1409.2378. [Online]. Available: https://arxiv.org/abs/1409.2378.

[9] A. Mokhov, "Algebraic graphs with class (functional pearl)," *SIGPLAN Not.*, vol. 52, no. 10, pp. 2–13, Sep. 2017, ISSN: 0362-1340. DOI: 10.1145/3156695.3122956.

[10] C. T. Edwards, "Double-elimination tournaments: Counting and calculating," *The American Statistician*, vol. 50, no. 1, pp. 27–33, Feb. 1996, ISSN: 1537-2731. DOI: 10.1080/00031305.1996.10473538. [Online]. Available: http://dx.doi.org/10.1080/00031305.1996.10473538.

[11] *Elo Rating System - Chess Terms*, [Online; accessed 19. Nov. 2023], Nov. 2023. [Online]. Available: https://www.chess.com/terms/elo-rating-chess.

[12] A. V. N. Dinh, N. P. H. Bao, M. N. A. Khalid, and H. Iida, "Simulating competitiveness and precision in a tournament structure: A reaper tournament system," *International Journal of Information Technology*, vol. 12, no. 1, pp. 1–18, Nov. 2019, ISSN: 2511-2112. DOI: 10.1007/s41870-019-00397-5. [Online]. Available: http://dx.doi.org/10.1007/s41870-019-00397-5.

[13] N. Pham, H. Bao, S. Xiong, and H. Iida, "Reaper tournament system," in Jun. 2017, pp. 16–33, ISBN: 978-3-319-73061-5. DOI: 10.1007/978-3-319-73062-2_2.

[14] D. Van Bulck and D. Goossens, "Handling fairness issues in time-relaxed tournaments with availability constraints," *Computers & Operations Research*, vol. 115, p. 104 856, Mar. 2020, ISSN: 0305-0548. DOI: 10.1016/j.cor.2019.104856. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2019.104856.

[15] P. C. Placek, "The impossibility of a perfect tournament," *Entertainment Computing*, vol. 45, p. 100 540, Mar. 2023, ISSN: 1875-9521. DOI: 10.1016/j.entcom.2022.100540. [Online]. Available: http://dx.doi.org/10.1016/j.entcom.2022.100540.

[16] A. Karpov, "A new knockout tournament seeding method and its axiomatic justification," *Operations Research Letters*, vol. 44, no. 6, pp. 706–711, Nov. 2016, ISSN: 0167-6377. DOI: 10.1016/j.orl.2016.09.003. [Online]. Available: http://dx.doi.org/10.1016/j.orl.2016.09.003.

[17] B. R. Sziklai, P. Biró, and L. Csató, "The efficacy of tournament designs," *Computers & Operations Research*, vol. 144, p. 105 821, Aug. 2022, ISSN: 0305-0548. DOI: 10.1016/j.cor.2022.105821. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2022.105821.

[18] S. P. Y. Fung, *Is this the simplest (and most surprising) sorting algorithm ever?* 2021. DOI: 10.48550/ARXIV.2110.01111. [Online]. Available: https://arxiv.org/abs/2110.01111.

[19] *AQL5 - Division 1 - Division 1 Finals*, [Online; accessed 17. Nov. 2023], 2023. [Online]. Available: https://kuachi.gg/cups/dacaa5af-a48a-4699-b181-a5de43f8c658/stage/1.