A practical study of modeling a highly generalised tournament structure

Abstract

In this paper I will document my attempt at creating a highly generalised model of the structure of a tournament. In doing so I will discuss the issues arrising (and some suggested solutions) when trying to create a data structure which is so highly generalised.

1 Introduction

In this paper[1] I will detail my attempt to model the structure of a tournament with a view to creating a piece of software which would allow users to interact with the tournament in an experimental way. I will discuss my inspiration for designing such a system and my motivation for trying to make it as highly generalised as possible. I will detail the design decisions which were taken and how I decided to represent a tournament structure in both a written form and as EDEN data structures. In implementing the system, using the rapidly developing tool tkeden, I had some problems, some of which were solved and some of which were sidestepped and avoided. I will detail these and the actions I took to remedy the situation. I will also attempt to suggest further improvements to the model and the tkeden environment. Finally I will suggest some areas of Empirical Modeling which I think would provide interesting and productive targets for further research.

2 Inspiration

When trying to think of something which would benefit from an empirical model, I found myself looking for things which could be modeled to show others how they worked or how they interacted with things around them, to take something we know about and allow others to experience it in an easier way in a computer model. I was also looking for things which we interact with in an empirical way in the real world. These things, when modeled, would not be of use merely as a research or learning aid but would be computer based replacements for their real world counterparts. When attending an Ultimate Frisbee tournament I found myself and others asking questions like, "If we lose the next game can we still finish in the top eight" or "In what circumstances would we have to play team x". I realized that we were interacting with the tournament structure in an experimental way. Although the Tournament Structure was published so that everyone knew when they needed to be on the pitch, if it were used in an experimental fashion like this, it provided another use. The problem we had was that a tournament structure is often highly complicated and knowing the effects of losing a game is a computationally difficult task, one our match-weary brains could rarely do well. It occurred to me that by posing questions like that to a computer model of the tournament structure, we could get more complicated and more meaningful results.

[1]Weighting of 60% report and 40% model

3 A Highly Generalised Tournament Structure

I did some research into different tournament structures used in not just Ultimate Frisbee but other sports. This revealed that tournament structures were many and varied and even if you split them into a series of structurally unrelated stages, which I later did, there were still a significant number of different formats. It was clear that to be of any practical use, I would have to develop the data structures and algorithms to deal with all of these types of tournament structures. I identified that a tournament could be split into atomic stages in which the order of games was unimportant. Before and after such a stage, every team could be assigned a position, each stage attempting to move this position towards the one actually deserved. It can be noted that in order to have a knockout tournament, you can assign the losers of a particular n games to positions $2n$ to $n + 1$, and then play a similar stage of $\frac{n}{2}$ games with the top n teams. The position of the bottom teams would therefore stay the same and the top teams would receive higher placings.

4 Tournament Notation

In order to describe a tournament structure in a way which can be entered into the model, I had to design a tournament notation. This was a two stage process, firstly I had to identify all of the information needed to fully describe a general tournament structure and secondly I needed to design a notation and implement it using the Agent Oriented Parser (AOP) which provides a simple framework for creating notations for the tkeden environment. I decided that a tournament could be broken down into atomic stages, within which the order of the games was irrelevant. Within each stage, the fixtures need to be enumerated, using the team's starting positions as the index (eg. the team who started the stage in 1^{st} vs. the team who started the stage in 13^{th}). A method of determining where a team stands at the end of the stage is also required. For this, I allowed two sections of EDEN code, one to perform some common setup operations (this section is executed once, when the tournament is loaded) and some other sections each containing an EDEN expression, which evaluates to the final position of the team starting in that position. For example if the 3^{rd} expression evaluates to 7 then the team which started the stage in 3^{rd}, finishes it in 7^{th}. To complete the notation, I devised an overall structure and defined some separators. The notation takes the form of a series of stages where each stage is surrounded by braces. Each stage consists of 3 sections separated by the following string, "`---`". In the first section there is a list of fixtures, one per line, team numbers separated by a colon. In the second section there is a section of EDEN code to be executed when the tournament is parsed. In the third section, there is a series of expressions, one per line, defining the finishing position of the teams. It later became evident that there would have to be another, optional, section to each stage which lists the games upon which a teams position depends. This takes the form of an array of fixture indexes, one for each team in the stage, and is the 3^{rd}section of a stage. If this information can't be supplied, nothing should be put between the 2 "`---`" separators.

5 Data Structures

In order to represent the structure of the tournament in the program, I have used the same form of information given in the notation and supplemented it with information given by the user, for example the result of a given fixture, and information inferred by the program, for example potential finishing positions. I have stored fixtures, placings, results and potential placings as my main working data. For the fixtures (`fixtures`), I have a three dimensional array where the first dimension is the stage to which the fixture belongs, the second dimension is the fixture in that stage which is being detailed and the third dimension is the team which is detailed (home or away), for instance if `fixtures[2][3][1]` has value 4 then the "home" team for the 3^{rd} game in the 2^{nd} stage is the team which started that stage in 4^{th}. I have used a similar format for the results array (`results`), however the values stored do not represent which team is to play but how many points they scored in that game, for instance `results[2][3][1] == 4;` would mean that the "home" team for the 3^{rd} game in the 2^{nd} stage scored 4 points. Initially both the placings (`placings`) and the potential placings (`potentialplacings`) were going to be stored in a two dimensional array where the first dimension was stage and the second was team. `placings` would store the final place given the current `results` and `potentialplacings` would store a list of the potential placings given that any game which doesn't have an associated result could be a win for either team or a draw. unfortunately, due to a limitation on tkeden**??**, I have had to use a similar mechanism to store this data, whereby you use a function to access a variable as if it was an element of an array.

6 Development Problems and their Solutions

6.1 Arrays of Formula Variables

6.1.1 Background

In Eden, things can be assigned to objects and an object can be one of four types, of which two are Read/Write Variables(RWV) and Formula Variables. A RWV is assigned a value which doesn't change, for example 2 or "fish". A formula variable is assigned an expression which defines its value, if the value of any of the variables in the expression change, then the value of the formula variable changes accordingly. This is the basis of a definitive programming language as values can be defined in terms of other values and know that any changes will propagate through the program as necessary. All data in EDEN has a type, for example int, char or, interestingly, list. This means that a formula variable may have a list as its value but you may not form arrays or lists of formula variables as lists can only contain an integer, a character, a string or another list. This is a problem which extends down to the system for maintaining dependencies in tkeden. To do this a graph of dependency is created, which is unaffected by changing a RWV, however by allowing a variable number of formula variables in the form of a list, the graph could be changed by changing a RWV.

6.1.2 The Problem

I would like to use an array of formula variables to store the expressions for calculating the placing and potential placing of all of the teams. I envisioned the elements of this list having individual dependency and allowing their current values to be referenced in the same way as a standard formula variable. Not being able to do this meant that I couldn't refer to an arbitrary number of formula variables using the same name and simply indexing it using an integer.

6.1.3 The Solution

In order to be able to refer to an arbitrary number of formula variables I designed some functions which took as arguments, among other things, the string representation of the name by which I would like to refer to the variable, including index. These functions read, wrote and got the size of my "virtual arrays"; this meant that I could refer to arrays, albeit in a cumbersome way, of formula variables. In order to make these functions, I parsed the name of the variable given to me and worked out the indexes. I used string concatenation to create a suitable name for the variable to represent this array element. For example `fish[1][2]` became `arr_fish_1_2`. I then used the execute function to either read this into a temporary variable whose value I then returned, or to set it equal, using "is", to an expression also given as an argument. In order to get the size of an array, I had to track the number of elements which were in the array and more importantly the largest index in the virtual array. By keeping track of this number in a variable whose name mirrored the name of the elements themselves, I could create a function which returned the size of the array. I did, however, have more of a problem with multidimensional arrays as I had to store the length of the array in all dimensions so by setting `fish[3][5]` I needed to check the size of `fish` and `fish[3]`. This provided a reasonable way to use arrays of formula variables. However, I found it cumbersome, especially in the interactive client where it was annoying to have to type the line `foo=readArray("fish[2][4]"); ?foo;` instead of simply `?fish[2][4];`.

6.2 Complex Data Structures Are Too Difficult to Construct

In Eden, the most complicated native data structure is the list and although it is possible to make some quite complicated data structures by building on this with list elements which are themselves lists, it would be nice to have a data structure similar to the C struct. Using this a datatype could be defined which contains a series of *named* components. This would mean statements which currently look like `fixtures[2][3][1]` could be rewritten as `stage[2].fixture[3].hometeam` which is clearer and allows for just as much flexibility and power, it however reduces the risk of unintentionally accessing an array element. The solution to this problem is complicated as it would be possible to design a definitive notation which could deal with these more complex data structures, however they would have to either be made inaccessible to EDEN or have their elements mapped into the EDEN namespace which would negate the benefit of having the notation. It would therefore be necessary to implement this in the EDEN language.

6.3 $\$t$ in the AOP

In order to make the tournament notation as general as was required I decided that the best way to represent which team would finish in a given position was to use an EDEN expression. By using the separators previously mentioned, I could identify the block and this could be referred to in an AOP action statement using $\$t$. I discovered however that rather than $\$t$ referring to a memory location where we should read the data to use, as a normal RWV does, $\$t$ simply gets replaced by the string it represents before the action statement is executed in EDEN. It is clear now why this is so: the action section of a AOP agent needs to be executed in EDEN, the $\$t$ is simply replaced then the whole action statement is given as an argument to execute(). Unfortunately this means that it is very difficult to allow certain EDEN code snippets in an AOP notation and be able to assign the string representation of them to a variable. Incidentally if I had been able to solve the first problem in a more satisfactory way, I could have simply written `place[1][2] is $t` and assuming the agent had parsed a valid EDEN expression, the action statement would have been valid. However I have to use a temporary variable to store the expression in before passing it to a function which allowed the use of virtual arrays, to do this I have had to write `tmp = "$t"; setArray("...",tmp);`. This causes the problem that when you include a double quote in the expression, even if it has a matching close quote, you may introduce syntax errors into the action statement as the close quote will close the quote around the $\$t$ and the rest will be unescaped. The only solution to this problem is to declare that all quotes appearing in the placing expression should be escaped with a \ character. I feel this is an inconvenience as it would be better to ask for any valid EDEN expression, which this escaped version is not.

7 Future Directions For Research into Highly Generalised Data Structures

Until the advent of a new form of data type which allows a more complicated design of data structure, I think it is most relevant to develop the lists type in EDEN. It would also be interesting to research into the implications of allowing EDEN snippets to be parsed in the AOP and also developing a way to alter the snippets in a complex way using a simple parser. For instance, you may want to prefix all variables in a snippet with a certain string. This is currently best achieved by requiring the snippet writer to add a "?1" to the beginning of all variable names, and using the macro function to replace it with the prefix, but this doesn't allow a very complicated transformation. It may be possible to extend the AOP for this purpose by creating a complement to "literal", "prefix" and the like for parsing either EDEN statements or EDEN expressions.

8 Conclusion

I believe that by using a highly generalised approach to creating the tournament structure being modeled I am designing data structures in an Empirical Modeling Mentality, whereby the user is free to experience the model in whatever way they feel, and more importantly that by using the model in this way they will gain the information about the model which is most relevant and means the most to them. By not prescribing anything more than the things which make a tournament more than just a series of unconnected games, I feel that the user is free to experiment not just with the effects of changing the results but with changing the structure. In order for this to be effective I think a better interface to this power should be developed as I have detailed in my paper. I feel excited by the prospect of Empirical Modeling developing into a mainstream method of programming, when the languages and tools are available to support it, and I feel it could have a significant influence on many applications like this where experimentation on a general data structure would be of great benefit.