

Decomposing Hackenbush

A Project in Combinatorial Game Theory

Joey Hammer
University of California, San Diego
<http://cgt.calculusfairy.com>

Table of Contents

Contributors to the Project	5
Early Observations	6
“Back of the Napkin” Calculations	
Decomposition Details	
Research Timeline	7
Project Software	8
Early Calculators	
Visual Hackenbush	
Introduction to the Project	10
Assumptions & Conventions	
Notation and Abbreviations	
Methods of Proof	
Combinatorial Game: Defined	14
Conditions for Analysis	
Rules of Play	
The Players	
Combinatorial Game Theory Fundamentals	18
Definition Zero	
Definition of a Number	
Zero is a Number	
Option-Value Relation for a Number	
Simplicity Rule	
Hackenbush Definitions	21
Hackenbush Stick	
Hackenbush Trunk	
Hackenbush Tree	
Hackenbush Stalk	
Instance of Branching	
Trunk Stalk	
Purity	
Impediment	
Highest Stick	

Table of Contents

Rules of Red-Blue Hackenbush	23
Whose Game?	
Hackenbush Gameplay	
Hackenbush is Hard	
Hackenbush Theorems	25
Tweedle-Dum and Tweedle-Dee Strategy	
Fundamental Theorem of Hackenbush (FTOH)	
Stalk Reduction Principle (SRP)	
Corollary 1	
Combinatorial Game Values for Hackenbush	31
Game Theoretic Values	
Zero	
Integer-Valued Games	
Notation and Values	
Fractional-Valued Games	
Beyond Values	
Hackenbush Stalks	38
Significance of Stalks	
Evaluating Stalks	
Stalks Algorithm	
Implementation	
Hackenbush Trees	41
Trouble with Trees	
Ordinal Sums	
Algorithm for Trees	
Implementation	

Table of Contents

Hammerian Trees	45
Introduction	
Hammerian Domain	
Definition: Hammerian Tree	
Hammerian Classification	
Method of Decomposition	
Other Observations	
Joshua Trees	51
Definition: Joshua Tree	
Joshua Classification	
Other Observations	
Conclusions	53
Summary	
Future of Hammerian Research	
Appendix A:	55
Hackenbush Game Tree Example	
Appendix B:	62
Visual Hackenbush 1.1 Documentation	
Software	89
References	90

Contributors to the Project

Research and Programming:

Hammer, Joey

Research Associates:

Lee, Jason

Trotter, Jeremy

Research Supervisor:

Haff, Leonard

Inspiration:

The inspiration for this project stemmed from the Combinatorial Game Theory class taken at UC San Diego and taught by Professor Haff and Jason Lee.

Inside one of the homework assignments was a problem which concerned finding the values of some Childish Hackenbush Trees. It soon became clear that there had to be a faster way to calculate the values to such trees (and other trees for different Hackenbush variations). The idea for writing an algorithm and assembling a program was born.

Early Observations

“Back of the Napkin” Calculations

In the Spring of 2003, my research associate, Jeremy Trotter, and I were enrolled in an experimental course based on Combinatorial Game Theory: Math 168A. Our discovery for the topic of research outlined in this paper was made while working on a homework assignment involving Red-Blue Hackenbush at *Pick Up Stix*®. The idea struck when I happened to see two sets of Hackenbush games: the first was a single Red-Blue Hackenbush Tree and the second was a pair of Red-Blue Hackenbush Stalks, both of which had the same combinatorial game value.

This sparked more than year’s worth of extensive undergraduate research and intensive software programming in the game of Hackenbush to seek out the parameters for and the logic behind this interesting decomposition.

Decomposition Details

Hammerian Trees *(details analyzed later)*

A Hammerian Tree is any Red-Blue Hackenbush Tree whose combinatorial game value is the same as the sum of the decomposed branches and its trunk.

Joshua Trees *(details analyzed later)*

A Joshua Tree is a Hammerian Trees whose combinatorial game value is the same as its trunk.

Research Timeline

- Spring 2003:** Math 168A: course in Combinatorial Game Theory
Pick Up Stix®: discovery of possibilities for tree decomposition
- Summer 2003:** Parameter Research: 200 trees calculated by hand
Programming: Stalks Algorithm and Binary Tree Calculator
- Winter 2004:** Parameter Research: Hammerian parameters solidified
Programming: Hackenbush Applet (Visual Hackenbush 1.0)
- Summer 2004:** Programming: Visual Hackenbush 1.1
Completion of Visual Hackenbush 1.1
- Fall 2004:** Continued research
- Winter 2005:** Formalization of theorems and proofs
- Spring 2005:** Results compiled and documented

Project Software

Early Calculators

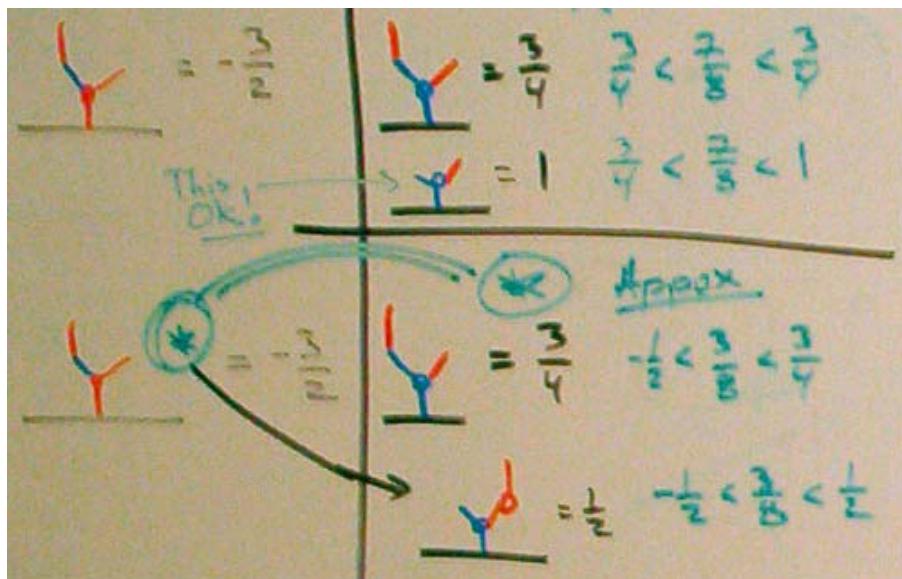
Undoubtedly, the most rewarding part of this project has been the opportunity to see the implementation of the Project Software. When I first signed on to start this project, Professor Haff and I originally spoke of the “intensive algorithmic programming” desired to help in the manifestation of the ideas for which my proposal had potential. Little did I know that it would take over a year to finally develop a useful package to assist in the calculations required for the project to progress.

Before the creation of the current package, there were two programs which were designed to perform simple computations:

- C Stalks Calculator (CSC)
- Hammerian Trees Algorithm

Both were command line driven programs written in C during the summer of 2003 and were used mainly to check the computations I was doing on my whiteboard: an arsenal of 200 Red-Blue Hackenbush Trees.

In December 2003, I had the chance to begin working on my real dream: a Hackenbush Applet, in which the user could draw any desired Hackenbush Tree and retrieve its Combinatorial Game Value. A month later, the graphical user interface had taken form and the computational backbone from the Stalks Algorithm had been properly adapted. However, I will reemphasize that it was far from my ambitions to have a package which allowed the user to draw any tree. In fact, such an achievement would not be fully realized until six months later.



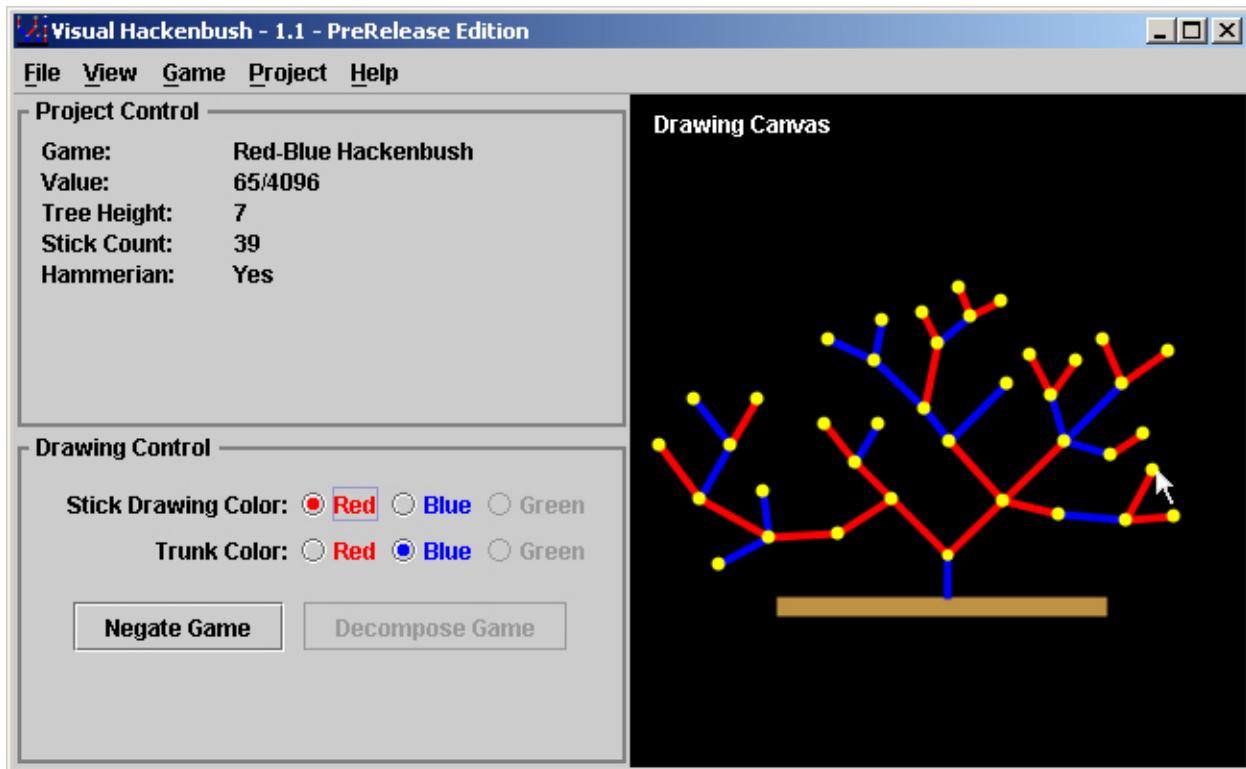
A photo of an early phase in the project.

Visual Hackenbush

During the beginning of the Spring Quarter of 2004 I finally reached the chapter on Hackenbush in *ONAG* (*On Numbers and Games*, Conway). There is in fact a complete algorithm for Red-Blue Hackenbush Trees laid out on page 88 of this text. With this development, the entire project could be refocused. It was no longer about working out a step-by-step process to calculate the Combinatorial Game Value of Red-Blue Hackenbush Trees, rather it became centered about my ideas for Hackenbush Tree classification and decomposition. Hence began the development of Visual Hackenbush.

The Visual Hackenbush package turned out to be everything I had dreamed for—and a whole lot more. Features such as the capabilities to save and open (*.hat) files and a fully operational log panel (not in the original plans) were integrated, enhancing the overall power of the software package.

Version 1.1 marked a huge milestone in this project's development, offering to the user the ability to calculate trees as fast as they could draw them into the interface, limited only by their imaginations within the realm of Red-Blue Hackenbush.



Visual Hackenbush 1.1 – PreRelease Screenshot.

See Appendix B for extensive documentation on Visual Hackenbush.

Introduction to the Project

Assumptions & Conventions

I have written this document to be as clear and precise as possible. So many times have I read papers and been totally confused over a single line simply because of the fancy/ technical wording utilized in describing a fairly intuitive concept and I want to avoid that even at the cost of the inflated length of this document.

I will assume that my readers have a firm understanding of the principles presented in higher mathematics. Mathematical Reasoning is certainly the foundation from which all conclusions of this document derive. However, I do not assume that my readers have any extended experience with such subjects as Real or Complex Analysis.

Furthermore, there is a simple fact that must be faced: Combinatorial Game Theory (CGT) is a branch of mathematics which is still in its infancy in comparison to most others such as Algebra or Combinatorics. As a result, I will not assume that my readers have any experience with CGT. It is highly recommended (for both knowledge and fun) that my readers pick up a copy of *Winning Ways* (WW) as a reference.

Most of what I will be dealing with here falls more along the lines of the heavier theory within the game of Red-Blue Hackenbush. Quite a bit was established in John Conway's *On Numbers and Games*, and so I strive for this document to supplement his thorough pioneer work.

From hereon, the reader should assume that we are dealing strictly with Red-Blue Hackenbush. Therefore if it is not explicitly stated as "Red-Blue Hackenbush," it should still be assumed. I'm positive that there are quite a few things presented in this document which can be applied to other variations of Hackenbush, yet in this document they are neither stated nor implied.

By convention, we refer to our two players as Left and Right and designate them with different genders in hopes to keep track of which one we are referring to throughout their gameplay. Left is female and Right is male. It should also be noted that since this paper is focused solely on the game of Red-Blue Hackenbush that we designate Left to be **bLue** and Right to be **Red**. Furthermore, we say that Left likes positive values and Right likes negative values. A more formal and detailed introduction to these players is certainly in order as we will be playing with them from hereon (see *The Players* section in **Combinatorial Game: Defined**).

Quite a few people have asked me why I sign the end of my proofs with "**GS!**" The answer is that it stands for "**Game Solved!**" since most of the proofs in CGT involve playing out the game in order to prove a conjecture or determine the game value.

Notation & Abbreviations

$\text{CGV}()$	-	Combinatorial Game Value; “game theoretic value”, or just “value” If we need to distinguish this from another type of value, we will do so explicitly.
branches_b	-	the set of all branches from a single branching node which have an initial stick color blue.
branches_r	-	the set of all branches from a single branching node which have an initial stick color red.
CGT	-	Combinatorial Game Theory
FTOH	-	Fundamental Theorem of Hackenbush
SRP	-	Stalk Reduction Principle

Notation Basics

Any game $G = \{ G^L \mid G^R \}$, where $G^L = \{ A, B, C, \dots \}$ and $G^R = \{ D, E, F, \dots \}$.

In other words, for any game G , we have two sets of options, one for Left and one for Right (notated respectively with a $|$ between them). Within each set of options are sub-games in the game tree. When a player takes a turn, they send the game G to one of the sub-games enumerated in their set of options. This sub-game then becomes the current game and the process is repeated. Also, one should think of G^L as a generic symbol that takes on any A, B, C, \dots , and similarly for G^R .

Abuse to Notation: $G = \{ A, B, C, \dots \mid D, E, F, \dots \}$.

I put this in here because the notation used in CGT can get very confusing due to its recursive nature. Above, we can see that we have omitted the set braces around the set of Left options as well as the set of Right options. This is common practice and is used regularly throughout this paper.

Notating Options of Options

We notate the options of a game option in the following manner: $G^L = \{ G^{LL} \mid G^{LR} \}$, since the position G^L is made of options for both players. Similarly, $G^R = \{ G^{RL} \mid G^{RR} \}$.

More Abuse to Notation:

Again, since enumerating the game tree options in CGT can get very wooly very quickly, we introduce another shorthand notation. $G = \{ G^L \mid G^R \}$ can be simply written without the set braces: $G = G^L \mid G^R$. In this paper, I try to stay with the braced notation since it is easier to distinguish the games and their options.

Methods of Proof

There are several methods of proof which we will need to employ in order to analyze Hackenbush and game theory fundamentals. In general they do not differ too much from those used in higher mathematics. I have outlined those with the significant changes below.

Gameset Proof

For the most part, to analyze games in CGT we generalize the game and its options and classify them and play them out until a verdict is reached. This paper utilizes this technique more than any other so it is important to understand what is happening right away. There are some simple, grounding examples given in the **Combinatorial Game Fundamentals** section after a few of the important concepts of CGT have been defined.

Proof by Induction

Due to the recursive nature of games and their notation, it becomes necessary to outline a method for Proof by Induction that is useful within the context of our analysis. Professor Haff outlined this in his text for Math 168A very clearly:

At this point we need the style of induction put forth in ONAG. Let $P(x)$ be a proposition whose truth or falsehood depends upon $x = \{ x^L \mid x^R \}$. Now assume all statements of the kind $P(x^L)$ and $P(x^R)$ are true (this is our inductive hypothesis). If this implies the truth of $P(x)$, then we conclude that P is true for all numbers.

Now, let $P(x, y)$ be a proposition whose truth or falsehood depends on x and y . If the truth of $P(x, y)$ can be inferred from the truth of all statements of the kind:

$P(x, y^L)$ and $P(x, y^R)$ for each fixed x and all values of y^L and y^R , and
 $P(x^L, y)$ and $P(x^R, y)$ for each fixed y and all values of x^L and x^R .

Then we conclude that $P(x, y)$ is true for all numbers x and y .

Proof by Gameplay

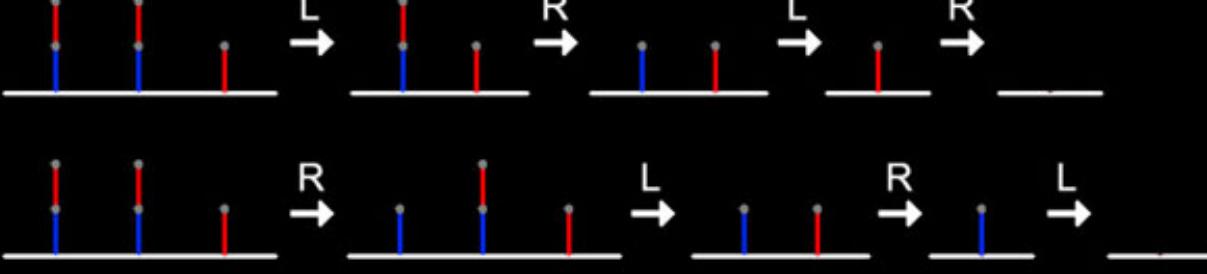
CGT is a very **visually oriented** branch of mathematics. Hence, we will use pictures or a series of pictures to show the gameplay of a game to prove that something is true or false. Moreover, using pictures helps save words since describing a game and its sub-options can become unnecessarily verbose.

The following is a proof to show that the Red-Blue Hackenbush game in which there is a single Blue stick connected to the ground and a single Red stick connected to the Blue stick has a combinatorial game value of $\frac{1}{2}$. It is done by adding a second copy of the game in question as well as a stalk with a value negative of the sum of the copies so that the entire sum of the stalks is equal to zero (later defined as a second player win).

Example a Typical Gameplay Proof in CGT:

Prove that  = $\frac{1}{2}$

Proof:



Since the Second player wins in both cases, the game is a Zero Game (has value = 0), which means that our original stalk is equal to $\frac{1}{2}$.

Proof of the first fractional value in Hackenbush.

Combinatorial Game: Defined

Conditions for Analysis

There is a certain set of game attributes a game must have in order to be classified as a combinatorial game. We require these specifications for the sake of analysis; otherwise we would have to take a lot more information into account such as probabilities and various types of valid endgames. We recognize that this would be so much information, in fact, that it would simply be impossible to complete any in-depth investigations without these limitations. Therefore, the following restrictions/ conditions are necessary to simplify the set of games so that we may carry out our analysis:

Two Players

In essence, Combinatorial Game Theory is study of two-person games. Strategies and methods presented in CGT can certainly be applied to games with more players, but this will obviously complicate the investigation. As you may or may not already know, gameplay structure breaks down into the form of a binary tree (like one would see in a data structures computer science course), which after a few levels of play can quickly get out of hand. Needless to say that adding another player would only make this situation even more difficult to handle.

No Chance

Combinatorial games do not allow dice, the shuffling of cards, or any other devices which lead to the need for probabilities and distributions. Otherwise, the outcome from each turn would be heavily dependent upon the factors of chance rather than the abilities of the players and the nature of the game itself—if we were interested in that, we'd really just be doing statistical analysis and not game theory.

Perfect Information

All combinatorial games require for all game data to be accessible to both players. That is to say that there is nothing hidden from a player's opponent. Everything pertinent to the current game being played is completely laid out on the game board for both players to see.

Turn-Based Gameplay

Players make moves by taking turns one at a time. This ensures that speed is NOT a factor that would also need to be included in our analysis. And surely it prevents the game play from dissolving into complete chaos!

Outcome Condition

In every combinatorial game, there must be an absolute winner: the first player to fulfill the winning condition (described below). This means that there is no possibility for a Tie or a Draw. It also prevents the allowances for player resignations or any other sort of premature game termination.

Victory Condition

In most combinatorial games, the winning condition is simple: the last player to make a valid move wins. However, there is another side of combinatorial game theory based around "Misère" play, in which the opposite condition is set in place. We will not address "Misère" play since it requires a completely different strategy, which, in fact, is much harder to analyze.

Rules of Play

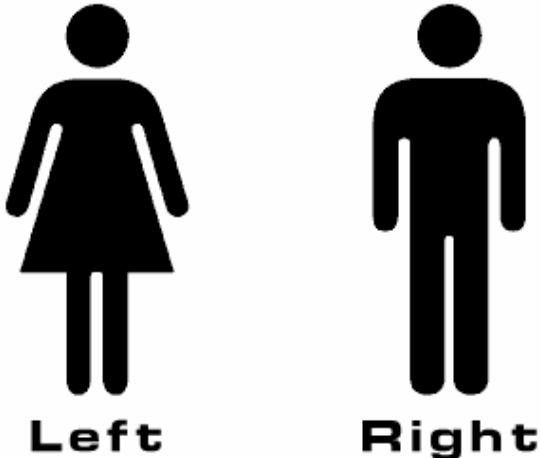
Below I have included a few other important pieces of information with an emphasis on parts that usually lead to confusion.

- We cannot quantify a game unless we know what is in it for both players since a value of a game position is INDEPENDENT of the order of players taking turns (first or second). In other words, *all* the options for *both* players must be enumerated in order to quantify the game.
- "_____ player CAN win" means that there is a strategy for that player to be victorious despite their opponent's moves.
- The names "First" and "Second" are NOT a part of the game analysis; that is, it is not given in the rules or positions of the game. These are adjectives we append to the players *once we begin to play and analyze*.
- Remember that we only make value judgments upon the *optimal* moves.

The Players

Introducing Left and Right

I'd like to introduce you to our two players: Left and Right.



Genders

As mentioned before, Left is female and Right is male for clarity in reference. So when speaking in terms of “she” and “her”, we are most certainly referring to Left. Likewise, when using the terms “he” and “him”, we are referring to Right.

Colors

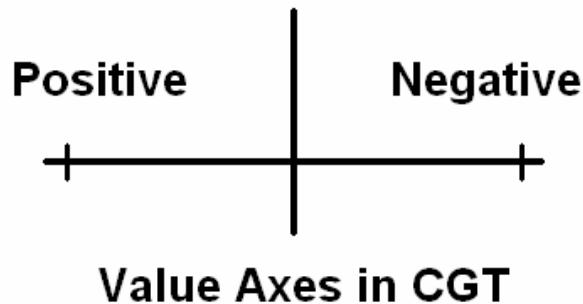
In the game of Red-Blue Hackenbush, we utilize colors to also help distinguish whose stick is whose. As a convention, we designate Left to be **bLue** and Right to be **Red**. Keep in mind that the colors also become nicknames for our players so it is not unusual to see a phrase like “when Blue makes her move” as a reference to Left inside our proofs and analysis.

Values

One of the core components to our analysis focuses on assigning game theoretic values to Hackenbush games. Both *ONAG* and *Winning Ways* jump right into assigning and using values without much of an introductory discourse, so we will try our best to address this here.

To start, we say that Left likes positive values and Right likes negative values. That is, a game with a positive value indicates that Left can win; similarly, a game with a negative value indicates that Right can win.

This leaves only infinitesimals and zero. As we will soon prove, all Red-Blue Hackenbush games evaluate to numbers, so we will not have to take infinitesimals into consideration at all in this paper even though they are a fundamental component to Combinatorial Game Theory. The value zero is special and will be defined as a basis for many other definitions including the definition of a number. This investigation is featured in the next section.



Player Summary

<u>LEFT</u>	<u>RIGHT</u>
Female	Male
Blue	Red
Positive	Negative

This concludes our introduction to the conventions and rules of play in Combinatorial Games. Now we embark into the more mathematical side of CGT starting with some fundamental definitions, concepts, and proofs. Once we have completed this groundwork, we can begin to analyze the game of Hackenbush!

Combinatorial Game Theory Fundamentals

Definition of Zero

Any game G in which the second player can always win has a game theoretic value of 0. This is called a zero game.

More explicitly: no matter who plays first, Left or Right, the second player will always have a winning strategy. This does not imply that the game ends after the second move of the game; however, it does imply that if the second player makes no mistakes, they are guaranteed to be the player who makes the final move of the game.

Definition of a Number

A game G is a number if all options of G (for both Left and Right) are numbers, and no left option is greater than or equal to any right option.

It should be noted that there are all sorts of games out there which are not numbers; however, we shall prove later that all games of Red-Blue Hackenbush have a numerical game theoretic value. (See the FTOH).

Zero is a Number

The game $G = \{ | \} = 0$ is a number.

Proof:

On the contrary, assume G is not a number. Then some option of G is not a number or some left option of G is greater than or equal to some right option. But in the first instance we have a contradiction because G has no options. The second one is also contradictory. We can have no inequalities of this kind because G has no options. Thus G is a number.

GS!

Option-Value Relation for a Number

For any number $x = \{x^L \mid x^R\}$ we have $x^L < x < x^R$.

Proof:

To show that $x^L < x$, we first look at $x^L \not\geq x$, which is to say: $x^L + \{-x^R \mid -x^L\} \not\geq 0$. If Right plays first in $x^L + \{-x^R \mid -x^L\}$ then he wins since he will send the difference to $x^L + (-x^L) = 0$ (so Left loses). Then, we note that $x^L \not\geq x$ is equivalent to $x^L < x$. Thus, $x^L < x$.

To show that $x < x^R$, we first look at $x^R \not\leq x$, which is to say: $x^R + \{-x^L \mid -x^R\} \not\leq 0$. If Left plays first in $x^R + \{-x^L \mid -x^R\}$ then she wins since she will send the difference to $x^R + (-x^R) = 0$ (so Right loses). Then, we note that $x^R \not\leq x$ is equivalent to $x < x^R$. Thus, $x < x^R$.

GS!

Simplicity Rule

Suppose for $x = \{x^L \mid x^R\}$ that some number z satisfies $x^L < z < x^R$ for all x^L and x^R , but no option of z satisfies the same condition. Then $x = z$.

Prove $x \geq z$:

For the sake of contradiction, assume that $x < z$ so that $x - z < 0$:

$$\{x^L \mid x^R\} + \{-z^R \mid -z^L\} < 0$$

This means that Right has a winning move in one of the two components x^R or $-z^L$. So either $x^R + (-z) < 0$ for some x^R OR $x + (-z^L) < 0$ for some $-z^L$. Keeping in mind that x and z are in canonical form, we can conclude that $x^R - z$ cannot be less than zero since $x < z$. Then Right must have a winning strategy in the move to $-z^L$:

$$x + (-z^L) < 0, \text{ that is to say: } x < z^L.$$

But now $x^L < x < z^L < z < x^R$ which yields $x^L < z^L < x^R$, a contradiction since no option of z satisfies these inequalities.

Prove $x \leq z$:

Now assume for the sake of contradiction that $x > z$ so that $x - z > 0$:

$$\{x^L \mid x^R\} + \{-z^R \mid -z^L\} > 0$$

This means that Left has a winning move in one of the two components x^L or $-z^R$. So either $x^L + (-z) > 0$ for some x^L OR $x + (-z^R) > 0$ for some $-z^R$. Keeping in mind that x and z are in canonical form, we can conclude that $x^L - z$ cannot be greater than zero since $x > z$. Then Left must have a winning strategy in the move to $-z^R$:

$$x - z^R > 0, \text{ that is to say: } z^R < x.$$

But now $x^L < z < z^R < x < x^R$ which yields $x^L < z^R < x^R$, a contradiction since no option of z satisfies these inequalities.

GS!

"The most important game-theoretical property of numbers is that given by the simplicity rule: if all the options G^L and G^R of some game G are known to be numbers, and each G^L is strictly less than each G^R , then G is itself a number, namely the simplest number x greater than every G^L and less than every G^R ."

— Theorem 11, Chapter 2, ONAG

Hackenbush Definitions

Hackenbush Stick

A Hackenbush Stick (we will just say “stick” from hereon) is the base component in any Red-Blue Hackenbush game. In game play, each stick counts as a move for either player provided it is their color. **NOTE:** a “move” or “turn” is not the same as a value.

For our analysis, we must clarify that a Hackenbush Stick is composed of the following:

Color:	[Red, Blue]	(one or the other)
Source:	a set of sticks from which this stick stems	(cannot be empty)
Branches:	a set of sticks which stem from this stick	(can be empty)

Hackenbush Trunk

A Hackenbush Trunk (we will just say “trunk” from hereon) is a special type of stick whose source is the ground of the game rather than another stick.

Therefore, by definition, a trunk is composed of the following:

Color:	[Red, Blue]	(one or the other)
Source:	the ground	
Branches:	a set of sticks which stem from this stick	(can be empty)

We also note that the color of the trunk determines the sign of the entire game extending from this trunk. (Proof – derives from the FTOH)

Hackenbush Tree

A Hackenbush Tree (we will just say “tree” from hereon) is a Red-Blue Hackenbush game in which there exists only one connecting stick to the ground (trunk) and each stick has a maximum of one source (parent) stick, excluding the trunk whose parent is the ground.

Hackenbush Stalk

A Hackenbush Stalk (we will just say “stalk” from hereon) is a special type of tree in which there may only be a maximum of one branching (child) stick.

Instance of Branching

We say that there is an instance of branching when the number of sticks stemming from a source is greater than one.

It should be noted that a Hackenbush Stalk **cannot** have *any* instances of branching.

Trunk Stalk

A Hackenbush Trunk Stalk (we will just say “trunk stalk” from hereon) is the *set* of sticks between the ground and the first instance of branching within a Hackenbush Tree.

Important Note:

This is not to be confused with the trunk itself. Even though the trunk is a part of the trunk stalk, the trunk itself is significant within and of itself.

Purity

We say that a Hackenbush Stalk is “pure” if it is completely composed of sticks of the same color.

Important Note:

Recall that these definitions (like all games in CGT) are recursive in nature, meaning that a stalk may be pure up to a certain stick—which makes analysis easier to split the stalk into the game below and the game above.

Look at sticks/ stalks/ trees which stem from any stick as a game itself.

Impediment

An impediment is the first stick to break a line of purity in a Hackenbush Stalk. In other words, it is the first stick with a color differing from the color of the pure stalk.

As a result:

A stalk with an impediment will always have a combinatorial game value less than a pure stalk of the same height (which is to say: has the same number of sticks).

Highest Stick

The “highest” stick is that which is the furthest from the ground by way of the number of sticks along the path connecting it to the ground.

Rules of Red-Blue Hackenbush

Whose Game?

Players

You already met the players in a previous section which explained all the conventions we may use with them in our analysis, but I have outlined them here for completeness.

Left = Blue

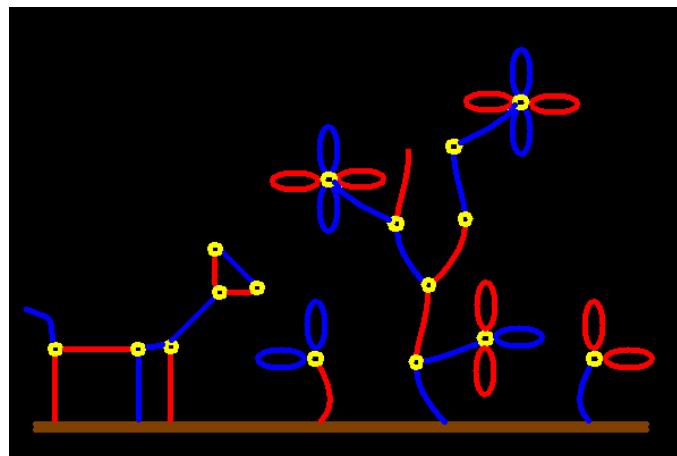
Right = Red

Hackenbush Gameplay

Game Format

The game board is created by arranging any sort of combination of red and blue sticks that extend from the *ground* or another stick. Before playing, the players must decide who will play Blue (Left) and who will play Red (Right) as well as who is to make the first move—it should be noted that these decisions do not have any affect upon the computations that are to follow.

A typical “game board” is similar to the figure below:



Hackenbush is a turn-based game such that each player takes one move at a time until one of the players can no longer move (that is to say that they are defeated).

Taking a Turn

A player moves by “hacking” a single stick from the game board of their own designated color. Remember that if any stick becomes disconnected from the ground, it too must be removed from the game board.

Winning the Game

A player wins the game when their opponent has no more moves remaining. While at first this may seem arbitrary and only dependent upon the number of sticks each player has, we reemphasize that every stick must be connected to the ground to still be playable and thus allows players to eliminate their opponent’s moves from the game board as well.

Hackenbush is Hard

“HACKENBUSH IS HARD!”

— *Chapter 7: Hackenbush, Winning Ways*

Conway et al. confess in Chapter 7 of *Winning Ways* that Red-Blue Hackenbush is difficult. “Although the values are ordinary numbers, it may be hard to work out exactly which ordinary number is the answer” (*Winning Ways* 211).

They go on to explain that from a “good” algorithm to determine the values of Red-Blue Hackenbush trees one could derive a “good” algorithm for finding the minimum spanning tree of a bipartite graph. They also argue that if such an algorithm for finding the values of a Red-Blue Hackenbush tree had a polynomial runtime, then the aforementioned algorithm for finding the minimum spanning tree could also be bounded by a polynomial runtime function. However, it has been previously determined that the algorithm for finding the minimum spanning tree is NP-complete. Therefore, the problem for evaluating Red-Blue Hackenbush positions is NP-hard. (*Winning Ways* 224)

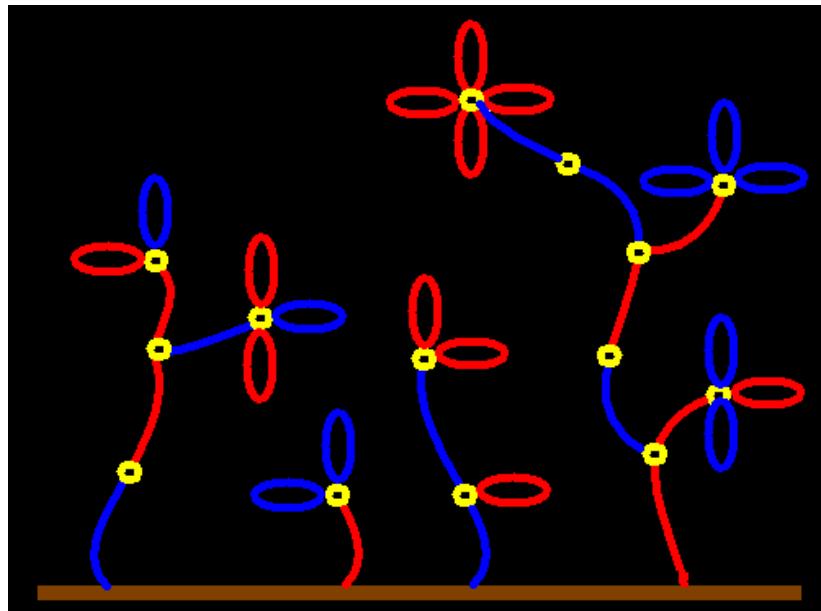
To be sure, it should not be mistaken that I have found such an algorithm—nevertheless I will reveal some new tactics to breaking down these problems into smaller, simpler ones. But first let’s take a look at some theorems which directly involve Hackenbush.

Hackenbush Theorems

Section A

Here we prove the **Fundamental Theorem of Hackenbush (FTOH)**. All other theorems stem from this foundation. The theorem is found on page 88 of *On Numbers and Games*, yet the proof is supplied from the course notes of Math 168A. It should be noted that nowhere have I found a “true” title to this theorem, yet its importance demands one for both significance and ease of reference. I have dubbed it the **Fundamental Theorem of Hackenbush** for those exact reasons.

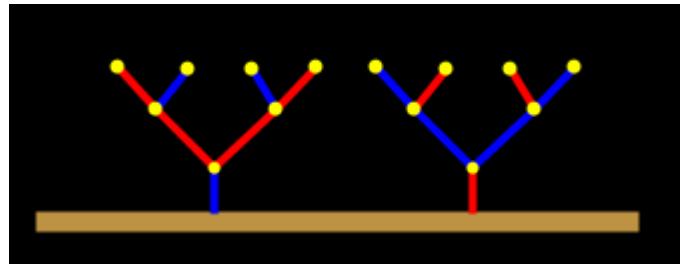
However, before we can properly prove the FTOH we must introduce a strategy of gameplay called “Tweedle-Dum and Tweedle-Dee” after the characters from Lewis Carroll’s tales of Alice in *Through the Looking Glass*. This strategy will serve as a Lemma to the **Fundamental Theorem of Hackenbush**.



A Typical Red-Blue Hackenbush Game.

Tweedle-Dum and Tweedle-Dee Strategy

In select game positions in which the moves of either player are matched with equivalent options, one should reply to their opponent's moves with an equal and corresponding move. Such a strategy is called Tweedle-Dum and Tweedle-Dee.



All right, I have to admit that there's not a whole lot of proof going on here, but the Tweedle-Dum and Tweedle-Dee Strategy shows its use many times throughout our analysis. Basically, the Tweedle Strategy is a method for keeping control throughout gameplay. If your opponent does one thing, you mirror their move until they no longer have any options, leaving you as the victor.

Employing the Tweedle-Dum and Tweedle-Dee Strategy guarantees a second player win by the nature of the method of gameplay. It should be clear that the players will play out the game and whoever began mirroring their opponent's moves will *always* have a move to respond with. Hence, it is the opponent who will run out of options first.

Important Note:

Tweedle-Dum and Tweedle-Dee cannot just be arbitrarily applied at any point in any game. The initial condition of a game with matching and equivalent options for both players must be met exactly.

CGT Note:

The definition given above is sufficient for our purposes. However, in the scope of all combinatorial games there are two classifications: partisan games and impartial games. Hackenbush is a partisan game and the Tweedle Strategy above is worded to hold tightly to partisan games. There is a more general way of expressing Tweedle-Dum and Tweedle-Dee so that it encompasses both classes of games but there is no need to address it here.

Fundamental Theorem of Hackenbush

- (i) When a blue edge in a Red-Blue Hackenbush diagram is erased, the value strictly decreases. Similarly, when a red edge is erased, it strictly increases.
- (ii) Every Red-Blue Hackenbush diagram is equal to a number.

Proof:

Suppose we start with a Hackenbush game position $G = \{G^L \mid G^R\}$ that has a value x . Now, suppose some blue stick b is erased, and denote the resulting game by G^L and its corresponding value by x^L . We need to argue that the value of the new position G^L is less than the value of original position G . To do this we set up a difference game D and show that Left can always win; mathematically: $D = G - G^L > 0$.

Left plays first:

If Left plays first in D , then she will play in the G component and send it to G^L which renders the difference game value as zero.

Right plays first:

If Right is first, then there are two possibilities:

1. Remove any red stick connected to the ground via a path of sticks which includes the stick b ; or
2. Remove a red stick whose connection to the ground is independent of the stick b .

In the first case, Left will respond by removing b , which brings the difference game to $G^L - G^L = 0$. In the second case, she will respond with a Tweedle-Dum and Tweedle-Dee Strategy.

In either case, Left can win.

If some red stick r is removed from the starting position G , then denote the result by G^R and its corresponding value by x^R . From symmetry, we argue that the value of the new position G^R is greater than the value of original position G . To do this we set up a difference game D and show that Right can always win; mathematically: $D = G - G^R < 0$.

In conclusion, we have $x^R > x$ and $x > x^L$ which implies $x^R > x^L$. This last inequality implies that x is a number.

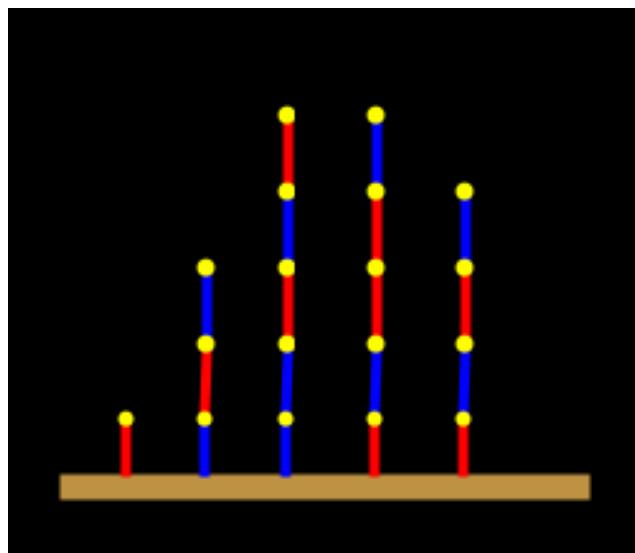
GS!

Section B

Here we first prove the **Stalk Reduction Principle** as a basis for the rest of the corollaries to follow.

The **Stalk Reduction Principle** is quite an intuitive concept and can be easily understood by the “common sense” realization that if a player has a stick further away from the ground than the stick they are choosing to hack, they are essentially squandering their resources (i.e.: wasting a playable move) and is therefore detrimental since we hold to the rule that the last player to make a move wins the game.

However, when applied to Hackenbush Trees in general, this concept loses its intuitiveness, yet grows to become an even more powerful tool. Its case tree grows exponentially with each new level of branching. Therefore, the method of proof adopted is similar to that which was presented in ONAG. Refer back to the section on *Methods of Proof* for a more general discussion.



A Set of Red-Blue Hackenbush Stalks.

Stalk Reduction Principle

For any Red-Blue Hackenbush Stalk, the “best” move for either player is always the highest valid move in the stalk.

Proof:

Suppose we start with a Hackenbush Stalk $S = \{S^L \mid S^R\}$ that has a value x . Now suppose that Left removes the highest blue stick in the stalk, resulting in the position S^L .

We now apply the **Fundamental Theorem of Hackenbush** where the original game position is given by $G = S^L$. This implies that: $G = \{G^L \mid G^R\} = \{S^{LL} \mid S^{LR}\}$. It follows that the value of G is greater than the value of G^L . In consequence, $S^L > S^{LL}$. More explicitly, the position S^L has a greater game theoretic value than any other position S^{LL} to which Left could send the original game S .

Similarly, by symmetry we reason that $S^R < S^{RR}$, where S^R is the position to which Right sends the original game S by removing the highest red stick.

GS!

Corollary 1

There is no good reason to cut a stick in the “trunk stalk” if a player can make a cut elsewhere (that is, in a branch somewhere else in the tree).

Game Setup:

Let G be a Red-Blue Hackenbush Tree. In game notation, $G = \{A, B \dots | X, Y \dots\}$. In G , there are two “best” moves (one in each set of options):

G^L for Left and G^R for Right. Therefore, $G = \{G^L | G^R\}$.

WTP:

Let T be the best option for Left, wherein Left hacks a blue stick in the trunk (it should be clear from the Stalk Reduction Principle that “best” simply means the highest blue stick in the trunk stalk). Let S be the option for Left, wherein Left hacks a blue stick not in the trunk (i.e.: somewhere in a branch higher in the tree). We wish to prove that the best move for Left cannot lie in the trunk stalk, provided there exists a different option in which Left does not hack a stick in the trunk stalk. Mathematically, $G^L \neq T$, if $\exists S$. Then, similarly for G^R .

Proof:

By the Stalk Reduction Principle, an option that is the result of hacking a stick in the trunk stalk cannot be greater than an option for which Left hacks a stick in a branch somewhere else in the tree. Therefore, since Left has an option that will result in a more positive game value in a branch somewhere else in the tree, she will not hack a stick in the trunk stalk.

Similarly, by the Stalk Reduction Principle, an option that is the result of hacking a stick in the trunk stalk cannot be less than an option for which Right hacks a stick in a branch somewhere else in the tree. Therefore, since Right has an option that will result in a more negative game value in a branch somewhere else in the tree, he will not hack a stick in the trunk stalk.

Hence, we have shown for both players that there is no good reason to cut a stick in the trunk stalk if they can make a cut elsewhere in the game.

GS!

Combinatorial Game Values for Hackenbush

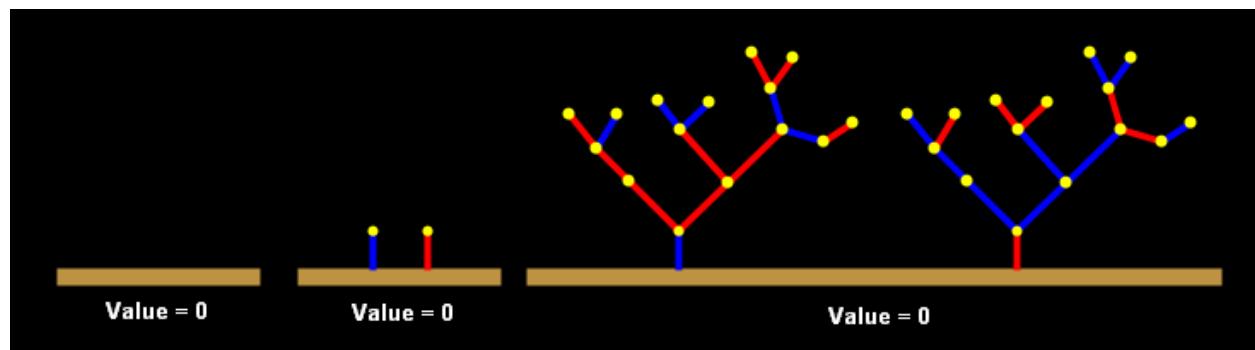
Game Theoretic Values

Until now, we have continued to speak of and use values of games in an abstract manner. But games are not abstract in practice—after all, all of this theory is supposed to help you play a better game of Hackenbush! In this section we will focus on concrete game theoretic values for Hackenbush to supplement the theory and proofs in the previous sections and provide a basis for the discourse to follow.

This section contains the foundational methods for evaluating game values upon which CGT is taught. It strives to show how we can attain a correspondence between games and their values. The subject of Numbers as Conway would describe is beyond the scope of this paper and should be sought in ONAG's *Zeroth Part: On Numbers*. There, Conway proves his Numbers are a field and shows many other properties in detail. For our purposes, we will attempt to highlight only the facets needed for the analysis to come and concentrate on the derived gameplay.

Zero

Already, we have been introduced to a very important game value: zero. A zero game was defined as a Second player win. Consequently, there are infinitely many different game representations for the value zero in Hackenbush. A few of these games are shown below:



Each of the games above is a Second player win. Let's examine each of them:

In the first case, we can determine that whoever goes first does not have a move to make and therefore loses the game.

In the second example, if Left goes first and takes the Blue stick, then Right will respond by taking the Red stick. Now it is Left's turn, but there are no more Blue sticks! Right (who went Second) wins. When we examine the game where Right goes first, we find that Left is the victor by a similar series of events. Thus, after examining **both** gameplay paths, we have determined that the Second player will always win. Therefore the second game is a zero game.

The third game looks far more complex, but there is a shortcut! As you may have already noticed, the secret to creating (and winning!) a zero game lies in the strategy we labeled Tweedle-Dum and Tweedle-Dee. Hence, we can say that by Tweedle-Dum and Tweedle-Dee, the third example is a Second player win.

Before moving on let's take another look at the second example. Notice that during the course of the game (at the end) the players have played the game out to be exactly the game in the first example. This is what we mean by "sending" one game to another. The players have played out moves from one game state to one of its sub-games, which, in turn, becomes the current game to be played.

Integer-Valued Games

Let's start defining some other numbers for Hackenbush games. It seems reasonable to evaluate game positions in terms of the number of moves a player can make. For example, if there is one Blue stick then this is worth one move for Left; we could argue the same for a single Red stick for Right.

If we recall the axes presented when we introduced the Players, we said that Left is favored by positive numbers and Right is favored by negative numbers. Therefore, it stands to reason that we may define a single Blue stick as having a value of positive one; and a single Red stick as having a value of negative one. Furthermore, this coincides with our previous example where a single Blue stick and a single Red stick made up a game we determined to be a Second player win. It must follow:

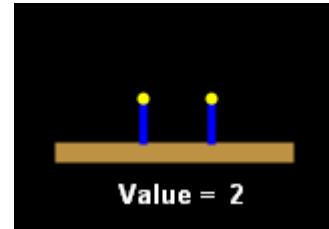
$$1 + (-1) = 0.$$

Naturally, such a statement comes at no surprise.

Continuing further into the integers, we ask the question of what does a pair of Blue sticks evaluate to?

Intuitively, this game has two moves for Left and no moves for Right, which by the same reasoning as above, should evaluate to positive two. Again, this follows mathematically:

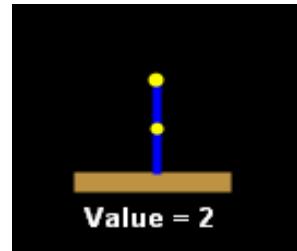
$$1 + 1 = 2.$$



This also seems quite natural and should come at no surprise.

However, let us investigate what happens when we stack the two sticks on top of each other so that only one is attached to the ground.

Once again, this game has two moves for Left and no moves for Right, which by the same reasoning as above, should evaluate to positive two. But wait! Certainly, the stalk shown here and the pair of stalks in the previous example cannot be the same game!



True. They are not the same game, yet they still have the same value. This makes sense since there are still two moves available for Left. Obviously, if Left chose to hack the trunk of the stalk then she would be squandering an available resource in the upper stick since this too is hacked with the bottom stick. By cutting the top stick, Left still has a stick remaining and is victorious, winning with a remaining value of positive one. Such a victory is better than winning without any left over resources and is therefore more desirable, making the top stick the *optimal move*.

Since we play with optimal players when performing our analysis, we will only consider the optimal moves. This makes sense in our notation too. So far we have ignored our notation throughout this discussion on values so that we might concentrate on the intuitive aspect of how these games and numerical values correspond.

Notation and Values

After our definition of a zero game, we proved that zero was a number and labeled it with the following gameset notation:

$$0 = \{ | \},$$

which says that there are no options for Left and that there are no options for Right. This corresponds exactly to the first representation of a zero game we examined, but what about the others? They certainly have options!

In fact, if we apply our “common-sense” evaluations to the notation, we find that we get the exact same end result! In the case of the second zero game example, we have one move for Left valued at negative one and we have one move for Right valued at positive one. To clarify, remember that we are talking about the sub-game values which are the game options produced after a player has taken their turn. Mathematically:

$$\{-1 | 1\} = 0.$$

Our result is a number since all the options are numbers and no Left option is greater than any of the Right options and the actual value is a product of the Simplicity Rule, through which we would say that 0 is the simplest number between the Left and Right options -1 and 1, respectively. Thus, the statement $\{-1 | 1\} = 0$ is true and follows from our definitions and notation properly.

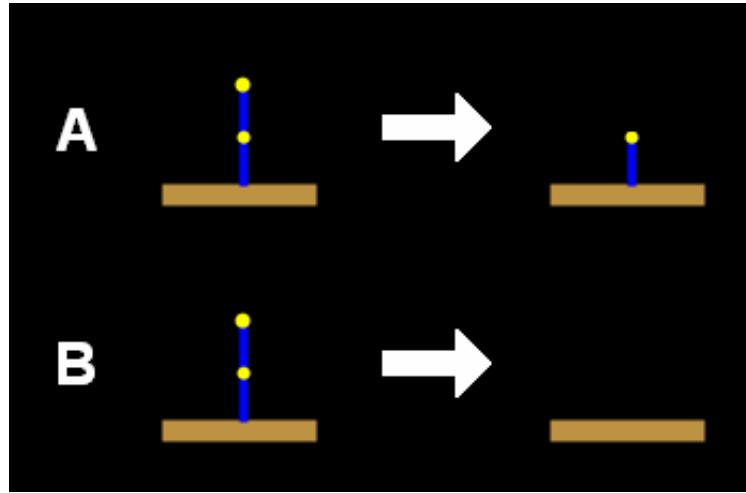
When we employ the gameset notation on the pair of Blue sticks, which we defined as having the value of positive two, we obtain the following gameplay breakdown:

Left: Left can take one of the Blue sticks (which one does not matter since the result is identical in either case), resulting in a sub-game with a single Blue stick, which we determined as having the value positive one.

Right: Right has no move.

The gameset options are filled as follows: $G = \{1 | \} = 2.$

Now we will apply this same process to our stacked example. In this game, we have two options for Left:



- A. Left takes the top stick, resulting in a sub-game with a single Blue stick, which we determined to be valued at positive one. [*optimal*]
- B. Left takes the trunk, resulting in the sub-game with no sticks, which we determined to be a zero game.

In our gameset notation, we write: $\mathbf{G} = \{1, 0 \mid \}$.

We know that 0 is not an option that Left would choose in the light of her other option 1. Therefore we may reduce the representation to:

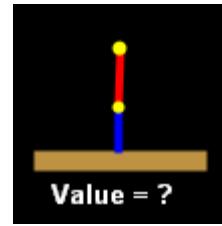
$$\mathbf{G} = \{1 \mid \},$$

which is the exact same representation we obtained for a game of value positive two! Hence, our intuition has served us right in our previous explanation.

Fractional-Valued Games

Now we ask a much harder question to evaluate by mere “common-sense.” What is the value of a game with a Blue trunk and a Red stick stacked on top of it?

Even though the title may have already given it away, let's reason it out before looking at the proof.



The gameplay is as follows:

Left: Left can take the Blue stick, which is the trunk of the stalk, resulting in a sub-game with no sticks, which is a zero game.

Right: Right can take the Red stick at the top of the stalk, resulting in a sub-game with a single Blue stick, which has a value of positive one.

We fill in the gameset options and obtain: $\mathbf{G} = \{0 | 1\}$.

With some further analysis, we can determine that this strange creature is in fact a number and by the Simplicity Rule we may determine that the value of G must be the simplest number between the Left option, zero, and the Right option, positive one. As of yet, we have not given any parameters on how to determine this “simplest” number, we have only proven that such a number exists.

For our purposes we look no further than the dyadic rational numbers. There is far more significance and theory behind why this is involving the birthdays of numbers and the ordering given to numbers by their birthdays. Since we will not be utilizing the concept of birthdays beyond this point, this further discussion is omitted but can be referenced in all three of the combinatorial game texts referenced (for a clear overview, I highly recommend reading Haff's text first: pages 30-55).

In general, we say that the smaller a number's denominator is, the simpler it is. This allows us to classify the integers as the simplest of numbers. We then proceed into the dyadic rational numbers (rational numbers with denominators that are powers of two) and determine that $\frac{n}{2}$ family is next in line, followed by the $\frac{n}{4}$ family, etc.

Proceeding along this path, we can now quantify our game in question with the simplest number between 0 and 1: $\frac{1}{2}$. Therefore:

$$\mathbf{G} = \{0 | 1\} = \frac{1}{2}.$$

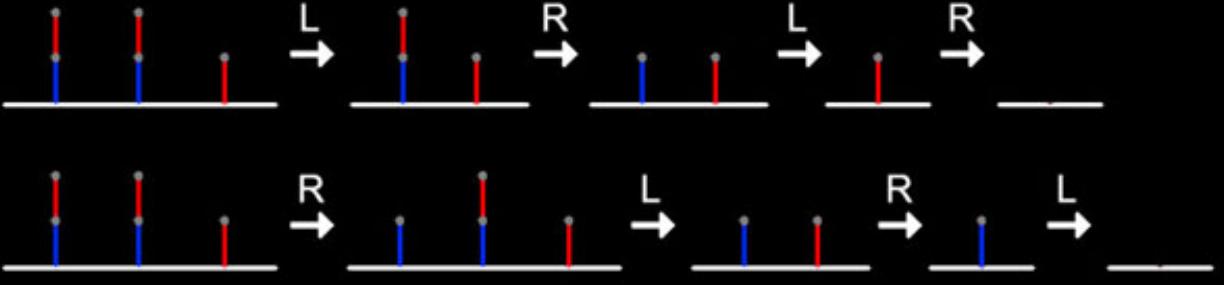
Maybe this is starting to sound a little suspicious without any proof. Not to worry! What follows is a proof that our game in question is in fact equivalent to the value of half of a move. To do this, we simply play out the sum of games which we will hypothesize to be a zero game. In our case, we wish to construct a game that includes two copies of the game in question and a lone Red stick to send the value back to zero. Mathematically,

$$\frac{1}{2} + \frac{1}{2} + (-1) = 0.$$

We certainly hope so! Here's the proof:

Prove that  = $\frac{1}{2}$

Proof:



Since the Second player wins in both cases, the game is a Zero Game (has value = 0), which means that our original stalk is equal to $\frac{1}{2}$.

Fantastic!

Beyond Values

There is much, much more beyond these simple value calculations that we have yet to touch upon. Considering the ground we have covered thusfar, we can calculate simple constructions in Hackenbush, yet must do so using the method of proof shown above which requires a complete delineation of the gameplay in order to attain the results. What if the Hackenbush game was a stalk 17 sticks high? Or even higher!? Such a fundamental problem must have a more general solution, and that is what we will look into next: the calculations of Hackenbush stalks.

Hackenbush Stalks

Significance of Stalks

Stalks are simple. They range from a single stick to a tall series of sticks stacked as high as the clouds, and come in various color patterns. More importantly, stalks are easy to calculate values for and are a good starting point for introducing the concept of evaluating Hackenbush games in a more general way than the methods shown in the previous section.

Notably, the primary goal behind this project was to determine if there was a method by which Hackenbush trees could be decomposed into a group of stalks. If possible, complicated trees could be quickly calculated by a much simpler means.

Evaluating Stalks

The first step toward calculating the values of complicated Hackenbush Tree structures is to determine how to calculate a simple case/type of tree. As alluded to earlier, a stalk is a special type of tree: a tree with no branching. This is an important factor and will immensely simplify the process required to calculate stalks in general.

It should be noted that there are a couple different ways by which to calculate the value of a stalk in Red-Blue Hackenbush. For a method utilizing binary strings to represent the sticks of the stalk, refer to Elwyn Berlekamp's Rule for Hackenbush Strings in *Winning Ways* (77-78). Another method, presented by Thea van Roode, says to start from the trunk and give each edge a value of 1 or -1 (depending upon the color) until there is a color change. Then halve each succeeding value and change the sign according to the color (WW 78). We shall examine van Roode's method in the algorithm below since it was the one I implemented in CSC.

Stalks Algorithm

Here we examine a complete block of pseudo-code outlining van Roode's method for calculating the values of Red-Blue Hackenbush Stalks.

- 1) Start at the base stick (trunk) and work upwards:
 - If trunk is red: Value = -1,
 - If trunk is blue: Value = 1.
- 2) Proceed to the next stick above and evaluate the following:
 - Repeat while the stick color is same as trunk
 - If trunk is red: Value = Value -1
 - If trunk is blue: Value = Value + 1
 - If no more sticks: DONE
- 3) Now that we have hit a change in color, start halving and adding according to color:
 - Loop until there are no more sticks to evaluate
 - If trunk is red: $\text{Value} = \left(\frac{2 \cdot \text{num} - 1}{2 \cdot \text{den}} \right)$,
 - If trunk is blue: $\text{Value} = \left(\frac{2 \cdot \text{num} + 1}{2 \cdot \text{den}} \right)$,

where *num* is the numerator of the current dyadic rational and where *den* is the denominator of the current dyadic rational.

If we look closer at Step 3 and break the formula up, the process seems clearer:

$$\text{Value}_{\text{red trunk}} = \frac{2 \cdot \text{num} - 1}{2 \cdot \text{den}} = \frac{\text{num}}{\text{den}} - \frac{1}{2 \cdot \text{den}}.$$

$$\text{Value}_{\text{blue trunk}} = \frac{2 \cdot \text{num} + 1}{2 \cdot \text{den}} = \frac{\text{num}}{\text{den}} + \frac{1}{2 \cdot \text{den}}.$$

Intuitively, this makes sense since if we add a stick of a color different than the trunk of the stalk the value should diminish (for the player who owns the trunk) since the opponent now has another move that they can make (we saw this in depth when calculating the value $\frac{1}{2}$). However, this extra move does not alter who will win the stalk game. In other words, by appending another stick to the stalk, the signage of the stalk game value will not change. We can argue this mathematically since we are continually halving the values we are appending to our running sum, which is similar to the

argument that $\sum_{i=1}^n \frac{1}{2^i}$ does not equal 1 for any finite number n.

As an example, if we started with a Blue stick and then appended any finite number of red sticks above it, the value of the game will stay positive even though the value of the game will become smaller with each addition of a move for Red.

This observation is truly significant and puts a lot of value (pun intended) into the trunk of the Hackenbush stalk (or tree). Of course, in the algorithm above, each step is entirely dependant upon the color of the trunk of the stalk.

Implementation

During the summer of 2003, I implemented the van Roode's algorithm in C and used it as a mechanism to check my work as I began to search for the parameters for decomposing a general Red-Blue Hackenbush Tree. I originally chose van Roode's algorithm over Berlekamp's Rule because I more naturally think in fractions rather than in binary and van Roode's algorithm was much closer to how I would formulate solutions to Hackenbush problems in general. It later became clear that van Roode's algorithm provided support for computing the values in a fractional form, which made it even more appealing since we almost never speak of Hackenbush games in decimal values in CGT.

```
Start from the bottom and work your way up
To EXIT: type "x" or "exit"

Game #01:    !rrbr
Value: -7/ 4
Game Value = -1.750000

Game #02:    !_
```

The program is called C Stalks Calculator (CSC). However, it is no longer available for download from my CGT site since I have implemented a Java applet which is interfaced with a graphically-based drawing panel (precursor to Visual Hackenbush) rather than CSC's command-line interface.

If you wish to use the applet, please visit: <http://cgt.calculusfairy.com/Software/> Then again, development for the applet has currently been suspended in the interest of the more complete tool: Visual Hackenbush, which can be downloaded for installation at the same URL. Stalks can be calculated in the current release of Visual Hackenbush through a graphical interface similar to that of the applet's, and the process is well documented in supplied User's Guide.

Hackenbush Trees

Trouble with Trees

The trouble with trees is that they have at least one instance of branching. When examining stalks, we were guaranteed that only a single stick would stem from our current position as we made our way up the stalk. This constraint simplified the calculation of the value of the stalk. In this section we remove this restriction and find how hard it actually is to calculate the values for these trees without any guidance. We will then utilize a technique presented in *ONAG* as the key component to Conway's complete theory for trees.

Before we reveal this key component, I would like to draw your attention to how extensive a game tree may become. In Appendix A, I have included a complete game tree to a sum of two Red-Blue Hackenbush trees. It should not come to a surprise that in order to arrive at such a solution takes a good amount of time and effort. What's more is that the question prompts for just gameplay and a winner without calculation, which would add another layer of complexity to the final solution, requiring the calculation of all of the intermediate values to build up the lists of options and their sub-options, and so on and so forth.

Thankfully, there is another way which is paved with the concept of Ordinal Sums.

Ordinal Sums

For a real number x , we say that the number $1:x$ has the first value from the sequence:

$$\frac{x+n}{2^{n-1}} = \frac{x+1}{1}, \frac{x+2}{2}, \frac{x+3}{4}, \frac{x+4}{8}, \frac{x+5}{16}, \dots$$

for which the numerator ($x + n$) is greater than 1. We call such a number $1:x$ the Ordinal Sum of 1 and x .

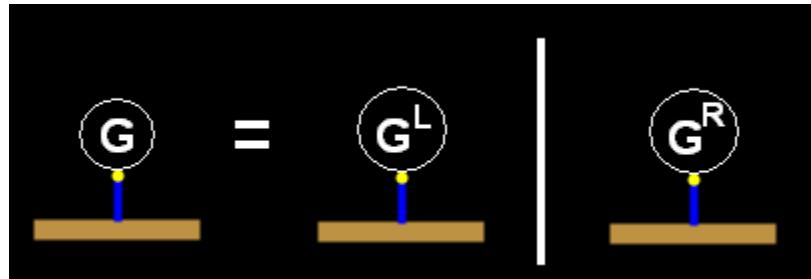
Similarly, for a real number x , we say that the number $(-1):x$ has the first value from the sequence:

$$\frac{x-n}{2^{n-1}} = \frac{x-1}{1}, \frac{x-2}{2}, \frac{x-3}{4}, \frac{x-4}{8}, \frac{x-5}{16}, \dots$$

for which the numerator ($x - n$) is less than -1.

Conway explains the relationship between the Ordinal Sum and Red-Blue Hackenbush trees in *ONAG* on page 88.

In essence, we find that for a Red-Blue Hackenbush tree that the value of the entire tree is dependent only on the value of the game position above the trunk. The reasoning behind this argument is simple and directly follows from Corollary 1. With this in mind, we can proceed to construct the gameset in a more general manner:



Without loss of generality, we have enumerated the options for a Red-Blue Hackenbush tree in the same spirit as previously discussed. The gameset $G = \{G^L \mid G^R\}$ is representative of all but one of the options available to the players in our generalization above; what is missing is the move for Left to hack the trunk, which is either a trivial option to take into account if there are no other moves for Left in the tree or we default to Corollary 1. In either case, the resulting gameplay is short and self-explanatory. Thus, we shall continue to concentrate on the much more interesting component to this generalization.

By the aforementioned method of proof by induction, we can apply the Ordinal Sum function to the gameset notation to reveal the following result:

$$1:x = \{0, 1:x^L \mid 1:x^R\}$$

Here we should recall the FTOH which says that all Red-Blue Hackenbush positions are numbers, and that by definition each of the Ordinal Sum components in the gameset above must also be numbers. Furthermore, notice that since zero is an option for Left, the Right option $1:x^R$ must be positive. In fact, this Ordinal Sum function is a mapping from all numbers onto the positive numbers in order of simplicity as outlined in the previous sections. As to be expected, this coincides with our intuition about games with Blue trunks since they guarantee the game's value to be positive despite what may be found further up the tree.

More can be said about the Ordinal Sum $1:x$ and its applications in CGT, but this is sufficient for our purposes in this paper. For more details, see *ONAG, Chapter 15: Ups, Downs, and Bynumbers*.

Algorithm for Trees

Here we examine a complete block of pseudo-code outlining a recursive implementation for Conway's method for calculating the values of Red-Blue Hackenbush Trees.

```

CalculateTreeValue(currentValue, sourcePositionInTree)
{
    // Initialize values
    stickValue      = 0
    branchSum       = 0
    branchVal       = 0
    currentStickId = 0
    branches = number of branches in sourcePositionInTree

    // Branch values (according to branch trunk color)
    blueBranchValue = 0
    redBranchValue  = 0

    /* If there are NO branches extending from the current position
     * (size of branch array = 0): then we cannot traverse any higher
     * Thus: simply return the current value */
    if(branches == 0)
        return(currentValue)

    /* Loop over the branches (at this point guaranteed at least 1)
     * and recursively traverse up the tree to find the value */
    for(branchIndex = 0; branchIndex < branches; branchIndex++)
    {
        currentStickId = id of branchIndex in sourcePositionInTree

        if(Color of currentStickId == RED)
            stickValue = -1
        else
            stickValue = 1

        // invoke recursion
        branchVal = calcTreeValue(stickValue, currentStickId)

        // Determine the stick color and add subtree value to it
        if(Color of currentStickId == RED)
            redBranchValue += branchVal
        else
            blueBranchValue += branchVal

        branchSum += branchVal
    }

    // Tree value is computed using an Ordinal Sum Algorithm
    value = CalculateOrdinalSum(currentValue, branchSum)
    return(value)
} // end CalculateTreeValue()

```

The main idea in this implementation is to use the instance of branching as a mechanism to promote the recursive nature to Conway's mathematics. Essentially, the function sets out to calculate the tree stemming from its current position without any other knowledge of where exactly in the tree at which it is being called. The only other data it receives is the value of the stick on top of which it is processing its calculations, which is also an indication of whether the stick is Red or Blue.

The function loops over all of the branches stemming from the current position. At each base stick to a branch, the stick value of 1 or -1 is assigned and then the function is recursively called to proceed up the branch until no more branches stem from the current position. A running sum of the branches is calculated and then, once the value of the tree above the current position is known, an Ordinal Sum can be run on this subtree. This result is returned so that other stacked recursive calls can be completed until there are no more to complete—at which point we will have calculated the value for the entire tree.

This algorithm handles both stalks and trees alike (since by definition stalks are a special type of tree with a maximum of one branching stick), which made it an excellent candidate to be the definitive calculation method in Visual Hackenbush.

Implementation

The trees algorithm was not implemented until Summer 2004, an entire year after the initial CGT value calculators, in one of my proudest achievements: Visual Hackenbush. It had been a top priority to assure that the user could draw the tree interactively and see the quantified results immediately and the trees algorithm was one which could cater to such a particular demand. Moreover, most of my previous analyses had been in binary trees, which are only a small class of Red-Blue Hackenbush trees, but the trees algorithm works for all trees no matter how many instances of branching the user decides to throw at it.

To be sure, the trees algorithm is far more powerful a tool than the simple stalks calculators; however, for an individual processing a series of stalks calculations is far simpler a task than tackling the bookkeeping issues which reside in the trees algorithm. Thus, it was still viable to search for a way in which a tree might be decomposed into several individual stalks, which is what we will examine next.

Hammerian Trees

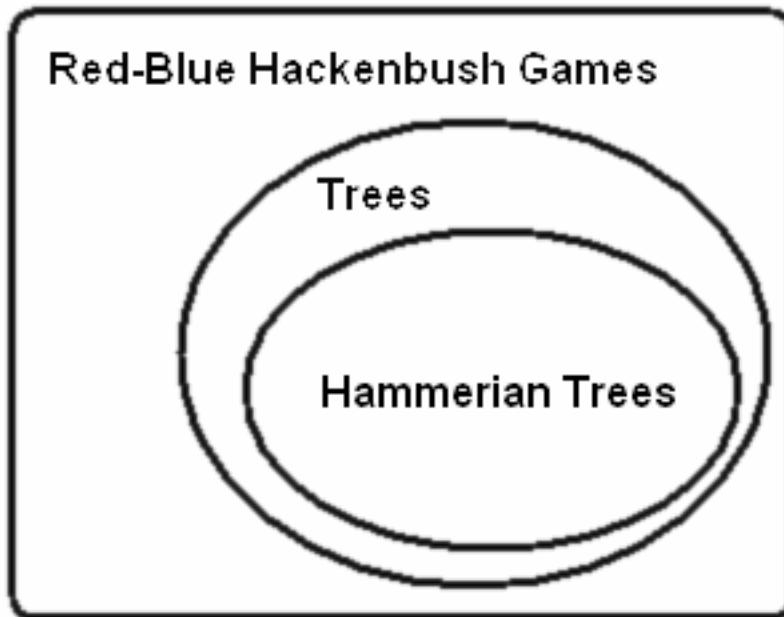
Introduction

Here we are at the threshold of the crux of the project: Hammerian Trees. With the aforementioned definitions and techniques, we are finally able to accurately describe and examine the first set of trees in Red-Blue Hackenbush that can be decomposed.

In this section, we define what it means for a tree to be Hammerian and the properties that can then be attributed to this classification. Also, we will characterize the process for decomposing a tree into its components so that it retains its game theoretic value.

Hammerian Domain

Before we go into extensive detail about what parameters define a tree as class Hammerian, it seems appropriate for us to place this class of trees in relation to the rest of Red-Blue Hackenbush games.

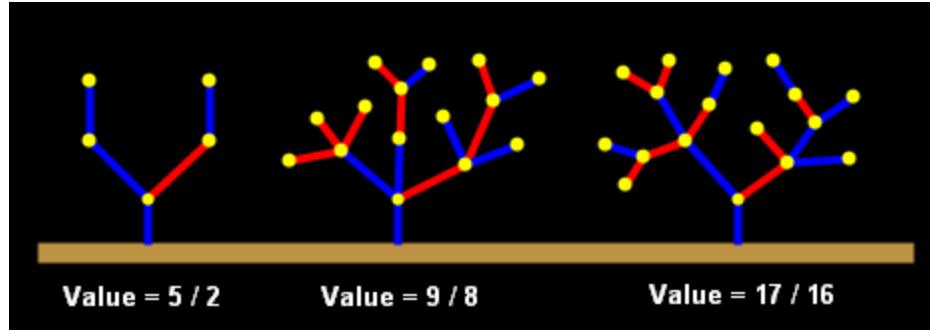


Hammerian Trees in the domain of all Red-Blue Hackenbush.

As shown in the diagram, Hammerian Trees are a subset of Red-Blue Hackenbush Trees, which are (not surprisingly) a subset of all Red-Blue Hackenbush Games.

Definition: Hammerian Tree

A Hammerian Tree is a Red-Blue Hackenbush game in which the branches may be severed from the full tree and planted separately without altering the combinatorial game value of the original game.



Examples of Hammerian Trees.

Hammerian Classification

In order for a Red-Blue Hackenbush Tree to be classified as Hammerian it must fulfill the following requirement:

If the trunk of the tree is pure Blue, $|\sum \text{CGV}(\text{branches}_b)| \geq |\sum \text{CGV}(\text{branches}_r)|$, where $\text{CGV}(\text{branches}_b)$ is the combinatorial game value of a branch whose lowest stick is colored Blue, and $\text{CGV}(\text{branches}_r)$ is the combinatorial game value of a branch whose lowest stick is colored Red. Similarly, if the trunk of the tree is pure Red, then $|\sum \text{CGV}(\text{branches}_b)| \leq |\sum \text{CGV}(\text{branches}_r)|$.

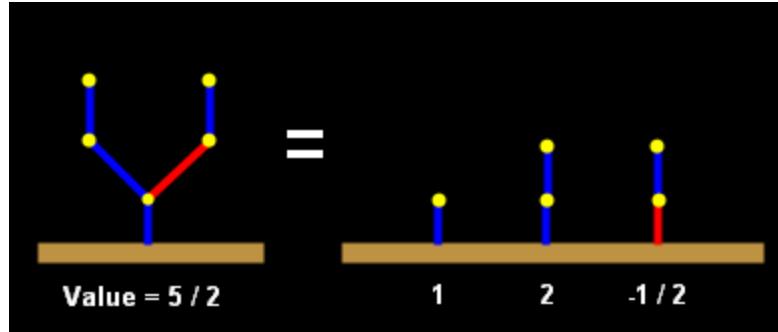
Otherwise, it can be proven that the tree cannot be decomposed.

In other words, the value of the tree is equal to the value of its decomposed components. However, one should be cautioned that the process of decomposition does not allow just any branch or stick to be cut from the tree and planted. Before proceeding too much further, this last statement requires further examination and so we shall look at the proper method by which trees can be decomposed.

Method of Decomposition

We will investigate the process of decomposing a Red-Blue Hackenbush Tree through a short series of examples.

Example 1:



First, we apply the definition to ensure that the tree above is in fact Hammerian. Since the trunk of the tree is pure Blue, we must check that the given inequality holds:

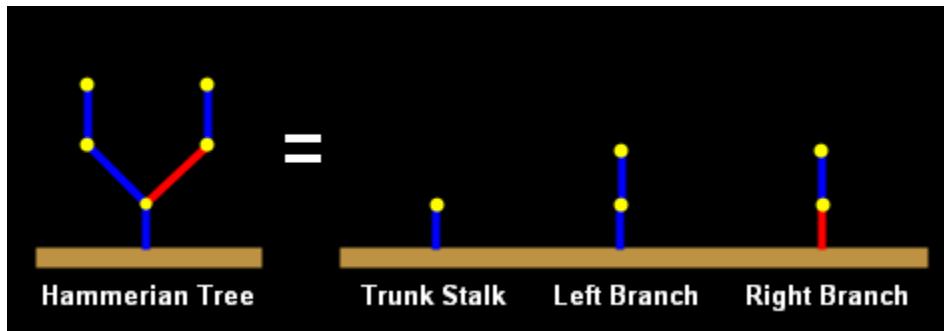
$$|\sum \text{CGV}(\text{branches}_b)| \geq |\sum \text{CGV}(\text{branches}_r)|.$$

In our case, we must check that the absolute value of the branch on the left (with the Blue base stick) is greater than or equal to the absolute value of the branch on the right (with the Red base stick). We recall our previous calculations of the individual branch stalks, rendering:

$$|\sum \text{CGV}(\text{branches}_b)| \geq |\sum \text{CGV}(\text{branches}_r)| = |2| \geq |\frac{1}{2}|.$$

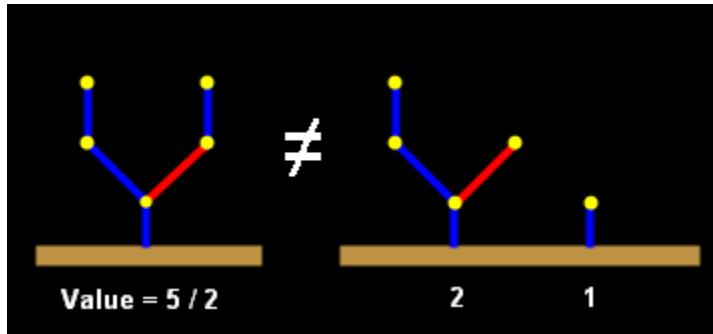
Thus, the tree above is Hammerian and is therefore able to be decomposed.

The decomposition, as shown above, is performed in the following manner: we remove all of the branches at the instance of branching and plant them in the ground as separate trees. This results in the following template:



IMPORTANT NOTE:

However, it should be noted that we must perform the decomposition upon the instance of branching in question. Otherwise, the resulting sum of subtrees is *not* guaranteed to be equal to the game theoretic value of the original tree. See example below:

Example 2:

Here we have the same original tree, yet with a “decomposition” differing from that exemplified in Example 1. Despite the fact that the tree has already been classified as Hammerian, the actual decomposition was performed incorrectly.

From the labeled values it is obvious that something is wrong since $\frac{5}{2} \neq 2 + 1$. Of course, such an observation cannot always be easily made since the extent of the tree and the number of instances of branching may be very large. Thus, it is of the utmost importance that this point be made clear.

The Blue stick of the right branch was severed and planted. This is in violation to our Hammerian classification parameter since the trunk of the right branch subtree is Red. The following condition must be satisfied by the branches within that subtree in order to be decomposed:

$$|\sum \text{CGV}(\text{branches}_b)| \leq |\sum \text{CGV}(\text{branches}_r)|.$$

In this case, we find that we do not satisfy the conditions since:

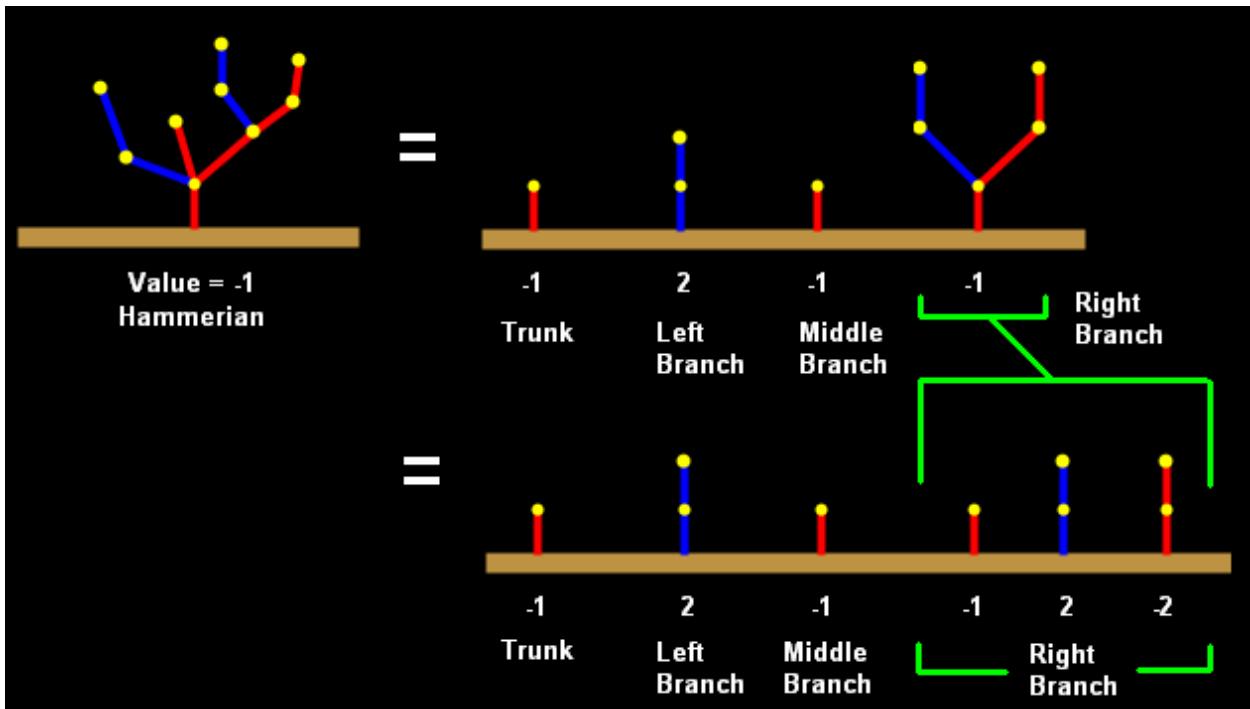
$$|\sum \text{CGV}(\text{branches}_b)| \leq |\sum \text{CGV}(\text{branches}_r)| \Rightarrow |1| \leq |0|$$

which is certainly a false statement.

Other Observations

Recursive Definition

The Hammerian classification parameter is truly recursive under the condition that the tree branching below the branching node in question is Hammerian.



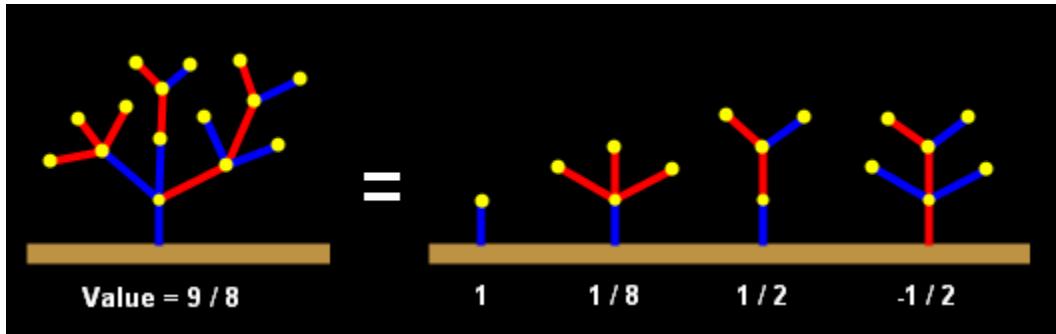
As exemplified by the decomposition above, we initially decompose the tree at the first instance of branching. This renders us with the first result: the trunk, the left branch stalk, the middle branch stalk, and the right branch subtree. Since the subtree is also Hammerian, it too can be decomposed.

Most importantly, we observe that throughout the process the value of the original tree was always equal to the sum of the decomposed components. This property is essentially an extension to the fact that trees are governed by their relationship with Ordinal Sums, which allows for the classification to percolate up the tree until there is an instance of branching that renders the tree (or subtree) non-Hammerian.

By the same logic, it can be argued that no instance of branching stemming after a non-Hammerian instance of branching is Hammerian. In other words, while working our way up the tree, if we reach an instance of branching that fails to meet the criteria required to be classified as Hammerian, then there cannot be an instance of branching further up that subtree that can be decomposed.

Multi-Branching and Solid-Color Branching

It is important to notice that the Hammerian classification parameter makes no specifications to the number of branches for either side of the inequality. So far, each of the dissected examples have dealt with binary trees, yet the classification stands for trees with any number of branches stemming from one single node.



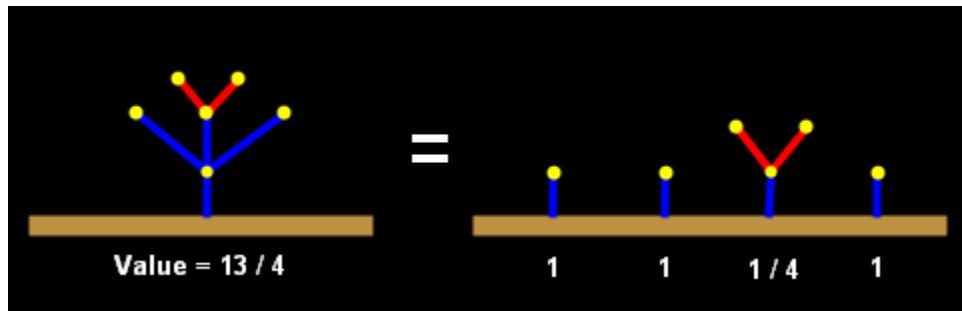
This means that it also applies at instances in Red-Blue Hackenbush Trees where all of the branches begin with the same color:

$$|\sum \text{CGV}(\text{branches}_b)| \geq |\sum \text{CGV}(\text{branches}_r)| = 0, \text{ no branches which begin with Red,}$$

and

$$|\sum \text{CGV}(\text{branches}_b)| = 0 \leq |\sum \text{CGV}(\text{branches}_r)|, \text{ no branches which begin with Blue.}$$

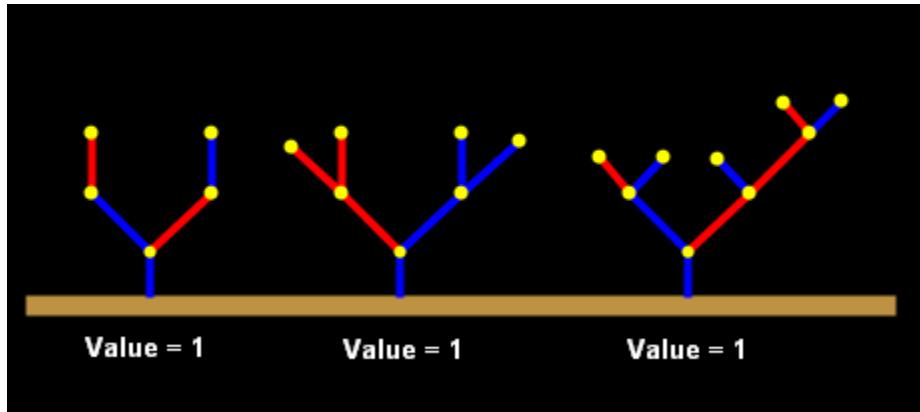
Such a case is shown below:



Joshua Trees

Definition: Joshua Tree

A Joshua Tree is a Red-Blue Hackenbush game in which the combinatorial game value of the tree itself is the same as the combinatorial game value of the trunk.



Examples of Joshua Trees.

Joshua Classification

In order for a Red-Blue Hackenbush Tree to be classified as a Joshua Tree it must fulfill the following requirement:

$$\sum \text{CGV}(\text{branches}) = 0,$$

where $\text{CGV}(\text{branches})$ is the combinatorial game value of each branch.

In other words, the combinatorial game value of the tree itself is the same as the combinatorial game value of the trunk. This also implies that the absolute value of branches_b is equivalent to the absolute value of branches_r. That is:

$$|\sum \text{CGV}(\text{branches}_b)| = |\sum \text{CGV}(\text{branches}_r)|.$$

Other Observations

The easiest way to simulate a Joshua Tree is by creating the branches in the same fashion as Tweedle-Dum and Tweedle-Dee. Essentially, this means that each subtree (branch) is constructed to have a combinatorial game value of zero; leading us to play out the game until all that remains is the trunk—confirming our intuition that the combinatorial game value should be the same as the trunk alone.

Also, it should be noted that the examples above may seem a little misleading since they all have a trunk stalk of size one. To be sure, the trunk stalk can be any size when considering the conditions for being a Joshua Tree.

Actually, there's not much more to say about Joshua Trees as they are simply Hammerian Trees that have the same value as their trunk and we just completed a relatively involved discussion on Hammerian Trees in the last section. In any case, Joshua Trees seemed to show up so often throughout my research that I needed to give them a name for ease of reference.

Conclusions

Summary

We have finally come to the end of our discourse on Hackenbush. Throughout this paper we have examined the game of Red-Blue Hackenbush with an introduction to the fundamentals of Combinatorial Game Theory and an in-depth analysis of the tree structures within the game. After establishing several precise definitions concerning trees and their components, we were able to prove a few general theorems about Hackenbush before jumping into the topics of Stalks, Trees and their relationship with the Ordinal Sum function. Finally, we examined the findings of my research by defining two classifications: Hammerian Trees and Joshua Trees, as well as the constraints required to decompose them into component parts which sum to be the same game theoretic value as the original tree.

This is by no means a completion to all the analysis for Hackenbush—which is to say that there is much, much more to be researched and documented. Here I have shown my perspective on solving Hackenbush through the technique of decomposition and problem reduction. Though these methods are only proven to be applicable on trees, there may very well be use for decomposition in the general case of all Red-Blue Hackenbush games. Furthermore, there still remains the question of non-Hammerian Trees and exactly why the decomposition does not apply to them in the same manner. It may be possible that there is a different sort of decomposition that governs their breakdown into easier to compute components. But alas, such theories have yet to be grounded and examined, and must wait as they are dependent upon the future of Hammerian research.

Future of Hammerian Research

I honestly can't say what the future of these classifications are since up to this point I have not found any other practical use for them with the exception of having the ability to calculate complicated trees into simpler structures. Problem reduction certainly is an important topic in both Mathematics and Computer Science, and I hope that one day this will be the lifeblood for my methods presented here.

On the other hand, the future of the standalone application Visual Hackenbush has a very promising future as it has already been in use for almost a full year. In fact, just recently, I received an email from Aaron Siegel, author of one of the most widespread Combinatorial Games computer applications: CG Suite, concerning the integration of the Visual Hackenbush Drawing Canvas into the CG Suite environment!

I certainly hope to continue my work and research in Red-Blue Hackenbush throughout my graduate studies.

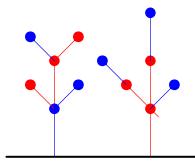
Appendix A: Hackenbush Game Tree Example

Preface

The following game tree is a complete solution to one of the homework problems given out in Math 168A. This particular solution was turned in by one of my top-students during the Winter 2004 course offering: Erik Hill. Erik graduated with a Bachelor's in Mathematics- Computer Science in June of 2004 and is currently in the CSE Master's program at UC San Diego. Erik and I have kept in contact with each other and now both work in the Experimental Game Lab as programmers under the direction of Sheldon Brown. This example is printed here with his permission.

Problem 1.5.2

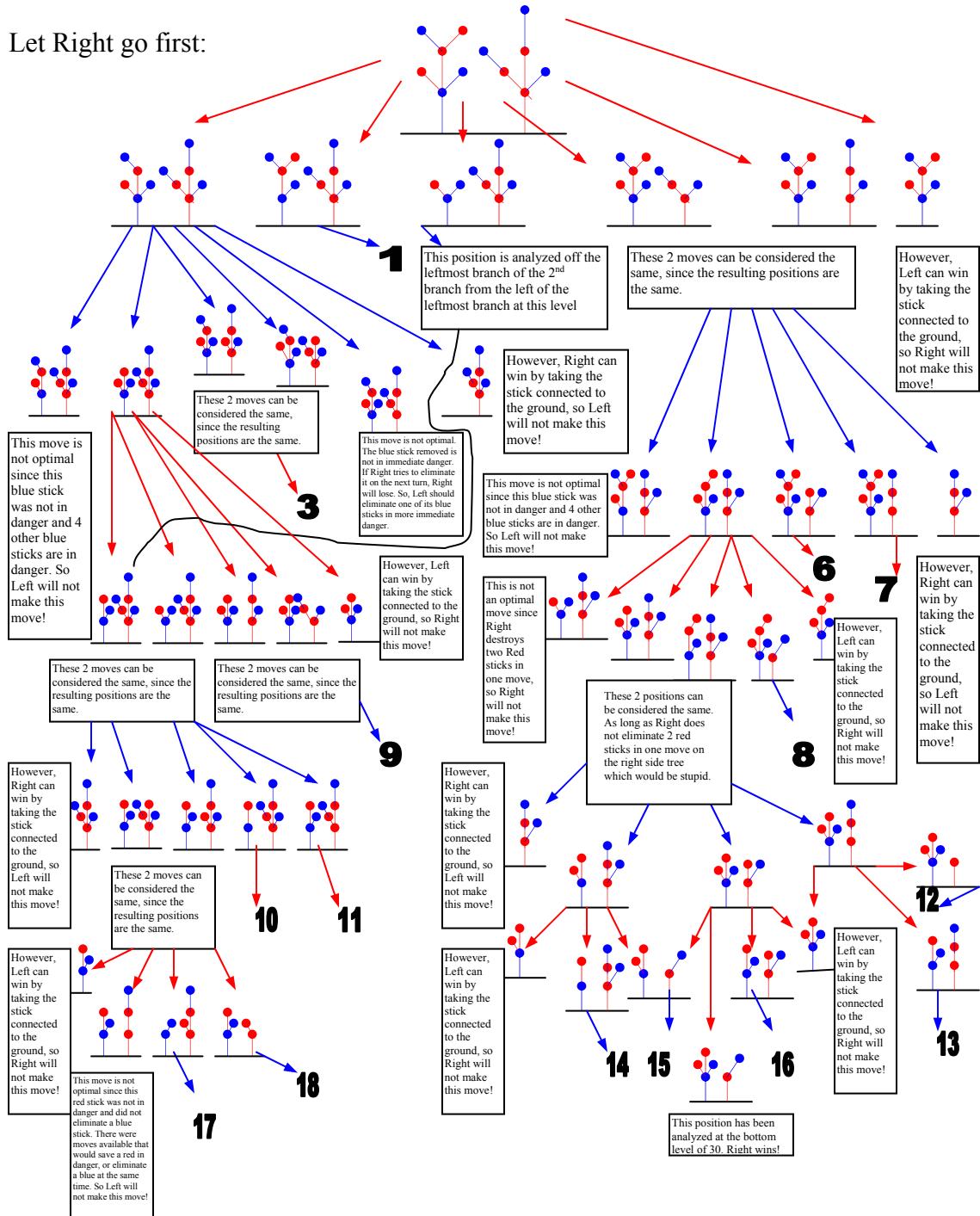
Show that the following figure is a winning position for Right:

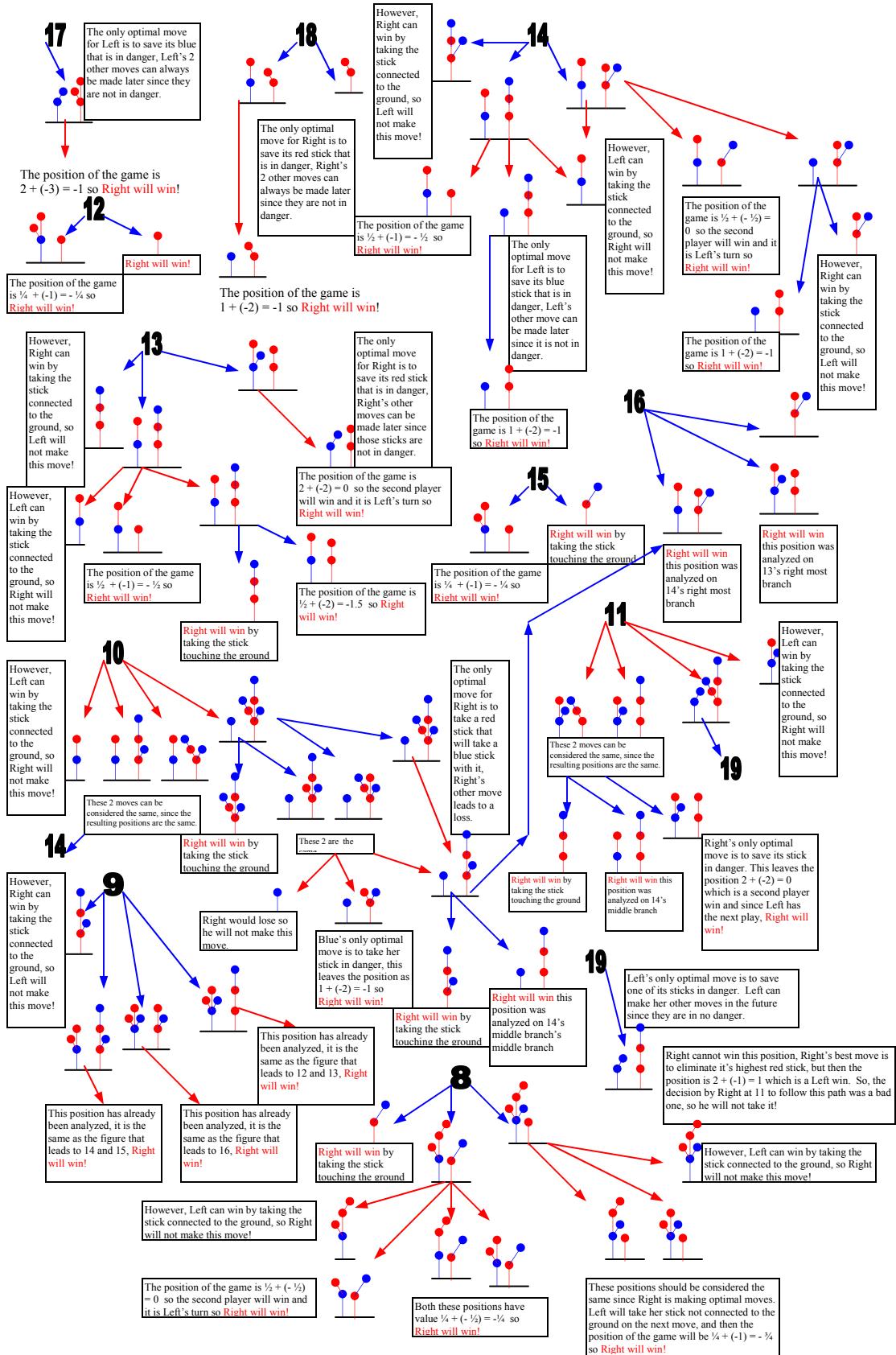


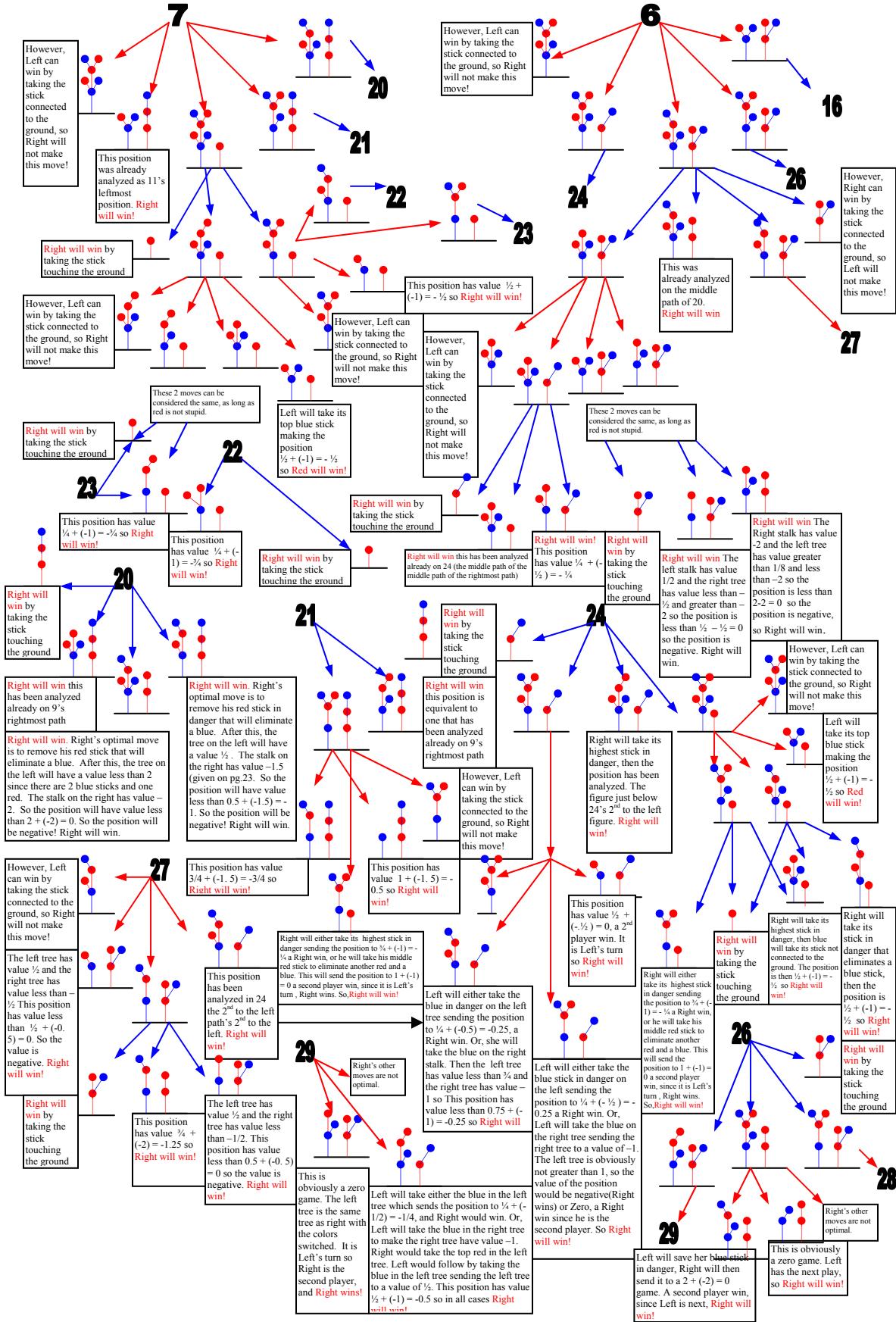
Do this by playing out the game.

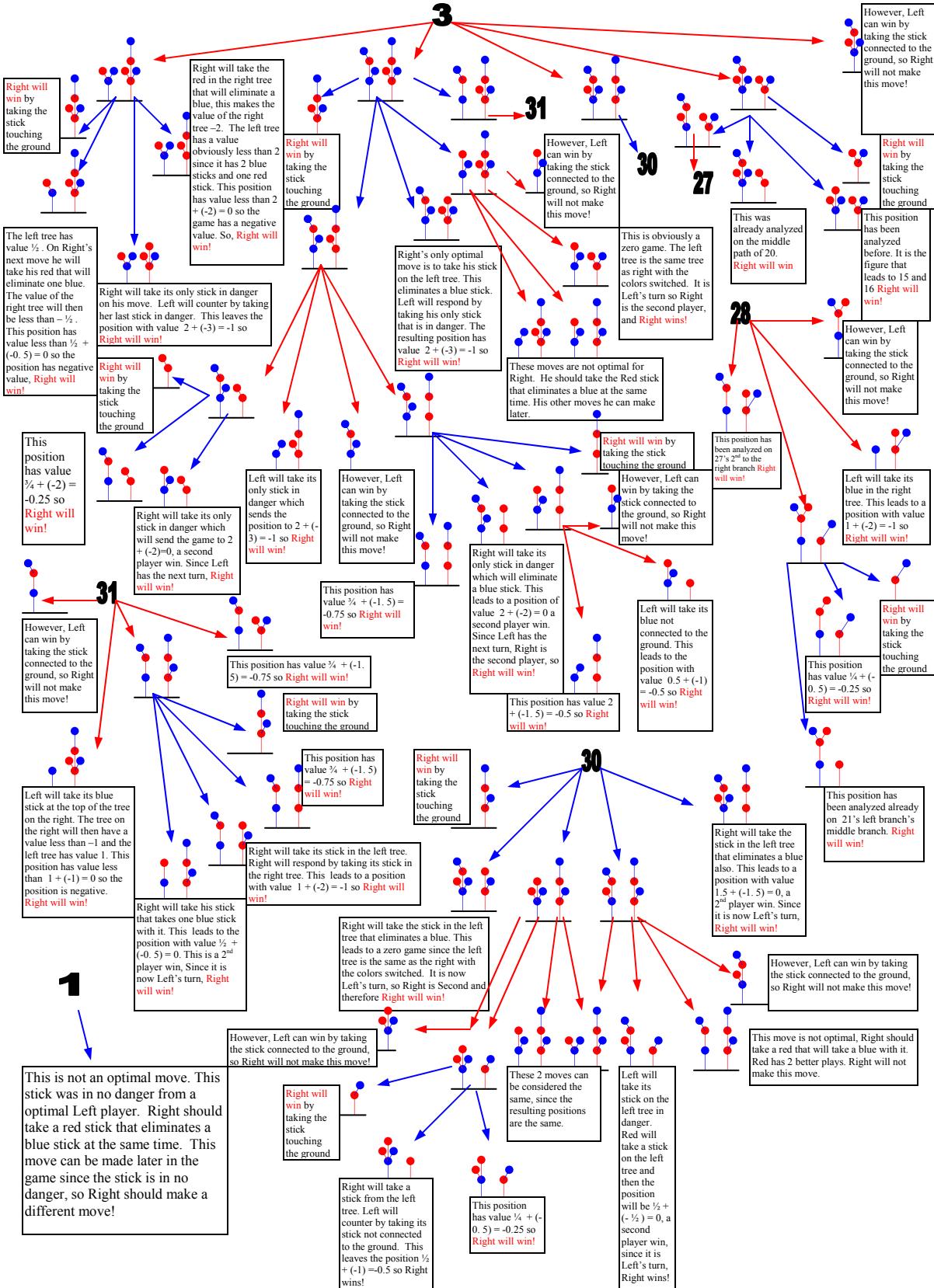
Erik Hill's Solution

Let Right go first:

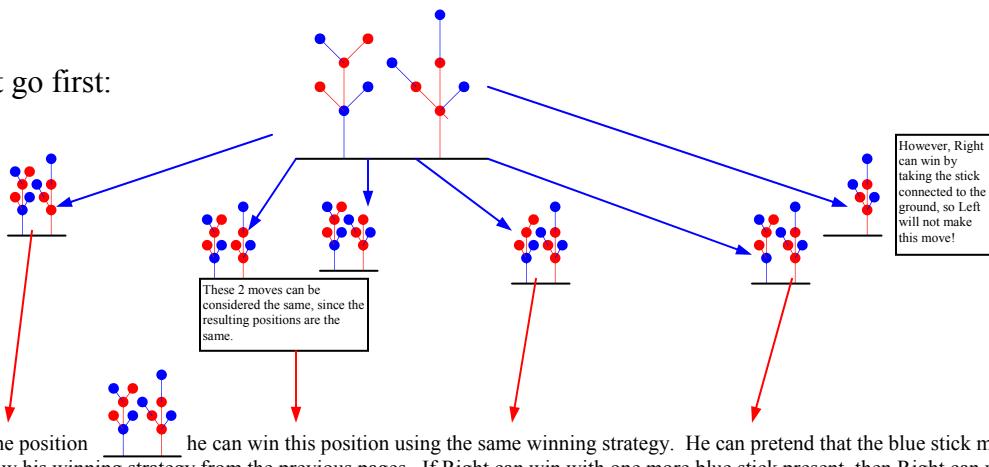








Now let Left go first:



Since Right won the position he can win this position using the same winning strategy. He can pretend that the blue stick missing is still there and follow his winning strategy from the previous pages. If Right can win with one more blue stick present, then Right can win when the blue stick is not present. This is due to the fact that the blue stick adds some positive factor to the value of the position. Since Right was the first player and won in the previous case, the starting position must have a negative value by Definition 1.3.1(iv).

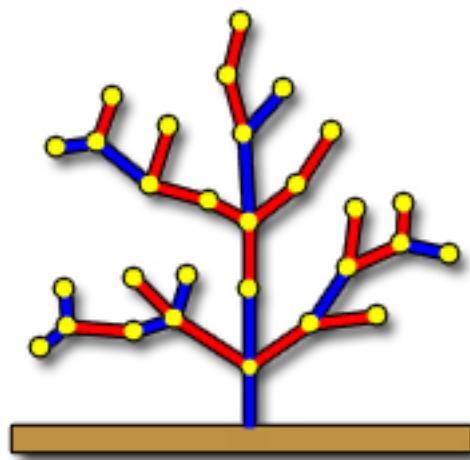
The position without the blue stick has a smaller value than the position with the blue stick, so these positions all have a value smaller than the negative value of .

So, all of these positions are negative. This implies **Right will win all of these positions!!!**

Note: This argument would not have worked if one of Left's move had eliminated any red sticks!

Visual Hackenbush

Version 1.1



Welcome to the
Visual Hackenbush Documentation

Visual Hackenbush v1.1
©2003-2004 Joey Hammer

Joey Hammer
University of California, San Diego
<http://cgt.calculusfairy.com>

Table of Contents

Introduction	5
Introduction to the Package	
About the Developer	
 Installation	6
System Requirements	
JRE 1.4 Installation	
Visual Hackenbush Installation	
 Hackenbush Tree Calculation Studio	7
Know Your Environment	
Visual Hackenbush Development	
Version 1.1 Capabilities	
 Menus	9
Toolbar	
File Menu	
View Menu	
Game Menu	
Project Menu	
Help Menu	
 Studio Main Frame	12
Project Control Display Panel	
Drawing Control	
Drawing Canvas	
Decomposition Canvas [currently unavailable]	
 Display Options Frame	15
Size Adjustment Sliders	
Color Adjustment ComboBoxes	

Table of Contents

Project Settings Frame	17
Calculation Settings	
Log Options	
Other Options	
Visual Hackenbush Log Frame	19
Log Overview	
Log Tools	
Visual Hackenbush Functionality	21
Drawing Canvas Revisited	
Moving Vertices	
Adding Sticks to the Tree	
Modifying Stick Colors	
Removing Sticks from the Tree	
Credits	27

Introduction

Introduction to the Package

Visual Hackenbush is a free Hackenbush Tree calculation environment. Its only requirement is the Java Runtime Environment, which can be freely downloaded from Sun Microsystems' website at <http://java.sun.com>.

Visual Hackenbush is still being developed at the time of this current release so there are potential bugs. Therefore, please send feedback (positive and negative welcome) to the web address below.

Visual Hackenbush
Version: 1.1
© 2003-2004 Joey Hammer
<http://cgt.calculusfairy.com>

Java Runtime Environment
Version: 1.4.1
© 2002 Sun Microsystems, Inc.
<http://java.sun.com>

About the Developer

Joey Hammer completed his degree in Mathematics-Computer Science at the University of California, San Diego. His current research is conducted in the field of Combinatorial Game Theory, primarily focused in the game of Hackenbush.

Joey first became interested in Combinatorial Games in the Fall of 2000. He later became one of the first students to enroll in an experimental course on CGT in the Spring of 2003: Math 168A taught by Professor Len Haff and Jason Lee. During the following year, Hammer served as Haff's Teaching Assistant for the Math 168A course for two quarters before graduating.

Hammer has also developed a web resource guide for Combinatorial Games, which can be found at: <http://cgt.calculusfairy.com>.

Installation

System Requirements:

Basically, just about any system should be able to handle Visual Hackenbush. However, in order to run it, you must have the Java Runtime Environment (JRE) installed on your system.

Note for Network Users:

In order to save Visual Hackenbush Trees (*.hat) files to your system, you must have write access privileges.

JRE 1.4 Installation:

1. Download the `jre-1.4.2` installation package from:
Sun Microsystems' website <http://java.sun.com> or
Math 168A Course Site <http://cgt.calculusfairy.com/Software/jre.exe>
2. Double-click on the `jre.exe` and follow the instructions.

Visual Hackenbush Installation:

1. Download the `Visual_Hackenbush_1.1-Setup.exe` installation setup from <http://cgt.calculusfairy.com/Software/VisualHackenbush/>.
2. Double-click on the installer and follow the instructions.
3. Simply double-click on the shortcut under CGT/Visual Hackenbush and you're ready to start drawing some trees!

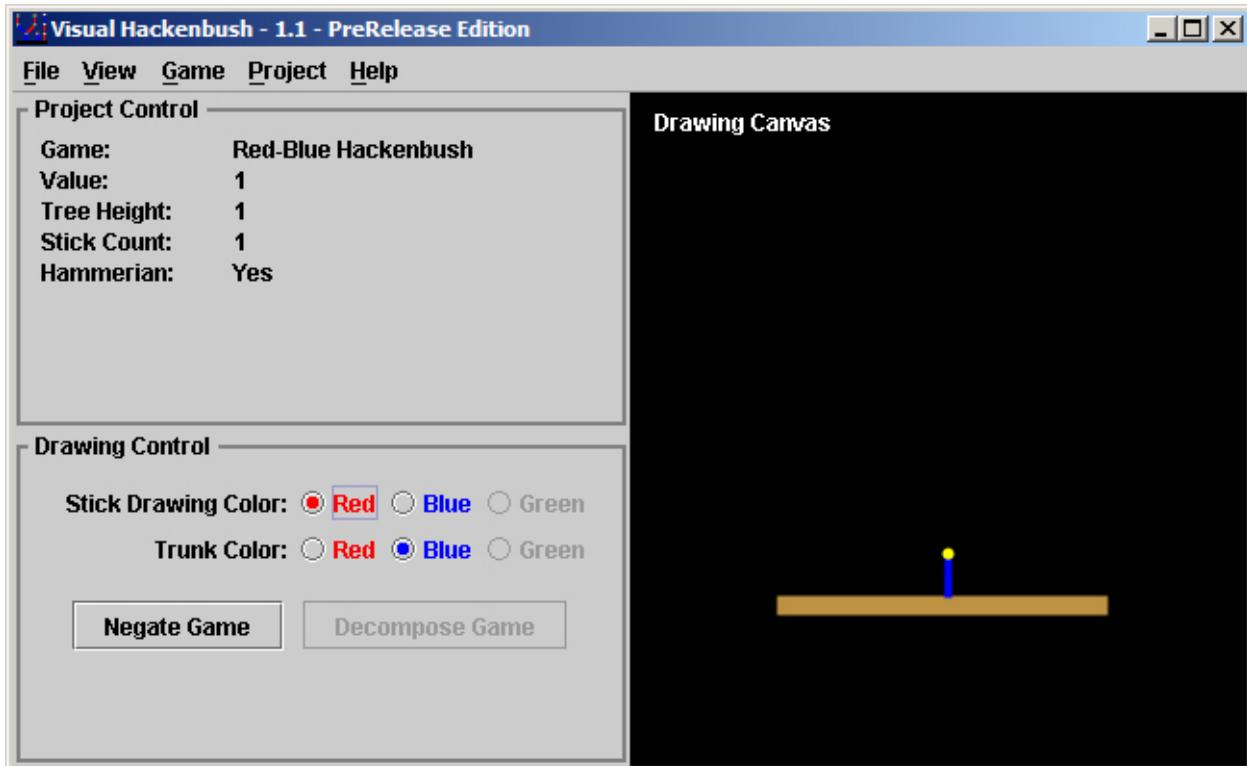
Hackenbush Tree Calculation Studio

Know Your Environment:

Visual Hackenbush is a windowed environment, composed of four separate window frames:

- **Visual Hackenbush Main Window Frame**
- **Display Options Window Frame**
- **Project Options Window Frame**
- **Visual Hackenbush Log Window Frame**

We will go through each frame in-depth and explain their features and functionality in the sections to follow. In this section, we will keep to introducing the capabilities of the overall application itself.



Hackenbush Tree Calculation Studio [Continued]

Visual Hackenbush Development:

Essentially, Visual Hackenbush was developed to be a personal tool for my research in the area of Hackenbush Tree Decomposition. I had initially wanted an applet that provided the user with a graphical interface inside of which they could draw the trees they wanted to analyze. To assist with the analysis, I wanted the program to have the ability to calculate the value of what had been directly drawn into the interface. This turned into an extensive project to say the least, but I'm proud to say that version 1.1 is the first real landmark for my hopes and dreams in the development of this software.

Furthermore, I have implemented several additional features to Visual Hackenbush since its conception, including a play-by-play log, adjustable display settings, and the ability to read and write Visual Hackenbush Trees (*.hat) files to name a few. In fact, the file input/ output option was so important that it required almost a complete overhaul to the software's backbone—including the constraint that the program be written as an application rather than an applet.

The final thing I should reemphasize is that most of Visual Hackenbush's functionality is solely based upon the tools I needed for my research. This is the reason why there is an entire panel dedicated to Hackenbush Tree Decomposition and a gamut of other tools which factor into labeling a tree as either Hammerian or Non-Hammerian. Whether these tools would be of value to other users was not one of my top priorities when developing this software; nevertheless, they are still available to those who may have a need for them or are interested in experimenting with their extended capabilities.

Version 1.1 Capabilities:

Upon completion, Visual Hackenbush will be able to calculate the combinatorial game value of any type of Hackenbush Tree you can throw at it (well, draw in it).

But since that day has not yet come, it is somewhat limited in its calculation capabilities. In 1.1 version release, Visual Hackenbush has the ability to calculate any tree governed by the rules of ***Red-Blue Hackenbush alone***.

Support for other games types, such as Red-Blue-Green Hackenbush, Childish Red-Blue Hackenbush, and Green Hackenbush, is in development and will be released as soon as they can be properly devised, written, and tested.

Menus

Toolbar:

Just like any other windowed program, Visual Hackenbush has a toolbar from which you can choose various options. I have also included key-bindings to most of the options found in the menus of the toolbar for quick access.



We now outline each of the menus:

File Menu:

- New:** **CTRL + N**
Selecting this option will clear the current Hackenbush Tree in the Drawing Canvas and reset the controls to their default settings.
- Open:** **--**
Selecting this option will bring up the “Open Dialog” window so that a previously saved (*.hat) file can be opened. The game type and button settings stored in the (*.hat) file will be automatically applied when a file is opened.
- Save:** **CTRL + S**
Depending upon the situation, selecting this option will either bring up the “Save As Dialog” window so that you can name and save your file in the location of choice, or it will simply save the currently active (*.hat) file without a prompt. If you do not wish to overwrite your currently active (*.hat) file, you should use the **Save As** menu option instead.
- Save As:** **--**
Selecting this option will bring up the Save As Dialog window so that you may name and save your (*.hat) file in the location of your choosing.
- Exit:** **CTRL + X**
Selecting this option will exit Visual Hackenbush. While this is equivalent to clicking the ‘X’ in the upper right-hand corner of the main window frame, choosing “Exit” does provide a safety net for your project: prompting you to save your current work before you exit.

Menus [Continued]

View Menu:

Drawing Canvas:

C

Selecting this option will change the right-hand panel to the Drawing Canvas if it is not already visible.

Decomposition Canvas:

D

Selecting this option will change the right-hand panel to the Decomposition Canvas if it is not already visible.

View Log:

Selecting this option will make the Visual Hackenbush Log visible.

Game Menu:

Red-Blue Hackenbush:

1

Selecting this option will create a new game and change the Hackenbush game type to Red-Blue Hackenbush. In consequence, the Green Stick and Trunk buttons in the Drawing Control Panel will be disabled.

Red-Blue-Green Hackenbush:

2

Selecting this option will create a new game and change the Hackenbush game type to Red-Blue-Green Hackenbush.

Childish Red-Blue Hackenbush:

3

Selecting this option will create a new game and change the Hackenbush game type to Childish Red-Blue Hackenbush. In consequence, the Green Stick and Trunk buttons in the Drawing Control Panel will be disabled.

Green Hackenbush:

4

Selecting this option will create a new game and change the Hackenbush game type to Green Hackenbush. In consequence, the Red and Blue Stick and Trunk buttons in the Drawing Control Panel will be disabled.

Menus [Continued]

Project Menu:

Calculate Value: **ENTER**
 Selecting this option will initiate a recalculation of the combinatorial game value of the tree currently visible in the Drawing Canvas. If the **Auto-Calculate** option is activated in the Project Settings Window, this option will be disabled since Visual Hackenbush will recalculate the tree value each time the tree is structurally modified (by this I mean that moving the vertices do NOT initiate a recalculation, however: the options of adding, subtracting, or changing the color of one or more sticks will most certainly initiate a recalculation).

Tighten Display: **T**
 Selecting this option will change the position of the vertices in the Drawing Canvas so that the structure of the resulting Hackenbush Tree is more clear. Visual Hackenbush will try its best to make the lengths of each stick as uniform as possible, and space the vertex positions far enough apart so that they are visually recognizable. [1.1– somewhat unpredictable results]*

Display Options: **O**
 Selecting this option will make the Display Options window frame visible. This window frame is described in detail later.

Project Settings: **S**
 Selecting this option will make the Project Settings window frame visible. This window frame is described in detail later.

Help Menu:

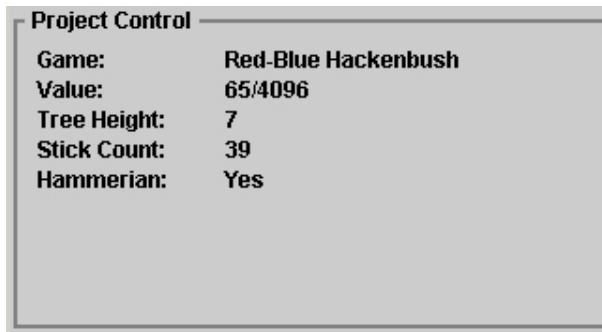
Contents: **--**
 Selecting this option will display a link to the location of the Visual Hackenbush Documentation.

About: **--**
 Selecting this option will display the copyright and version information for the release of Visual Hackenbush currently in use.

***Tighten Display:** may place connection nodes on top of each other resulting in two or more sticks being drawn on top of each other. These “hidden” sticks could lead to confusion, it is suggested you check for overlapping nodes after using this functionality until this situation is remedied and updated.

Studio Main Frame

Project Control Display Panel:



This is Visual Hackenbush's information and calculation results display. At current release, it displays the following data:

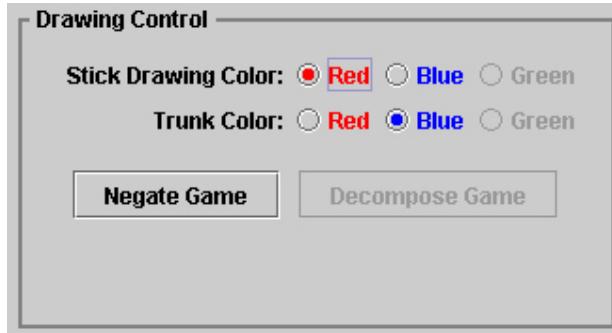
- Hackenbush Game Type
- Combinatorial Game Value
- Height of the Current Tree
- Number of Sticks (which compose the current tree)

Additionally, to supplement my own personal research, the panel also displays whether or not the current tree is considered Hammerian.

Certainly as the functionality and algorithmic capabilities in later releases of Visual Hackenbush progress, There are plans for displaying more information about the trees constructed via the Drawing Canvas (hence the reason for the unused space at the bottom of the panel).

Studio Main Frame [Continued]

Drawing Control Panel:



This is Visual Hackenbush's drawing control center. Its main purpose is to control parameters of the Drawing Canvas in the right panel. At current release, the user has the following options to adjust:

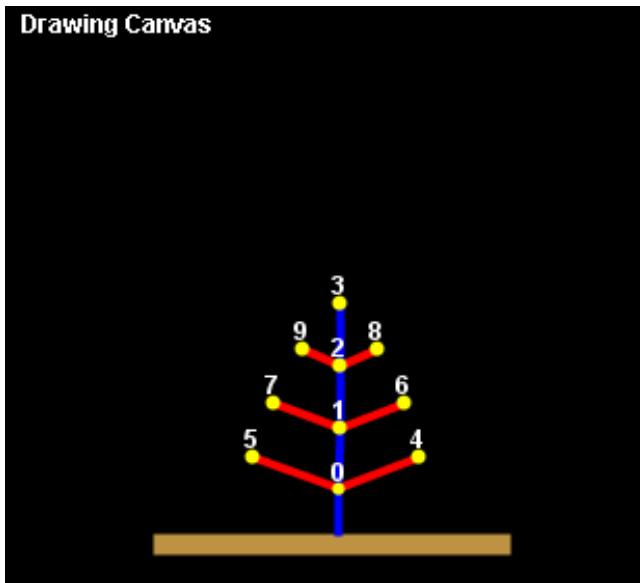
- Stick Drawing Color
- Trunk Color
- Game Negation
- Show the Game Decomposition Panel [1.1- not functional]

Of course, both the Stick Drawing Color and Trunk Color buttons will adjust as the user changes the overall Hackenbush game type.

It should be noted that the Trunk Color buttons are the only way to adjust the trunk of the current tree in the Drawing Canvas since there is no special popup menu which handles the trunk. This prevents the user from removing the trunk, leaving them with nothing to work with: a trivial zero game.

Studio Main Frame [Continued]

Drawing Canvas:



This is the essence of Visual Hackenbush. The Drawing Canvas is where 90% of the action is, so let's properly examine its components in detail.

NOTE: We explain the functionality in a later section.

Vertices:

Vertices are the connections between sticks (shown in yellow above). Each one has an identification number which can be displayed by activating the option in the Project Settings window frame.

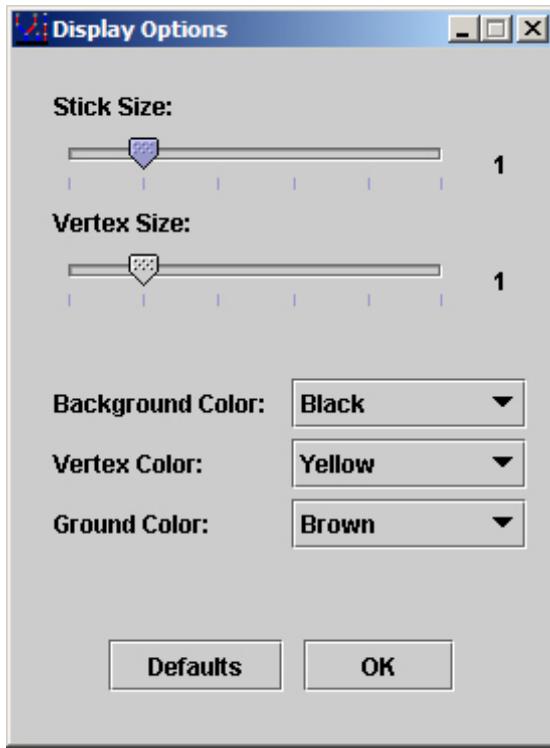
Trunk:

The trunk is a very special stick—it is the only stick directly connected to the ground (this may change in later versions of Visual Hackenbush). It cannot be removed. Its corresponding vertex is labeled as 0. Its color can only be adjusted via the Drawing Control Center.

Sticks:

Sticks are the components which make up any Hackenbush Tree you can imagine. The user may add, remove, and change the colors of any stick in the tree (with the aforementioned restriction of the trunk, which cannot be added or removed). Their visual lengths do NOT affect the combinatorial game value.

Display Options Frame



Size Adjustment Sliders:

Stick Size:

Allows the user to uniformly adjust the thickness of the sticks displayed in the Drawing Canvas. Also affects the sticks drawn after adjustment.

Default Size: 1.

Range: [0, 5].

Vertex Size:

Allows the user to uniformly adjust all the radii of the vertices displayed in the Drawing Canvas. Also affects the vertices drawn after adjustment.

Default Size: 1.

Range: [0, 5].

Display Options Frame [Continued]

Color Adjustment ComboBoxes:

Background Color:

Allows the user to adjust the background color displayed in the Drawing Canvas.

Default Color: Black.

Choices: [Black, White].

Vertex Color:

Allows the user to adjust the color of each of the vertices displayed in the Drawing Canvas. Also affects the vertices drawn after adjustment.

Default Color: Yellow.

Choices: [Gray, White, Yellow].

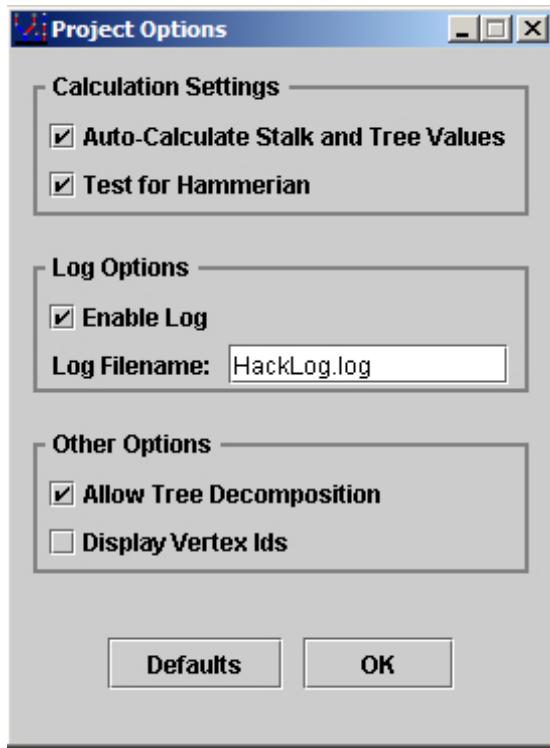
Ground Color:

Allows the user to adjust the ground color displayed in the Drawing Canvas.

Default Color: Brown.

Choices: [Black, Brown, Gray, White].

Project Settings Frame



Calculation Settings:

Auto-Calculate Stalk and Tree Values:

Toggles the option for Visual Hackenbush to automatically calculate the combinatorial game value for the tree being constructed in the Drawing Canvas.

Default Value: ON.

Test for Hammerian:

Toggles the option for Visual Hackenbush to automatically test and calculate whether the tree being constructed in the Drawing Canvas fulfills the parameters to be classified as Hammerian or Non-Hammerian.

Default Value: ON.

Project Settings Frame [Continued]

Log Options:

Enable Log:

Toggles the option for Visual Hackenbush to keep a log of the actions done within the Drawing Canvas and Hackenbush Project.

Default Value: ON.

Log Filename:

Allows the user to specify the name of the log file to be used when the log is saved via the “Save Log” button on the Log Frame. It should be noted that this filename *cannot* be changed if the log is disabled.

Default Filename: HackLog.log.

Other Options:

Allow Tree Decomposition:

Toggles the option for Visual Hackenbush to calculate the Hammerian Decomposition of Hackenbush Trees so that the user may examine the results on the Decomposition Canvas. [1.1- not available]

Default Value: ON.

Display Vertex Ids:

Toggles the option for Visual Hackenbush to label each vertex as it is added to the tree in the Drawing Canvas.

Default Value: OFF.

Visual Hackenbush Log



Log Overview:

The Visual Hackenbush Log is a very important tool to creating trees. It stores to a text file all the commands and processes in the order in which the user performs them upon the Drawing Canvas. The fact that this can be saved and reopened in a simple Text Editor (such as Notepad) means that the user can recreate a tree or pattern via a readable plaintext. This is quite different from the Hackenbush Tree files (*.hat) which are structured for space optimization, making them much more cryptic to read. Additionally, the (*.hat) files are simply a snapshot of the Hackenbush Tree structure at the time it was saved and does not record the steps leading up to the saved product.

Furthermore, the log records *very* detailed information which cannot be found in the Project Control Display Panel, such as a stick's source ID, a stick's tree level, and even the identification numbers from the sticks removed.

Visual Hackenbush Log [Continued]

Log Tools:

Mark:

Allows the user to insert a separator line and a numbered “mark” in the log, complete with timestamp. This particular feature comes in handy if the user wishes to examine the log from a particular point in their project work in the Drawing Canvas. This is better than simply clearing the log since all previous entries remain in the log.

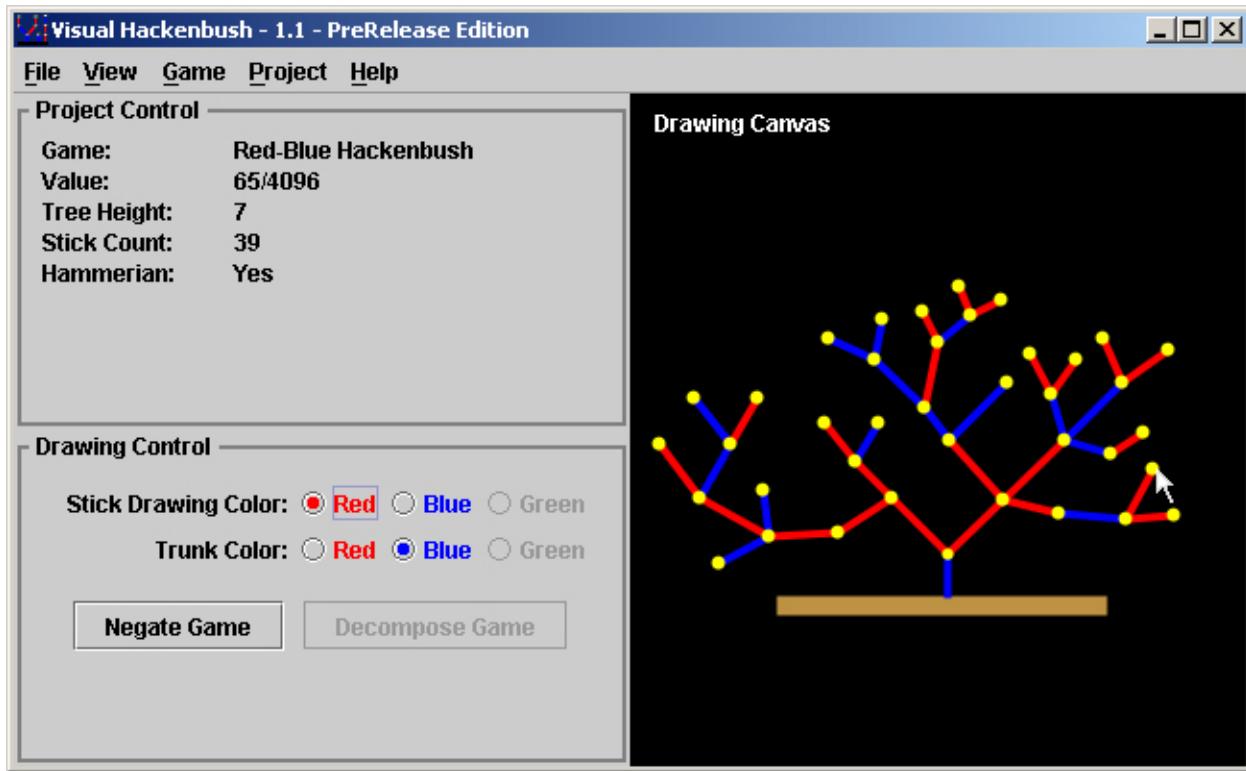
Save Log:

Allows the user to save the log to a file. The name of the log file is retrieved from the “Filename” text field found in the Project Settings Frame.

Clear:

Allows the user to clear the current log. Note that this *cannot* be undone!

Visual Hackenbush Functionality



Drawing Canvas Revisited:

The Drawing Canvas is equipped with MouseEvent Listeners, which means that any mouse movement, click, or dragging performed over the panel is recorded and tested to determine if the user is attempting to modify the tree.

In fact, mouse clicks and dragging are the main methods by which the user can construct a Hackenbush Tree in the Drawing Canvas, as well as the feature which sets Visual Hackenbush apart from other Combinatorial Games Software.

Since this is the most essential part of understanding Visual Hackenbush, a simple, tutorial-like description is provided below to introduce each piece of the Drawing Canvas Graphical User Interface.

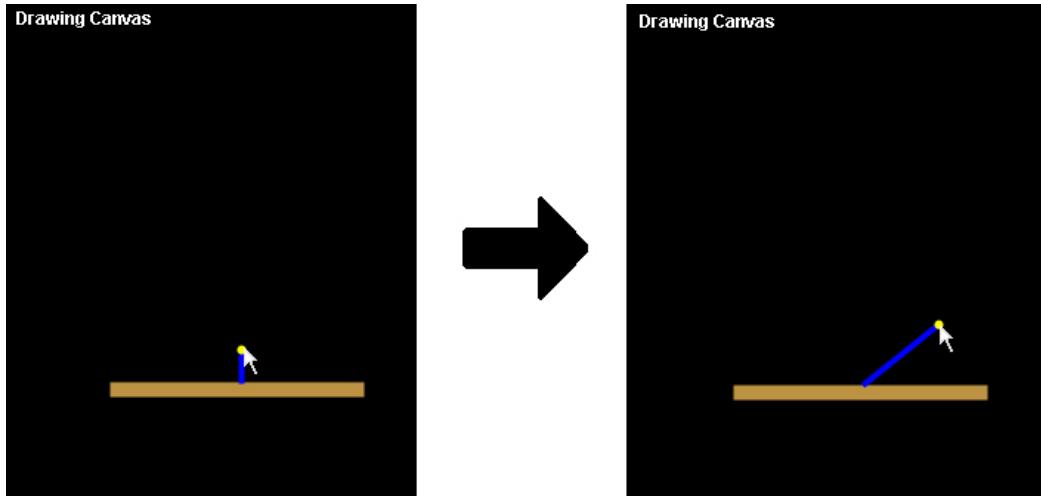
Visual Hackenbush Functionality [Continued]

Moving Vertices:

The Drawing Canvas can easily get cluttered with vertices and although it does not alter the value or underlying structure of the Hackenbush Tree the user has the option of adjusting the position of the stick connection vertices.

LEFT MOUSE CLICK on the desired connection vertex and **DRAG** the cursor (and vertex) to a new position.

You should notice that the sticks will stay connected to the vertices as they are moved around. Sticks may be stretched or shrunk at the user's whim, yet these actions will not modify the combinatorial game value of the tree.



CAUTION: It is possible to drag a connection vertex on top of another connection vertex, which means that the tree may look connected at that one vertex when it really is not. A quick way to double check the integrity of a vertex is by simply "wiggling" it by dragging it back and forth to reconfirm that the sticks that appear connected to it are actually connected. One could also activate the **Display Vertex Ids** option in the Project Settings window to be extra cautious.

Another complexity which may result in careless vertex positioning is the instance where two (or more) sticks lie on top of each other and as a result are disguised as a single stick. If you become unsure of your display and suspect multiplicity in your sticks, you should first count the sticks displayed and compare this with the **Stick Count** in the Project Control Display Panel or activate the **Display Vertex Ids** option in the Project Settings window rather than manually checking each connection vertex.

Visual Hackenbush Functionality [Continued]

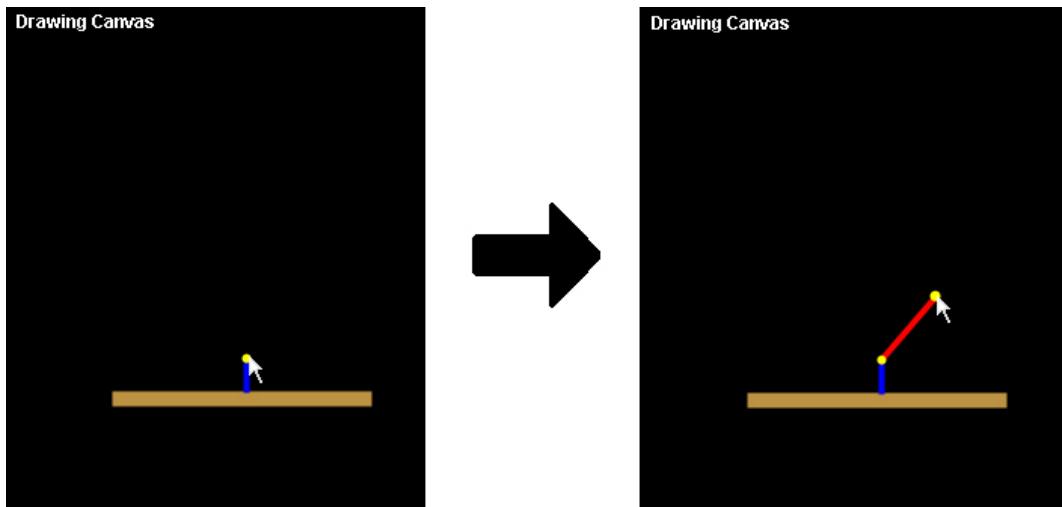
Adding Sticks to the Tree:

Drawing a Hackenbush Tree on a piece of paper or a whiteboard may appear as simple as drawing a bunch of connected lines, yet, in reality, there is quite a bit of data required to properly construct any tree-like structure. Imagine attempting to describe a particular Hackenbush Tree to someone without the use of any visual aids. Take it one step further: ask them to calculate its value based on the description you just gave them. For any tree larger than a few sticks, this can prove to be quite a difficult task! Thankfully, Visual Hackenbush is the solution!

In fact, thanks to its graphical interface adding a new stick to the Hackenbush Tree is quite easy:

RIGHT MOUSE CLICK on any connection vertex and **DRAG** the cursor (and vertex) to a desired position.

By this action, Visual Hackenbush will create a new connection vertex on top of the original and position it where the user releases the right mouse button. You will notice that the new stick will appear as you drag the new connection vertex away from the original. Also notice that the color of this new stick is the currently selected **Stick Drawing Color** in the Drawing Control Panel.



SPECIAL NOTE: All of the details concerning the creation of the new stick are recorded in the Visual Hackenbush Log (provided that it is enabled).

Visual Hackenbush Functionality [Continued]

CAUTION: *It is possible to create a connection vertex by accident! This occurs if the user performs a right click and releases on top of the original vertex position, resulting in a stick length which is obstructed by its connection vertex. Thus, the graphical user interface will process this as the command to create a new stick. To avoid this scenario, the user should always drag out the connection vertex to a reasonable distance away from the original.*

If you do in fact create an unwanted stick: don't panic, you can always remove it to resume the previous tree structure. See the section on "Removing Sticks from the Tree" for details.

Visual Hackenbush Functionality [Continued]

Modifying Stick Colors:

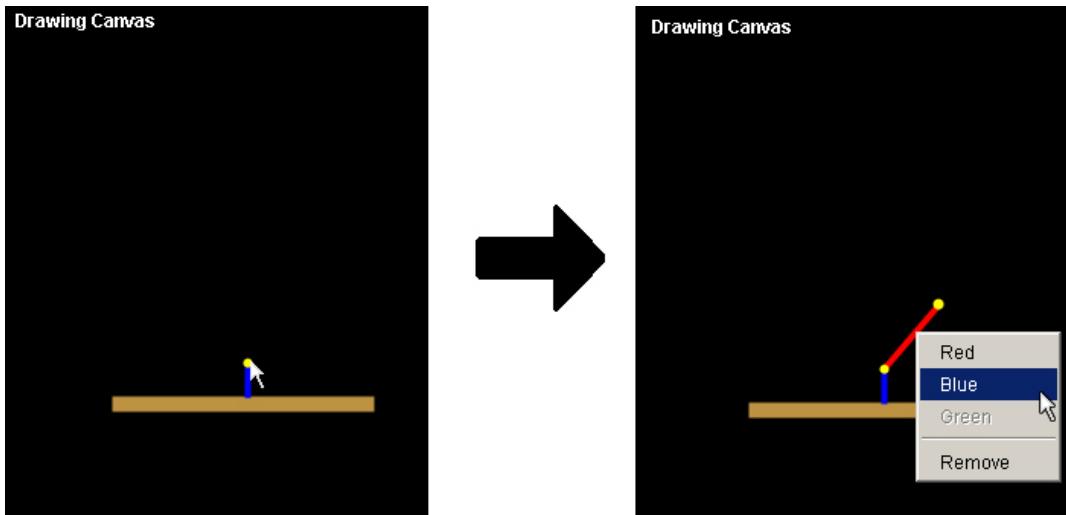
Of course, your Hackenbush Tree would be quite boring if you only drew red sticks on top of that blue trunk. After all, the game is Red-Blue Hackenbush. There are two ways in which the user may change stick colors:

- 1) Adjust the **Stick Drawing Color** BEFORE drawing the stick in the Drawing Control Panel.
- 2) Modify the stick color AFTER it is already drawn by the method below.

The former is pretty self-explanatory so we examine the latter in consequence. To change the color of a stick that has already been grafted into the current tree:

RIGHT MOUSE CLICK on the desired stick and **SELECT** the new color for the stick from the popup menu.

This action will not only change the appearance of the stick of the tree in the Drawing Canvas, but also the combinatorial game value of the tree. If the **Auto-Calculate** option is active the value displayed in the Project Controls Display Panel will change automatically to compensate for the modification.



SPECIAL NOTE: Only the colors available to the game type will be available to choose from in the popup menu. This prevents an unknown stick color from entering into a game where its character is unknown.

IMPORTANT: There is no popup menu for the trunk of the tree. You must use the Drawing Controls Panel to modify the trunk color.

Visual Hackenbush Functionality [Continued]

Removing Sticks from the Tree:

Finally, there is the issue of removing any unwanted sticks or subtrees from the current tree drawn in the Drawing Canvas. You may wish to do this because you accidentally added a stick where you did not want to, or maybe want to calculate the value of tree if a player decides to hack a particular stick.

For whatever reason you may have for its use, the method by which you remove a stick is similar to the way you would modify the color:

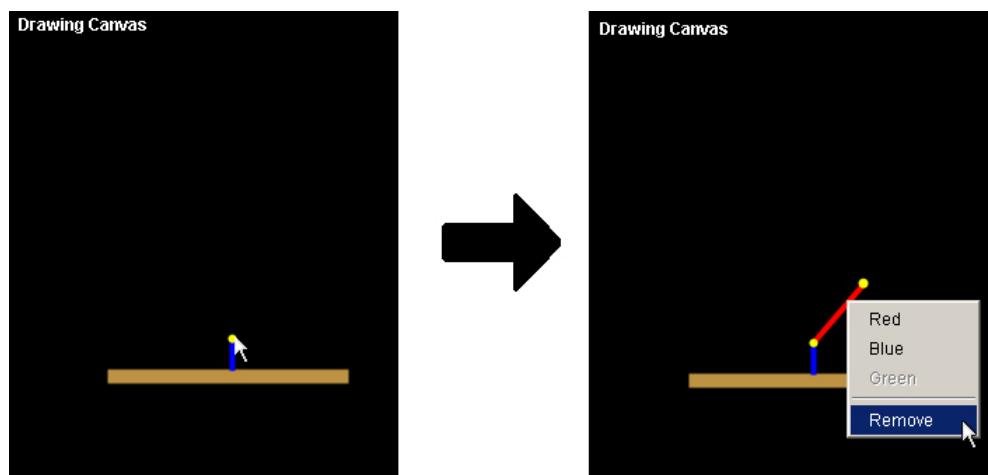
RIGHT MOUSE CLICK on the desired stick and
SELECT REMOVE from the popup menu.

Take note of the following bullets:

- There is NO UNDO feature as of version 1.1, so make sure the stick you are selecting to remove is really the stick you want to remove!
- Removing a stick obeys the same rules laid out in Hackenbush:
"If a stick is no longer (indirectly) connected to the ground after the removal of a stick, it too must fall away."

Therefore, if the stick you are about to remove is the parent to a subtree then the subtree will also be removed as a result of the removal.

NOTE: If you wish to examine which sticks were removed, you can always consult the log (provided it was active at the time of the removal).



IMPORTANT: You CANNOT remove the trunk of the tree.

Software

Combinatorial Games Suite (CG Suite). Aaron Siegel, 2003.

C Stalks Calculator (CSC). Joey Hammer, 2003.

Hammerian Trees Algorithm (HTA). Joey Hammer, 2003.

Hackenbush Applet. Joey Hammer, 2004.

Visual Hackenbush 1.1. Joey Hammer, 2004.

References

- Carroll, Lewis. *Through the Looking Glass*.
- Conway, J. *On Numbers and Games*. Second Edition.
- Conway, J. Berlekamp, E., Guy, R. *Winning Ways*. Vol. 1.
- Haff, L. R. *Course Notes for Math 168A: Combinatorial Game Theory*.