The Tricolorability Game

James McKeown

7/28/2011

Abstract

Rules: Players take turns coloring the strands of a knot projection. The winner is the last person able to color a strand. Crossings of a knot are always comprised of three strands; one over and two under not necessarily distinct. Completed crossings must contain strands all one color, or all different colors. A strand cannot be colored if it breaks this property.

Since this is a two-player finite game of perfect information with no ties or draws, one of the two players has a winning strategy. This is to say that either the first player can force a win (from here on referred to as $P1$), or the 2nd player can force a win (from here on referred to as $P2$).

This paper is interested in how such a winning strategy can be found for any knot projection with n crossings. The method is rather brute force.

Introduction to the Tricolorability Game

Rules: Players take turns coloring the strands of a knot projection. The winner is the last person able to color a strand. Crossings of a knot are always comprised of three strands; one over and two under not necessarily distinct. Completed crossings must contain strands all one color, or all different colors. A strand cannot be colored if it breaks this property.

Since this is a two-player finite game of perfect information with no ties or draws, one of the two players has a winning strategy. This is to say that either the first player can force a win (from here on referred to as $P1$), or the 2nd player can force a win (from here on referred to as $P2$). To find such a strategy we will find all maximal knot colorings and use them to construct a game tree. We will then prune our game tree from the point of view of Player 1 to see if he has a winning strategy.

Part I

Finding Complete Games.

Definition 1 *A complete game or endgame is a maximal coloring of a knot.*

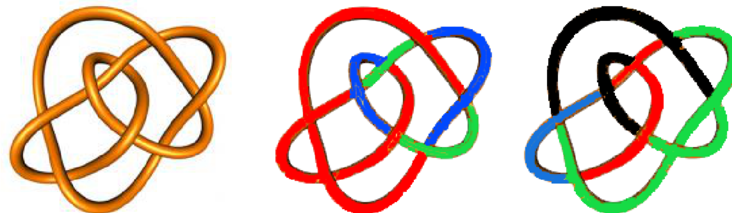

Figure 1: A knot and two complete games on that knot

The Methodology:

1. Label the strands of a projection starting at 0.
2. Represent crossings as sets of three strands.
   Note that if a game has more than one crossing with the same unordered set representation, it suffices to consider this set only once. (Think about the trefoil and you will soon see this.) This notion of *reduced set representation* will be important later.
3. Color strands all possible ways, allowing for uncolored strands. (4^n possible ways)

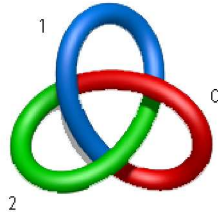

Figure 2: The trefoil has *set representation* $[(0,1,2),(0,1,2),(0,1,2)]$ and *reduced set representation*$[(0,1,2)]$

4. Check crossings to see if the colorings are valid. Throw out invalid colorings. Given a coloring k we check to make sure that for all crossings we either have

   (a) the same color on all three strands

   (b) a different color on all three strands

   if this fails, then coloring k is invalid and can be thrown out.

5. Check if the valid games are complete. Find a blank strand and color it red. Check to see if the resulting gamestate is valid (as in step 4).

   If it is colorable the game is incomplete.

   Otherwise color the strand green or blue and again check for validity.

   If none of the uncolored strands can be colored the game is complete.

We now have all completed games.

Lemma 1 *If all complete games on a knot projection have an odd number of colored strands the game is $P1$ Likewise if all complete games on a knot projection have an even number of colored strands the game is $P2$*

Proof. If all complete games have an odd number of colored strands then Player 1 will necessarily be the last player to color making him the winner. Likewise, if all complete games have an even number of colored strands then Player 2 will necessarily be the last player to color, making her the winner. □

Using this lemma we can find that the trefoil, figure 2, is $P1$ since all of its complete games have three colored strands. Very few games are trivial enough to use this theorem. In general it will be more complicated to find which player has a winning strategy given a set of complete games.

Part II

Creating a Winning Strategy from The Set of Completed Games

We will construct a game tree and use alpha-beta pruning to find which player has a winning strategy. This is how many computerized tic-tac-toe and chess games develop strategy.

1. Since on his first move Player 1 can color any strand any color, let us arrange the complete games by fixing a single strand. There are $3n$ first moves Player 1 can make by coloring any of the n strands one of three colors. But without loss of generality Player 1 only has n first moves. (Coloring the first strand red will have the same outcome as coloring it blue or green under perfect play.) We take our set of completed games C and create n subsets $S_0, S_1, S_2, ..., S_n$based on single shared red strands. These sets are not disjoint.

   For example, for $S_1 = \{x \in C \mid strand\#1\, of\, x\, is\, red\ \}$and $S_2 = \{x \in C \mid strand\#2\, of\, x\, is\, red\ \}$ $S_1 \cap S_2 = \{x \in C \mid strand\#1\, and\, strand\#2\, are\, red\ \}$which need not be null.

2. We now form subsets of $S_0, S_1, S_2, ..., S_n$ by fixing any second strand.

   Note that this step is more complex.

   For one, we have lost generality. It may matter how we color the 2nd strand. We do retain some generality, however. We could just consider if the second strand's color to be red r or not red $\neg r$.

For example, here are the subsets of S_0

$$\begin{aligned}
S_{0,1r} &= \{x \in S_0 \mid strand\#1\, of\, x\, is\, red\ \} \\
S_{0,1\neg r} &= \{x \in S_0 \mid strand\#2\, of\, x\, is\, notred\ \} \\
S_{0,2r} &= \{x \in S_0 \mid strand\#2\, of\, x\, is\, red\ \} \\
S_{0,2\neg r} &= \{x \in S_0 \mid strand\#3\, of\, x\, is\, notred\ \} \\
&\vdots \\
S_{0,nr} &= \{x \in S_0 \mid strand\#n\, of\, x\, is\, red\ \} \\
S_{0,n\neg r} &= \{x \in S_0 \mid strand\#n\, of\, x\, is\, notred\ \}
\end{aligned}$$

3. Form subsets of these new sets by fixing another strand. We now have no generality left to exploit. Continue on in this fashion until every new set formed is empty.

   Definition 2 *A set with k subscripts is a member of the $kth - generation$.*

   This is where our problem blows up. At each new generation, we can have $3n$ times as many sets as the last generation.

   Fortunately, there can be no more than n generations (we can't fix more strands than we have).

4. Construct a tree diagram where the sets from the previous steps act as vertices.

   Set$S_{a,b,\ldots,x,y}$ from the $ith - generation$ is connected by an edge
   to set $S_{a,b,\ldots,x}$ from the $i-1th - generation$ if $S_{a,b,\ldots,x,y} \subseteq S_{a,b,\ldots,x}$.

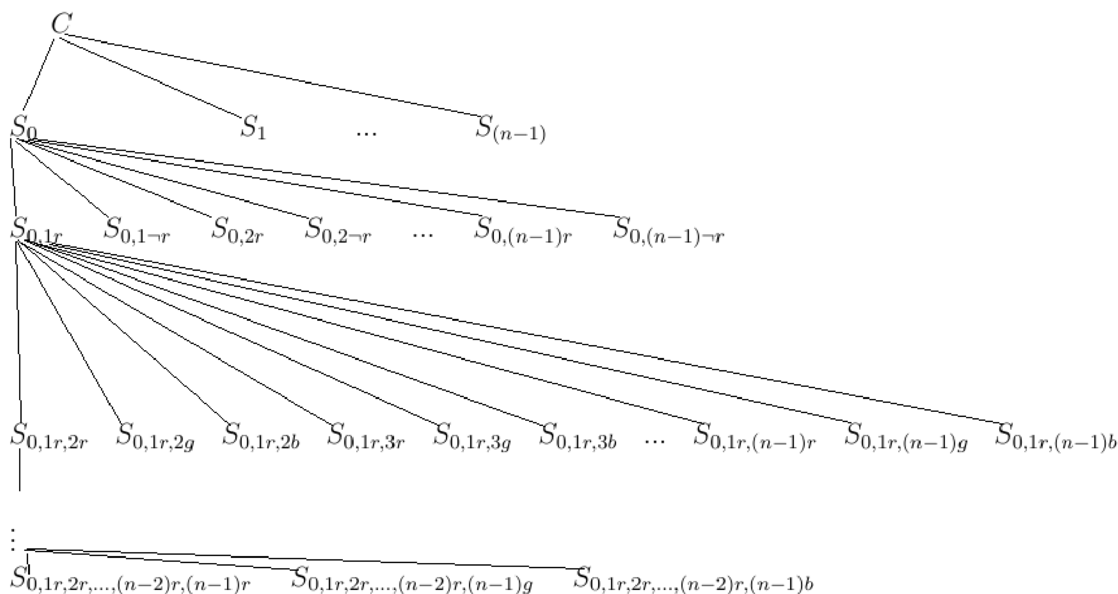

Figure 3: Vertices in this tree are sets of complete games with fixed strands.

This tree is at most n generations deep but its bottom could be $n(2n)(3n)^{n-2}$sets wide! Fortunately, many of these sets are empty.

We can now begin to refer to these sets as nodes, or vertices in the tree.
We will look at this tree much as one would look at a family tree.

Definition 3 *Call a node x a parent of y if it is one generation above y and there exists and edge between*

Definition 4 *y is a child of x iff x is a parent of y*

Definition 5 *Call node b a sibling of node a if a and b have the same parent.*

Definition 6 *Call a node a $leaf$ if it has no children.*

The Tree Game

Rules: Players take turns picking a direction to travel down the game tree. The first player to reach a dead end (a leaf) wins.

Lemma 2 *Playing the Tricolorability game is then equivalent to this new **Tree Game**.*

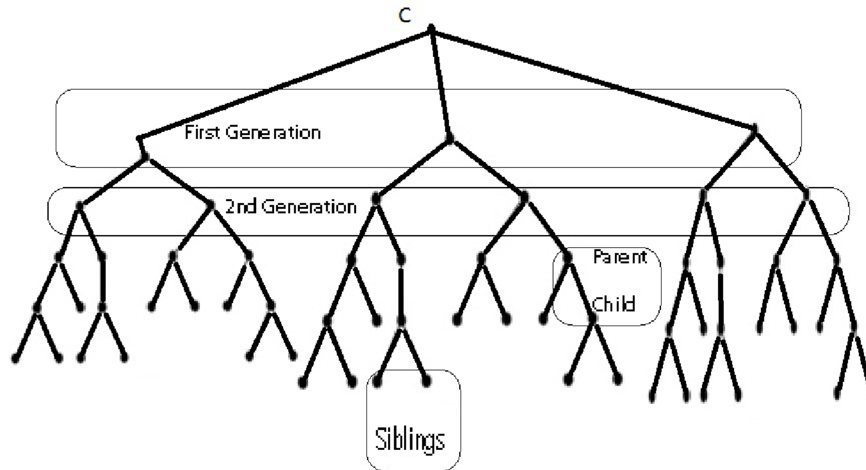

Figure 4: Tree Terminology

Proof. This statement is logically equivalent to saying that there is a bijective mapping from any game play on a knot to any path in this tree starting at C. This is easy to see. If in the Tricolorability Game as a first move Player 1 colors strand one blue, this action maps to Player 1 moving from C to S_1 and from there on treating "blue" in the Tricolorability Game as "red" in the Tree Game and viceversa. Then if Player 2 colors strand two red this maps to moving from S_1 to $S_1, 2b$. If a strand cannot be colored, this corresponds to an empty set and a nonexistent node in the tree. So we see that this map is bijective. □

We will now alter our tree to help see if player 1 has a winning strategy.

Definition 7 *A player's gameplan-tree is the subtree of the tree game connected to C (the 0 generation node) with all detrimental edges removed.*

Definition 8 *A detrimental edge for Player 1 is an edge that if followed from an even generation to an odd generation will cause him to lose.*

Constructing Player 1's Gameplan-Tree

Player 1 wants to avoid branches that end on even generations. Consider a leaf v from an even generation. Player 1 will lose if he goes to v's parent node. Thus the edge e between v's grandparent and parent is *detrimental*. Consequentally, the whole subgraph in red is no longer a part of player 1's $gameplan - tree$. Notice that v's grandparent is also of even generation. If it has no other children we call it a pseudo-leaf and treat it as we did the last leaf. Continue on in this fashion until there are no leaves or pseudo-leaves remaining. The resulting tree is our $gameplan - tree$.

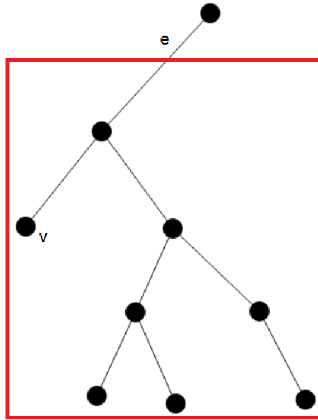

Figure 5: The boxed in area is removed from Player 1's $gameplan - tree$

Lemma 3 *Player 2 can never exit Player 1's* $gameplan - tree$.

Proof.
Player 1 always moves from an even generation to an odd generation.
Player 2 always moves from an odd generation to an even generation.
By constructing Player 1's $gameplan - tree$ only edges from even generation to odd generation were removed from the game tree.
Player 1 only limited his own moves, not those of Player 2. □

Theorem 1 *A game is $P1$ iff Player 1's $gameplan - tree$ has an odd endgame.*

$\leftarrow$ **Proof.**

If Player 1's $gameplan - tree$ has an odd endgame then the game is $P1$. Say Player 1's $gameplan - tree$ has an odd endgame and the game was $P2$.

Then by Lemma 3, Player 2 could win without going outside of Player 1's $gameplan - tree$.

But, by nature of its construction, there are no even generation endgames in Player 1's $gameplan - tree$. $\rightarrow\leftarrow$

$\rightarrow$ **Proof.**

If the game is $P1$ then Player 1's $gameplan - tree$ has an odd endgame.

Contrapositive: If Player 1's $gameplan - tree$ does not have an odd endgame then the game is $P2$.

If Player 1's $gameplan - tree$ does not have an odd endgame by virtue of being a $gameplan - tree$, it has no endgames at all!

Player 1 will be forced to move outside of $gameplan - tree$ and as soon as he does, he has taken a *detremental* edge and must lose.$\square$

Theorem 2 *Games that have the same reduced set notation have identical strategy and outcomes.*

This was suggested back when we were constructing our algorithm to find completed games but we didn't have the tools to prove it until now.

Proof. As was discussed with the Trefoil example, it is apparent that expressing a knot k with set notation will yield the same set of complete games as expressing k in reduced set notation. This implies that they will produce the same game tree, which, as we have seen, has identical gameplay to the tricolorability game. $\square$

Another Approach

This method for finding a winning strategy will not require that we construct the whole tree, just the terminating nodes. But, we will need a new way to distinguish them.

Definition 9 *A gameplay is a complete game with added information about the order in which strands were colored.*

Each completed game with k colored strands actually represents $k!$ different gameplays.
For example here are the complete games and their corresponding gameplays for the trefoil.

Complete Game	Gameplay	Complete Game	Gameplay	···
111	012	123	0 1 2	···
	021		0 2 1	···
	120		10 2	···
	102		1 2 0	···
	201		2 1 0	···
	210		2 0 1	···

We will now take a set of all gameplays and prune it down to see if Player 1 has a winning strategy. This pruning strategy is analogous to our old one. We have just changed notation. Our nodes are no longer sets. In fact, we will not be storing all of the nodes of our tree- just the leaves, each represented by a single gameplay.

Pruning the Set of Gameplays

1. Create a set of all gameplays G from our set of complete games C.

   Notice that we don't actually need to consider all of G. We again have some generality. We only need to consider the set of gameplays G' where the first strand to be colored is red. And of the gameplays in G' we only need to consider the set of gameplays G'' where the second strand is either red or green (not red).

2. Find a gameplay g with an even number of colored strands. Uncolor the last (kth) strand and look for other gameplays with the identical strands up to $k-1$. These games correspond to the leaves that are lost when the gameplan tree is pruned in figure 5. Prune out those gameplays.

3. Uncolor another strand of g. (This corresponds to moving up to v's grandparent in figure 5). Look for other gameplays that share its colored beginning.

   (a) If you find some throw away g and go back to 2.

   (b) If you don't find any, repeat step 3.

Continue on like this until all gameplays with an even number of colored strands are removed. If there are any gameplays left, the game is $P1$. Otherwise it is $P2$.

Example: Consider a 5-twisted projection of the unknot.

Figure 6: an n-twist projection of the unknot

Complete Games	Gameplays
33301	⋮ 0124 ⋮
33333	⋮ 01234 ⋮
33302	⋮ 0124 ⋮
33011	⋮ 0143 ⋮
⋮	⋮

So first we pick a gameplay with an even number of colored strands, say 0124. (This corresponds to vertex v in figure 5.) Uncoloring the 4, we check to see if there are any other gameplays with first strands 012. (This corresponds to checking to see if v has any siblings and removing them and their children from the tree.) We see that 01234 and 0124 both do so we remove them from our set of gameplays. Then we uncolor the 2 and check for complete games with first strands 01. (This corresponds to checking to see if v has any aunts or uncles in our tree.) We see that 0143 falls into this set and since this set is non-empty, we remove0124 from our set of gameplays and search for another gameplay with an even number of colored strands.

Part III

Python Program to Find All Complete Games on Any Knot Projection

This Program takes the number of strands n, and crossing information in the form

$$[[(0,2,3),(1,3,4),(4,6,2),...]$$

as arguements. It returns all complete games as strings of length n with 0 corresponding to an uncolored strand, $1,2$, and 3 corresponding to red, green, and blue. It then returns the number of complete games with all strands colored, one strand uncolored, two strands uncolored... n strands uncolored(obviously 0). Finally, it returns the total number of complete games and the total number of valid games. This program was created in collaboration with John McKeown and its code can be downloaded from www.johnmckeown.t15.org

```
def base10toN(num,n):
    """Change a  to a base-n number.
    Up to base-36 is supported without special notation."""
    new_num_string=''
    current=num
    while current!=0:
        remainder=current%n
        if 36>remainder>9:
            remainder_string=num_rep[remainder]
        elif remainder>=36:
            remainder_string='('+str(remainder)+')'
        else:
            remainder_string=str(remainder)
        new_num_string=remainder_string+new_num_string
        current=current/n
    return new_num_string

def base4(num):
    return base10toN(num,4)

def correct_size(base4string,n):
    if (len(base4string)<n):
        base4string=('0'*(n-len(base4string)))+base4string
        return (base4string)
    else:
        return (base4string)


valid=[]



def validchk(x):
    k=len(crossings)
    i=0
    while (i<k):
        if (x[int(crossings[i][0])]=='0' or x[int(crossings[i][1])]=='0' or x[int(crossings[i][2])]=='0'):
```

```
            i=i+1
        else:
            if (x[int(crossings[i][0])]==x[int(crossings[i][1])] and
      x[int(crossings[i][1])]==x[int(crossings[i][2])] or
      x[int(crossings[i][0])]!=x[int(crossings[i][1])] and
      x[int(crossings[i][1])]!=x[int(crossings[i][2])] and
      x[int(crossings[i][0])]!=x[int(crossings[i][2])]):
                i=i+1
            else:
                return False
    valid.append(x)
    return True



complete=[]
def completechk(x):
    k=len(x)
    i=0
    while(i<k):
        if (x[i]=='0'):
            y=x
            mylist=list(y)
            mylist[i]='1'
            y=''.join(mylist)
            if(validchktwo(y)==True):
                return False
            else:
                mylist=list(y)
                mylist[i]='2'
                y=''.join(mylist)
                if(validchktwo(y)==True):
                    return False
                else:
                    mylist=list(y)
                    mylist[i]='3'
                    y=''.join(mylist)
                    if(validchktwo(y)==True):
                        return False
                    else:
                        i=i+1
        else:
            i=i+1
    complete.append(x)
    return True

def howManyZeros(string):
a=len(string)
i=0
numzeros=0
while(i<a):
if (string[i]=='0'):
numzeros=numzeros+1
i=i+1
else:
i=i+1
```

```
return numzeros


def validchktwo(x):
    k=len(crossings)
    i=0
    while (i<k):
        if (x[int(crossings[i][0])]=='0' or x[int(crossings[i][1])]=='0' or x[int(crossings[i][2])]=='0'):
            i=i+1
        else:
            if (x[int(crossings[i][0])]==x[int(crossings[i][1])] and
     x[int(crossings[i][1])]==x[int(crossings[i][2])] or
     x[int(crossings[i][0])]!=x[int(crossings[i][1])] and
     x[int(crossings[i][1])]!=x[int(crossings[i][2])] and
     x[int(crossings[i][0])]!=x[int(crossings[i][2])]):
                i=i+1
            else:
                return False
    return True

def XgamesYuncolored():
    numzeros=0
    while(numzeros<n):
numgames=0
     for y in complete:
if(howManyZeros(y)==numzeros):
numgames=numgames+1
print 'there are {} games with {} uncolored strands.'.format(numgames,numzeros)
numzeros=numzeros+1



while 1==1:
    valid=[]
    complete=[]
    n=input("How Many Strands: ")
    crossings=eval('input("Number your strands starting at 0 and give us the
 crossings in the following form [(0,2,3),(1,3,4),(4,6,2)]: ")')

    i=0
    while (i<(4**int(n))):
        validchk(correct_size(base4(i),int(n)))
        i=i+1

    for x in valid:
        completechk(x)


    print '\n\n'
    print complete
    print '\n\n'
    XgamesYuncolored()
    print '\n\n There are {} complete games on this knot projection. \n\n'.format(len(complete))
```

Computation and Future Research and Development

As we have discussed, game trees are very large. Even the standard trefoil projection from figure 2 has nine complete games and 54 leaves in its game tree. Because all of the leaves are of generation three, Theorem 1 quickly told us the trefoil was $P1$. We didn't even need to look at its game tree. But what if we did? We would have needed to consider around 100 sets. What about for more complex knots? How difficult is it to find all complete games? How many gameplays are there for a typical knot projection? Is our pruning method computationally reasonable to find a winning strategy?

As an experiment I used our program to find the number of complete games on an n-twist projection of the unknot.

3 strands	9 complete games
4 strands	21 complete games
5 strands	75 complete games
6 strands	195 complete games
7 strands	459 complete games
8 strands	1233 complete games
9 strands	3075 complete games

The complete games on the 9 strand projection took about 10 seconds for my computer to find. The complete games on the 10 strand projection filled all of my computer's memory and it couldn't finish the calculation. This is likely because our Python program does not store complete games and valid games as well as it could. A list of all valid games and all complete games is stored in RAM, not on a harddrive so we have limited space to use. Games are stored as strings which take up more space than other possible types. This could be improved.

As discussed earlier, each of these complete games with k colored strands represents $k!$ different gameplays, each corresponding to a single leaf in our tree. For our nine strand twist projection of the unknot

there are 3 games with 0 uncolored strands.
there are 42 games with 1 uncolored strands.
there are 906 games with 2 uncolored strands.
there are 2076 games with 3 uncolored strands.
there are 48 games with 4 uncolored strands.

There are thus $(9! * 3) + (8! * 42) + (7! * 906) + (6! * 2076) + (5! * 48) = 8,848,800$ gameplays. But given the generallity talked about in step 1 of our second method, we only need to consider a subset of these (namely the gameplays who have their first move coloring a strand red and their second move coloring a strand red or green). To give context, there are 1,073,741,824 bytes in a gigabyte. Hypothetically, if a single gameplay took up 20 bytes then it would take about 1.77 gb to store all gameplays for our 9-twist projection of the unknot. This isn't ideal but it isn't insurmountable. If someone wrote a program that converted complete games to gameplays and stored them efficiently, it would be possible to implement our second pruning method to determine which player has a winning strategy on such a knot. This is most straight forward continuation of our research.