Ryan Tate
30015498

NE and SPE tutorial

Do you want to be able to solve any decision tree for all its Nash Equilibria?

Do you want to be able to solve perfect information decision trees for a single Subgame Perfect Equilibrium?

This tutorial is to show how to use the functions I created for drawing decision trees and solving them for Nash Equilibria and Subgame Perfect Equilibrium. Currently other game theory packages solve specific types of games, but this will work for any game as long as you can figure out how to create a decision tree for it. This will give you a data frame with the equilibria and payoffs and will draw decision trees with those equilibria marked.

The first function, tree.node, is the one used to create the decision trees in R.

```
tree.node<-function(play=NA,node="start.node",act=NA,pay=NA,prob=NA,tree,remove.node=FALSE){
```

The inputs for this are play, node, act, pay, prob, tree, and remove.node. The input take this structure: (play=integer, node="character" or node=c("character", "character"), act= c("character", "character"), pay=c(numeric, numeric), prob=c(numeric, numeric), tree=game, remove.node=FALSE or remove.node=TRUE).

Play takes the number of the player who controls the node; if that player would be a nature node set play to 0 (this function does not accept player names). The default NA is used for nodes that no one controls.

Node is the name of the node which is the action that leads to that node. If you have a multi-node info set use c() to set the location of each node in the info set. The default start.node should only be used for the start node.

Act is used for decision nodes by using c() and inputting all the possible actions the player can choose from at that node. Use the default of NA for terminus nodes.

Pay is used the set the payoff of the terminus node when it is reached and again you use c() and list payoffs in player order. Use the default of NA for non-terminus nodes.

Prob takes c() frequencies between 0 and 1 with the order having to match the order of the actions they are reporting the probabilities of. The default of NA is used for all non-probability nodes.

Tree is the game that you are currently building. This is important because you can only add one node/info set at a time so in order to build most games you have to use this to add nodes to a game.

Ryan Tate
30015498

Remove.node should usually be false; only set it to true if there is a node you want to remove from your game. If that is the case all you need is the node's name and remove.node to be true. Note that is addition to removing nodes this function knows if a node already exist in a decision tree. If you input a node that already exists it will overwrite the old version of the node instead of having two nodes with the same name (i.e. if you want to change a node you don't have to remove it and re-add it; just change the values and the function with overwrite it for you).

```
game<-list(NA,NA,list(NA),list(NA),list(NA))
game<-tree.node(play=1,act=c("cooperate`1","defect`1"),tree=game)
game<-tree.node(play=2,node=c("cooperate`1","defect`1"),act=c("cooperate`2","defect`2"),tree=game)
game<-tree.node(node="cooperate`2~cooperate`1",pay=c(3,3),tree=game)
game<-tree.node(node="cooperate`2~defect`1",pay=c(5,0),tree=game)
game<-tree.node(node="defect`2~cooperate`1",pay=c(0,5),tree=game)
game<-tree.node(node="defect`2~defect`1",pay=c(1,1),tree=game)
```

Here is how to input the classic prisoner's dilemma. You can see how each line is adding a new node/info set to the game and that if a node does not have a defined feature it is left out and handled by the default value. There are a couple important things to note:

First, the first line is critical as it sets up the structure of the decision tree that is used in R; this line should not be changed (except 'game' can be changed if you want a different name for your game).

Next, all actions from different info sets must have different names. If you want actions to appear to have the same name, use '`' at the end of the action name followed by a unique tag. '`' will tell the other function to drop the end when presenting you with outputs.

Line 3 is an example of how to input an info set with more than one node. Note that when you do this it will print the new name for the actions that originate from each of the nodes. This is important because you must use those names when naming nodes that are produced by those actions as lines 4-7 do. The naming convention is 'action_a~node_A'. The '~' that's inserted in the name is very important for the code to understand this is an info set and thus '~' should not be used in another action or node names.

```
> game<-tree.node(play=2,node=c("cooperate`1","defect`1"),act=c("cooperate`2","defect`2"),tree=game)
[1] "cooperate`2~cooperate`1" "defect`2~cooperate`1"
[1] "cooperate`2~defect`1"  "defect`2~defect`1"
```

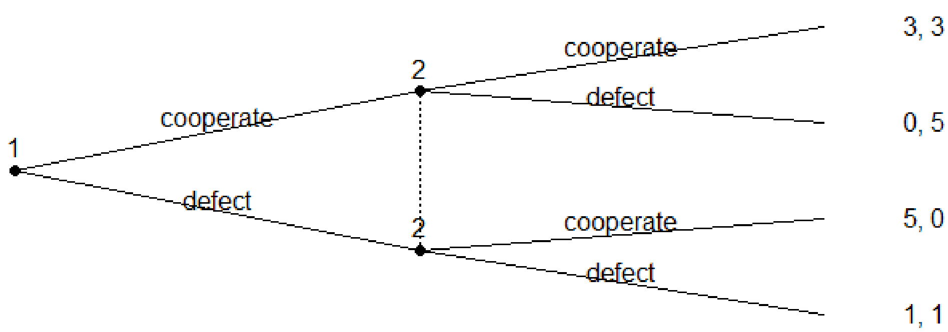
Lastly, the order in which you add nodes to the tree does not matter, but the output of some functions are in the order the nodes were input. This is purely aesthetic but I do recommend adding nodes in the order in which they are reached in the decision tree.

So to summarize, to input a decision tree, break the decision tree into its nodes/info sets. For each node/info set include all information that node controls and leave any information that is not a part of that node as the default value. Make sure your node and action names pair up and are different from all other pairs. And don't forget the first line that sets up the structure.

The next function, tree.draw, is the one used to draw the decision trees in R.

```
tree.draw<-function(tree,equilibrium=NA)
```

The only input you should use is tree run your game through this function. Equilibrium is only for other functions to pass the equilibrium they solved for to tree.draw. Because tree.node does not check to see if all parts of a tree connect or are complete (because of it node by node input), instead all the other function including this one have this check. In the cases where certain nodes don't connect or are not complete you will get an error message telling you only one of the offending nodes or actions.



Here is the prisoner's dilemma as depicted by tree.draw. Note that this function is not meant to produce final products but instead to give you a visual representation of the stored game to double check you inputted all your nodes correctly. The dotted line between the two player 2 controlled nodes is how this function represents shared info sets (Warning! the draw.tree function can currently only handle 2 nodes in 1 shared info set). While this function is not meant to produce beautiful decision tree, if you want to modify how far above the line or node the words appear, you can search the code for 'offset' because all text offsets have been commented with that word.

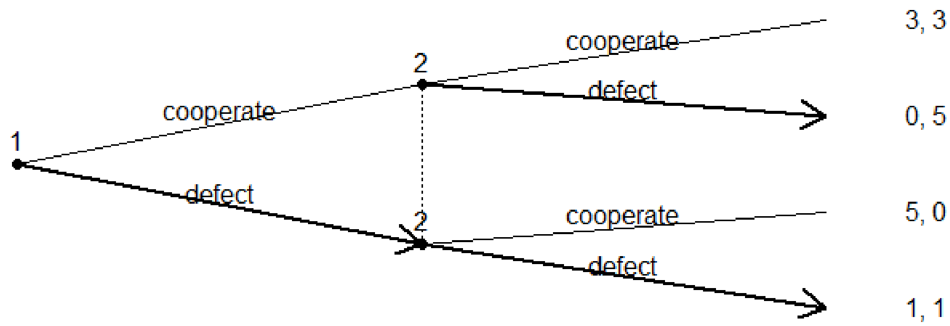
The first solution function, tree.solveNE, solves for Nash Equilibria (the bedrock of game theory) in R.

```
tree.solveNE<-function(tree){
```

It's only input is tree which takes any decision tree. This function outputs all pure Nash Equilibria for a given game. This output will be in the form of a data frame that shows the actions chosen by each player as well as the payoffs received for each Nash Equilibrium. Also it outputs using tree.draw decision trees showing all the Nash Equilibria with chosen action marked with arrows (there is one decision tree per Nash Equilibria). If no pure strategy Nash Equilibrium exists the function will simple print a message telling you so. Here is the output for the prisoner's dilemma.

Ryan Tate
30015498

```
> tree.solveNE(game)
       1      2 p1 p2
1 defect defect  1  1
```



The last function, tree.solveSPE, is used to solve for Subgame Perfection (a more logically refined Nash Equilibrium) in R.

```
tree.solveSPE<-function(tree)
```

This function's only input is the game you want solved, but this time it will refine the Nash Equilibrium down to its single Subgame Perfect Equilibrium. Because this is a more rigorous Equilibrium it has more restrictions on what games it can handle. It can't solve games with tied nodes and it currently can't solve games with imperfect information, in both cases it will print a message telling you the reason it couldn't solve the game. When it does solve the game this functions output is very similar to the tree.solveNE function. It will return a data frame with the action chosen by each player at SPE and their payoffs, along with a decision tree with the SPE actions marked with arrows. Note that because the prisoner's dilemma has imperfect information if you input it into the tree.solveSPE function you will get the following output.

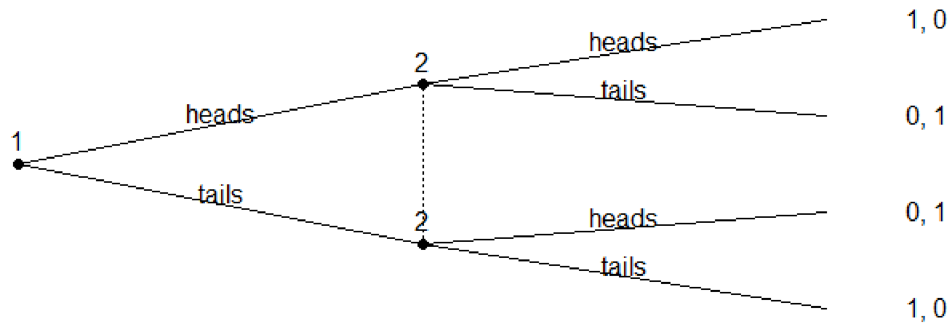
```
> tree.solveSPE(game)
[1] "There is imperfect information, can't solve for SPE"
NULL
```

This is end for the main part of the tutorial. It is not recommended for novices to change the function code, but if you wish to modify the code of these functions they have been commented to help give a vague idea about what each part of the code is doing. The codes are at the end of this tutorial and attached in another file. The rest of this tutorial shows how to input 4 more games as a demonstration for those that learn best by seeing examples. It's useful because it shows how the input change based on the type of game.

Ryan Tate
30015498

The first game is matching pennies.

```
game2<-list(NA,NA,list(NA),list(NA),list(NA))
game2<-tree.node(play=1,act=c("heads`1","tails`1"),tree=game2)
game2<-tree.node(play=2,node=c("heads`1","tails`1"),act=c("heads`2","tails`2"),tree=game2)
game2<-tree.node(node="heads`2~heads`1",pay=c(1,0),tree=game2)
game2<-tree.node(node="tails`2~heads`1",pay=c(0,1),tree=game2)
game2<-tree.node(node="heads`2~tails`1",pay=c(0,1),tree=game2)
game2<-tree.node(node="tails`2~tails`1",pay=c(1,0),tree=game2)
```



This game is very similar to the prisoner's dilemma. The main thing to note is because of the changes in payoffs when we run this game through tree.solveNE, we get the following output because there is no pure strategy Nash Equilibrium.

```
> tree.solveNE(game2)
[1] "No pure Nash Equilibrium"
NULL
```

The next game is the centipede game.

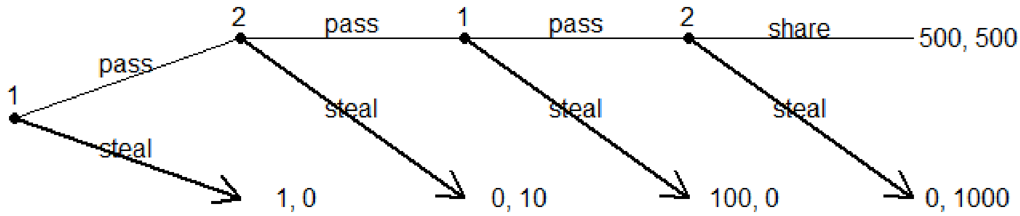
```
game3<-list(NA,NA,list(NA),list(NA),list(NA))
game3<-tree.node(play=1,act=c("pass`1","steal`1"),tree=game3)
game3<-tree.node(play=2,node="pass`1",act=c("pass`2","steal`2"),tree=game3)
game3<-tree.node(play=1,node="pass`2",act=c("pass`3","steal`3"),tree=game3)
game3<-tree.node(play=2,node="pass`3",act=c("share","steal`4"),tree=game3)
game3<-tree.node(node="steal`1",pay=c(1,0),tree=game3)
game3<-tree.node(node="steal`2",pay=c(0,10),tree=game3)
game3<-tree.node(node="steal`3",pay=c(100,0),tree=game3)
game3<-tree.node(node="share",pay=c(500,500),tree=game3)
game3<-tree.node(node="steal`4",pay=c(0,1000),tree=game3)
```

```
> tree.solveNE(game3)
      1     2     1     2 p1 p2
1 steal steal  pass share  1  0
2 steal steal steal share  1  0
3 steal steal  pass steal  1  0
4 steal steal steal steal  1  0
```

This game has perfect information and the actions are marked with `` because each player can choose the same action over and over. This game has multiple Nash Equilibria, but only one SPE.

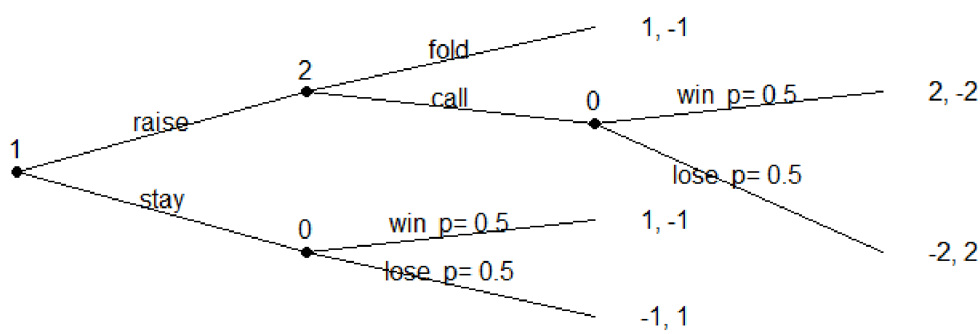
Ryan Tate
30015498

```
> tree.solveSPE(game3)
      1     2     1     2 p1 p2
1 steal steal steal steal  1  0
```



The next game is about poker.

```
game4<-list(NA,NA,list(NA),list(NA),list(NA))
game4<-tree.node(play=1,act=c("raise","stay"),tree=game4)
game4<-tree.node(play=2,node=c("raise"),act=c("fold","call"),tree=game4)
game4<-tree.node(play=0,node=c("stay"),act=c("win`stay","lose`stay"),prob=c(0.5,0.5),tree=game4)
game4<-tree.node(play=0,node=c("call"),act=c("win`call","lose`call"),prob=c(0.5,0.5),tree=game4)
game4<-tree.node(node="win`stay",pay=c(1,-1),tree=game4)
game4<-tree.node(node="lose`stay",pay=c(-1,1),tree=game4)
game4<-tree.node(node="fold",pay=c(1,-1),tree=game4)
game4<-tree.node(node="win`call",pay=c(2,-2),tree=game4)
game4<-tree.node(node="lose`call",pay=c(-2,2),tree=game4)
```



This game shows how to use nature nodes with probabilities controlling their decisions. It is important to note, that it may be tempting to represent nature with as single information set, but don't, since that will interfere with the functions abilities to understand the tree (nature is always considered omnipotent). While this game has two NE it can't be solved for a single SPE since there is a tied node.

```
> tree.solveNE(game4)
      1    2 p1 p2
1 raise call  0  0
2  stay call  0  0
```

```
> tree.solveSPE(game4)
[1] "There is a tie, can't solve for a single SPE"
NULL
```

Ryan Tate
30015498

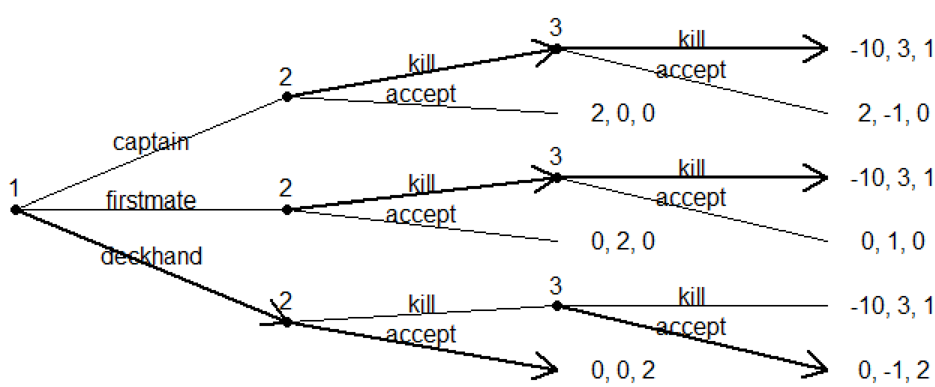
The last game is a simplifed Dread Pirate Nash game.

```
game5<-list(NA,NA,list(NA),list(NA),list(NA))
game5<-tree.node(play=1,act=c("captain","firstmate","deckhand"),tree=game5)
game5<-tree.node(play=2,node="captain",act=c("kill`c2","accept`c2"),tree=game5)
game5<-tree.node(play=2,node="firstmate",act=c("kill`f2","accept`f2"),tree=game5)
game5<-tree.node(play=2,node="deckhand",act=c("kill`d2","accept`d2"),tree=game5)
game5<-tree.node(play=3,node="kill`c2",act=c("kill`c3","accept`c3"),tree=game5)
game5<-tree.node(play=3,node="kill`f2",act=c("kill`f3","accept`f3"),tree=game5)
game5<-tree.node(play=3,node="kill`d2",act=c("kill`d3","accept`d3"),tree=game5)
game5<-tree.node(node="accept`c2",pay=c(2,0,0),tree=game5)
game5<-tree.node(node="accept`f2",pay=c(0,2,0),tree=game5)
game5<-tree.node(node="accept`d2",pay=c(0,0,2),tree=game5)
game5<-tree.node(node="kill`c3",pay=c(-10,3,1),tree=game5)
game5<-tree.node(node="accept`c3",pay=c(2,-1,0),tree=game5)
game5<-tree.node(node="kill`f3",pay=c(-10,3,1),tree=game5)
game5<-tree.node(node="accept`f3",pay=c(0,1,0),tree=game5)
game5<-tree.node(node="kill`d3",pay=c(-10,3,1),tree=game5)
game5<-tree.node(node="accept`d3",pay=c(0,-1,2),tree=game5)
```

This game is to show that while all the other games are ones that are easy to do by hand, it is possible to solve games that would take forever to do by hand. This game has 26 NE but only one SPE

```
> tree.solveNE(game5)
           1      2      2      2      3      3      3  p1 p2 p3
1    captain   kill   kill   kill   kill   kill   kill -10  3  1
2  firstmate   kill   kill   kill   kill   kill   kill -10  3  1
3    captain accept   kill   kill accept   kill   kill   2  0  0
4    captain accept accept   kill accept   kill   kill   2  0  0
5    captain accept   kill accept accept   kill   kill   2  0  0
6    captain accept accept accept accept   kill   kill   2  0  0
7  firstmate   kill accept   kill   kill accept   kill   0  2  0
8  firstmate   kill accept accept   kill accept   kill   0  2  0
9    captain accept   kill   kill accept accept   kill   2  0  0
10   captain accept accept   kill accept accept   kill   2  0  0
11   captain accept   kill accept accept accept   kill   2  0  0
12   captain accept accept accept accept accept   kill   2  0  0
13  deckhand   kill   kill accept   kill   kill accept   0  0  2
14  deckhand   kill accept accept   kill   kill accept   0  0  2
15   captain accept   kill   kill accept   kill accept   2  0  0
16   captain accept accept   kill accept   kill accept   2  0  0
17   captain accept   kill accept accept   kill accept   2  0  0
18   captain accept accept accept accept   kill accept   2  0  0
19 firstmate   kill accept   kill   kill accept accept   0  2  0
20  deckhand   kill   kill accept   kill accept accept   0  0  2
21 firstmate   kill accept accept   kill accept accept   0  2  0
22  deckhand   kill accept accept   kill accept accept   0  0  2
23   captain accept   kill   kill accept accept accept   2  0  0
24   captain accept accept   kill accept accept accept   2  0  0
25   captain accept   kill accept accept accept accept   2  0  0
26   captain accept accept accept accept accept accept   2  0  0

> tree.solveSPE(game5)
         1      2    2    2      3    3    3 p1  p2  p3
1 deckhand accept kill kill accept kill kill  0   0   2
```



Ryan Tate
30015498

You can see a wide variety of games in this tutorial but it is nothing compared to the infinite number of complex games these functions can solve. The next section goes over what large chunks of the function code do.

Function: tree.node

The first part is for removing nodes.

```
tree.node<-function(play=NA,node="start.node",act=NA,pay=NA,prob=NA,tree,remove.node=FALSE){ # inputs and defaults
  if(remove.node){ # checks to see if removing node
    for(node.i in 1:length(node)){ # cycles through nodes that are being removed
      if(length(which(tree[[2]]==node[node.i]))>0){ # checkes to see if those nodes exist
        tree[[1]][which(tree[[2]]==node[node.i])]<-NA # replaces all of the node's info with NA's
        tree[[3]][[which(tree[[2]]==node[node.i])]]<-NA
        tree[[4]][[which(tree[[2]]==node[node.i])]]<-NA
        tree[[5]][[which(tree[[2]]==node[node.i])]]<-NA
        tree[[2]][which(tree[[2]]==node[node.i])]<-NA
      }else{
        print(paste("Node",node[node.i],"does not exist",sep=" ")) # warning if node doesn't exist
      }
    }
```

The second part is for adding or changing nodes.

```
  }else{ # if adding nodes
    outcome<-NA
    for(node.i in 1:length(node)){ # cycles through nodes that are being added
      if(!is.na(act[1])){ # checks if node has actions
        if(length(node)>1){ # checks to see if more than one node is being added
          for(act.i in 1:length(act)){
            outcome[act.i]<-paste(act[act.i],node[node.i],sep="~") # creates unique action names for multi-node info sets
          }
          print(outcome) # prints unique actions so they can be used for the next node
        }else{
          outcome<-act # if only one node it transfers act to outcome
        }
      }
      if(length(which(tree[[2]]==node[node.i]))>0){ # if node already exist it will overwrite that node
        tree[[1]][which(tree[[2]]==node[node.i])]<-play
        tree[[3]][[which(tree[[2]]==node[node.i])]]<-outcome
        tree[[4]][[which(tree[[2]]==node[node.i])]]<-pay
        tree[[5]][[which(tree[[2]]==node[node.i])]]<-prob
      }else{
        if(length(which(is.na(tree[[2]])))>0){ # if a node is NA this will overwrite it
          tree[[1]][which(is.na(tree[[2]]))[1]]<-play
          tree[[3]][[which(is.na(tree[[2]]))[1]]]<-outcome
          tree[[4]][[which(is.na(tree[[2]]))[1]]]<-pay
          tree[[5]][[which(is.na(tree[[2]]))[1]]]<-prob
          tree[[2]][which(is.na(tree[[2]]))[1]]<-node[node.i]
        }else{
          tree[[1]][length(tree[[1]])+1]<-play # if there is no node to overwite this will create node to add to the list
          tree[[2]][length(tree[[1]])]<-node[node.i]
          tree[[3]][[length(tree[[1]])]]<-outcome
          tree[[4]][[length(tree[[1]])]]<-pay
          tree[[5]][[length(tree[[1]])]]<-prob
        }
      }
    }
  }
  return(tree) # returns the decision tree
}
```

Ryan Tate
30015498

Function: tree.draw

The first part checks to see if the function can work with this tree.

```
tree.draw<-function(tree,equilibrium=NA){ # draws a tree given a decision tree, equilibrium is used for solving function to pass solutions to this function
  for(node.i in 1:length(tree[[2]])){ # this checks to see your tree is complete
    if(!is.na(tree[[2]][node.i])){
      check.i<-0
      for(node.j in 1:length(tree[[2]])){
        if(!is.na(tree[[3]][[node.j]][1])){
          for(act in tree[[3]][[node.j]]){
            if(length(which(tree[[2]]==act))==0){ # checks that all actions lead to a corresponding node
              print(paste("ERROR! Action ",act," does not connect to a node"))
              return()

            }else{
              if(tree[[2]][node.i]==act) # checks that the node had a corespoding action
                check.i<-1
            }
          }
        }
      }
      if(check.i==0){
        if(tree[[2]][node.i]!="start.node"){ # checks that the node had a corespoding action, but makes an exception for the start.node
          print(paste("ERROR! Node ",tree[[2]][node.i]," does not connect to an action"))
          return()
        }
      }
    }
  }
```

The next part figures out the order of the nodes and stores important characteristic about the structure.

```
  order<-list("start.node")
  connections<-list(NA)
  info.set<-list(NA)
  round.i<-0
  while(round.i<length(order)){ # builds list for determining structure of the tree
    round.i<-round.i+1
    sequence.i<-1
    for(node.i in 1:length(order[[round.i]])){
      if(is.na(tree[[3]][[which(tree[[2]]==order[[round.i]][node.i])]][1])){ # this is for dead end nodes
        connections[[round.i]][node.i]<-NA
        info.set[[round.i+1]]<-NA
      }else{
        connections[[round.i]][node.i]<-length(tree[[3]][[which(tree[[2]]==order[[round.i]][node.i])]]) # this is for connecting node
        if(any(grepl("~",tree[[3]][[which(tree[[2]]==order[[round.i]][node.i])]]))){ # this is for marking information sets
          info.set[[round.i]][node.i]<-1
        }else{
        }
        if(length(order)==round.i){ # this is for making the next slice of the tree
          order[[round.i+1]]<-NA
          connections[[round.i+1]]<-NA
          info.set[[round.i+1]]<-NA
        }
        for(act in tree[[3]][[which(tree[[2]]==order[[round.i]][node.i])]]){ # this is for determing what nodes are in the next slice of the tree
          order[[round.i+1]][sequence.i]<-act
          sequence.i<-sequence.i+1
        }
      }
    }
  }
```

The last part actually plots the decision trees.

```
  plot(0,xlim=c(1,length(order)+0.5),ylim=c(1,0),col=0,axes=FALSE) # creates empty plot, note that the y axis is flipped
  info.set.i<-0
  for(round.i in 1:(length(order))){# cycles through slices of the tree
    segment.i<-1
    for(node.i in 1:length(order[[round.i]])){# cycles through nodes in a slice
      if(!is.na(connections[[round.i]][node.i])){ # node text and point
        text(x=round.i,y=node.i/(length(order[[round.i]])+1)-0.05,labels=tree[[1]][which(tree[[2]]==order[[round.i]][node.i])]) # -0.05 is offset
        points(x=round.i,y=node.i/(length(order[[round.i]])+1),pch=19)
        if(!is.na(info.set[[round.i]][node.i])){ # checks if part of info set
          if(info.set.i>0){ # checks if another info set has been found
            segments(x0=round.i,y0=node.i/(length(order[[round.i]])+1),x1=info.set.node[1],y1=info.set.node[2],lty=3) # connects nodes in info set
          }else{
            info.set.i<-1
            info.set.node<-c(round.i,node.i/(length(order[[round.i]])+1)) # stores info about node in info set
          }
        }
        for(act.i in 1:connections[[round.i]][node.i]){
          text.i<-strsplit(tree[[3]][[which(tree[[2]]==order[[round.i]][node.i])]][act.i],"~")
          if(length(which(equilibrium==text.i[[1]][1]))>0){
            arrows(x0=round.i,y0=node.i/(length(order[[round.i]])+1),x1=round.i+1,y1=segment.i/(length(order[[round.i+1]])+1),lwd=2) # arrow for equilibrium choice
          }else{
            segments(x0=round.i,y0=node.i/(length(order[[round.i]])+1),x1=round.i+1,y1=segment.i/(length(order[[round.i+1]])+1)) # default line
          }
          if(!is.na(tree[[5]][[which(tree[[2]]==order[[round.i]][node.i])]][act.i])){ # text for probabilities
            text.i<-strsplit(text.i[[1]][1],"`")
            text.i<-paste(text.i[[1]][1]," p=",tree[[5]][[which(tree[[2]]==order[[round.i]][node.i])]][act.i])
            text(x=(round.i+round.i+1)/2,y=(node.i/(length(order[[round.i]])+1)+segment.i/(length(order[[round.i+1]])+1))/2-0.025,labels=text.i) # -0.025 is offset
          }else{
            text.i<-strsplit(text.i[[1]][1],"`") # text for actions
            text(x=(round.i+round.i+1)/2,y=(node.i/(length(order[[round.i]])+1)+segment.i/(length(order[[round.i+1]])+1))/2-0.025,labels=text.i[[1]][1]) # -0.025 is offset
          }
          segment.i<-segment.i+1
        }
      }else{ # text for probabilities
        text(x=round.i+0.25,y=node.i/(length(order[[round.i]])+1),labels=paste(tree[[4]][[which(tree[[2]]==order[[round.i]][node.i])]],collapse=", ")) # +0.25 is offset
      }
    }
  }
}
```

Ryan Tate
30015498

Function: tree.solveNE

The first part checks to see if the function can work with this tree.

```
tree.solveNE<-function(tree){ # just requires decsion tree
  for(node.i in 1:length(tree[[2]])){ # this checks to see your tree is complete
    if(!is.na(tree[[2]][node.i])){
      check.i<-0
      for(node.j in 1:length(tree[[2]])){
        if(!is.na(tree[[3]][[node.j]][1])){
          for(act in tree[[3]][[node.j]]){
            if(length(which(tree[[2]]==act))==0){ # checks that all actions lead to a corresponding node
              print(paste("ERROR! Action ",act," does not connect to a node"))
              return()
            }else{
              if(tree[[2]][node.i]==act) # checks that the node had a corespoding action
                check.i<-1
            }
          }
        }
      }
      if(check.i==0){
        if(tree[[2]][node.i]!="start.node"){ # checks that the node had a corespoding action, but makes an exception for the start.node
          print(paste("ERROR! Node ",tree[[2]][node.i]," does not connect to an action"))
          return()
        }
      }
    }
  }
```

This part finds all possible combination of decisions possible

```
  node.act<-list(NA)
  node.act.i<-1
  imp.info<-NA
  for(node.i in 1:length(tree[[2]])){ # cycles through node
    if(is.na(tree[[5]][[node.i]][1])){ # is it a non-probability node?
      if(!is.na(tree[[3]][[node.i]][1])){ # does is have an action?
        if(any(grepl("~",tree[[3]][[node.i]]))){ # is it part of an info set
          act<-strsplit(tree[[3]][[node.i]][1],"~")
          if(!is.na(match(act[[1]][1],imp.info))){ # has the info set already been recorded?
            break
          }else{ # if not mark that info set has been recorded
            for(act.i in 1:length(tree[[3]][[node.i]])){
              act<-strsplit(tree[[3]][[node.i]][act.i],"~")
              imp.info<-c(imp.info,act[[1]][1])
            }
          }
        }
        node<-NA
        for(act.i in 1:length(tree[[3]][[node.i]])){ # record each action in node
          act<-strsplit(tree[[3]][[node.i]][act.i],"~")
          node[act.i]<-act[[1]][1]
        }
        node.act[[node.act.i]]<-node # store those actions, grouped by node
        node.act.i<-node.act.i+1
      }
    }
  }
  temp.table<-expand.grid(node.act,stringsAsFactors=FALSE)
  decision.table<-as.data.frame(matrix(data=NA,nrow=length(temp.table[,1]),ncol=length(temp.table[1,])+max(tree[[1]],na.rm=TRUE)+1))
  decision.table[1:length(temp.table[,1]),1:length(temp.table[1,])]<-temp.table # create table that has all possible decision combinations
```

This part finds the payoff for each combination of decisions.

```
  for(row.i in 1:length(decision.table[,1])){ # for each combination find out payoff
    temp.tree<-tree # so tree can be rest for each combination
    while(is.na(temp.tree[[4]][[which(temp.tree[[2]]=="start.node")]][1])){ # cycle while no payoff for combination
      for(node.i in 1:length(temp.tree[[2]])){ # cylces through nodes
        if(is.na(temp.tree[[4]][[node.i]][1])){ # does it have no payoff?
          if(!is.na(temp.tree[[3]][[node.i]][1])){ # does is have actions?
            if(!is.na(temp.tree[[5]][[node.i]][1])){ # is it a probability node
              for(act in temp.tree[[3]][[node.i]]){ # cycles through actions in node
                if(is.na(temp.tree[[4]][[which(temp.tree[[2]]==act)]][1])){ # checks to see if all actions have payoffs
                  break
                }
                if(act==temp.tree[[3]][[node.i]][length(temp.tree[[3]][[node.i]])]){ # if all actions have payoffs
                  temp.tree[[4]][[node.i]]<-0
                  for(prob.i in 1:length(temp.tree[[5]][[node.i]])){ # deterimines payoff based on probability of each action
                    temp.tree[[4]][[node.i]]<-temp.tree[[4]][[node.i]]+temp.tree[[4]][[which(temp.tree[[2]]==temp.tree[[3]][[node.i]][prob.i])]]*temp.tree[[5]][[node.i]][prob.i]
                  }
                }
              }
            }else{ # if not probability node
              for(act in temp.tree[[3]][[node.i]]){ # cycles through actions in node
                act.s<-strsplit(act,"~")
                if(!is.na(match(act.s[[1]][1],temp.table[row.i,]))){ # if action matches a decision then copy payoff from that action to this node
                  temp.tree[[4]][[node.i]]<-temp.tree[[4]][[which(temp.tree[[2]]==act)]]
                }
              }
            }
          }
        }
      }
    }
    decision.table[row.i,(length(temp.table)+1):(length(decision.table[1,])-1)]<-temp.tree[[4]][[which(temp.tree[[2]]=="start.node")]] # transfer decision payoff to table
  }
```

Ryan Tate
30015498

This next part finds the NE.

```
col.names<-NA
for(col.i in 1:length(temp.table)){ # cycles through columns that contain decisions
  for(node.i in 1:length(tree[[2]])){ # cycles through nodes
    if(!is.na(tree[[3]][[node.i]][1])){ # does the node have actions?
      act<-strsplit(tree[[3]][[node.i]],"~")
      if(!is.na(match(decision.table[1,col.i],act[[1]][1]))){ # is one of the actions in this column?
        col.names[col.i]<-tree[[1]][[node.i]] # determines player who controls that node's decisions
        break
      }
    }
  }
}
colnames(temp.table)<-col.names # rename decision column for temp.table with player who controls that node's decisions
for(row.i in 1:(length(decision.table[,1]))){ # checking to see if each row is a Nash Equilibrium
  for(row.j in 1:length(decision.table[,1])){ # cycle through row that it will be compared against
    if(max(tree[[1]],na.rm=TRUE)==1){ # checks to see if there is only one player
      decision.table[which(decision.table[,length(temp.table)+1]!=max(decision.table[,length(temp.table)+1])),length(decision.table[1,])]<-1 # if so mark row i if lower than row j
    }else{
      for(play.i in 1:max(tree[[1]],na.rm=TRUE)){ # cycles through players
        if(all(decision.table[row.i,which(colnames(temp.table)!=play.i)]==decision.table[row.j,which(colnames(temp.table)!=play.i)])){ # are all different action under this player's control
          if(max(decision.table[row.i,length(temp.table)+play.i],decision.table[row.j,length(temp.table)+play.i])!=decision.table[row.i,length(temp.table)+play.i]){ # is row i lower than row j for this player?
            decision.table[row.i,length(decision.table[1,])]<-1 # if so mark row i
          }
        }
      }
    }
  }
}
```

This last part makes the output more presentable and passes the equilibria to tree.draw.

```
  if(length(which(is.na(decision.table[,length(decision.table[1,])])))>0){ # are there any unmarked rows?
    for(NE.i in which(is.na(decision.table[,length(decision.table[1,])]))){ # cycle through all unmarked rows and pass equlibria to the draw function
      NE<-as.vector(decision.table[NE.i,1:(length(temp.table[1,]))])
      tree.draw(tree=tree,equilibrium=NE)
    }
    NE.table<-as.data.frame(matrix(data=NA,nrow=length(which(is.na(decision.table[,length(decision.table[1,])]))),ncol=length(decision.table[1,])-1))
    NE.table.i<-1 # creating a nicer Nash equilbrium table to return
    for(NE.i in which(is.na(decision.table[,length(decision.table[1,])]))){ # cycle through NE
      for(col.i in 1:length(temp.table)){
        text.i<-strsplit(decision.table[NE.i,col.i],"`") # this removes unique action markers that are unwanted in the output
        decision.table[NE.i,col.i]<-text.i[[1]][1]
      }
      NE.table[NE.table.i,]<-decision.table[NE.i,1:length(NE.table[1,])] # transfers NE from old table to new table
      NE.table.i<-NE.table.i+1
    }
    colnames(NE.table)<-c(col.names,paste("p",1:max(tree[[1]],na.rm=TRUE),sep="")) # rename decision column for NE.table with player who controls that node's decisions and recieve each payoff
    return(NE.table) # return all NE and their payoffs
  }else{ # if all rows are marked
    print("No pure Nash Equilibrium")
    return()
  }
}
```

Ryan Tate
30015498

Function: tree.solveSPE

The first part checks to see if the function can work with this tree.

```
tree.solveSPE<-function(tree){ # just requires decsion tree
  for(node.i in 1:length(tree[[2]])){ # this checks to see your tree is complete
    if(!is.na(tree[[2]][node.i])){
      check.i<-0
      for(node.j in 1:length(tree[[2]])){
        if(!is.na(tree[[3]][[node.j]][1])){
          for(act in tree[[3]][[node.j]]){
            if(length(which(tree[[2]]==act))==0){ # checks that all actions lead to a corresponding node
              print(paste("ERROR! Action ",act," does not connect to a node"))
              return()

            }else{
              if(tree[[2]][node.i]==act) # checks that the node had a corespoding action
                check.i<-1
            }
          }
        }
      }
      if(check.i==0){
        if(tree[[2]][node.i]!="start.node"){ # checks that the node had a corespoding action, but makes an exception for the start.node
          print(paste("ERROR! Node ",tree[[2]][node.i]," does not connect to an action"))
          return()
        }
      }
    }
  }
  if(any(grepl("~",tree[[2]]))){ # checks to see if there are any multi-node info sets
    print("There is imperfect information, can't solve for SPE")
    return()
  }
```

The next part solves for SPE.

```
  SPE<-list(NA,NA)
  SPE.i<-1
  while(is.na(tree[[4]][[which(tree[[2]]=="start.node")]][1])){ # cycles until there is a payoff for the start.node
    for(node.i in 1:length(tree[[2]])){ # cycles through nodes
      pay<-list(NA)
      pay.i<-1
      if(is.na(tree[[4]][[node.i]][1])){ # does this node have a payoff?
        if(!is.na(tree[[3]][[node.i]][1])){ # does this node have actions?
          for(act in tree[[3]][[node.i]]){ # cycles through actions
            if(is.na(tree[[4]][[which(tree[[2]]==act)]][1])){ # checks to see if all the node's actions have payoffs
              break
            }
            if(tree[[1]][node.i]>0){ # makes sure nature isn't controling this node
              pay[[1]][pay.i]<-tree[[4]][[which(tree[[2]]==act)]][tree[[1]][node.i]] # adds this node's player's payoff for each action
              pay.i<-pay.i+1
            }
            if(act==tree[[3]][[node.i]][length(tree[[3]][[node.i]])]){ # if all the node's actions have payoffs
              if(!is.na(tree[[5]][[node.i]][1])){ # check to see if probability node
                tree[[4]][[node.i]]<-0
                for(prob.i in 1:length(tree[[5]][[node.i]])){ # deterimines payoff based on probability of each action
                  tree[[4]][[node.i]]<-tree[[4]][[node.i]]+tree[[4]][[which(tree[[2]]==tree[[3]][[node.i]][prob.i])]]*tree[[5]][[node.i]][prob.i]
                }
              }else{
                if(length(which(pay[[1]]==max(pay[[1]])))>1){ # check to see if there is a tie
                  print("There is a tie, can't solve for a single SPE") # fuction can't handle ties
                  return()
                }else{ # otherwise compare node's player's payoff for each action to select which action player would do
                  tree[[4]][[node.i]]<-tree[[4]][[which(tree[[2]]==tree[[3]][[node.i]][which(pay[[1]]==max(pay[[1]]))])]] # copy that action's payoff to this node
                  SPE[[1]][SPE.i]<-tree[[3]][[node.i]][which(pay[[1]]==max(pay[[1]]))] # record action decided
                  SPE[[2]][SPE.i]<-tree[[1]][node.i] # record player that chose this action
                  SPE.i<-SPE.i+1
                }
              }
            }
          }
        }
      }
    }
  }
```

The last part makes the output more presentable and passes the equilibrium to tree.draw.

```
  tree.draw(tree=tree,equilibrium=SPE[[1]]) # pass decided action to draw function
  for(SPE.i in 1:length(SPE[[1]])){
    text.i<-strsplit(SPE[[1]][SPE.i],"`") # this removes unique action markers that are unwanted in the output
    SPE[[1]][SPE.i]<-text.i[[1]][1]
  }
  SPE.table<-as.data.frame(matrix(data=NA,nrow=1,ncol=length(SPE[[1]])+max(tree[[1]],na.rm=TRUE))) # create table for output
  SPE.table[1,length(SPE[[1]]):1]<-SPE[[1]] # adds decisions to table
  SPE.table[1,(length(SPE[[1]])+1):length(SPE.table[1,])]<-tree[[4]][[which(tree[[2]]=="start.node")]] # adds payoff
  colnames(SPE.table)<-c(SPE[[2]][length(SPE[[2]]):1],paste("p",1:max(tree[[1]],na.rm=TRUE),sep="")) # adds players who control each decision and recieve each payoff
  return(SPE.table) # return SPE and it's payoff
}
```