Node Examples

Christoph Glur

2015-02-15

Contents

About Trees

Trees are ubiquitious is mathematics, computer science, data sciences, finance, and in many other fields. Trees are useful always when we are facing hierarchies. Some examples where trees are useful:

- in decision theory (c.f. decision trees)
- in finance, e.g. to classify financial instruments into asset classes
- in routing algorithms
- in computer science and programming (e.g. binary search trees, xml)
- e.g. for family trees

R provides some tree-like structures on various low levels. For example, environments can be seen as nodes in a tree. Also, R provides various packages that deal with tree-like structures (especially in the area of decision theory). Yet, there is no high-level tree data structure that could be used equally conveniently and generically as, say, data.frame.

As a consequence, people often try to resolve hierarchical problems in a tabular fashion, for instance with data.frames (or in Excel sheets). But hierarchies don't marry with tables and various workarounds are usually required.

This package tries to offer an alternative. The tree package allows creating hierarchies by the use of the Node object. Node provides basic traversal and search operations. You can decorate Nodes with attributes and methods, by that extending the package to your needs.

Also, the package provides convenience methods to print trees nicely, and to convert it to a data.frame for integration with other packages.

Tree Creation

Let's start by creating a tree of nodes. In our example, we are looking at a company, Acme Inc., and the tree reflects its oranisational structure. The root (level 0) is the company. On level 1, the nodes represent departments, and the leaves of the tree represent projects the company considers for next year:

```r
library(ahp)
acme <- Node$new("Acme Inc.")
  accounting <- acme$AddChild("Accounting")
    software <- accounting$AddChild("New Software")
    standards <- accounting$AddChild("New Accounting Standards")
  research <- acme$AddChild("Research")
    newProductLine <- research$AddChild("New Product Line")
    newLabs <- research$AddChild("New Labs")
  it <- acme$AddChild("IT")
    outsource <- it$AddChild("Outsource")
    agile <- it$AddChild("Go agile")
    goToR <- it$AddChild("Switch to R")

print(acme)
```

```
##                       levelName
## 1  Acme Inc.
## 2  * Accounting
## 3  * * New Software
## 4  * * New Accounting Standards
## 5  * Research
## 6  * * New Product Line
## 7  * * New Labs
## 8  * IT
## 9  * * Outsource
## 10 * * Go agile
## 11 * * Switch to R
```

Custom Attributes

Now, let's associate some costs with the projects:

```r
software$cost <- 1000000
standards$cost <- 500000
newProductLine$cost <- 2000000
newLabs$cost <- 750000
outsource$cost <- 400000
agile$cost <- 250000
goToR$cost <- 50000
```

And some probabilities that the projecs will be executed in the next year:

```r
software$p <- 0.5
standards$p <- 0.75
newProductLine$p <- 0.25
newLabs$p <- 0.9
outsource$p <- 0.2
agile$p <- 0.05
goToR$p <- 1
```

Converting to data.frame

We can now convert the tree into a data.frame:

```r
acmedf <- as.data.frame(acme)
acmedf
```

```
##                        levelName
## 1  Acme Inc.
## 2  * Accounting
## 3  * * New Software
## 4  * * New Accounting Standards
## 5  * Research
## 6  * * New Product Line
## 7  * * New Labs
## 8  * IT
## 9  * * Outsource
## 10 * * Go agile
## 11 * * Switch to R
```

Adding the cost as a column to our data.frame is easy, by using the Get method. We'll explain the Get method in more detail below.

```
acmedf$level <- acme$Get("level")
acmedf$cost <- acme$Get("cost")
acmedf
```

```
##                        levelName level    cost
## 1  Acme Inc.                         0      NA
## 2  * Accounting                      1      NA
## 3  * * New Software                  2 1000000
## 4  * * New Accounting Standards      2  500000
## 5  * Research                        1      NA
## 6  * * New Product Line              2 2000000
## 7  * * New Labs                      2  750000
## 8  * IT                              1      NA
## 9  * * Outsource                     2  400000
## 10 * * Go agile                      2  250000
## 11 * * Switch to R                   2   50000
```

We could have achieved the same result in one go:

```
as.data.frame(acme, "level", "cost")
```

```
##                        levelName level    cost
## 1  Acme Inc.                         0      NA
## 2  * Accounting                      1      NA
## 3  * * New Software                  2 1000000
## 4  * * New Accounting Standards      2  500000
## 5  * Research                        1      NA
## 6  * * New Product Line              2 2000000
## 7  * * New Labs                      2  750000
## 8  * IT                              1      NA
## 9  * * Outsource                     2  400000
## 10 * * Go agile                      2  250000
## 11 * * Switch to R                   2   50000
```

Internally, the same is called when printing a tree:

```
print(acme, "level", "cost")
```

```
##                        levelName level    cost
## 1  Acme Inc.                         0      NA
## 2  * Accounting                      1      NA
## 3  * * New Software                  2 1000000
## 4  * * New Accounting Standards      2  500000
## 5  * Research                        1      NA
## 6  * * New Product Line              2 2000000
## 7  * * New Labs                      2  750000
## 8  * IT                              1      NA
## 9  * * Outsource                     2  400000
## 10 * * Go agile                      2  250000
## 11 * * Switch to R                   2   50000
```

Using Get when converting to data.frame and for printing

Above, we saw how we can add the name of an attribute to the ellipsis argument of the as.data.frame. However, we can also add the results of the Get method to the as.data.frame directly. This allows for example formatting the column in a specific way. Details of the Get method are explained in the next section.

```
as.data.frame(acme,
              "level",
              probability = acme$Get("p", format = FormatPercent)
             )
```

```
##                        levelName level probability
## 1  Acme Inc.                        0            
## 2  * Accounting                     1            
## 3  * * New Software                 2    50.00  %
## 4  * * New Accounting Standards     2    75.00  %
## 5  * Research                       1            
## 6  * * New Product Line             2    25.00  %
## 7  * * New Labs                     2    90.00  %
## 8  * IT                             1            
## 9  * * Outsource                    2    20.00  %
## 10 * * Go agile                     2     5.00  %
## 11 * * Switch to R                  2   100.00  %
```

Get Method

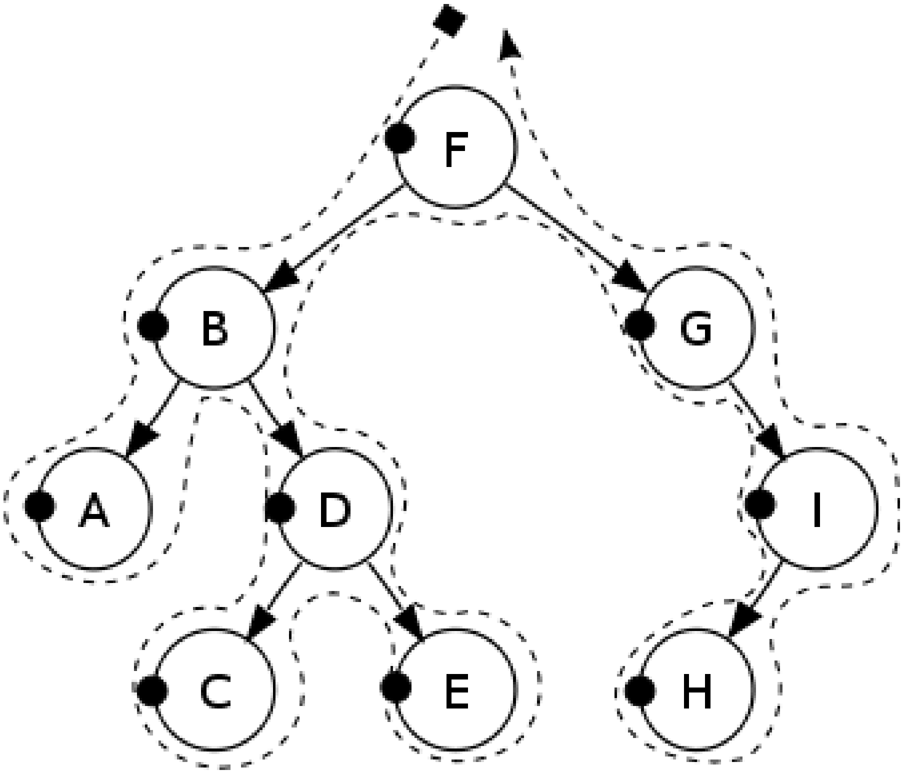
Tree traversal is one of the core concepts of trees. See for example here: http://en.wikipedia.org/wiki/Tree_traversal The Get Method traverses the tree, and collects values from each node. It then returns a vector containing the collected values.

Additional features of the Get Method are: * execute a function on each node, and append the function's result to the returned vector * execute a Node Method on each node, and append the Method's return value to the reuturned vector * assign the function or method return value to a Node's attribute

Traversal Order

The Get method can traverse the tree in various ways. This is called traversal order.

Pre-Order The default traversal mode is pre-order.



This is what is used e.g. in the as.data.frame method:

```
as.data.frame(acme, "level")
```

```
##                          levelName level
## 1  Acme Inc.                           0
## 2  * Accounting                        1
## 3  * * New Software                    2
## 4  * * New Accounting Standards        2
## 5  * Research                          1
## 6  * * New Product Line                2
## 7  * * New Labs                        2
## 8  * IT                                1
## 9  * * Outsource                       2
## 10 * * Go agile                        2
## 11 * * Switch to R                     2
```

```
data.frame(level = acme$Get('level'))
```

```
##                           level
## Acme Inc.                     0
## Accounting                    1
## New Software                  2
## New Accounting Standards      2
## Research                      1
## New Product Line              2
## New Labs                      2
## IT                            1
## Outsource                    2
## Go agile                     2
## Switch to R                  2
```

Post-Order The post-order traversal mode first returns children:

[post-order]](postorder.png)

We can use it like this on the Get method:

```
data.frame(level = acme$Get('level', traversal = "post-order"))
```

```
##                            level
## New Software                   2
## New Accounting Standards       2
## Accounting                     1
## New Product Line               2
## New Labs                       2
## Research                       1
## Outsource                      2
## Go agile                       2
## Switch to R                    2
## IT                             1
## Acme Inc.                      0
```

Ancestor This is a non-standard traversal mode that does not traverse the entire tree. Instead, the ancestor mode starts from a node, and then walks the tree along the path from ancestor to ancestor, up to the root:

```
data.frame(level = agile$Get('level', traversal = "ancestor"))
```

```
##             level
## Go agile        2
## IT              1
## Acme Inc.       0
```

Get using a function

Pass a function to the Get method You can pass a standard R function to the Get method. For example:

```
ExpectedCost <- function(node) {
  result <- node$cost * node$p
  if(length(result) == 0) result <- NA
  return (result)
}

data.frame(acme$Get(ExpectedCost))
```

```
##                            acme.Get.ExpectedCost.
## Acme Inc.                                      NA
## Accounting                                     NA
## New Software                             500000
## New Accounting Standards                 375000
## Research                                     NA
## New Product Line                         500000
## New Labs                                 675000
## IT                                           NA
## Outsource                                 80000
## Go agile                                  12500
## Switch to R                               50000
```

The requirement for the function (ExpectedCost in the above example) is that: * the first argument of the function is a Node * it needs to return a scalar

```
library(magrittr)
ExpectedCost <- function(node) {
  result <- node$cost * node$p
  if(length(result) == 0) {
    if (node$isLeaf) result <- NA
    else {
      node$children %>% sapply(ExpectedCost) %>% sum -> result
    }
  }
  return (result)
}

data.frame(ec = acme$Get(ExpectedCost))
```

Use Recursion

```
##                               ec
## Acme Inc.                2192500
## Accounting                875000
## New Software              500000
## New Accounting Standards  375000
## Research                 1175000
## New Product Line          500000
## New Labs                  675000
## IT                        142500
## Outsource                  80000
## Go agile                   12500
## Switch to R                50000
```

Add Parameters to the Passed Function The Traverse method accepts an ellipsis (...). Any additional parameters with which Traverse is called will be passed on the the ExpectedCost function: This allows us to make this more flexible:

```
ExpectedCost <- function(node, fun = sum) {
  result <- node$cost * node$p
  if(length(result) == 0) {
    if (node$isLeaf) result <- NA
    else {
      node$children %>% sapply(function(x) ExpectedCost(x, fun = fun)) %>% fun -> result
    }
  }
  return (result)
}

data.frame(ec = acme$Get(ExpectedCost, fun = mean))
```

```
##                              ec
## Acme Inc.                357500
## Accounting               437500
## New Software             500000
## New Accounting Standards 375000
## Research                 587500
## New Product Line         500000
## New Labs                 675000
## IT                        47500
## Outsource                 80000
## Go agile                  12500
## Switch to R               50000
```

Assigning values using Get

We can tell the Get method to assign the value to a specific attribute for each Node it traverses. This is especially useful if the attribute parameter is a function, as in the previous examples :

```
acme$Get(function(x) x$p * x$cost, assign = "expectedCost")
```

```
##                Acme Inc.               Accounting             New Software
##                       NA                       NA                   500000
## New Accounting Standards                 Research         New Product Line
##                   375000                       NA                   500000
##                 New Labs                       IT                Outsource
##                   675000                       NA                    80000
##                 Go agile              Switch to R
##                    12500                    50000
```

```
print(acme, "p", "cost", "expectedCost")
```

```
##                       levelName    p    cost expectedCost
## 1  Acme Inc.                      NA      NA           NA
## 2  * Accounting                   NA      NA           NA
## 3  * * New Software             0.50 1000000       500000
## 4  * * New Accounting Standards 0.75  500000       375000
## 5  * Research                     NA      NA           NA
## 6  * * New Product Line         0.25 2000000       500000
## 7  * * New Labs                 0.90  750000       675000
## 8  * IT                           NA      NA           NA
## 9  * * Outsource                0.20  400000        80000
## 10 * * Go agile                 0.05  250000        12500
## 11 * * Switch to R              1.00   50000        50000
```

Combine Assignment and Calculation

In the above Recursion example, we recurse for each node to all descendants straight to the leaf, by that repeating the same calculations various times.

We can avoid these repetitious calculations by piggy-backing on pre-calculated values. Obviously, this requires us to traverse the tree in post-order mode: We want to start calculating at the leafs, and then walk back towards the root.

In the following example, we calculate the average expected cost. As this depends now only of a node's children, and because we walk the tree in post-order mode, we can be sure that our children have the value calculated when we traverse the parent.

```
ExpectedCost <- function(node, variableName = "avgExpectedCost", fun = sum) {
  #if the "cache" is filled, I return it. This stops the recursion
  if(!is.null(node[[variableName]])) return (node[[variableName]])

  #otherwise, I calculate from my own properties
  result <- node$cost * node$p

  #if the properties are not set, I calculate the mean from my children
  if(length(result) == 0) {
    if (node$isLeaf) result <- NA
    else {
      node$children %>%
      sapply(function(x) ExpectedCost(x, variableName = variableName, fun = fun)) %>%
      fun -> result
    }
  }
  return (result)
}

acme$Get(ExpectedCost, fun = mean, traversal = "post-order", assign = "avgExpectedCost")
```

```
##             New Software New Accounting Standards               Accounting
##                   500000                   375000                   437500
##         New Product Line                 New Labs                 Research
##                   500000                   675000                   587500
##                Outsource                 Go agile              Switch to R
##                    80000                    12500                    50000
##                       IT                Acme Inc.
##                    47500                   357500
```

```
print(acme, "cost", "p", "avgExpectedCost")
```

```
##                          levelName    cost    p avgExpectedCost
## 1  Acme Inc.                            NA   NA          357500
## 2  * Accounting                         NA   NA          437500
## 3  * * New Software                1000000 0.50          500000
## 4  * * New Accounting Standards     500000 0.75          375000
## 5  * Research                           NA   NA          587500
## 6  * * New Product Line            2000000 0.25          500000
## 7  * * New Labs                     750000 0.90          675000
## 8  * IT                                 NA   NA           47500
## 9  * * Outsource                 400000 0.20           80000
## 10 * * Go agile                  250000 0.05           12500
## 11 * * Switch to R                50000 1.00           50000
```

Formatting Get

We can pass a formatting function to the Get method, which will convert the returned value to a human readable string for printing.

```
PrintMoney <- function(x) {
  format(x, digits=10, nsmall=2, decimal.mark=".", big.mark="'", scientific = FALSE)
}

as.data.frame(acme, cost = acme$Get("cost", format = PrintMoney))
```

```
##                       levelName         cost
## 1  Acme Inc.                              NA
## 2  * Accounting                           NA
## 3  * * New Software             1'000'000.00
## 4  * * New Accounting Standards   500'000.00
## 5  * Research                             NA
## 6  * * New Product Line         2'000'000.00
## 7  * * New Labs                   750'000.00
## 8  * IT                                   NA
## 9  * * Outsource                  400'000.00
## 10 * * Go agile                   250'000.00
## 11 * * Switch to R                 50'000.00
```

Note that the format is not used for assignment with the assign parameter, but only for the values returned by Get:

```
acme$Get("cost", format = PrintMoney, assign = "cost2")
```

```
##                Acme Inc.               Accounting             New Software
##                     "NA"                     "NA"           "1'000'000.00"
## New Accounting Standards                 Research         New Product Line
##             "500'000.00"                     "NA"           "2'000'000.00"
##                 New Labs                       IT                Outsource
##             "750'000.00"                     "NA"             "400'000.00"
##                 Go agile              Switch to R
##             "250'000.00"              "50'000.00"
```

```
as.data.frame(acme, cost = acme$Get("cost2"))
```

```
##                       levelName    cost
## 1  Acme Inc.                         NA
## 2  * Accounting                      NA
## 3  * * New Software             1000000
## 4  * * New Accounting Standards  500000
## 5  * Research                        NA
## 6  * * New Product Line         2000000
## 7   * * New Labs                     750000
## 8   * IT                                 NA
## 9   * * Outsource                    400000
## 10  * * Go agile                     250000
## 11  * * Switch to R                   50000
```

Set Method

The Set method is the counterpart to the Get method. It takes a vector or a single value as an input, and traverses the tree in a certain order. Each node is assigned a value from the vector.

Assigning Values

The same could be achieved with the Set method:

```
ec <- acme$Get(function(x) x$p * x$cost)
acme$Set("expectedCost", ec)
as.data.frame(acme, "p", "cost", "expectedCost")
```

```
##                          levelName    p    cost expectedCost
## 1  Acme Inc.                         NA      NA           NA
## 2  * Accounting                      NA      NA           NA
## 3  * * New Software                0.50 1000000       500000
## 4  * * New Accounting Standards 0.75  500000       375000
## 5  * Research                        NA      NA           NA
## 6  * * New Product Line            0.25 2000000       500000
## 7  * * New Labs                    0.90  750000       675000
## 8  * IT                              NA      NA           NA
## 9  * * Outsource                   0.20  400000        80000
## 10 * * Go agile                    0.05  250000        12500
## 11 * * Switch to R                 1.00   50000        50000
```

Deleting Attributes

The Set method can also be used to assign a single value directly. For example, to remove the avgExpectedCost, we assign NULL on each node like this:

```
acme$Set("avgExpectedCost", NULL)
```

Chaining

Note that we can chain the arguments:

```
acme$Set("avgExpectedCost", NULL)$Set("expectedCost", NA)
as.data.frame(acme, "avgExpectedCost", "expectedCost")
```

```
##                          levelName avgExpectedCost expectedCost
## 1  Acme Inc.                                    NA           NA
## 2  * Accounting                                 NA           NA
## 3   * * New Software                          NA            NA
## 4   * * New Accounting Standards              NA            NA
## 5   * Research                                NA            NA
## 6   * * New Product Line                      NA            NA
## 7   * * New Labs                              NA            NA
## 8   * IT                                      NA            NA
## 9   * * Outsource                             NA            NA
## 10 * * Go agile                               NA            NA
## 11 * * Switch to R                            NA            NA
```

A Word on Null and NA

Also note that setting a value to NA or to NULL looks equivalent when printing to a data.frame, but internally it is not:

```
acme$avgExpectedCost
```

```
## NULL
```

```
acme$expectedCost
```

```
## [1] NA
```

The reason is that NULL is always converted to NA for printing, and when using the Get method.

Aggregate

For simple cases, you don't have to write your own function to pass along to the Get method. For example, the Aggregate method provides a shorthand for the often used case when a parent is the aggregate of its children values:

```
acme$Aggregate("cost", sum)
```

```
## [1] 4950000
```

We can use this in the Get method:

```
acme$Get("Aggregate", "cost", sum)
```

```
##                Acme Inc.              Accounting             New Software
##                  4950000                 1500000                  1000000
## New Accounting Standards                Research         New Product Line
##                   500000                 2750000                  2000000
##                 New Labs                      IT                Outsource
##                   750000                  700000                   400000
##                 Go agile             Switch to R
##                   250000                   50000
```

This is equivalent of:

```
GetCost <- function(node) {
  result <- node$cost
  if(length(result) == 0) {
    if (node$isLeaf) stop(paste("Cost for ", node$name, " not available!"))
    else {
      node$children %>% sapply(GetCost) %>% sum -> result
    }
  }
  return (result)
}

acme$Get(GetCost)
```

```
##                Acme Inc.               Accounting             New Software
##                  4950000                  1500000                  1000000
## New Accounting Standards                 Research         New Product Line
##                   500000                  2750000                  2000000
##                 New Labs                       IT                Outsource
##                   750000                   700000                   400000
##                 Go agile              Switch to R
##                   250000                    50000
```

Sorting

You can sort an entire tree by using the Sort method on the root. The method will sort recursively and, for each node, sort the children by a child attribute. As before, the child attribute can also be a function or a method (e.g. of a sub class of Node, see below).

```
acme$Get(ExpectedCost, assign = "expectedCost")
```

```
##                Acme Inc.               Accounting             New Software
##                  2192500                   875000                   500000
## New Accounting Standards                 Research         New Product Line
##                   375000                  1175000                   500000
##                 New Labs                       IT                Outsource
##                   675000                   142500                    80000
##                 Go agile              Switch to R
##                    12500                    50000
```

```
acme$Sort("expectedCost", decreasing = TRUE)
print(acme, "expectedCost")
```

```
##                        levelName expectedCost
## 1  Acme Inc.                          2192500
## 2  * Research                         1175000
## 3  * * New Labs                        675000
## 4  * * New Product Line                500000
## 5  * Accounting                        875000
## 6  * * New Software                    500000
## 7  * * New Accounting Standards        375000
## 8  * IT                                142500
## 9  * * Outsource                       80000
## 10 * * Switch to R                     50000
## 11 * * Go agile                        12500
```

Naturally, you can also sort a subtree by calling Sort on the subtree's root node.

Subclassing Node

We can create a subclass of Node, and add custom methods to our subclass. This is very natural to users with experience in OO languages such as Java, Python or C#:

```
library(R6)
MyNode <- R6Class("MyNode",
                    inherit = Node,
                    lock = FALSE,

                    #public fields and function
                    public = list(
                        p = NULL,
                        cost = NULL,
                        AddChild = function(name) {
                          child <- MyNode$new(name)
                          invisible (self$AddChildNode(child))
                        }
                    ),

                    #active
                    active = list(
                      expectedCost = function() {
                        if ( is.null(self$p) || is.null(self$cost)) return (NULL)
                        self$p * self$cost
                      }
                    )
                )
```

The AddChild utility function in the subclass allows us to construct the tree just as before.

The expectedCost function is now a Method, and we can call it in a more R6-ish way.