Optimization - Graph Theory Applications

Karen Ficenec

October 23, 2017

```
#start by clearing the workspace
rm(list=ls())

#load the required packages...
library(igraph)
library(optrees)
```

Adjacency Matrix to List Conversion

For when a function needs an adjacency list as input (e.g. the msTreeKruskal() function)

We're going to generate an adjacency matrix and then turn it into an adjacency list.

The adjacency matrix will have all the distances or weights between all nodes in your network. For your application you can fill in actual weights, but here we will just generate random data as an example.

```
#choose n (the number of nodes in your network)
n <- 1000
#generate numbers from a random uniform distribution
#runif automatically generates them between 0 and 1.
d <-  runif(n*n)
#set everything less than 0.8 to NA, so that each non-NA
#space has a 20% chance of occuring.
#this simulates 2 nodes that have no weight or connection between them.
d[d<0.8] <- NA
#initialize a matrix of size n x n
d <- matrix(d, nrow=n, ncol=n)
#set all diagonal entries to zero because that represents
#mapping something to itself.
diag(d) <- NA
#make the matrix symmetric because (for example) the distance
#from point A to point B is the same as from point B to point A.
d[upper.tri(d)] = t(d)[upper.tri(d)]

#The function below will convert an adjacency matrix
#into an adjacency list.

AdjMatrix2List <- function(d) {
  #worst case every element of the matrix will be filled
  #so there will by (n x n) connections that need to be mapped
  #by the adjacency list.
  size <- nrow(d)*ncol(d)
  #initialize an index and the adjacency list matrix
  indx <- 1
  adjList <- matrix(nrow=size, ncol=3,
                    dimnames = list(list(),list("head","tail","weight")))
  #loop through the matrix and for all non-NA values, add
```

```
  #that connection to the adjacency list, with the start &
  #end nodes and the weight.
  #since i and j are tracking rows and columns in the adjacency
  #matrix, indx is needed to track rows in the new adjacency list.
  for (i in 1:nrow(d)){
    for (j in 1:ncol(d)){
      if (is.na(d[i,j]) == FALSE) {
        adjList[indx,] = c(i,j,d[i,j])
        indx <- indx + 1
      }
    }
  }
  #we initially set the adjacency list data structure to the worst
  #case size of (nxn).  Now we can chop off any rows in the
  #adjacency list that were not used, or were filled with a zero
  #weight.
  keepIndx <-  (adjList[,3] != 0 & is.na(adjList[,3]) == FALSE )
  adjList <- adjList[keepIndx,]
  #the function returns the adjacency list.
  adjList
}

#call the adjacency list function
AdjList <- AdjMatrix2List(d)
#the whole thing is 1000s of rows long, so just print the
#head and tail to see that the data is in the desired form.
head(AdjList)
```

```
##      head tail    weight
## [1,]    1    4 0.9245954
## [2,]    1    7 0.8685949
## [3,]    1    9 0.9308500
## [4,]    1   12 0.9510365
## [5,]    1   14 0.8736912
## [6,]    1   15 0.9776121
```

```
tail(AdjList)
```

```
##           head tail    weight
## [199535,] 1000  988 0.8319268
## [199536,] 1000  989 0.8331223
## [199537,] 1000  991 0.8766523
## [199538,] 1000  992 0.8645679
## [199539,] 1000  994 0.8196286
## [199540,] 1000  998 0.9412361
```

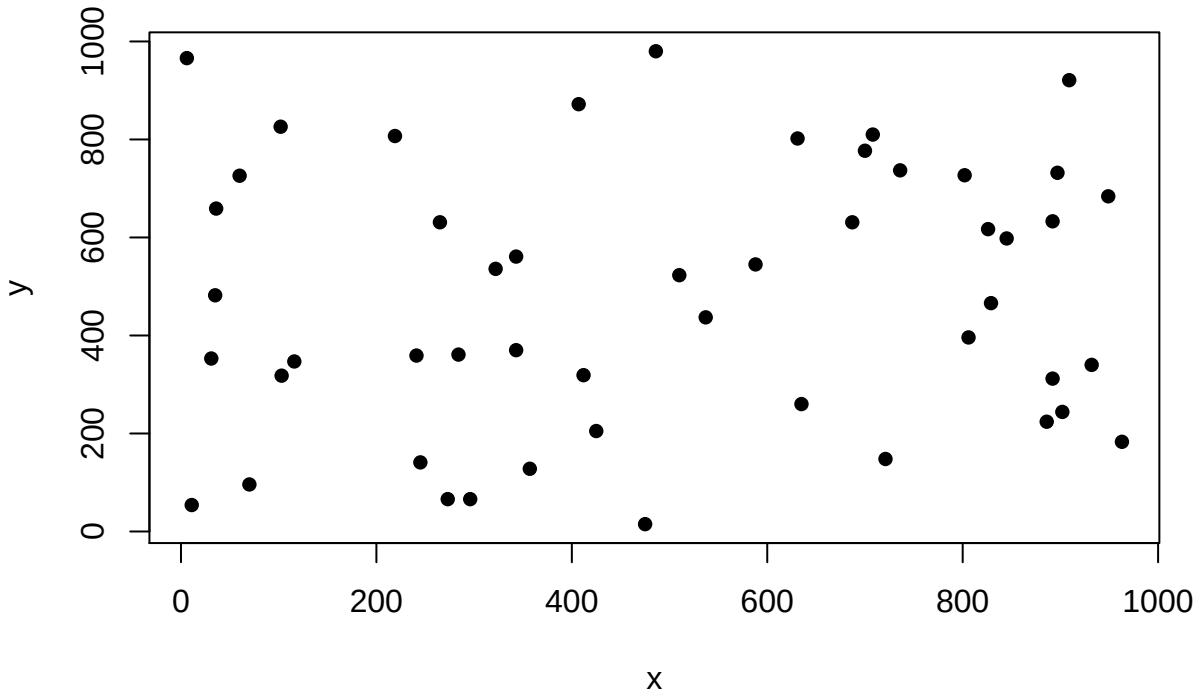
```
#note: you can also check that it looks right by changing
#n to a small number (like 10) and looking at the the whole
#adjacency list, which I did first. (not shown)
```

Euclidean Minimum Spanning Tree

We want to build a minimum spanning tree where the weights are the distances between points on a graph. We will create distances between every possible pair of points which is the definition of a complete graph.

And then use the msTreeKruskal() function to find the minimum spanning tree, which we will plot.

```
#pick a number of points
n2 <- 50
#generate integer numbers between 0 and 1000 as your x &
#y coordinates
x <- round(runif(n2)*1000)
y <- round(runif(n2)*1000)

#plot the points
plot(x,y,pch=16)
```



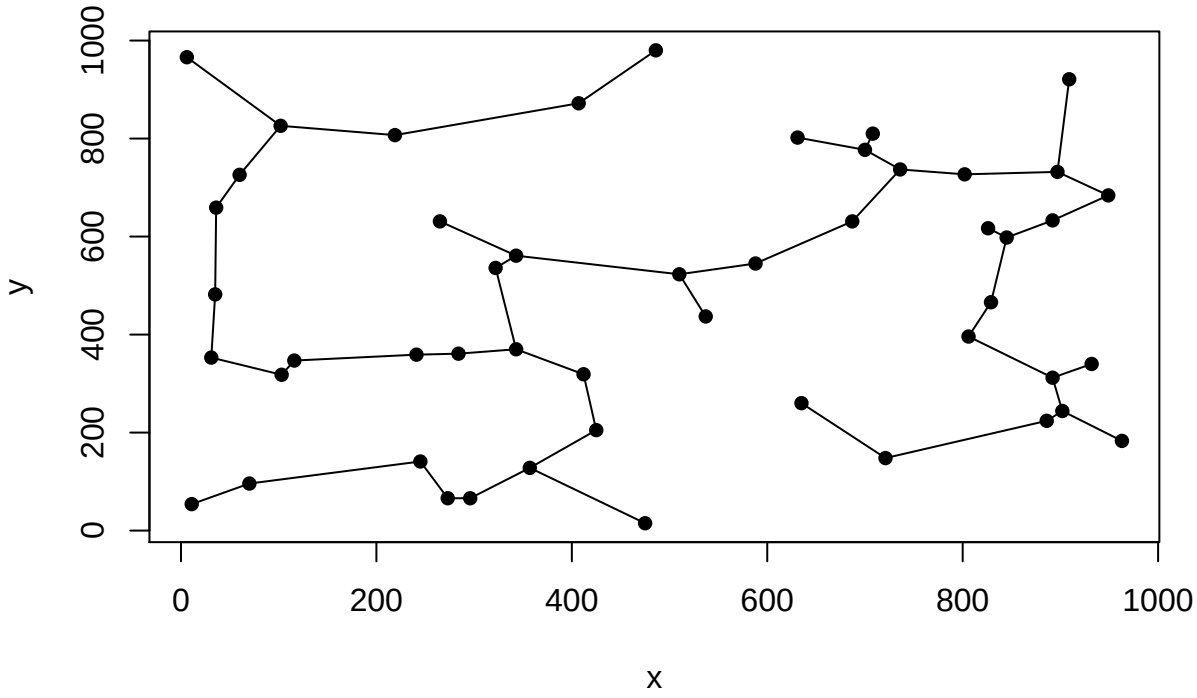

```
#the function below will compute the euclean distance between all pairs of
#points on a graph.
createAdjMatrix <- function(x,y){
  len <- length(x)
  #initialize a list that will hold all the points
  pts <- list(rep(0, len))
  #initialize a matrix that will contain all edge weights (distances)
  #between every pair of points on the graph.
  adjMatrix <- matrix(nrow = len, ncol=len)
  #make a list containing a list for each pair of x,y points
  for (i in 1:len) {pts[[i]] <- list(x[i], y[i])}
  #loop through to compute the distance between each pair of points
  for (i in 1:len){
    for (j in 1:len){
      distance = sqrt((pts[[i]][[2]]-pts[[j]][[2]])^2 + (pts[[i]][[1]]-pts[[j]][[1]])^2)
```

```
      #i & j refer to certain points, 2 is the y coord and 1 is the x coord in the
      #above indices.  This is the formula for computing euclidean distance.
      adjMatrix[i,j] <- distance
    }
  }
  adjMatrix
}

#create an adjacency matrix using the x & y lists above.
d1 <- createAdjMatrix(x,y)
#turn the adjacency matrix into an adjacency list.
ds1 <- AdjMatrix2List(d1)

#call Kruskal's algorithm to get the minimum spanning tree.
ds.mst <- msTreeKruskal(1:n2,ds1)
#the arcs of the minimum spanning tree are displayed in the
#$tree.arcs output.
ds.mst
```

```
## $tree.nodes
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
##
## $tree.arcs
##       ept1 ept2    weight
##  [1,]   21   33  23.00000
##  [2,]    2   36  25.61250
##  [3,]    5   20  26.87006
##  [4,]    9   11  31.78050
##  [5,]   19   30  32.64966
##  [6,]   15   48  33.95585
##  [7,]   12   24  43.04649
##  [8,]   28   32  48.82622
##  [9,]   15   22  53.81450
## [10,]    5   10  58.60034
## [11,]   24   31  59.68249
## [12,]    7   22  66.75328
## [13,]   32   36  68.73136
## [14,]   17   37  70.76722
## [15,]   40   49  71.16881
## [16,]   14   38  72.42237
## [17,]   15   45  73.38937
## [18,]   25   26  73.68175
## [19,]   10   17  76.48529
## [20,]    1   11  80.05623
## [21,]   33   39  80.05623
## [22,]    4   35  81.04320
## [23,]   31   41  85.80210
## [24,]   36   42  86.26703
## [25,]   21   46  86.97701
## [26,]   13   35  90.13878
## [27,]    7   37  95.13149
## [28,]   46   47 102.72780
```

```
## [29,]   30   50 104.80458
## [30,]    8   40 108.46197
## [31,]   41   47 114.73883
## [32,]   22   29 116.77757
## [33,]    8   27 118.53270
## [34,]   26   32 120.21647
## [35,]    9   12 125.57468
## [36,]    1   43 129.06200
## [37,]    4   29 131.13733
## [38,]    5   25 132.96616
## [39,]   23   34 133.80957
## [40,]   16   18 141.20906
## [41,]    3   46 163.37993
## [42,]   19   31 167.32304
## [43,]    8   44 169.75276
## [44,]   30   35 171.26879
## [45,]   43   49 177.00282
## [46,]   38   39 180.69311
## [47,]    2   16 181.66177
## [48,]    6   37 189.38057
## [49,]   27   34 198.91958
##
## $stages
## [1] 231
##
## $stages.arcs
##  [1]   1   3   5   7   9  11  13  15  17  19  21  23  27  29  31  33  35
## [18]  37  39  45  47  49  53  55  57  63  67  75  79  81  91  93  95  99
## [35] 107 109 113 115 117 133 161 165 169 171 179 197 199 215 231
```

```r
#The function below will access the list of arcs from the output
#of a minimum spanning tree algorithm.
#The segments function will take the x & y coordinate of the head of
#each arc (in column 1 of arcList) to the x & coordinates of the tail of
#each arc (column 2 in the arcList) in order to draw a line between them.
plot.mst <- function(arcList, xs, ys) {
  segments(xs[arcList[,1]],ys[arcList[,1]],xs[arcList[,2]],ys[arcList[,2]])
}

#make a plot and draw in the segments of the minimum spanning tree
#using the plot.mst function above.
plot(x,y,pch=16)
plot.mst(ds.mst$tree.arcs, x, y)
```



Unique Application of Minimum Spanning Tree:

Agents in Hostile Territory

(Given as a thought experiment - with process explained but no numbers)

The government has agents in hostile territory and wants to spread a message among all the agents with the minimum probability of detection. Agents can rendezvous with a certain probability of detection a.k.a. interception; and the government has estimated these probabilities of interception for each possible rendezvous between agents.

In this case, the nodes of the graph are the agents, and the arcs (or connections) are the probability of interception. We want to find the set of rendezvous' with the smallest probability of detection, in other words - the highest probability of safe and secure transfer of information.

To use the actual probabilities of interest, I think we would need to compute 1-p(interception) = p(safety) and then multiply p(safety) for all of the rendezvous's taking place.

E.g. p(safetyRendezvous1) $\times$ p(safetyRendezvous2) $\times \ldots \times$ p(safetyRendezvousX)

This would get the overall probability of safety after all rendezvous had taken place. (And 1-this probability would be the overall probability of interception.) And we want to maximize this overall p(safety).

When using the minimum spanning tree functions provided in most packages to find a maximum product such as this, we have to be very careful. In order to use an MST algorithm to **maximize** something, you must take the **negative** of all the inputs; and in order to use an MST algorithm to minimize the *product* you need to take the *log* of all inputs.

Doing these two things in the wrong order would result in taking a negative log (which is not a real number).

If you are given the p(interception) for all rendezvous's, the first step is to compute 1-(p(interception)) this will give you p(safety) for all the edges (possible rendezvous's). Because you are interested in the product of all of these p(safety)'s, the next step is to take the lg of each edge. Then when min spanning tree sums lg(p(safety1)) + lg(p(safety2)), it will actually be finding lg(p(safety1)×p(safety2)); which gives us the product that we're looking for.

The next situation to address is that we want to maximize this p(safety), but the minimum spanning tree package will be trying to minimize it. Because all of our probabilities are between 0 and 1, the lg of all of them will be negative. So instead of trying to maximize the sum of those negative numbers, we can minimize the sum of their positive values.

E.g. Say you are choosing two edges and you can choose -2 + -3 = -5 or -2 + -4 = -6. The maximum that we want is -2 and -3 = -5. If you took the opposite signs 2+3=5 and 2+4=6, and found the minimum-as the min spanning tree will- you'd still be choosing the 2 and the 3.

Because the numbers that we want to maximize the sum of are all negative, we can instead take their absolute value and minimize them.

So, to recap, we will take -lg(1-p(interception)), and compute the minimum spanning tree on that. We can input those values into Kruskal's algorithm (or Prim's) and this will find the spanning tree with the minimum chance of detection in O(ElogV) time (where E represent all possible rendevous -edges- and V all agents -nodes). Based on this tree, we could tell our agents with whom they should rendevous.

If you then wanted to find the overall p(safety) you would need to unpack what we did. So, if the tree has an overall minimum cost... first take it's negative, then take 2 and raise it to the number that you have (to undo the lg). This will give the overall p(safety) and 1-that will give the overall p(interception).

Optimal Project Scheduling

We have a set of tasks A-J that must be completed in a certain order. We want to know when we can start and finish each task and get the overall project done in the minimal amount of time. Some events will lie on the critical path (and thus have no flexibility in their start and end dates), but other events will have some flexibility (a.k.a. slack) in when they can begin and end to still have the overall project finish as early as possible.

The labels actually have names A-J, however we will need them in numeric form for the shortest path algorithms, so they are listed as 1-10 here. It is easy to get the letters back by accessing the letters() function in R. E.g. letters[1] gives "a" and LETTERS[1] gives "A"

```
s.labels <- c(1,2,3,4,5,6,7,8,9,10)
```

s.nodes gives the duration (in days) of each task.

```
s.nodes <- c(90, 15, 5, 20, 21, 25, 14, 28, 30, 45)
```

s.pred gives the event that is required to be completed before a certain event of interest can start. The event of interest is simply the index position in the s.pred list.
For example, the first position in the list (index 1 corresponding to event A) has NA in s.pred because no event is required to start before A. The second position in s.pred (index 2 corresponding to event B) has a value of 1 because event 1 (event A) must take place before B can occur. As a final example, the 7th position in s.pred (index 7 corresponding to event G) has a list of 3 and 6 because events C and F must take place before event G can occur. The s.pred list will be used to build the arcs of the directed graph. When we build the arcs, the weight will be the duration of the predecessor's node. We will actually have to take the negative

of this value, because we will be using a shortest path alogorithm, but we actually want the longest path. We want the longest path because that will ensure that every event that needs to happen before another event has had time to occur (if we took the shortest path, then we'd get to events before all of their predecessors had been completed).

```
s.pred <- list(NA, 1,2,7,4,1,list(3,6),4,1,list(4,9))
```

```
#The function below counts the total number of elements when there are
#nested lists.  This will be used to initialize an adjacency list large
#enough to hold all the arcs in the directed graph (one for each predecessor
#relationship).
elemsInListOfLists <- function(x){
  count <- 0
  for (i in x) {
    if (class(i) == "list") {
      count <- count + length(i)
      #counts the number of elements when they're contained inside a list.
    }
    #counts the other (non-nested) predecessor relationships,
    #excluding values of NA which do not get mapped.
    else if (is.na(i) == FALSE) {
      count <- count + 1
    }
  }
  count
}

#initialize a matrix that will becomes the adjacency list
s.adjList <- matrix(nrow=elemsInListOfLists(s.pred), ncol = 3,
                    dimnames = list(list(),list("head","tail","weight")))

#this loops through each event and examines it's predecessors.
#if the predecessor value it not NA, then it treats it as a list
#to look for all required predecessors.
#There are several indices to get inside the list of lists, and then
#each predecessor is made into an arc and added to the adjacency list.
cnt <- 1
for (i in 1:length(s.pred)){
  if (is.na(s.pred[i]) == FALSE){
    #going inside inner lists in case there are multiple predecessors.
    for (z in 1:length(s.pred[i][[1]])){
      s.adjList[cnt,] <- c(s.pred[i][[1]][[z]], s.labels[i],
                           as.integer(-s.nodes[s.pred[i][[1]][[z]]]))
                           #The third column is the weight given by the predecessor node.
                           #if the labels were letters, you could use this code instead:
                           #as.integer(-s.nodes[match(s.pred[i][[1]][[z]], s.labels)]))
      cnt <- cnt + 1
    }
  }
}
s.adjList
```

```
##       head tail weight
##  [1,]    1    2    -90
##  [2,]    2    3    -15
```

```
##  [3,]    7    4    -14
##  [4,]    4    5    -20
##  [5,]    1    6    -90
##  [6,]    3    7     -5
##  [7,]    6    7    -25
##  [8,]    4    8    -20
##  [9,]    1    9    -90
## [10,]    4   10    -20
## [11,]    9   10    -30
```

We will need to use the Bellman-Ford algorithm for the shortest path since we are using negative edge weights.

```
ESsolu <- getShortestPathTree(s.labels, s.adjList, algorithm = "Bellman-Ford")
```

```
##
##  Shortest path tree
##  Algorithm: Bellman-Ford
##  Stages:  9 | Time:  0
##  ------------------------------
##       head     tail    weight
##          1        2       -90
##          1        6       -90
##          1        9       -90
##          2        3       -15
##          4        5       -20
##          4        8       -20
##          4       10       -20
##          6        7       -25
##          7        4       -14
##  ------------------------------
##                    Total = -384
##
##  Distances from source:
##  ------------------------------
##     source     node  distance
##          1        2       -90
##          1        3      -105
##          1        4      -129
##          1        5      -149
##          1        6       -90
##          1        7      -115
##          1        8      -149
##          1        9       -90
##          1       10      -149
##  ------------------------------
```

The shortest path algorithm with negative edge weights has given us the longest path (to ensure that all predeccesors are met), and thus the earliest start times for each event.

```
ES <- -ESsolu$distance
```

By adding the duration of each event, we obtain their earliest finish times. (The earliest and latest start and finish times will be shown together in a table once they have been computed. For now let's just look at the largest value in earliest finish, which is the earliest completion date of the overall project)

```
EF <- (ES + s.nodes)
EF
```

```
##  [1]  90 105 110 149 170 115 129 177 120 194
```

So, the earliest completion date is 194 days from when the project begins (or since it began on November 1st, this is May 14th).

Now, in order to find the latest possible start and finish times, we have to work backwards. We will reverse the direction of all of the arcs, make sure that the edge weight corresponds to the new predecessor (previous tail node) and again take their negative when putting them into the shortest path formula. This will give the times *from the finish time* of the latest possible finish. For example, if we get -104, that is 104 days from the end of the project; so 194-104 = 90 days into the project.

```
#so, when we flip things around, we first need to change the weights in
#our adjacency list to their new predecessor (their previous tail node).
#because the weight of the new predecessor is just the tail of the old arc
#we can simply reference the weight of column 2 (the tails) in the old adjacency matrix.
cnt <- 1
wtCol <- numeric(length(s.adjList[,2]))
for (i in s.adjList[,2]) {
  wtCol[cnt] <- s.nodes[i]
  cnt <- cnt + 1

}

#initialize the transpose adjacency list
transposeAdjList <- matrix(nrow = (length(s.adjList[,2])), ncol = 3,
                           dimnames = list(list(), list("head","tail", "weight")))
#switch the heads and tails, and take the negative of the weights just computed.
transposeAdjList[,1] <- s.adjList[,2]
transposeAdjList[,2] <- s.adjList[,1]
transposeAdjList[,3] <- -wtCol
```

Because there was 3 end nodes, there are now 3 possible start nodes. We can create a dummy variable (a pre-start node) that goes to these 3 nodes with weight zero. We're going to pretend that we don't know which nodes were the end nodes in the original graph, or how many there are, so that way we can make the computer find them. :)

```
#start by initializing a vector to hold the (previuos end nodes, now) possible start nodes.
v = numeric()

#find all endpoints on the original graph, so you know all possible
#start nodes in the transpose.
#the endpoints in the original graph will not show up in column 1 (head of an arc)
#of the original adjacency list, because they are the end nodes and thus not the
#head/start of anything.
#therefore, the end nodes will have NA when we try to match them to a value in the
#first column of the adjacency matrix.
for (x in s.labels) {if (is.na(match(x, s.adjList[,1])) == TRUE){v <- append(v, x)}}

#add the dummynode to the list
s.labelsWithDummy <- c(1,2,3,4,5,6,7,8,9,10,11)
#the dummy node is the final one added to the list (11 in this case).
dummyNode <- (length(s.labelsWithDummy))

#add on the connections to the dummy node
for (i in v) {transposeAdjList <- rbind(transposeAdjList, c(dummyNode, i, 0))}
transposeAdjList
```

```
##       head tail weight
##  [1,]    2    1    -15
##  [2,]    3    2     -5
##  [3,]    4    7    -20
##  [4,]    5    4    -21
##  [5,]    6    1    -25
##  [6,]    7    3    -14
##  [7,]    7    6    -14
##  [8,]    8    4    -28
##  [9,]    9    1    -30
## [10,]   10    4    -45
## [11,]   10    9    -45
## [12,]   11    5      0
## [13,]   11    8      0
## [14,]   11   10      0
```

Now we can use the traspose adjacency list to call the shortest path function (using the Bellman-Ford algorithm for negative edge weights) to find the latest finish date.

```
LFdata <- getShortestPathTree(s.labelsWithDummy, transposeAdjList,
                              algorithm = "Bellman-Ford", source.node = 11)
```

```
##
##  Shortest path tree
##  Algorithm: Bellman-Ford
##  Stages:  10 | Time:  0.01
##  ------------------------------
##       head     tail    weight
##          3        2        -5
##          4        7       -20
##          6        1       -25
##          7        3       -14
##          7        6       -14
##         10        4       -45
##         10        9       -45
##         11        5         0
##         11        8         0
##         11       10         0
##  ------------------------------
##                    Total = -168
##
##  Distances from source:
##  ------------------------------
##     source     node  distance
##         11        1      -104
##         11        2       -84
##         11        3       -79
##         11        4       -45
##         11        5         0
##         11        6       -79
##         11        7       -65
##         11        8         0
##         11        9       -45
##         11       10         0
##  ------------------------------
```

Remember, because we are essentially starting from the end, the numbers are given in reference to the end of the project, so -104 is 104 days from the end of the project which ends on day 194. So, to get the latest finish days from the output of the algorithm we have to add them (they're negative values) to 194. Also we need to index 1:10 on the results so as not to include the dummy variable.

```
LF <- (194 + LFdata$distance[1:10])
```

We can subtract the duration of each event from these values to get the latest possible start days.

```
LS <- (194 + LFdata$distance[1:10] - s.nodes)
```

Lastly, we can calculate the slack, which is the difference between the earilest start time and the lastest possible start time. (Or equivalently between the earliest finish time and the latest finish time.) If an event has zero slack, this means that it has no flexibility in start and end days, if the project is to finish in minimal time. When an event has no slack, it is said to lie on the critical path.

```
slackV <- LF - EF
slackdf <- data.frame(events = letters[s.labels], slack = slackV)
slackdf
```

```
##    events slack
## 1       a     0
## 2       b     5
## 3       c     5
## 4       d     0
## 5       e    24
## 6       f     0
## 7       g     0
## 8       h    17
## 9       i    29
## 10      j     0
```

Tasks A, D, F, G, and J are on the critical path. And tasks B, C, E, H and I have scheduleing flexibility (slack). Tasks I and E have the most slack at 29 and 24 days respectively, H has 17 days of slack, and tasks B & C have the least (positive) slack at just 5 days.

And the full tables are shown below (first in days and then in dates)...

```
fullDaydf <- data.frame(Events = LETTERS[s.labels], EarliestStart = ES,
                        LatestStart = LS, EarliestFinish = EF,
                        LatestFinish = LF, slack = slackV)
fullDaydf
```

```
##    Events EarliestStart LatestStart EarliestFinish LatestFinish slack
## 1       A             0           0             90           90     0
## 2       B            90          95            105          110     5
## 3       C           105         110            110          115     5
## 4       D           129         129            149          149     0
## 5       E           149         173            170          194    24
## 6       F            90          90            115          115     0
## 7       G           115         115            129          129     0
## 8       H           149         166            177          194    17
## 9       I            90         119            120          149    29
## 10      J           149         149            194          194     0
```

```
ESdates <- format((as.Date("2017-11-01") + ES), "%b %d, %Y")
EFdates <- format((as.Date("2017-11-01") + EF), "%b %d, %Y")
LSdates <- format((as.Date("2017-11-01") + LS), "%b %d, %Y")
LFdates <- format((as.Date("2017-11-01") + LF), "%b %d, %Y")
```

```
fullDatedf <- data.frame(Events = LETTERS[s.labels],
                         Earliest_Start = ESdates, Latest_Start__ = LSdates,
                         Earliest_Finish = EFdates, Latest_Finish__ = LFdates)
fullDatedf
```

```
##    Events Earliest_Start Latest_Start__ Earliest_Finish Latest_Finish__
## 1       A   Nov 01, 2017   Nov 01, 2017    Jan 30, 2018    Jan 30, 2018
## 2       B   Jan 30, 2018   Feb 04, 2018    Feb 14, 2018    Feb 19, 2018
## 3       C   Feb 14, 2018   Feb 19, 2018    Feb 19, 2018    Feb 24, 2018
## 4       D   Mar 10, 2018   Mar 10, 2018    Mar 30, 2018    Mar 30, 2018
## 5       E   Mar 30, 2018   Apr 23, 2018    Apr 20, 2018    May 14, 2018
## 6       F   Jan 30, 2018   Jan 30, 2018    Feb 24, 2018    Feb 24, 2018
## 7       G   Feb 24, 2018   Feb 24, 2018    Mar 10, 2018    Mar 10, 2018
## 8       H   Mar 30, 2018   Apr 16, 2018    Apr 27, 2018    May 14, 2018
## 9       I   Jan 30, 2018   Feb 28, 2018    Mar 01, 2018    Mar 30, 2018
## 10      J   Mar 30, 2018   Mar 30, 2018    May 14, 2018    May 14, 2018
```