Appendix: Algorithms

Dalia Camacho, Gabriela Vargas, Elizabeth Monroy

Naive Pattern Matching

Here we show the algorithm used for naive pattern search.

```
naive_patternSearching<-function(P,S,maxErr){
  # Given the patterns P and S we transform them as
  # a vector contianing its characters
  P <- substring(P,1:nchar(P),1:nchar(P))
  S <- substring(S,1:nchar(S),1:nchar(S))
  # We find the length of the patterns
  l <- length(P)
  N <- length(S)
  # We begin the pattern search
  for(i in 0:(N-l+1)){
    Err <- 0
    j   <- 1
    k   <- 1
    while(j<=N & k<=l & Err<=maxErr){
      if(P[k]==S[i+j]){
        k<-k+1
        j<-j+1
      }else{
        if((i+j+1) <= N & (k+1) <= l & P[k]==S[(i+j+1)]){
          if(P[k+1]==S[i+j]){
            #Transposition
            Err <- Err+1
            j   <- j+2
            k   <- k+2
          }else{
            #Insertion
            Err<-Err+1
            j <- j + 2
          }
        }else{
          #Deletion
          if((k+1)<=l & P[k+1]==S[i+j]){
            Err<-Err+1
            k <-k+2
          }else{
            #Substitution
            Err<-Err+1
            k <- k+1
            j <- j+1
          }
        }
      }
    }
    # Return first occurence of the pattern
```

```
    if(Err<=maxErr){
      return(list(position=i+1, errors=Err))
    }
  }
  if(Err>maxErr){
    return("Pattern not found")
  }
}
```

We tried the algorithm on the following examples:

```
maxErr <- 1

S <- "Amlover"
P <- "Amlo"
naive_patternSearching(P,S,maxErr)
```

```
## $position
## [1] 1
##
## $errors
## [1] 0
```

```
P <- "Axlo"
naive_patternSearching(P,S,maxErr)
```

```
## $position
## [1] 1
##
## $errors
## [1] 1
```

```
P <- "love"
naive_patternSearching(P,S,maxErr)
```

```
## $position
## [1] 3
##
## $errors
## [1] 0
```

```
P <- "lobe"
naive_patternSearching(P,S,maxErr)
```

```
## $position
## [1] 3
##
## $errors
## [1] 1
```

Algorithm to defined the shift tree

This function generates a shift tree from a set of patterns.

```
library(data.tree)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
DefineShiftTree <- function(patterns){

  #Maximum shift value
  maxShift  <- min(nchar(patterns))

  # Create blocks of the suffixes with length 2
  Bs        <-unique(unlist(lapply(patterns,
                                   function(i){
                                     substr(i,(nchar(i)-1),nchar(i))
                                   })))

  # Extract all characters in the patterns
  Chars   <- unique(unlist(lapply(patterns,
                                  function(i){
                                    substring(i,1:nchar(i), 1:nchar(i))
                                  }
  )))
  # Create vector with all characters from the patterns that
  # do not correspond to the first character of any of the blocks
  CharsErr1 <- c(Chars[-which(Chars%in%substring(Bs,1,1))], "not")

  # Create vector with all characters from the patterns that
  # do not correspond to the second character of any of the blocks
  CharsErr2 <- c(Chars[-which(Chars%in%substring(Bs,2,2))], "not")

  # Define the maximum number of shifts for all the substrings of size two
  # that do not correspond to any suffix
  Shifts <- matrix(unlist(lapply(patterns,
                                 function(i){
                                   rbind(substring(i,1:(nchar(i)-1), 2:(nchar(i))),(nchar(i)-2):0)
                                 }
  )), ncol=2, byrow=TRUE)

  Shifts[,2] <- lapply(Shifts[,2], function(i){min(maxShift, as.numeric(i))}) %>% unlist()

  if(any (duplicated(Shifts[,1]))){
    doubles    <- which(duplicated(Shifts[,1]))
    stringDob  <- Shifts[doubles,1]
    uniqueDob  <- unique(Shifts[doubles,1])
    for (i  in uniqueDob) {
      whichstr <- which(Shifts[,1]==i)
      Shifts[whichstr,2] <- min(as.numeric(Shifts[whichstr,2]))
    }
    Shifts <- unique(Shifts)
```

```
}
Shifts     <- Shifts[-which(Shifts[,2]=="0"),]
Shifts     <- cbind(Shifts, substring(Shifts[,1],1,1))
Shifts     <- cbind(Shifts, substring(Shifts[,1],2,2))




Tree <- Node$new("Root")
for (i  in 1:length(Bs)) {
  Tree$AddChild(substring(Bs[i],1,1))$AddChild(substring(Bs[i],2,2))$AddSibling("Error")
  Tree$children[[i]]$children[[1]]$shift <- 0
  Tree$children[[i]]$children[[1]]$error <- 0
  Tree$children[[i]]$children[[1]]$pointer <- i

  Tree$children[[i]]$children[[2]]$shift   <- 0
  Tree$children[[i]]$children[[2]]$error   <- 1
  Tree$children[[i]]$children[[2]]$pointer <- i


}
# Add all the branches that arise from a first error
Tree$AddChild("Error")

for (i in 1:length(CharsErr1)) {
  # Add character from pattern
  Tree$Error$AddChild(CharsErr1[i])

  for (j in 1:length(unique(substring(Bs,2,2)))) {
    Tree$Error$children[[i]]$AddChild(unique(substring(Bs,2,2))[j])
    Tree$Error$children[[i]]$children[[j]]$shift   <-0
    Tree$Error$children[[i]]$children[[j]]$error   <-0
    Tree$Error$children[[i]]$children[[j]]$pointer <- j

  }

  # Add node for the second error
  Tree$Error$children[[i]]$AddChild("Error")

  # Add all the nodes that arise from a second error
  for (j in 1:length(CharsErr2)) {
    Tree$Error$children[[i]]$Error$AddChild(CharsErr2[j])
    Tree$Error$children[[i]]$Error$children[[j]]$error <- 2
    Tree$Error$children[[i]]$Error$children[[j]]$shift <- maxShift
  }
  # If the character corresponds to the first character of
  # any substring of size two (without suffixes)
  # find the entry of the tree that matches the substrings
  # starting with this character and adjust shift value
  if(CharsErr1[i]%in%Shifts[,3]){
    mark <- which(Shifts[,3]==CharsErr1[i])
    for (j in mark) {
```

```
        if(is.null( Tree$Error$children[[i]]$Error$children[[Shifts[j,4]]])==FALSE){
          Tree$Error$children[[i]]$Error$children[[Shifts[j,4]]]$shift <- Shifts[j,2]
        }
      }
    }
  }
  return(Tree)
}
```

As an exmaple consider the tree constructed from the following patterns.

```
Tree <- DefineShiftTree(c("Amlo", "Amlover","Amlofest"))
Tree
```

```
##                                levelName
## 1   Root
## 2    ¦--l
## 3    ¦   ¦--o
## 4    ¦   °--Error
## 5    ¦--e
## 6    ¦   ¦--r
## 7    ¦   °--Error
## 8    ¦--s
## 9    ¦   ¦--t
## 10   ¦   °--Error
## 11   °--Error
## 12       ¦--A
## 13       ¦   ¦--o
## 14       ¦   ¦--r
## 15       ¦   ¦--t
## 16       ¦   °--Error
## 17       ¦       ¦--A
## 18       ¦       ¦--m
## 19       ¦       ¦--l
## 20       ¦       ¦--v
## 21       ¦       ¦--e
## 22       ¦       ¦--f
## 23       ¦       ¦--s
## 24       ¦       °--not
## 25       ¦--m
## 26       ¦   ¦--o
## 27       ¦   ¦--r
## 28       ¦   ¦--t
## 29       ¦   °--Error
## 30       ¦       ¦--A
## 31       ¦       ¦--m
## 32       ¦       ¦--l
## 33       ¦       ¦--v
## 34       ¦       ¦--e
## 35       ¦       ¦--f
## 36       ¦       ¦--s
## 37       ¦       °--not
## 38       ¦--o
## 39       ¦   ¦--o
## 40       ¦   ¦--r
```

```
## 41       ¦   ¦--t
## 42       ¦   °--Error
## 43       ¦       ¦--A
## 44       ¦       ¦--m
## 45       ¦       ¦--l
## 46       ¦       ¦--v
## 47       ¦       ¦--e
## 48       ¦       ¦--f
## 49       ¦       ¦--s
## 50       ¦       °--not
## 51       ¦--v
## 52       ¦   ¦--o
## 53       ¦   ¦--r
## 54       ¦   ¦--t
## 55       ¦   °--Error
## 56       ¦       ¦--A
## 57       ¦       ¦--m
## 58       ¦       ¦--l
## 59       ¦       ¦--v
## 60       ¦       ¦--e
## 61       ¦       ¦--f
## 62       ¦       ¦--s
## 63       ¦       °--not
## 64       ¦--r
## 65       ¦   ¦--o
## 66       ¦   ¦--r
## 67       ¦   ¦--t
## 68       ¦   °--Error
## 69       ¦       ¦--A
## 70       ¦       ¦--m
## 71       ¦       ¦--l
## 72       ¦       ¦--v
## 73       ¦       ¦--e
## 74       ¦       ¦--f
## 75       ¦       ¦--s
## 76       ¦       °--not
## 77       ¦--f
## 78       ¦   ¦--o
## 79       ¦   ¦--r
## 80       ¦   ¦--t
## 81       ¦   °--Error
## 82       ¦       ¦--A
## 83       ¦       ¦--m
## 84       ¦       ¦--l
## 85       ¦       ¦--v
## 86       ¦       ¦--e
## 87       ¦       ¦--f
## 88       ¦       ¦--s
## 89       ¦       °--not
## 90       ¦--t
## 91       ¦   ¦--o
## 92       ¦   ¦--r
## 93       ¦   ¦--t
## 94       ¦   °--Error
```

```
## 95       ¦       ¦--A
## 96       ¦       ¦--m
## 97       ¦       ¦--l
## 98       ¦       ¦--v
## 99       ¦       ¦--e
## 100      ¦       °--... 3 nodes w/ 0 sub
## 101      °--... 1 nodes w/ 15 sub
```

Search Algorithm

First we define the algorithm that will be used to search on a string once the suffix matched to at most one error and the corresponding pattern has been selected from the set.

```
# This algorithm receives as input a pattern and a string already divided by characters
# The current error and the maximum error
# As ourput it returns the position in which the pattern was found
# and the number of errors. If the pattern did not match it returns a -1
# This search starts from the last characters of both the string and the pattern
searchInverted<- function(P,S, Err, maxErr){
  j   <- length(S)
  k   <- length(P)
  while(j>0 & k>0 & Err<=maxErr){
    if(P[k]==S[j]){
      k<-k-1
      j<-j-1
    }else{
      if((k-1)>0 & (j-1)>0){
        if(P[k-1]==S[j]){
          if(P[k]==S[j-1]){
            #Transposition
            Err <- Err+1
            j   <- j-2
            k   <- k-2
          }else{
            #Insertion
            Err<-Err+1
            j <- j - 2
          }
        }
      }else{
        #Deletion
        if((j-1)>0){
          if(P[k]==S[j-1]){
            Err<-Err+1
            k <-k-2
          }
        }else{
          #Substitution
          Err<-Err+1
          k <- k-1
          j <- j-1
        }
      }
```

```
    }
  }
  # Return first occurence of the pattern
  if(Err<=maxErr){
    return(list(position=j, errors=Err))
  }else{
    return(-1)
  }
}
```

Now we construct the function that finds the matches of severan strings.

```
approx_multi_patternSearch <- function(P,S, maxErr=1){
  maxShift <- min(nchar(P))
  Tree     <- DefineShiftTree(P)
  P        <- lapply(P, function(i){substring(i,1:nchar(i),1:nchar(i))})
  S        <- lapply(S, function(i){substring(i,1:nchar(i),1:nchar(i))})
  NStrings <- length(S)
  for (ni in 1:NStrings) {
    Si     <- S[[ni]]
    N      <- length(Si)
    Matchi <- NULL
    i      <-  maxShift - 1
    while (i<N){
      pointer <- NULL
      # Check if first character matches
      if(is.null(Tree$children[[Si[i]]])){
        # Check if first character is in any of the patterns
        if(is.null(Tree$Error$children[[Si[i]]])){
          # Check if second character is in the pattern
          if(is.null(Tree$Error$not$children[[Si[i+1]]])){
            # Check if second character is in any of the patterns
            if(is.null(Tree$Error$not$Error$children[[Si[i+1]]])){
              i <- i + maxShift
            }else{
              # second character is in a pattern
              i <- i + as.numeric(Tree$Error$not$Error$children[[Si[i+1]]]$shift)
            }
          }else{
            # Second character is last character
            pointer <- as.numeric(Tree$Error$not$children[[Si[i+1]]]$pointer)
            error   <- as.numeric(Tree$Error$not$children[[Si[i+1]]]$error)
          }
          # Check if second character is last character
        }else if(is.null(Tree$Error$children[[Si[i]]]$children[[Si[i+1]]])){
          if(is.null(Tree$Error$children[[Si[i]]]$Error$children[[Si[i+1]]])){
            i <- i + as.numeric(Tree$Error$children[[Si[i]]]$Error$not$shift)
          }else{
            i <- i + as.numeric(Tree$Error$children[[Si[i]]]$Error$children[[Si[i+1]]]$shift)
          }
        }else{
          #Second character is last
          pointer <- as.numeric(Tree$Error$children[[Si[i]]]$children[[Si[i+1]]]$pointer)
          error   <- as.numeric(Tree$Error$children[[Si[i]]]$children[[Si[i+1]]]$error)
```

```
      }
    }else{# First Character matches check error in second character
      if(is.null(Tree$children[[Si[i]]]$children[[Si[i+1]]])){
        # Check if second character in any pattern

        pointer <- as.numeric(Tree$children[[Si[i]]]$Error$pointer)
        error   <- as.numeric(Tree$children[[Si[i]]]$Error$error)

      }else{
        pointer <- as.numeric(Tree$children[[Si[i]]]$children[[Si[i+1]]]$pointer)
        error   <- as.numeric(Tree$children[[Si[i]]]$children[[Si[i+1]]]$error)
      }
    }
    if(is.null(pointer)==FALSE){
      Pi  <- P[[pointer]][1:(length(P[[pointer]])-2)]
      Si  <- Si[1:(i-1)]
      sol <- searchInverted(Pi,Si,error, maxErr)
      if(sol[1] != -1){
        Matchi <- c(string=ni, pattern=pointer,position=sol$position, errors=sol$errors)
        i <- i+N
      }else{
        i <- i+1
      }
    }
  }
  if(is.null(Matchi) || is.null(pointer)){
    Matchi <-c(string=ni, pattern = -1, position=-1, errors=-1)

  }
  if(ni==1){
    Matches <- data.frame(Matchi)
  }else{
    Matches <- cbind(Matches, Matchi)
  }
 }
 return(Matches)
}
```

Consider the following example.

```
approx_multi_patternSearch(c("Amlo", "Fest"),c("festival","Amlofest", "HOLa"))
```

```
##          Matchi Matchi Matchi
## string        1      2      3
## pattern       2      1     -1
## position      0      0     -1
## errors        1      0     -1
```