Algorithm to Calculate the Distribution of the Longest Path Length of a Stochastic Activity Network with Continuous Activity Durations

Lawrence M. Leemis, Matthew J. Duggan, John H. Drew, Jeffrey A. Mallozzi, Kerry W. Connell
Department of Mathematics, The College of William & Mary, Williamsburg, VA 23187

February 8, 2006

Note: Indentation is used to show nesting.

Parameters: The $n \times m$ node-arc incidence matrix N, PDFs of m activity durations in array $dist$, number of nodes n, number of arcs m

Procedure name: `GetDistribution`

Returned value: CDF, $F_{T_n}(t)$

local $a, b, c, i, j, k, l, insert, top, Paths, source, incoming, numPaths, tempPaths, count, S,$ $marker, counter, intOrder, first, second, numIntegrals, tempSet, upper_i, lower_i, temp,$ $lim, finished, maxSub, integrand, solution, tempSoln$

```
insert ← 1                                          some preprocessing to use algorithm
for (i ← 1; i ≤ n; i ← i + 1)                       rearrange columns to put matrix in usable form
    for (j ← 1; j ≤ m; j ← j + 1)                   sorts columns by source node
        if (N[i, j] = 1)
            N ← swapcol(N, j, insert)
            insert ← insert + 1
top ← 1                                             now sorts by destination node for arcs with same source
insert ← 1
while (top ≤ n)
    for (i ← 1; i ≤ n; i ← i + 1)
        for (j ← 1; j ≤ m; j ← j + 1)
            if ((N[i, j] = −1) and (N[top, j] = 1))
                N ← swapcol(N, j, insert)
                insert ← insert + 1
    top ← top + 1
```

```
for (i ← 1; i ≤ 50; i ← i + 1)                          50 is an arbitrarily large number of rows
    for (j ← 1; j ≤ n; j ← j + 1)                                    initialize Paths matrix
        Paths[i, j] ← 0
for (j ← 1; j ≤ m; j ← i + 1)                                        initialize source matrix
    source[j] ← 0
for (i ← 1; i ≤ n; i ← i + 1)                                      initialize incoming matrix
    incoming[i] ← 0
for (i ← 1; i ≤ m; i ← i + 1)
    for (j ← 1; j ≤ n; j ← j + 1)
        if N[i, j] = −1
            incoming[i] ← incoming[i] + 1              find number of arcs coming into each node
        if N[i, j] = 1
            source[j] ← i                         find the node from which each arc emanates
numPaths ← GetPaths(n, 1, 1, Paths)                              begin Step 1 of algorithm
Paths ← delrows(Paths, (numPaths + 1) .. 50)                               remove empty rows
for (i ← 1; i ≤ numPaths; i ← i + 1)                           initialize tempPaths matrix
    for (j ← 1; j ≤ n; j ← j + 1)
        tempPaths ← 0
for (i ← 1; i ≤ numPaths; i ← i + 1)                        reverse order of nodes in path
    count ← 0
    for (j ← 1; j ≤ n; j ← j + 1)
        if (Path[i, j] > 0)
            count ← count + 1
    for (a ← 1; a ≤ count; a ← a + 1)
        tempPaths[i, a] = Paths[i, count + 1 − a]
Paths ← tempPaths
for (i ← 1; i ≤ m; i ← i + 1)                                    begin Step 2 of algorithm
    S[i] ← 0                                                              initialize array S
for (i ← 1; i ≤ numPaths; i ← i + 1)   matrix S distinguishes single- and multiple-use arcs
    for (j ← 1; j ≤ n; j ← j + 1)
        if (Paths[i, j] > 0)
            S[Paths[i, j]] ← S[Paths[i, j]] + 1
for (a ← 1; a ≤ m; a ← a + 1)
    if (S[a] > 1)
        S[a] ← 1                                      S[a] = 1 if arc a is a multiple-use arc
    else
        S[a] ← 0                                        S[a] = 0 if arc a is a single-use arc
k ← 0                                                            begin Step 3 of algorithm
for (i ← 1; i ≤ m; i ← i + 1)                             k is the number of multiple-use arcs
    k ← k + S[i]
for (i ← 1; i ≤ numPaths; i ← i + 1)                     begin Steps 4 and 5 of algorithm
    marker[i] ← 1
```

```
counter ← 0
for (i ← 1; i ≤ k; i ← i + 1)                                    initialize intOrder array
    intOrder[i] ← 0
while (intOrder[k] = 0)
    first ← {}                                   set of nodes currently at the beginning of a path
    second ← {}                                         set of nodes appearing later on a path
    for (i ← 1; i ≤ numPaths; i ← i + 1)                   store current first arc on each path
        if (Paths[i, marker[i]] > 0)
            if (S[Paths[i, marker[i]]] = 1)
                first ← first ∪ {Paths[i, marker[i]]}
    for (i ← 1; i ≤ numPaths; i ← i + 1)                 store all subsequent arcs on each path
        for (j ← marker[i] + 1; Paths[i, j] > 0; j ← j + 1)
            if (S[Paths[i, j]] = 1)
                second ← second ∪ {Paths[i, j]}
    first ← first − (first ∩ second)                          separates out only initial nodes
    for (i ← 1; i ≤ ‖first‖; i ← i + 1)                store all initial arcs in intOrder array
        counter ← counter + 1
        intOrder[counter] ← first[i]
    for (i ← 1; i ≤ numPaths; i ← i + 1)                              continue traversing paths
        if (Paths[i, marker[i]] > 0)
            if (({Paths[i, marker[i]]} ∩ first = {Paths[i, marker[i]]})
             or (S[Paths[i, marker[i]]] = 0))
                marker[i] ← marker[i] + 1
numIntegrals ← 1                                                 begin Step 6a of algorithm
for (i ← 1; i ≤ k; i ← i + 1)                     initialize upper and lower limits of integration
    upper_1[1, i] ← t
    for (j ← 2; j ≤ 4; j ← j + 1)
        upper_1[j, i] ← 0
        lower_1[j − 1, i] ← 0
for (i ← 1; i ≤ k; i ← i + 1)                             begin Steps 6b and 7 of algorithm
    first ← {}                                              there are at most two incident arcs
    second ← {}
    for (j ← 1; j ≤ numPaths; j ← j + 1)                          find paths preceding current arc
        for (h ← 2; h ≤ n; h ← h + 1)
        if (Paths[j, h] = intOrder[i])
            tempSet ← {}
            for (l ← h − 1; l > 0; l ← l − 1)
                if (S[Paths[j, l]] = 1)
                    for (a ← 1; a ≤ k; a ← a + 1)
                        if (intOrder[a] = Paths[j, l])
                            temp ← temp ∪ {x_a}
            if (first = {})                                                    store first path
```

```
            first ← temp
          else                              there is already one path found, store second path
            second ← temp
      if (second = {})                          there is no maximum expression in the limit
        upper_1[2, i] ← first
        upper_1[3, i] ← {}
        upper_1[4, i] ← {}
      else                                         there is a maximum expression in the limit
        temp ← first ∩ second                    collect common terms in maximum expression
        first ← first − temp
        second ← second − temp
        upper_1[2, i] ← temp
        upper_1[3, i] ← first
        upper_1[4, i] ← second
for (i ← 1; i ≤ 4; i ← i + 1)
    for (j ← 1; j ≤ k; j ← j + 1)
        upper_1[i, j] ← convert(upper_1[i, j], ' + ')                  sum all terms in the set
finished ← 0                                              begin Steps 8 and 9 of algorithm
while (!finished)           repeat until all maximum expressions are eliminated (Step 9)
    for (i ← 1; i ≤ numIntegrals; i ← i + 1)
        for (j ← 1; j ≤ k; j ← j + 1)
            if ((upper_i[3, j] ≠ 0) or(upper_i[4, j] ≠ 0))        max expression in the upper limit
                numIntegrals ← numIntegrals + 1
                tempSet ← upper_i[3, j] ∪ upper_i[4, j]
                maxSub ← maximumX_i(tempSet)                       returns the highest subscript
                lim ← solve(upper_i[3, j] = upper_i[4, j], x_maxSub)
                a ← upper_i[3, j]
                b ← upper_i[4, j]
                upper_numIntegrals ← upper_i                              copy current matrices
                lower_numIntegrals ← lower_i
                for (c ← j; c ≤ k; c ← c + 1)     set the limit as one term in the max expression
                    if ((upper_i[3, c] = a) and (upper_i[4, c] = b))
                        if (upper_i[1, c] ≠ 0)              there is a term in the first row of upper_i
                            upper_i[2, c] ← upper_i[2, c] + a                augment current value
                            upper_i[3, c] ← 0
                            upper_i[4, c] ← 0
                            upper_numIntegrals[2, c] ← upper_numIntegrals[2, c] + b
                            upper_numIntegrals[3, c] ← 0
                            upper_numIntegrals[4, c] ← 0
                        else                        there is no term in the first row of upper_i
                            upper_i[2, c] ← a                                 replace current value
                            upper_i[3, c] ← 0
```

$upper_i[4, c] \leftarrow 0$
$upper_{numIntegrals}[2, c] \leftarrow b$
$upper_{numIntegrals}[3, c] \leftarrow 0$
$upper_{numIntegrals}[4, c] \leftarrow 0$
if (nops(lim) > 1) — if there are multiple terms in the new limit
if ($x_{maxSub} \in a$)
$lower_i[1, maxSub] \leftarrow 0$
$lower_i[2, maxSub] \leftarrow lim$
$lower_i[3, maxSub] \leftarrow 0$
$upper_{numIntegrals}[1, maxSub] \leftarrow 0$
$upper_{numIntegrals}[2, maxSub] \leftarrow 0$
$upper_{numIntegrals}[3, maxSub] \leftarrow lim$
$upper_{numIntegrals}[4, maxSub] \leftarrow 0$
else
$lower_{numIntegrals}[1, maxSub] \leftarrow 0$
$lower_{numIntegrals}[2, maxSub] \leftarrow lim$
$lower_{numIntegrals}[3, maxSub] \leftarrow 0$
$upper_i[1, maxSub] \leftarrow 0$
$upper_i[2, maxSub] \leftarrow 0$
$upper_i[3, maxSub] \leftarrow lim$
$upper_i[4, maxSub] \leftarrow 0$
else — there is only one term in the new limit
if ($x_{maxSub} \in a$)
$lower_i[1, maxSub] \leftarrow lim$
$lower_i[2, maxSub] \leftarrow 0$
$lower_i[3, maxSub] \leftarrow 0$
$upper_{numIntegrals}[1, maxSub] \leftarrow lim$
$upper_{numIntegrals}[2, maxSub] \leftarrow 0$
$upper_{numIntegrals}[3, maxSub] \leftarrow 0$
$upper_{numIntegrals}[4, maxSub] \leftarrow 0$
else
$lower_{numIntegrals}[1, maxSub] \leftarrow lim$
$lower_{numIntegrals}[2, maxSub] \leftarrow 0$
$lower_{numIntegrals}[3, maxSub] \leftarrow 0$
$upper_i[1, maxSub] \leftarrow lim$
$upper_i[2, maxSub] \leftarrow 0$
$upper_i[3, maxSub] \leftarrow$
$upper_i[4, maxSub] \leftarrow 0$
for ($j \leftarrow 1; j \leq k; j \leftarrow j + 1$)
if (($lower_i[2, j] \neq 0$) or ($lower_i[3, j] \neq 0$)) — max expression in the lower limit
$numIntegrals \leftarrow numIntegrals + 1$
$tempSet \leftarrow lower_i[2, j] \cup lower_i[3, j]$

$maxSub \leftarrow \text{maximum}X_i(tempSet)$ returns the highest subscript
$lim \leftarrow \text{solve}(lower_i[2,j] = lower_i[3,j], x_{maxSub})$
$a \leftarrow lower_i[2,j]$
$b \leftarrow lower_i[3,j]$
$upper_{numIntegrals} \leftarrow upper_i$
$lower_{numIntegrals} \leftarrow lower_i$
for $(c \leftarrow j; c \leq k; c \leftarrow c+1)$ set the limit as one term in the max expression
  if $((lower_i[2,c] = a)$ and $(lower_i[3,c] = b))$
    $lower_i[1,c] \leftarrow lower_i[1,m] + a$
    $lower_i[2,c] \leftarrow 0$
    $lower_i[3,c] \leftarrow 0$
    $lower_{numIntegrals}[1,c] \leftarrow lower_{numIntegrals}[1,c] + b$
    $lower_{numIntegrals}[2,c] \leftarrow 0$
    $lower_{numIntegrals}[3,c] \leftarrow 0$
if (nops$(lim) > 1)$ if there are multiple terms in the new limit
  if $(x_{maxSub} \in a)$
    $lower_i[1, maxSub] \leftarrow 0$
    $lower_i[2, maxSub] \leftarrow lim$
    $lower_i[3, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[1, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[2, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[3, maxSub] \leftarrow lim$
    $upper_{numIntegrals}[4, maxSub] \leftarrow 0$
  else
    $lower_{numIntegrals}[1, maxSub] \leftarrow 0$
    $lower_{numIntegrals}[2, maxSub] \leftarrow lim$
    $lower_{numIntegrals}[3, maxSub] \leftarrow 0$
    $upper_i[1, maxSub] \leftarrow 0$
    $upper_i[2, maxSub] \leftarrow 0$
    $upper_i[3, maxSub] \leftarrow lim$
    $upper_i[4, maxSub] \leftarrow 0$
else there is only one term in the new limit
    $lower_i[1, maxSub] \leftarrow lim$
    $lower_i[2, maxSub] \leftarrow 0$
    $lower_i[3, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[1, maxSub] \leftarrow lim$
    $upper_{numIntegrals}[2, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[3, maxSub] \leftarrow 0$
    $upper_{numIntegrals}[4, maxSub] \leftarrow 0$
  else
    $lower_{numIntegrals}[1, maxSub] \leftarrow lim$
    $lower_{numIntegrals}[2, maxSub] \leftarrow 0$

```
                    lower_{numIntegrals}[3, maxSub] ← 0
                    upper_i[1, maxSub] ← lim
                    upper_i[2, maxSub] ← 0
                    upper_i[3, maxSub] ← 0
                    upper_i[4, maxSub] ← 0
        finished ← 1
        for (g ← 1; g ≤ numIntegrals; g ← g + 1)
            for (j ← 1; j ≤ k; j ← j + 1)
                if ((lower_g[2, j] ≠ 0) or (lower_g[3, j] ≠ 0) or (upper_g[3, j] ≠ 0) or
                  (upper_g[4, j] ≠ 0))              test to determine if step 8 needs to be repeated
                    finished ← 0                                                        end while
for (i ← 1; i ≤ k; i ← i + 1)                                     begin Step 10 of algorithm
    dist[intOrder[i]] ← subs(x ← x_i, dist[intOrder[i]])
for (i ← 1; i ≤ m; i ← i + 1)
    if (S[i] = 0)                             find arcs on path on which single-use arcs appear
        temp = 0
        for (j ← 1; j ≤ numPaths; j ← j + 1)
            for (g ← 1; g ≤ n; g ← g + 1)
                if Paths[j, g] = i
                    for (l ← 1; l ≤ n; l ← l + 1)
                        if ((Paths[j, l] > 0) and (S[Paths[j, l]] = 1))
                            for (y ← 1; y ≤ k; y ← y + 1)
                                if (intOrder[y] = Paths[j, l])
                                    temp ← temp + x_y
        dist[i] ← ∫_{x_i=0}^{t−temp} dist[i] dx_i                  finds conditional CDF of single-use arcs
integrand ← 1
for (i ← 1; i ≤ m; i ← i + 1)
    integrand ← integrand ∗ dist[i]                                            constructs integrand
solution ← 0                                  sets up and evaluates k-fold integrals to find solution
for (i ← 1; i ≤ numIntegrals; i ← i + 1)                          begin Step 11 of algorithm
    tempSoln ← integrand
    for (j ← k; j > 0; j ← j − 1)
        tempSoln ← ∫_{x_j=lower_i[1,j]}^{upper_i[1,j]−upper_i[2,j]} tempSoln dx_j
    solution ← solution + tempSoln
return(solution)                                          return F_{T_n}(t): end of GetDistribution
```

Parameters: Starting node *node*, size of step *step*, current path number *path*, matrix of paths through the network *Paths*

Procedure name: `GetPaths`

Returned value: number of paths through network *numPaths* local $i, j, found, numpaths, total$

$i \leftarrow 1$ — initialize local variables

$found \leftarrow 0$

$numpaths \leftarrow 0$

$total \leftarrow 0$

while $(found < incoming[node])$ — while not all paths through a node are found

  if $N[node, i] = -1$ — arc i enters node *node*

    $numpaths \leftarrow$ GetPaths$(source[i], step + 1, path + total, Paths)$ — recursive step

    for $(j \leftarrow 0; j < numpaths; j \leftarrow j + 1)$

      $Paths[path + j + total, step] \leftarrow i$

    $total \leftarrow total + numpaths$

    $found \leftarrow found + 1$

  $i \leftarrow i + 1$

if $(total = 0)$

  $Paths[path, step] \leftarrow 0$

  $total \leftarrow 1$

return$(total)$ — end of GetPaths

Note: The function `GetPaths` is called recursively within `GetDistribution`.

The functions `swapcol` and `delrow` are included in Maple's `linalg` package and these functions swap columns of a matrix and delete rows of matrix, respectively. Maple also includes the set operations `intersect`, `union` and `minus` which are utilized in the implementation of the algorithm. The `convert(set, '+')` function takes a set a converts it to a sum of all the elements in the set.

Three basic Maple functions used in the implementation of the algorithm are: `copy`, which was used to copy one matrix variable to another leaving two independent copies of the matrix; `nops`, which was used to determine the number of operations in a set or function; and `solve`, which was used to solve a given equation for a specified variable in the equation.

The function `maximum`X_i is a simple function the user would have to implement that would return the x_i in a set that has the largest value of i.