Exploring Criticality Numbers For Kahn Process Networks

David Hofstee,

Philips Research Laboratories,
R. Holstlaan 4 WDCp-048, 5656 AA Eindhoven, The Netherlands

Delft University of Technology,
Faculty of Information Technology and Systems,
Computer Engineering Laboratory,
Mekelweg 4 (15th floor), 2628 CD Delft, The Netherlands

E-mail: D.Hofstee@ewi.tudelft.nl

July 18, 2003

Contents

List Of Figures III

List Of Tables V

ABSTRACT 1

1 Introduction 3

2 Background 7
- 2.1 Kahn Process Networks 7
- 2.2 Converting KPN Applications to Directed Weighted Graphs 8

3 Criticality Numbers For Acyclic KPNs 11
- 3.1 Introduction 11
- 3.2 General Objectives for Criticality Function 11
- 3.3 Formal Definition of Valid and Saturation Property 14
- 3.4 Criticality Function for Acyclic Kahn Process Networks 14
  - 3.4.1 Introduction to the Acyclic Algorithm 14
  - 3.4.2 Proof for Validity of Acyclic Algorithm 18
  - 3.4.3 Proof for Saturation Property of Acyclic Algorithm 21
  - 3.4.4 Complexity of Acyclic Algorithm 23
- 3.5 Example 23

4 Criticality Function for Cyclic Kahn Process Networks **27**
4.1 Introduction 27
4.2 Definition of Valid Criticality Function of Cyclic KPNs 27
4.3 Extra Definitions 29
4.4 Algorithm for Cyclic KPNs 33
4.4.1 Introduction 33
4.4.2 Motivation for Cyclic Algorithm 33
4.4.3 Algorithm for Acyclic KPNs 39
4.4.4 Example 41

5 Applications of Criticality Numbers **47**
5.1 Introduction 47
5.2 Visualization of Criticality Numbers 47
5.2.1 Introduction 47
5.2.2 Time Criticality Diagram 47
5.2.3 Accumulated Criticality Function 49
5.3 Adaptation of DAG Scheduling for Cyclic Weighted Task Graphs . . . 49
5.4 Executing a KPN on CAKE: Scheduling vs. Design Space Exploration . 51
5.5 Reducing Memory Needs 51

6 Implemented Parts **53**
6.1 Introduction 53
6.2 Tracer for Kahn Process Networks on Philips' CAKE Platform 53
6.3 Implementation of Criticality Function in C++ 54

7 Conclusions and Future Work **57**

A Source Code Acyclic Algorithm **59**
A.1 Introduction 59
A.2 ascf.cpp 60
A.3 Critlib.hpp 61

Bibliography **69**

List of Figures

1.1 "Y-Chart: a general scheme for heterogeneous system design":[2]. . . . 4

2.1 Example of code, using the select function. 8
2.2 A KPN application (left) is traced (center: directed acyclic graph of trace) and converted into weighted graph (right). 9

3.1 A graph and the corresponding time/process diagram. 12
3.2 Begin situation: example of a DAG with timing constraints *begin* and *end* (left) and after one iteration of the acyclic algorithm (right). 15
3.3 Graph G with the resulting criticality number. Indicated in the node is *name/criticality/weight*. 24
3.4 Graph after iterations of criticality algorithm. Indicated in the node is *name/0/weight*. 25

4.1 A KPN graph and possible traces. 28
4.2 Graph with cycles. 29
4.3 Possible trace from Fig. 4.2. The letter at a node indicate the process that generated the trace. 30
4.4 A cyclic graph and possible trace. 31
4.5 A *Cycle* in a *Cycle*. 32
4.6 KPN with two *Cycle*s that share some feed-forward nodes. 33
4.7 Example *Cycle*. 34
4.8 Example *Cycle* with indication of cut line. 35

4.9 *Feed-forward* part of the cycle. 35
4.10 *Feedback and feed-forward* part of the graph reworked. 36
4.11 **WRONG** j. Double time criticality diagram, j=25. 37
4.12 Final DAG which the acyclic algorithm can analyze. 38
4.13 Double time criticality diagram, j=19. 39
4.14 Example of cyclic graph. 42
4.15 First *Cycle* with attributes. 42
4.16 Second *Cycle* with attributes. 43
4.17 First *Cycle*, second *Cycle* removed. 43
4.18 Graph G with the first *Cycle* removed. 44
4.19 Graph G with *weight/criticality number* indicated inside node. 45

5.1 Time/criticality diagram of the example in Fig. 3.3. 48
5.2 Example of more complex time/criticality diagram of a random graph with 500 nodes. 48
5.3 Function cACC of the criticality function also depicted by the time/criticality diagram in Fig. 5.2. 50
5.4 Weighted DAG of KPN application 52

6.1 Two graphs generated by RandomLocalDigraph, n=200. Visualized by Dot 55
6.2 Automatically generated time/criticality diagram. Visualized by Gnuplot. 56

List of Tables

3.1 Several criticality functions for Fig. 3.1(a). c.f. stands for criticality function. 12

Exploring Criticality Numbers For Kahn Process Networks

Abstract

A Kahn process network is a model of task parallel programs. An implementation of Kahn process networks, Y-chart Application Programmers Interface, is used to program Philips' CAKE multiprocessor system. To execute an application, the tasks are mapped to processors and performance can be assessed by measuring total execution time. We introduce a function c, named criticality function, which indicates that process P can be executed on a processor which is slower by a factor of $c(P)^{-1}$ compared to the fastest processor. $c(P)$ has a value between 0 and 1. This thesis presents an algorithm which finds a criticality function, based on trace data, which does not incur a performance penalty, i.e., certain processes may be slowed down without an increase in execution time of the application. Slowing down the execution of processes can have advantages. For example, power consumption decreases when the clock frequency of a processor is reduced. Therefore, power can be saved by executing the process on a slower device. Another advantage is that the system designer can use the criticality function to see where (and by how much) to optimize the system. The criticality function shows which process is relatively critical and which is not.

1 Introduction

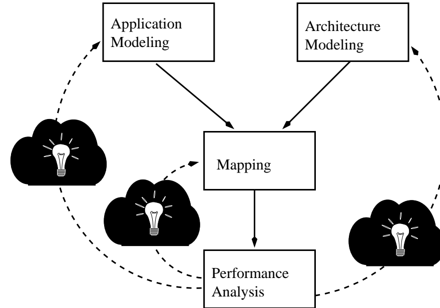
Kahn processing networks (KPNs[4]) is a model for task parallel programs. In this model, concurrent processes communicate via unidirectional FIFO's of unlimited size. This is modeled by a graph where nodes represent processes and directed edges represent FIFO's. Data is transported to/from the nodes via its directed edges. The process receives data via ingoing edges and sends data via outgoing edges. Executing a KPN application on a multiprocessor platform requires mapping each node to a processor.

Philips' implementation of KPN is part of Y-chart Application Programmers Interface (YAPI[2]) system design. YAPI provides concurrent C++ based on KPN with limited size FIFO's and support for nondeterministic events. It is used for the CAKE[6] heterogeneous multiprocessor platform. The CAKE platform has been created to overcome the verification problems that large systems on chip have. CAKE is formed by a set of equal tiles. Each tile is a heterogeneous multiprocessor system. The tiles are connected via a torus network. This method only requires the verification of a single tile to prove the correctness of a much larger system.

Four questions arise when using YAPI for system design[2] (see Fig. 1.1). First, how can an (existing) application be converted into a KPN? In [2] this is called *Application Modeling*. Second, how are the processes of this KPN mapped to a multiprocessor platform (*Mapping*)? Third, how is the architecture modeled (*Architecture Modeling*)? Fourth, what is the total performance of the system (*Performance Analysis*)? Performance analysis is supposed to help the system architect make (better) decisions about the other three questions.

When this research was started, it first focused on performance analysis. The specific

Figure 1.1: "Y-Chart: a general scheme for heterogeneous system design":[2].



question was: what is the number of (homogeneous) processors a KPN application can effectively use? This regardless of the actual mapping. This is a measure that indicates how well a program is converted into a KPN. Second, together with the actual number of processors used when mapped, it indicates how effective the mapping is. Third, it indicates how much of the architecture can be effectively used.

It was quickly discovered that if, for example, 2 processors are used 100% of the time during execution of an application, this does not provide a lot of information. If this is caused by two processes which run in parallel, no other mapping is possible. If this is caused by three processes, where one process uses a processor 100% of the execution time and two processes both need a processor 50% of the execution time, it is a possibility to use two processors. Another mapping can be made. Since the mapping is onto a multiprocessor platform the second and third process can be mapped onto a second and third processor. This can be advantageous for e.g. power dissipation, which is almost proportional to V_{dd}^2 for a given amount of tasks [8]. Because V_{dd} is proportional to execution speed, power dissipation is almost proportional to execution speed squared.

The answer to the question "what is the maximum number of processors that can effectively be used?" does not help to convert an application into a KPN application. It does not indicate where and why the conversion is successful or not. It only indicates the total performance when mapping is performed.

It can be seen that processor use, as described above, does not give a mapping tool any information on the possible mappings. Worse, our target platform is heterogeneous, and what does it mean when 2 processors are used, when they are not equal? How do they compare? So what makes a mapping of tasks onto various processors successful? Or if that mapping is not possible, how can the architecture modeling be done in such a way that it is possible to find an effective mapping?

The effectiveness of the mapping of a process to a processor is influenced by many factors, such as instruction set architecture, clock frequency, cache configuration, etc. For both practical and methodological reasons, it was chosen to quantify this effectiveness in a scalar. This means that for each task, if mapped on its own processor, the

necessary clock frequency for the processor is calculated. A single processor architecture is used. When this frequency is determined for all tasks, one of the tasks requires the highest clock frequency. All clock frequencies are normalized to this frequency. The result is a function c that maps each task/process of the KPN to a value in the range $(0, 1]$.

This function c, named criticality function, indicates that process P can be executed on a processor which is a factor $c(P)^{-1}$ slower than the fastest processor. This thesis presents two algorithms which can calculate a criticality function. First, an algorithm that can operate on an acyclic KPN is presented. Then this algorithm is extended to find a criticality function for cyclic KPNs.

A criticality function makes it possible to map a KPN application to a heterogeneous multiprocessor platform. It can be identified which processes should be mapped to fast processors and which might be scheduled on slower ones. If performance problems arise then it is clear where they originate from: mostly from processes with a high criticality number (criticality function value for that node) because they take up too much processing time. Either the code of that process needs to be optimized (Application Modeling) or more specialized hardware can be used to execute this process faster (Architecture Modeling). A non-optimal mapping should be more easily identifiable.

This thesis is organized as follows. Because our analysis is not performed on KPNs, but on weighted directed graphs, Chapter 2 briefly describes how these graphs are obtained from a KPN. Chapter 3 describes and partly defines the requirements that criticality functions have to meet before they can be considered useful. It then continues and provides an algorithm for acyclic KPNs which meets these requirements. Chapter 4 amends the algorithm for acyclic KPNs so that it can be used for cyclic KPNs. Chapter 5 elaborates on a few applications for criticality functions. Chapter 6 describes the implementation of the trace utility for KPNs on the YAPI/CAKE platform and the implemented algorithm from Chapter 3. Chapter 7 finishes this thesis with conclusions and possible future work.

2 Background

2.1 Kahn Process Networks

Kahn processing networks (KPNs[4]) is a model of task parallel programs. In KPNs, concurrent processes communicate via unidirectional, unbounded FIFO's. This is modeled by a graph where nodes represent processes and directed edges represent FIFO's. Data is transported to/from the nodes via its directed edges. The process receives data via ingoing edges and sends data via outgoing edges.

To quote [2]: "Each of the processes performs sequential computation on its private state space. The computation actions are interleaved with communication actions that read data from input channels and write data to output channels. Read actions are blocking, i.e., a process that read from an empty channel stalls until the channel has sufficient data to complete the read action. Write actions are non-blocking because the channels have unbounded capacity. A well-known property of a KPN is that it is deterministic, i.e., the stream of data that travels along each channel is determined by the given input data; it does not depend on the order in which the processes are executed. For this reason, an application programmer can combine processes that represent signal processing functions into process networks without specifying their order of execution. Moreover, a system designer can exploit the concurrency between the processes by using processing elements that operate in parallel."

A KPN cannot be implemented since unbounded FIFO's do not exist, they have to be simulated. Second, KPNs have the disadvantage that non-deterministic interaction, e.g.

Figure 2.1: Example of code, using the select function.

```
if (select(fifo1, fifo2)==0)
  read(fifo1, x);
else
  read(fifo2, x);
```

user interaction, is not possible. This is overcome in Y-chart Application Programmers Interface (YAPI[2]). YAPI adds channel selection, which means the program can select any channel for a read action. In Fig. 2.1 there is an example of how this is done in YAPI. If data is present in *fifo1*, then *select()* returns 0. Otherwise, *select()* returns 1. If none of the buffers has data, the process is blocked. Another change is that the size of FIFO's is dynamically adjusted in YAPI. For our purposes, the difference between YAPI and KPN is irrelevant. We do not focus on the non-deterministic behavior and the FIFO's are generally not an issue in the YAPI/CAKE system[1].

2.2 Converting KPN Applications to Directed Weighted Graphs

This paragraph explains how a KPN application is modeled as a weighted directed graph. This graph will be used by proposed algorithms to obtain a criticality function.

We extended the CAKE framework[6] developed at Philips so that the execution of KPNs can be traced. First, the conversion of acyclic KPNs is explained. An acyclic KPN is supplied a small test input and executed on a single processor/simulator. When the result is produced, a weighted DAG $G = (V, E, w)$ can be constructed whose nodes correspond to the processes of the KPN, which has an edge (u, v) if the process corresponding to node u communicated with the process corresponding to node v, and where the weight $w(v)$ of node v matches the execution time of the corresponding process. In some cases (in particular, when a process is preempted while it is being executed), a process may correspond to two or more nodes.

For cyclic KPNs, a trace is also generated. For cyclic KPNs, a trace can be seen as unrolling the KPN graph into an acyclic trace. To obtain the weighted cyclic graph, the reverse operation is performed. In Fig. 2.2 this is illustrated. A KPN application is traced. The result is a weighted graph with the same topology as the original KPN application. It is assumed that no process is preempted while being executed and traced, which is a fair assumption in the YAPI/CAKE system.

We further assume that the *relative* execution time of each process is independent of the input data. In other words, if node u requires time $w_1(u)$ on input I_1 and time $w_2(u)$ on input I_2, and node v requires time $w_1(v)$ on input I_1 and time $w_2(v)$ on input I_2, then $w_1(u)/w_1(v) = w_2(u)/w_2(v)$. Many signal processing applications have this property.

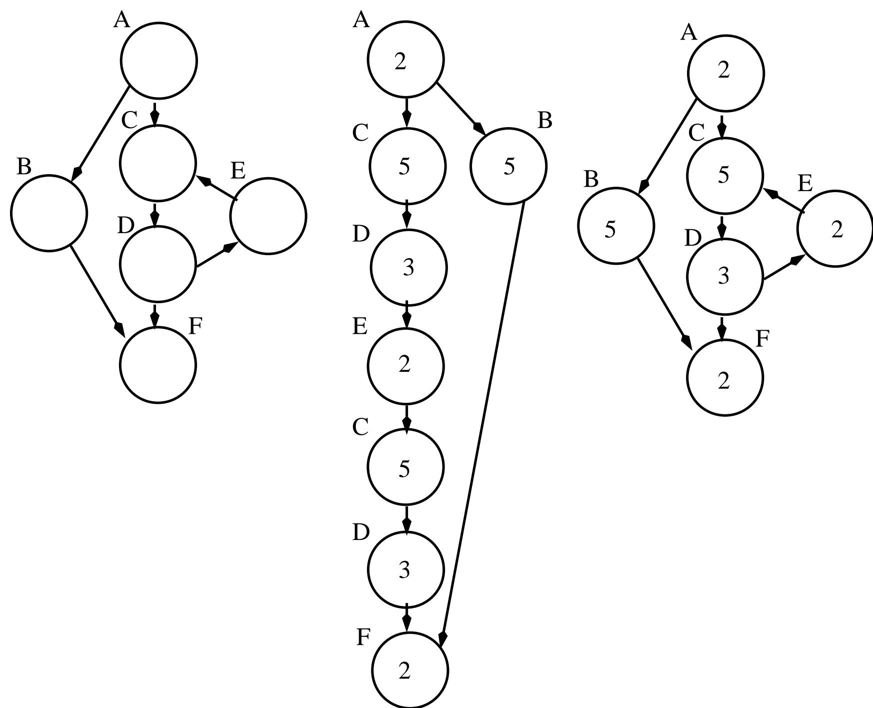


Figure 2.2: A KPN application (left) is traced (center: directed acyclic graph of trace) and converted into weighted graph (right).

Criticality Numbers For Acyclic KPNs

3

3.1 Introduction

In this chapter it is first explained what objectives were identified for criticality functions and how they guide decisions that were made in the algorithms. Then two of those objectives for criticality functions are defined formally and an algorithm is presented which meets these objectives for acyclic Kahn process networks. Finally, an example is provided. In the next chapter, an algorithm for cyclic KPNs is given.

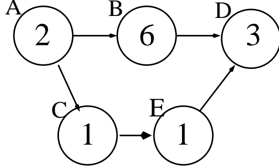
3.2 General Objectives for Criticality Function

Consider the weighted directed graph G, depicted in Fig. 3.1(a). In this figure, the weight w of each node equals the (minimum) processing time it requires. When node A has been processed, nodes B and C can start. When C finishes, E can start, and when B and E have been processed, node D can start processing. In Fig. 3.1(b) a diagram is given that illustrates the execution when each process is executed on a separate processor.

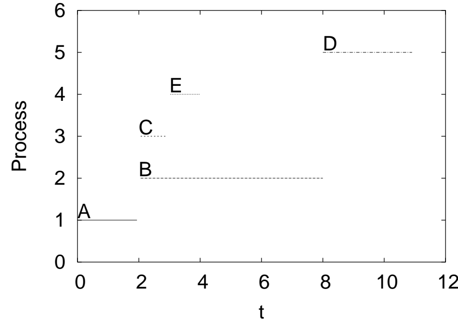
The criticality function $c : V \to (0, 1]$ indicates that process/node P can be executed on a processor which is a factor $c(P)^{-1}$ slower than the fastest processor. Many criticality functions can be found for any given KPN application. Table 3.1 illustrates the wide range of criticality functions with a few examples. It also illustrates the "valid" property, cost and "saturation" which will all be explained.

Figure 3.1: A graph and the corresponding time/process diagram.

(a) Graph corresponding to a KPN



(b) Time/Process diagram



The total execution time can be determined by calculating:

$$\text{total exec time} = \frac{w(A)}{c(A)} + \max\left\{\frac{w(B)}{c(B)}, \left(\frac{w(C)}{c(C)} + \frac{w(E)}{c(E)}\right)\right\} + \frac{w(D)}{c(D)}. \tag{3.1}$$

The cost function of Table 3.1 is given as:

$$\text{cost} = \sum_{v\in G} w(v)c(v). \tag{3.2}$$

This cost function indicates that a process which is executed at half speed is half as expensive to execute (even though it would take twice the time). A decreasing cost for the first four rows in the cost column in Table 3.1 is shown. In real life, doubling execution speed more than doubles the cost. For example, to indicate the cost of power dissipation, $w(v)c(v)^2$ would be a better cost function.

Table 3.1: Several criticality functions for Fig. 3.1(a). c.f. stands for criticality function.

criticality function	c(A)	c(B)	c(D)	c(C)	c(E)	total exec time	cost
homogeneous c.f.	1	1	1	1	1	11	13
valid c.f.	1	1	1	0,5	0,5	11	12
valid & saturated c.f.	1	1	1	0,5	0,25	11	11,75
valid & sat. & cost eff. c.f.	1	1	1	0,33	0,33	11	11,66
random & invalid c.f.	0,9	0,8	0,7	0,4	0,3	14,008	9,4

For each KPN, there is a minimum execution time which is determined by the critical path (when a fixed clock speed is assumed). Criticality functions which preserve this minimum execution time are useful. This property is called validity and it is defined formally in Section 3.3.

Another consideration for a criticality function is the cost of using the processors that the processes are mapped to. A processor is considered a resource, and a fast processor is considered a more valuable resource than a slow one. The criticality function determines the lowest speed a processor can have so that the process meet its deadlines. A criticality number which is chosen higher than necessary is considered inefficient because it rules out the use of/mapping to a less valuable resource.

The algorithms that are presented later use this cost *intuition*. The algorithms all select a path from the KPN graph and assign a criticality number to all the nodes from that path. This number is equal for all nodes from this path. The next iteration of the algorithms selects a path of nodes which do not yet have a criticality number. This next iteration does the same, with the condition that the criticality number is equal or lower to the one from the previous iteration. The algorithm iterates until all nodes have a criticality number. The idea is that a lower criticality number for a process allows for a mapping to a slower processor which is considered less valuable than a high speed processor. Thus the total cost can go down.

Another objective for a criticality function is the saturation property. Saturation indicates that the criticality number for each individual process cannot be lower, or else the validity criterion would not be met. A lower criticality number would violate its deadline and therefore make the total execution time larger than the original critical path length.

Several observations can be made from the criticality functions in Table 3.1. First, it can be seen that all valid criticality functions have the same execution time as the homogeneous criticality function (where for all nodes v, $c(v) = 1$) which always guarantees a minimum execution time. Second, it can be seen that a valid criticality function has a lower or equal cost compared to the homogeneous criticality function. Third, a saturated criticality function can have a lower cost than a non-saturated criticality function. Fourth, even saturated criticality functions can be cost-optimized. Fifth, there are many other criticality functions. E.g. the one named *random* has a lower cost than any of the valid criticality functions. The algorithms we present would find the fourth criticality function from Table 3.1.

To conclude, validity, cost and saturation guide the choices that we made for the algorithms. This is not complete. If more is known about the target platform or the application then other choices may make more sense. In Chapter 5 an example of this is given. The algorithms we created find criticality functions. These functions (and their corresponding algorithms) should only be seen as reference implementations of the set of functions that can be found.

3.3 Formal Definition of Valid and Saturation Property

Consider graph G from Fig. 3.1(a) again. In this example, the time needed by node B ($w(B) = 6$) is larger than $w(C) + w(E) = 2$. Because of this, node C and E could be executed on a processor that is $(w(C) + w(E))/w(B) = 1/3$ times as fast (in other words, 3 times as slow) as the processor that executes node B without incurring a performance penalty. This is what criticality numbers indicate. They indicate by how much a process is (allowed to be) slowed down. The goal is to create criticality numbers that do not increase the overall execution time. Obviously, the nodes on the critical path (in this case (A, B, D)) should have a criticality of 1, since they cannot be slowed down without incurring a performance penalty. Criticality numbers are properties of the network program and do not depend on the architecture it is executed on.

The following definition formally defines when a criticality assignment to an acyclic graph is valid.

Definition 1. *Given a weighted DAG $G = (V, E, w)$ with critical path length L. A criticality function $c : V \to (0, 1]$ is* valid *if for every path $p = (v_1, v_2, \ldots, v_n)$ in G*

$$\sum_{v \in p} w(v)/c(v) \leq L.$$

This condition ensures that there is no performance penalty if each node v is slowed down by a factor of $c(v)^{-1}$.

Definition 2. *Given a weighted DAG $G = (V, E, w)$ with critical path length L. A valid criticality function $c : V \to (0, 1]$ is* saturated *if for every node u there exists a path p through v such that*

$$\sum_{v \in p} w(v)/c(v) = L.$$

This condition states that each node v cannot be slowed down by more that a factor of $c(v)^{-1}$ because this will increase the time needed to execute the DAG.

3.4 Criticality Function for Acyclic Kahn Process Networks

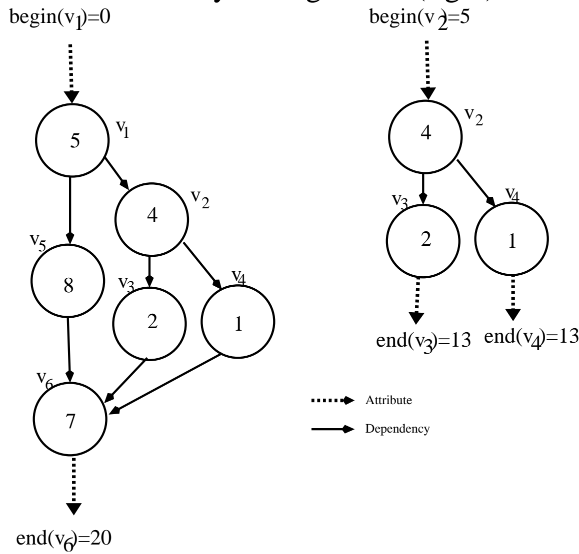
3.4.1 Introduction to the Acyclic Algorithm

An algorithm to obtain a valid saturated criticality function for an acyclic Kahn process network is presented here. It is different from the algorithm in [3] which was published earlier. The algorithm will be referred to as the *acyclic algorithm* because it finds a function for acyclic KPNs. To explain how the acyclic algorithm works we start of with a weighted DAG $G = (V, E, w)$. Node v in G has two attributes, *begin*(v) and *end*(v).

begin(v) is the earliest point in time node v can start processing and *end*(v) is the latest point in time node v must be finished.

Nodes without a predecessor (initial nodes) have their *begin* attribute set to 0. Nodes without a successor (final nodes) have their *end* attribute set to (critical path length) *L*. All remaining nodes do not have meaningful *begin* or *end* attributes yet. In the algorithm, nodes with valid *begin* attributes are put in set B, similar for *end* attributes and set U. An example of a begin situation is given in Fig. 3.2 on the left.

Figure 3.2: Begin situation: example of a DAG with timing constraints *begin* and *end* (left) and after one iteration of the acyclic algorithm (right).



The acyclic algorithm has an initialization (see Alg. 2) and a body (see Alg. 1). The algorithm finds a path starting at a node with a valid *begin* attribute and ending at a node with a valid *end* attribute. It selects the path $p = v_1, \ldots, v_n$ with the highest ratio of *pathlength* to *time available*. The *pathlength* represents the time needed to complete execution of nodes on that path. The time available is calculated by *end*(v_n)-*begin*(v_1), which is always more or equal to the time needed. The execution of nodes can be slowed down with that ratio (called criticality number). By taking the maximum criticality number, it is ensured that criticality numbers are decreasing each iteration. This makes it impossible to get criticality values larger than 1. In the algorithm, p is called $path_{v_1}(v_n)$. The criticality number $c(v_i)$ equals :

$$v_i \in p : c(v_i) = pathlength_{v_1}(v_n)/(end(v_n) - begin(v_1)) \tag{3.3}$$

and $pathlength_{v_1}(v_j)$ is defined by:

$$pathlength_{v_1}(v_j) = w(v_1) + \cdots + w(v_j). \tag{3.4}$$

The acyclic algorithm has the same criticality number for all nodes on path p, which has the most restrictive *begin* and *end* attributes. Other choices could have been made which result in different criticality functions. This would violate the cost intuition mentioned earlier.

Then, path p is cut out and nodes with edges to path p get valid *begin* and/or *end* attributes. The result is a graph with *begin* and *end* attributes. The algorithm can iterate again until the graph is empty. Each iteration, the initialization of *pathlength* and *path* needs to be done again. New *begin* and *end* attributes are added to B and U and other paths are removed because p is removed.

Algorithm 1 Body of acyclic algorithm.

1 *InitializeAttributes*
2 **while** $V \neq \emptyset$ **do**
3   *InitializePathAndPathlength()*
4   *DeterminePathlengthAndPath()*
5   *DeterminePathWithHighestCriticality()*
6   *RemovePathWithHighestCriticality()*
7   reorganize V such that it is again topologically ordered.
8 **elihw**

Algorithm 2 *InitializeAttributes()*, set *begin* and *end* attributes for entry and exit nodes.

9 Consider weighted directed acyclic graph $G = (V, E, w)$ with
10 critical path length L
11
12 let V be the topologically sorted $\{v_1, \ldots, v_n\}$
13 let $pred(v)$ be defined as set: $\{u|(u, v) \in E\}$
14 let $succ(v)$ be defined as set: $\{u|(v, u) \in E\}$
15
16 $B = \{u|pred(u) = \emptyset\}$
17 $U = \{w|succ(w) = \emptyset\}$
18
19 **for** each $v \in B$ **do**
20   *begin*$(v) = 0$
21 **for** each $v \in U$ **do**
22   *end*$(v) = L$

The first iteration the algorithm finds a critical path because it has the largest *pathlength / time available* ratio (of 1). All *begin* attributes were set to 0 and all *end* attributes were set to the critical path length L. The largest path length is equal to the critical path

Algorithm 3 *Initialize$PathAndPathlength$()*, resetting $path$ and $pathlength$ for a new iteration.

```
23  for each v ∈ V do
24      for each w ∈ B do
25          pathlength_w(v) ← −1
26          path_w(v) ← ∅
27
28  for each v ∈ B do
29      pathlength_v(v) ← w(v)
30      path_v(v) ← v
```

Algorithm 4 *Determine$PathlengthAndPath$()*.

```
31  for each v ∈ V do                 //has to be done in topological order
32      for each u ∈ pred(v) do       //has the effect that u is updated before v
33          for each w ∈ B do
34              if (pathlength_w(u) ≠ −1) then
35                  if (pathlength_w(v) < pathlength_w(u) + w(v)) then
36                      pathlength_w(v) ← pathlength_w(u) + w(v)
37                      path_w(v) ← path_w(u) ∪ {v}
```

length so the ratio becomes 1. The DAG that remains after one iteration is depicted in Fig. 3.2 on the right.

Algorithm 5 *DeterminePathWithHighestCriticality()*.

```
38  c_max ← 0
39  for each u ∈ U do
40      for each v ∈ B do
41          if (pathlength_v(u) ≠ −1) then
42              c ← pathlength_v(u)/(end(u) − begin(v))
43              if (c > c_max) then
44                  c_max ← c
45                  end_max ← u
46                  beg_max ← v
47                  path_max ← path_v(u)
```

Algorithm 6 *RemovePathWithHighestCriticality().*

```
48 for each v ∈ path_max do
49     c(v) ← c_max
50     begin(v) ← begin(beg_max) + (pathlength_{beg_max}(v) − w(v))/c(v)
51     end(v) ← end(end_max) − (pathlength_{beg_max}(end_max) − pathlength_{beg_max}(v))/c(v)
52     for each u ∈ pred(v) do
53         if (u ∉ U) then
54             end(u)=begin(v)
55         if (end(u) > begin(v)) then
56             end(u) ← begin(v)
57         remove (u, v) from E
58         U ← U ∪ {u}
59     for each u ∈ succ(v) do
60         if (u ∉ B) then
61             begin(u)=end(v)
62         if (begin(u) < end(v)) then
63             begin(u) ← end(v)
64         remove (v, u) from E
65         B ← B ∪ {u}
66     V ← V \ {v}
67     B ← B \ {v}
68     U ← U \ {v}
```

3.4.2 Proof for Validity of Acyclic Algorithm

This section provides proof that the acyclic algorithm yields a valid criticality function. Theorem 1 uses Lemma's 1 to 3 to proof that the criticality function is valid and in the range (0,1]. In the next paragraph, the saturation property is proven.

Lemma 1. *If the acyclic algorithm would generate an invalid criticality function, then there exists a begin(v) smaller than 0 or an end(v) larger than L for at least one node v after applying the acyclic algorithm.*

Proof. Suppose the acyclic algorithm generates an invalid criticality function $c_{invalid}$ and there exists at least one path $p = (v_1, \ldots, v_n)$ for which:

$$\sum_{v \in p} w(v)/c_{invalid}(v) > L \tag{3.5}$$

holds. From lines 50-51 it follows that:

$$\mathit{end}(v) - \mathit{begin}(v) = \mathit{end}(v_n) - \mathit{begin}(v_1) - \frac{pathlength_p(v_n) - pathlength_p(v) + pathlength_p(v) - w(v)}{c(v)} \tag{3.6}$$

and from line 42 it can be seen that $c(v)$ equals:

$$c(v) = pathlength_p(v_n)/(\textbf{\textit{end}}(v_n) - \textbf{\textit{begin}}(v_1)). \tag{3.7}$$

So that for all nodes v:

$$c(v) = w(v)/(\textbf{\textit{end}}(v) - \textbf{\textit{begin}}(v)) \tag{3.8}$$

is true after the acyclic algorithm is finished. So Equation 3.5 can be rewritten into:

$$\sum_{v \in p} (\textbf{\textit{end}}(v) - \textbf{\textit{begin}}(v)) > L. \tag{3.9}$$

Because node v_i should only start after its predecessors, and v_{i-1} is one, $\textbf{\textit{end}}(v_{i-1}) \leq \textbf{\textit{begin}}(v_i)$ should be true.

Two situations can occur. First, both vertices were removed in the same iteration. Then, using Eq. 3.4 to rewrite *begin* and *end* attributes from the algorithm, for any node u_{i-1} and u_i that are both removed in an iteration of the algorithm, $\textbf{\textit{end}}(u_{i-1}) = \textbf{\textit{begin}}(u_i)$:

$$\textbf{\textit{end}}(u_{i-1}) = \textbf{\textit{end}}(u_n) - \frac{\left(\sum_{v \in u_1,\ldots,u_n} w(v) - \sum_{v \in u_1,\ldots,u_{i-1}} w(v)\right)}{c(u_{i-1})} \tag{3.10}$$

and

$$\textbf{\textit{begin}}(u_i) = \textbf{\textit{begin}}(u_1) - \frac{\left(\sum_{v \in u_1,\ldots,u_i} w(v) - w(u_i)\right)}{c(u_i)} \tag{3.11}$$

If these equations are set equal, and $c(u)$ is assumed the same for all nodes of $(u_1, \ldots, u_n)$ then

$$c = \frac{\sum_{v \in u_1,\ldots,u_n} w(v)}{\textbf{\textit{end}}(u_n) - \textbf{\textit{begin}}(u_1)} \tag{3.12}$$

results. This exact equation is in the algorithm. Second, if the vertices are not removed in the same iteration, $\textbf{\textit{begin}}(v_i) \geq \textbf{\textit{end}}(v_{i-1})$ is upheld by lines 52-58. So $\textbf{\textit{begin}}(v_i) - \textbf{\textit{end}}(v_{i-1}) \geq 0$.

Then Equation 3.9 can be rewritten as:

$$\begin{aligned} &end(v_n) - begin(v_1) - \\ &\textbf{(}begin(v_n) - end(v_{n-1}) + \cdots + \textbf{\textit{begin}}(v_2) - \textbf{\textit{end}}(v_1)\textbf{)} \quad > L \end{aligned} \tag{3.13}$$

which can be rewritten as:

$$end(v_n) - begin(v_1) - \textbf{(}\delta\textbf{)} > L \quad (\delta \geq 0) \tag{3.14}$$

which equals:

$$\textbf{\textit{end}}(v_n) - \textbf{\textit{begin}}(v_1) > L + \delta \quad (\delta \geq 0). \tag{3.15}$$

This can be simplified into:

$$\textit{end}(v_n) - \textit{begin}(v_1) > L. \quad (3.16)$$

This can only be true if the lemma is true. □

Lemma 2. *In the acyclic algorithm, the lowest begin attribute is assigned to an initial node of the original graph. Also, the highest end attribute is assigned to an exit node of the original graph.*

Proof. Entry nodes are put in set B, similar to initial exit nodes in U. Every node that is cut out in an iteration is part of a path $(v_1, \ldots, v_m)$. v_1 was an element of B and v_m was an element of U. Both equations:

$$\textit{begin}(v_1) \leq \textit{begin}(v_2) \leq \cdots \leq \textit{begin}(v_m) \quad (3.17)$$

$$\textit{end}(v_1) \leq \cdots \leq \textit{end}(v_{m-1}) \leq \textit{end}(v_m) \quad (3.18)$$

are true. It holds that the smallest *begin* attributes are from nodes that were in B and the largest *end* attributes are from nodes that were in U. If a node v was later added to B then predecessor u of v was part of a path which was cut out in an earlier iteration. *begin(u)* was smaller than *begin(v)*. Therefore *begin*(v) is always larger than the smallest *begin* attribute of an initial node in the original graph. If a node w was later added to U a similar proof holds. □

Lemma 3. *The acyclic algorithm assigns a begin(v) $\geq$ 0 and an end(v) $\leq L$ attribute to all nodes v in G.*

Proof. First, the smallest *begin* attributes are from elements that are initial nodes. Second, initially those *begin* attributes are set to 0. A similar proof holds for the *end* attribute. □

Lemma 4. *Each iteration of the acyclic algorithm yields a lower or equal criticality value.*

Proof. Assume that iteration n yields a criticality of c_n and $path_n$ is cut out of V. Suppose iteration $n+1$ yields a higher criticality. There are two ways for that to happen. First, $path_{n+1} = (v_1, \ldots, v_m)$ has a c_{n+1} and both v_1 and v_m were part of B and U in iteration n and their *begin* and *end* attributes are unchanged. The lemma holds because the relation was calculated on line 39-41 explicitly in iteration n. The only difference is that some nodes are removed in iteration n. This can make the path between v_1 and v_m *shorter* in iteration $n+1$. The *begin* and *end* attributes are not changed. This means c_{n+1} can only be equal or lower. The second thing that can happen: v_1 and/or v_m are in B and/or U, or the *begin* or *end* attributes are changed, because of iteration n. This means either v_1 and/or v_m was a predecessor and/or successor of a node of $path_n$ (line 52/59). If $path_{n+1}$ would yield a higher c, then another path should have been constructed in

iteration n (e.g. $(u_1, \ldots, u_i, v_1, \ldots, v_m)$). This path has an c' which is higher than c_n and lower than c_{n+1}. Since the maximum c (i.e.. c_n) was selected in iteration n, the lemma holds. □

Theorem 1. *The acyclic algorithm yields a valid criticality function.*

Proof. In the first iteration, a path is cut out of V with criticality equal to 1. Since all iterations of the acyclic algorithm yield equal or smaller criticality values, the range of criticality values is guaranteed.

The acyclic algorithm assigns a *begin*$(v) \geq 0$ and an *end*$(v) \leq L$ for all nodes v in G. Because of Lemma 1, this would not be possible for an invalid criticality function. It is therefore valid. □

3.4.3 Proof for Saturation Property of Acyclic Algorithm

Saturation means that each node v cannot be slowed down by *more* than a factor of $c(v)^{-1}$ because this will increase the time needed to execute the DAG.

Theorem 2. *The acyclic algorithm determines a saturating criticality function.*

Proof. Each node is cut out in an iteration in the acyclic algorithm. Let node v_i be a node from $path_j = (v_1, \ldots, v_i, \ldots, v_n)$ which is cut out in iteration j. Also, and there must exist one path p for which:

$$\sum_{v \in p} w(v)/c(v) = L. \tag{3.19}$$

Path p is divided into three pieces (q, $path_j$ and s). Then Eq. 3.19 equals:

$$\sum_{v \in q} w(v)/c(v) + \sum_{v \in path_j} w(v)/c(v) + \sum_{v \in s} w(v)/c(v) = L. \tag{3.20}$$

If there exists a path q such that:

$$\sum_{v \in q} w(v)/c(v) = \textit{begin}(v_1) - 0, \tag{3.21}$$

there exists a path s such that:

$$\sum_{v \in s} w(v)/c(v) = L - \textit{end}(v_n) \tag{3.22}$$

and

$$(\textit{begin}(v_1) - 0) + \sum_{v \in path_j} w(v)/c(v) + (L - \textit{end}(v_n)) = L \tag{3.23}$$

holds, then the Theorem is true.

$$(\textit{begin}(v_1) - 0) + \sum_{v \in path_j} w(v)/c(v) + (L - \textit{end}(v_n)) = L \tag{3.24}$$

equals

$$\textit{end}(v_n) - \textit{begin}(v_1) = \sum_{v \in path_j} w(v)/c(v) \tag{3.25}$$

and because $c(v)$ is equal for all nodes of $path_j$, it equals

$$c(v_i) = pathlength_{v_1}(v_n)/\textit{end}(v_n) - \textit{begin}(v_1) \tag{3.26}$$

and that is used in the acyclic algorithm (line 42).

Path q of Eq. 3.21 and path s of Eq. 3.22 need to exist. In iteration k ($< j$), *begin*(v_1) was assigned to vertex v_1. The first possibility is that *begin*(v_1)=0, which is done at initialization. In that case, v_1 has no predecessors and Eq. 3.21 holds because q is empty. The second possibility is that predecessor u of v_1 assigned a non-zero value to *begin*(v_1). This is done in lines 59-65. u is in $path_k = (u_1, \ldots, u, \ldots, u_n)$. We can again subdivide path q and rewrite Eq. 3.21 into:

$$\sum_{v \in q'} w(v)/c(v) + \sum_{v \in (u_1, \ldots, u)} w(v)/c(v) = \textit{end}(u) = \textit{begin}(v_1) \tag{3.27}$$

The second summation of Eq. 3.27 equals *end*(u)-*begin*(u_1) which is proven later. Then Eq. 3.27 equals to:

$$\sum_{v \in q'} w(v)/c(v) = \textit{begin}(u_1). \tag{3.28}$$

This process can be recursively iterated until the *begin* equals 0. Similar proof holds for set s.

Proof for:

$$\sum_{v \in (u_1, \ldots, u)} w(v)/c(v) = \textit{end}(u) - \textit{begin}(u_1) \tag{3.29}$$

can be rewritten into:

$$\sum_{v \in (u_1, \ldots, u)} \textit{end}(v) - \textit{begin}(v) = \textit{end}(u) - \textit{begin}(u_1). \tag{3.30}$$

If in such a path *end*(u_{i-1}) $=$ *begin*(u_i) holds, Eq. 3.30 is true. Using Eq. 3.4 to rewrite *begin* and *end* attributes from the algorithm gives

$$\textit{end}(u_{i-1}) = \textit{end}(u) - \frac{\left(\sum_{v \in u_1, \ldots, u} w(v) - \sum_{v \in u_1, \ldots, u_{i-1}} w(v)\right)}{c(u_{i-1})} \tag{3.31}$$

and

$$\textit{begin}(u_i) = \textit{begin}(u_1) - \frac{\left(\sum_{v \in u_1, \ldots, u_i} - w(u_i)\right)}{c(u_i)} \tag{3.32}$$

If these equations are set equal, and $c()$ is assumed the same for all nodes of $(u_1, \ldots, u)$ then

$$c = \frac{\sum_{v \in u_1,\ldots,u} w(v)}{\textit{end}(u) - \textit{begin}(u_1)} \tag{3.33}$$

results. This exact equation is in the algorithm. Therefore the theorem holds. □

3.4.4 Complexity of Acyclic Algorithm

The complexity of the acyclic algorithm is $O(|V|^3|E|)$, where the $|.|$ operator returns the number of elements of a set. Each iteration, at least 1 node is removed from the graph. This means the number of iterations is smaller or equal to $|V|$. From the algorithm, Alg. 5 has the highest complexity. First the number of elements in V, $pred(v)$ and set B are estimated:

- In iteration n, there are maximally $|V|_n = |V| - n + 1$ nodes in V.
- The nested **for** loop is executed $|V|_n$ times maximally.
- Set B contains $|V|_n$ elements at most.
- The $\sum_{v \in V} |pred(v)|$ equals the number of edges in iteration n. This will be estimated at $|E|$.

Then the maximum number of operations can be expressed as:

$$\sum_{i=1}^{|V|} |V|_i \, |E| \, |V|_i \,\text{constant} \tag{3.34}$$

which (for large $|V|$) converges to $|V|^3|E|$ which makes the complexity $O(|V|^3|E|)$.

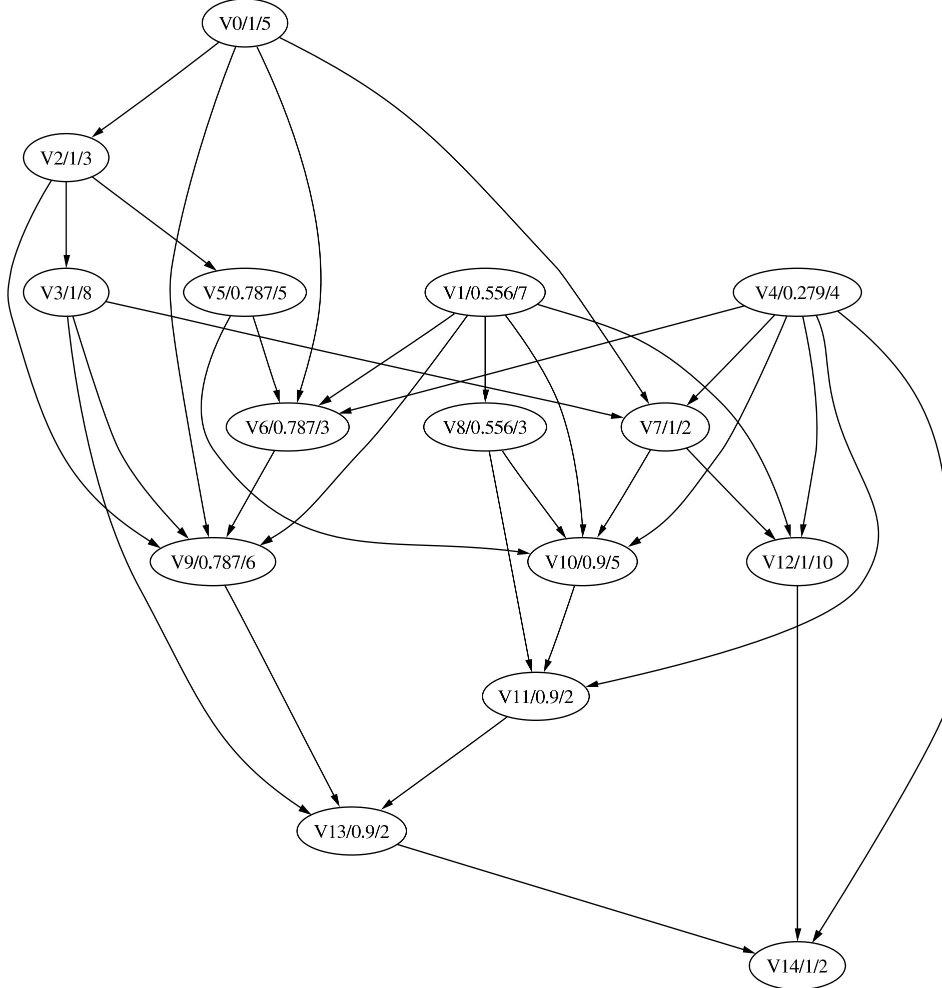
3.5 Example

Let Fig. 3.3 depict graph G. The name/criticality/weight is indicated in the node. The different parts of Alg. 1 will be calculated for one iteration.

In *InitializeAttributes()*, set B and U are determined. The topological order is already determined by the index number of the node. Here, for example, $V12$ has an index number 12. Set B is the set with vertices without predecessors: $\{V0, V1, V4\}$. Set U is the set with vertices without successors: $\{V14\}$.

In *InitializePathAndPathlength()* the three pairs of attributes, matching the three nodes in B are reset. This means for each node v, $path_{V0}(v) = path_{V1}(v) = path_{V4} = \emptyset$. Also, $pathlength_u(v)$ is set to -1 in a similar way. Then, for nodes u in B, $path_u(u) = \{u\}$ and $pathlength_u(u) = w(u)$.

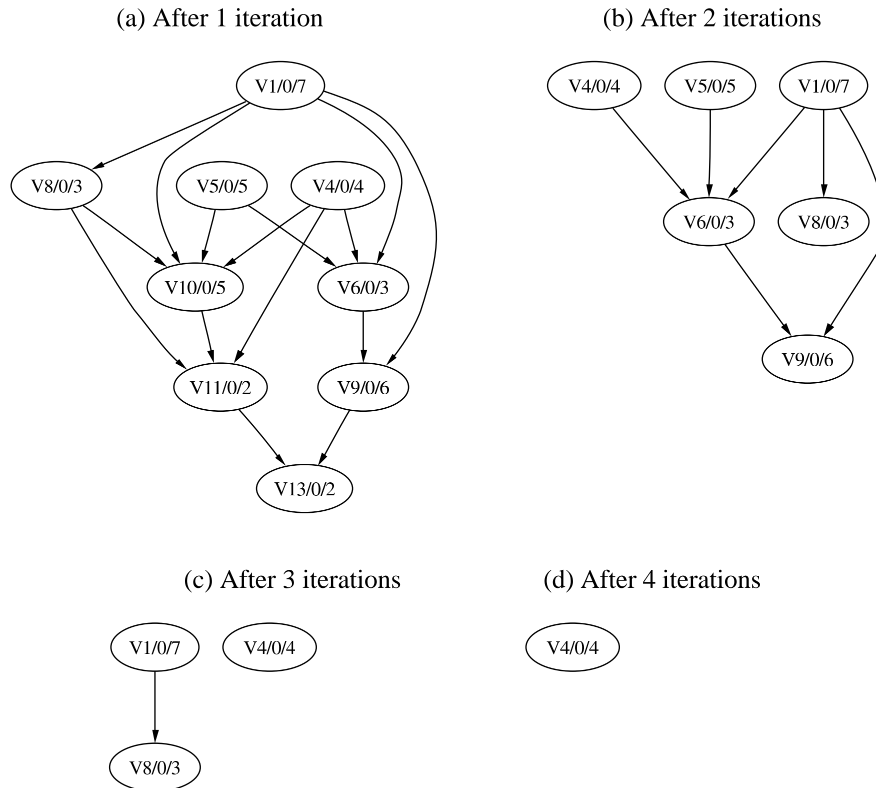
Figure 3.3: Graph G with the resulting criticality number. Indicated in the node is *name/criticality/weight*.



In *DeterminePathAndPathlength()* each node has to update the $path$ and $pathlength$ attributes. This is done in topological order, which has the effect that all predecessors of a node v have updated before v is updated. The effect for node $V7$ in the first iteration: $path_{V0}(V7) = 5 + 3 + 8 + 2 = 18$ and $path_{V0}(V7) = \{V0, V2, V3, V7\}$ which means that there is a path from $V0$ to $V7$ and the path length (sum of weights) is 18. There is another node in B with which $V7$ can form a path ($V4$). This makes $pathlength_{V4}(V7) = 4 + 2 = 6$ and $path_{V4}(V7) = \{V4, V7\}$. The updating is done for all nodes $\{V0, \ldots, V14\}$.

In *DeterminePathWithHighestCriticality()*, the possibility of a paths from nodes in B to nodes in U, is checked. If attribute $pathlenght_u(V14) \neq -1$ then it is possible to form a path $(u, V14)$. In $path$ the nodes of the largest path between u

Figure 3.4: Graph after iterations of criticality algorithm. Indicated in the node is *name/0/weight*.



(a) After 1 iteration (b) After 2 iterations (c) After 3 iterations (d) After 4 iterations

and $V14$ are given and the path length is known. Now, with the *begin* and *end* attributes a criticality number can be calculated. The highest criticality number of all possible paths, from nodes in B to nodes in U, is selected. In the first iteration, $path_{V0}(V14) = \{V0, V2, V3, V7, V12, V14\}$ and $pathlength_{V0}(V14) = 30$. Given *begin*(V0)=0 and *end*(V14)=30 this results in a criticality number of 1. The first iteration always finds a path with acriticality of 1.

In *RemovePathWithHighestCriticality()* this path $path_{V0}(V14)$ is removed from the graph. What remains of the graph G after iteration 1 is shown in Fig. 3.4(a). Also for the other iterations (except the last, which would show the empty set). Also, set B is expanded with nodes $(V5, V6, V9, V10, V13)$. Set U is expanded with $(V1, V4, V13)$. All nodes that were removed from G, were also removed from set B and set U if they were in there.

After 5 iterations, graph G is empty and the criticality number for all nodes is determined. In Fig. 3.3 these criticality nubers can be seen. This is an overview of the nodes that were removed (and in which iteration):

Iteration 1 Removed $\{V0, V2, V3, V7, V12, V14\}$, criticality = 1

Iteration 2 Removed $\{V10, V11, V13\}$, criticality = 0.9

Iteration 3 Removed $\{V5, V6, V9\}$, criticality = 0.78

Iteration 4 Removed $\{V1, V8\}$, criticality = 0.556

Iteration 5 Removed $\{V4\}$, criticality = 0.279

Criticality Function for Cyclic Kahn Process Networks

4

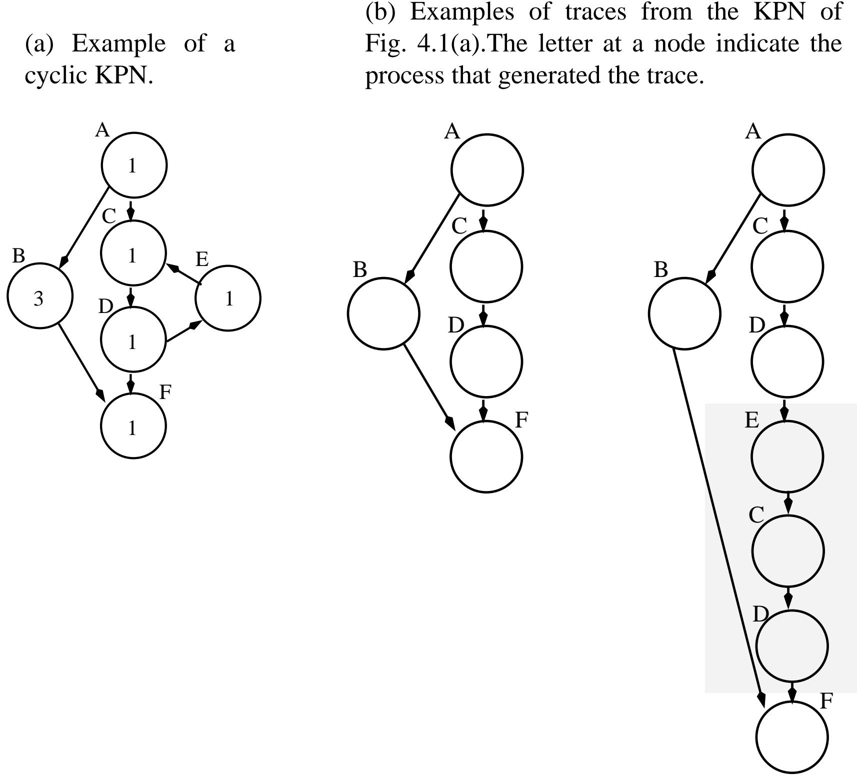
4.1 Introduction

This chapter defines the valid property for criticality functions of cyclic KPNs. It then presents an algorithms which produces a valid criticality function for these cyclic KPNs. This algorithm is called *cyclic algorithm* because it operates on a cyclic KPN. The saturation property cannot be obtained in general, which will be proven by a small example.

4.2 Definition of Valid Criticality Function of Cyclic KPNs

In Chapter 3 the validity property was described for acyclic KPNs. The property ensures that the total execution time is equal to the minimum execution time. This minimum execution time was set to the critical path length L. Cyclic KPNs do not have a fixed critical path length L, it can only be known at the time of execution. The easiest method to obtain the critical path length L of a cyclic KPN is by observing its trace. A cyclic KPN, such as Fig. 4.1(a), has an infinite set of possible traces. In Fig. 4.1(b), two possible traces are given. On the left trace, not all nodes from the cycle are executed. On the right trace, all nodes from the cycle are executed at least once. This shows that the weighted graph of a cyclic KPN does not show how many times the nodes of the cycle are executed.

Figure 4.1: A KPN graph and possible traces.



(a) Example of a cyclic KPN.

(b) Examples of traces from the KPN of Fig. 4.1(a).The letter at a node indicate the process that generated the trace.

A criticality function for cyclic weighted graph G is valid if all possible traces from G are valid. For trace G', a criticality function c' is defined. Each node from G' corresponds to a node in G and they have the same criticality number. This is also true for the weight of the node. Graph G' is a weighted directed acyclic graph with a critical path length L' and the equation from the previous chapter is reused:

Definition 3. *Let $G = (V, E, w)$ be a cyclic directed weighted directed graph. Let weighted DAG $G' = (V', E', w')$ be one of its traces. Each node in G' has a unique corresponding node in G. Both nodes have the same criticality and weight. Let function c' be the criticality function of G' which corresponds to c of G. Then the criticality function of G is valid if each corresponding criticality function c', for all possible traces G', is valid. Let trace G' have critical path length L'. Let function c' be* valid *for G' if for every path $p' = (v'_1, v'_2, \ldots, v'_n)$ in G':*

$$\sum_{v' \in p'} \frac{w'(v')}{c'(v')} \leq L'. \tag{4.1}$$

The other property of acyclic KPNs, saturation, cannot be obtained in general for cyclic KPNs. Consider Fig. 4.1(b) it can be derived that path (A, B, F) is not saturated

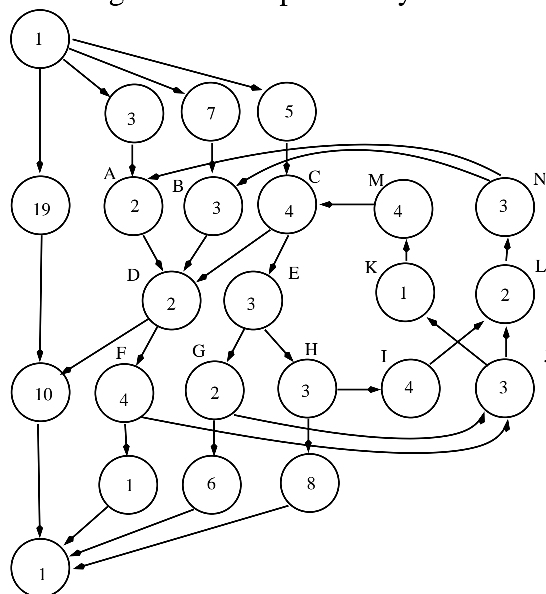
for every possible trace of Fig. 4.1(a). For the trace in Fig. 4.1(b) on the left, a criticality function for which $w(C)/c(C)+w(D)/c(D) = 3$ is true and $c(A) = c(B) = c(F) = 1$, can be considered saturated. The same criticality function for the right trace cannot be considered saturated because node B has at least 7 time units so its criticality number can be lower than 1 without an increase in total execution time. However, the first trace required $c(B) = 1$.

4.3 Extra Definitions

A classification for nodes of a cyclic KPN is presented here. A node can be part of an interdependent group of cycles, that we call *Cycle*. If it is, the node is classified as either a feedback node or a feed-forward node. This classification is used in next paragraph where the algorithm for the cyclic KPNs is discussed. Also, a few particular sorts of *Cycle*s is introduced in this paragraph.

In graphs, a cycle is defined by a route where the begin node and end node are the same (e.g. $(v_0, v_1, \ldots, v_n, v_0)$). If such a cycle is present in a KPN, the algorithm to find a criticality function from the previous chapter is no longer applicable, since it requires a directed acyclic graph.

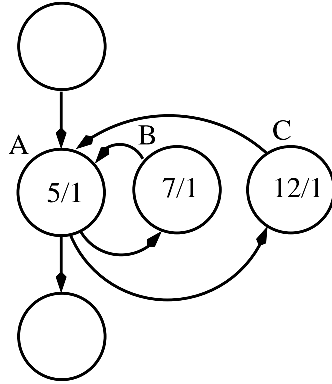
Figure 4.2: Graph with cycles.



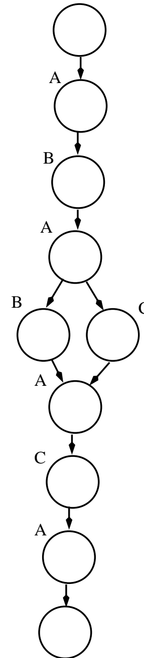
Many cycles exist in Fig. 4.2, such as (A, D, F, J, L, N, A) and (C, E, G, J, K, M, C). In our model, Fig. 4.2 could generate a trace such as Fig. 4.3. The two cycles can be identified as acyclic paths in this trace. From the trace it can also be observed that the cycles have mutual dependencies. The nodes from the

Figure 4.3: Possible trace from Fig. 4.2. The letter at a node indicate the process that generated the trace.

Figure 4.4: A cyclic graph and possible trace.



(a) Two *Cycles* with *weight/criticality* indicated in node.



(b) Possible trace from Fig. 4.4(a). The letter at a node indicate the process that generated the trace.

first cycle cannot be executed if the nodes from the other cycle are not also executed. There is a set of cycles which depend on each other. Either all nodes from the cycles in the set execute, or no complete cycle can be executed at all.

In the trace it can be observed that that set of cycles is executed twice (gray areas). After that, a subset of nodes from the set of cycles (nodes A to H) is executed. Those nodes are called the *feed-forward* nodes. The nodes which are part of the set of cycles, but which are not feed-forward nodes are called *feedback* nodes (nodes I to M). The feed-forward nodes are always executed once, the feedback nodes are only executed in conjunction with the feed-forward nodes. Feed-forward and feedback are terms which are borrowed from control engineering. We have not succeeded in obtaining an algorithm which unambiguously identifies the feedback/feed-forward property of a node.

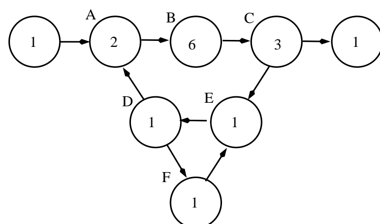
Any cycle which has its feedback nodes weakly connected to feedback nodes from another cycle is part of the cycle set. Weakly connectedness, the existence of a path between nodes if the graph would be undirected, from one feedback node to another implies that the feedback nodes depend on each other. So, feedback nodes of one cycle depends on the feedback nodes of the other one. From now on, *Cycle* (in *italic* with a capital c) refers nodes of this set of cycles which have their feedback nodes weakly

connected.

An example of two *Cycle*s is in Fig. 4.4(a). It is given that node A is a feed-forward node and nodes B and C are feedback nodes. There are two *Cycle*s, because node B and C are not weakly connected. This means there is no dependency of node B and node C to another. A valid criticality function is when all nodes have a criticality of 1. This is because only process A determines whether, and how many times, either *Cycle* ((A, B, A) or (A, C, A)) is executed. Fig. 4.4(b) is a possible trace from Fig. 4.4(a) which shows this.

There exist more exotic forms of graphs. For example, a *Cycle* can be nested, see Fig. 4.5. For *Cycle* $(A, \ldots, F)$, nodes (A, B, C) are feed-forward and nodes (D, E, F) are feedback. For *Cycle* (D, E, F), nodes (D, E) are feed-forward and node F is feedback.

Figure 4.5: A *Cycle* in a *Cycle*.

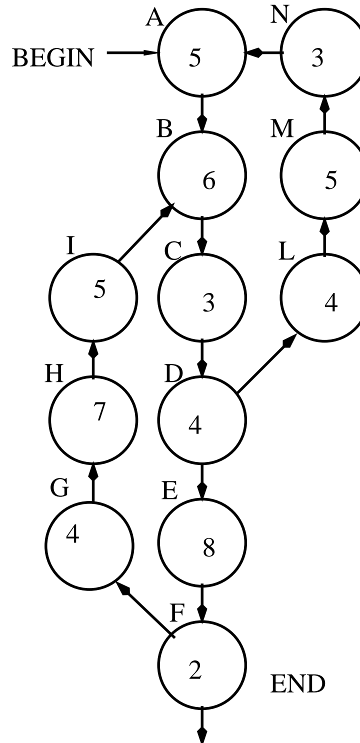


It is not necessary that these *Cycle*s execute the same number of times. Basically, if *Cycle* (D, E, F) can be replaced by a virtual DAG and the removed *Cycle* consists of either feedback or feed-forward nodes from the other *Cycle*, they have to be seen as separate *Cycle*s.

A *Cycle* in a *Cycle* can also be solved by the acyclic algorithm. The order of processing of the example in Fig. 4.5 would be to first look at $(A, \ldots, F)$ and while looking at that *Cycle*, recursively determine the criticality function of *Cycle* (D, E, F). In Section 4.4.4, an example of this is shown.

Another form of graph is seen in Fig. 4.6. Here, two *Cycle*s mix feed-forward nodes and non-*Cycle* nodes. Node A is a feed-forward node for one *Cycle*, but not in the second *Cycle*. For node E and F the reverse is true. In contrast to Fig. 4.4(a) these *Cycle*s share a common data path. Secondly, the result of one of the *Cycle*s is needed in the next iteration of that *Cycle* and vice versa. An extra condition can therefore be set: both *Cycle*s execute equally often. The result is that those *Cycle*s can be seen as a single *Cycle*. It is as if their feedback nodes have a dependency.

Figure 4.6: KPN with two *Cycle*s that share some feed-forward nodes.



4.4 Algorithm for Cyclic KPNs

4.4.1 Introduction

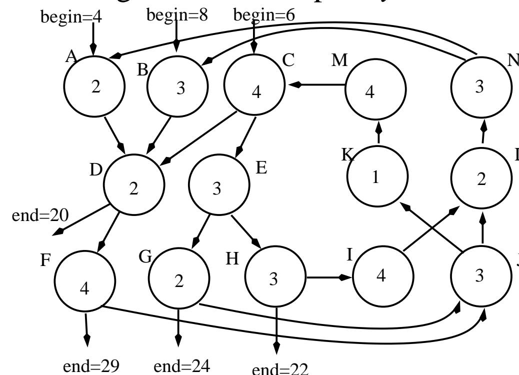
An algorithm for a valid criticality function for cyclic Kahn process networks is presented here. It is called cyclic algorithm because it operates on cyclic KPNs. In this cyclic algorithm, the acyclic algorithm is reused. First a motivation for part of the algorithm is given and then the algorithm itself. In the last paragraph a second example is given.

The cyclic algorithm operates as follows: it determines a criticality function for nodes of a *Cycle*. The nodes of the *Cycle* are removed and *begin* and *end* attributes are assigned where edges where removed. This is iterated until all *Cycle*s from the graph are removed. For the remaining acyclic KPN, the acyclic algorithm is used. The example *Cycle* that is used to explain the algorithm (Fig. 4.7) is part of a larger KPN (Fig. 4.2).

4.4.2 Motivation for Cyclic Algorithm

First, the criticality function for separate *Cycle*s of the cyclic KPN is determined. The *Cycle* is separated from G and it is converted to an acyclic graph by removing edges

Figure 4.7: Example *Cycle*.



which break all cycles. Then the acyclic algorithm can be used to obtain the criticality function of nodes from the *Cycle*.

When executing a single *Cycle*, i.e. nodes from a *Cycle*, two things can happen. First, only the feed-forward nodes are executed (nodes A to H of Fig. 4.7). Second, (all feedback and feed-forward nodes from) the *Cycle* executes. If the feed-forward part of the graph is represented by the letters *ff* and the feedback part is represented by the letters *fb* then the trace of a *Cycle* shows a pattern. If the *Cycle* does not execute completely, the trace is represented by *ff*. If the *Cycle* is executed once, it is can be represented by *ff-fb-ff*. If the *Cycle* is executed twice: *ff-fb-ff-fb-ff* (this exact pattern can be seen in Fig. 4.3), etc.

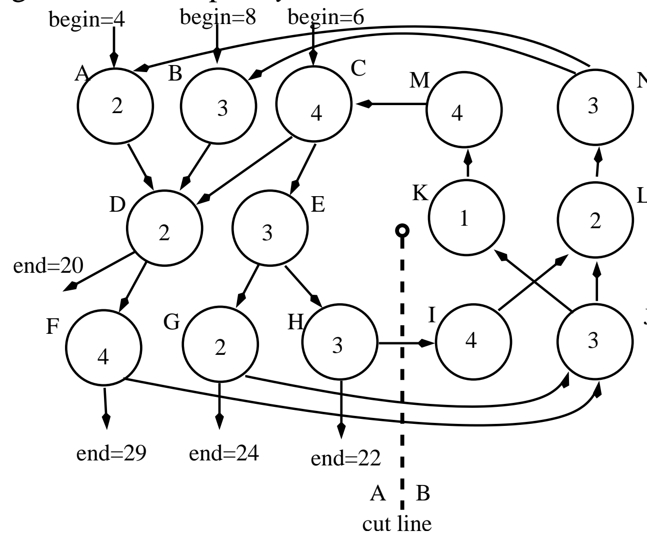
The complexity of finding the valid criticality function is reduced by reducing this pattern. Finding the criticality function for the *ff-fb-ff-... -fb-ff* trace is reduced to finding the criticality function for the *fb-ff* part. The cut, the removal of edges which break all cycles, is therefore made at all edges from feed-forward to feedback nodes (see Fig. 4.8).

Before the *Cycle* itself is analyzed, two concepts need to be explained. First the concept of periodicity. Second, the effect of a *Cycle* on execution time and how this time can be emulated by a fixed delay.

If a *Cycle* is executed a number of times, a node is executed periodically. This period is called j. For example, each time node I is executed, it is j time units later. Similarly, each time that node H is done, it is j time units later from the moment that node I could start the last time.

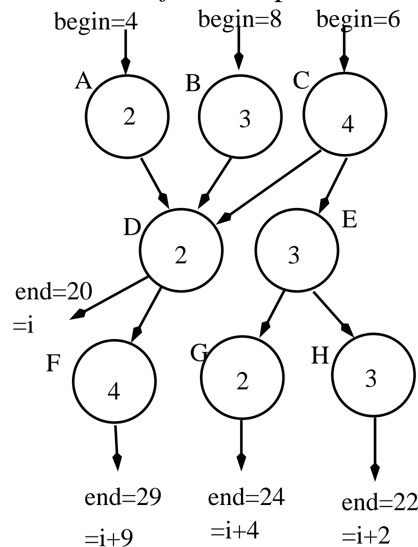
If a *Cycle* is executed a number of times, then it is possible that most of the execution time of the KPN is spent executing nodes of this *Cycle*. This means that nodes of this *Cycle* must start executing as soon as possible. Also, nodes which receive data from (nodes of) the *Cycle* must start executing as soon as possible. Virtually, the *Cycle* has the effect of a variable delay. This delay is between edges of feed-forward nodes and non-*Cycle* nodes which depend on (data through) that edge. This delay can be either be 0, j, $2j$, etc. A fixed delay is introduced for all these edges. Suppose that all feedback nodes are removed from graph G. This graph will be acyclic. The fixed delay should

Figure 4.8: Example *Cycle* with indication of cut line.



have two effects. The first effect is that there should be a path with a criticality of 1 through the feed-forward nodes of the *Cycle*. This guarantees that data is available for feed-forward nodes as soon as possible. Second, non-*Cycle* nodes which receive data should process this without delay. The net effect is that a second critical path is created which is equally long as the other one.

Figure 4.9: *Feed-forward* part of the cycle.

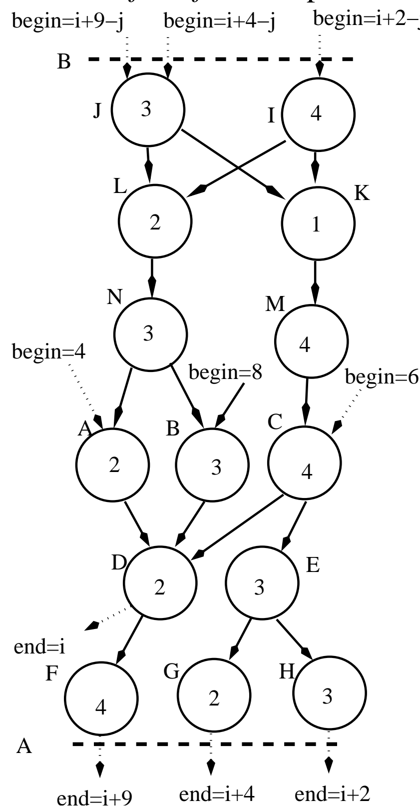


First, this fixed delay will be calculated. To do this, the feedback nodes are temporarily removed. The feed-forward nodes of the *Cycle* are shown in Fig. 4.9. From the

whole KPN (see Fig. 4.2) it can be seen that node A cannot start earlier than t=4. For node B it is t=8 and for node C it is t=6. The critical path length is 31. For validity, node D must end execution before t=20. For node F, G and H these values are also in Fig. 4.9. If these values are used as *end* attributes then all (non-*Cycle*) successors of F, G and H will have a criticality number of 1. A variable i is now introduced. This i is set to the smallest latest finish time. This is 20, for node D. From other *end* attributes, 20 is subtracted and variable i is added. So *end*(F)=29 becomes *end*(F)=i+9. If i is 20, the delay is included. Now it must be determined what the minimal i value is. The difference is the fixed delay.

To determine the minimum i, the earliest time that nodes D, F, G and H can be finished must be determined. Using the *begin* attributes, for node D it is 13 (*begin*(B)+w(B)+w(D)), for node F it is 17, for node G it is 15 and for node H it is 16. If $i = 14$, all nodes have an *end* attribute that is larger or equal to their earliest moment that they can finish execution. The fixed delay is 20-14=6 time units. This means that for nodes of the *Cycle*, i equals 14. For other nodes, i is 20.

Figure 4.10: *Feedback and feed-forward* part of the graph reworked.

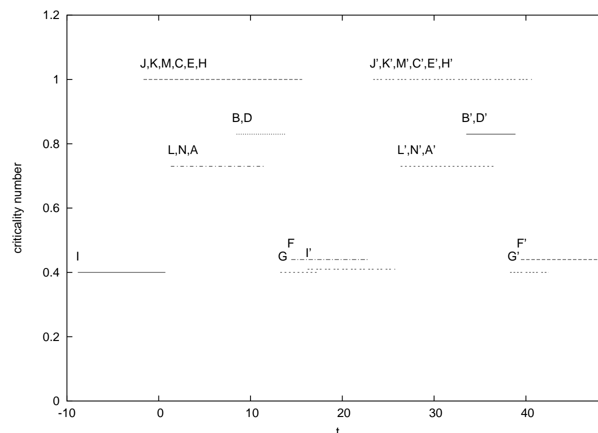


Now Fig. 4.8 can be converted into Fig. 4.10. Because of period j, *begin*(I) equals *end*(H)-j. This expresses that each time node H is finished, it is j time units later than

the last time. Variable i is already known but j still needs to be determined. For now we allow multiple *begin* attributes. Later, for each node the largest *begin* attribute and the smallest *end* attribute will be selected. To determine j, two steps are needed. First a preliminary j is determined. Second, the *begin* and *end* attributes are changed a little which effectively reduces j.

In the first (intermediate) step, variable i is eliminated and j is determined in the same way i was. Node D can be done at 37-j, F at 41-j, G at 40-j and H at 41-j. Variable $j = 25$ and it creates a critical path from node J to H. Just to see what happens, j is filled in and the acyclic algorithm is used to find a criticality function. The criticality-time diagram is given in Fig. 4.11. In this diagram the line with label I (left bottom) indicates that process I has a criticality number of 0.4 and *begin*(I)=-9 and *end*(I)=1. The plot is generated twice with a shift of j. What should be noticed is the distinct lack of a process with a criticality number of 1 between t=16 and t=23. This criticality function would slow down all processes for 7 time units. As if there is no critical path. The value of j is not yet correct.

Figure 4.11: **WRONG** j. Double time criticality diagram, j=25.

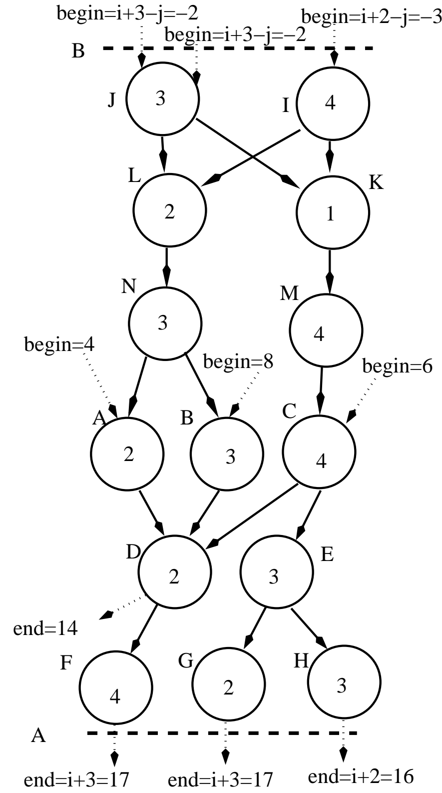


The second step will give the right value for j. It can be seen that for each edge $e(u, v)$ that was cut ((F, B), (G, B) and (H, I)) slack time is present. This slack time consists of three components. First, there is the amount of time that can be saved by setting *end*(u) earlier. Second, if data is available from u at *end*(u) and it is not used because node v has to wait for other data, it can be waste of time. The expression

$$\textit{begin}(v) + j - \textit{end}(v) \tag{4.2}$$

gives that amount of slack. Third, if *begin*(v) can be set to a larger value, this can save time. The sum of these time slacks is different per edge. The minimum value of this slack is the amount of time that can be saved per execution of a *Cycle*. E.g. edge (F, B) has time slack. Attribute *end*(F) can be lowered with 6 to 17, $\textit{begin}(J) + j - \textit{end}(F) = 0$

Figure 4.12: Final DAG which the acyclic algorithm can analyze.



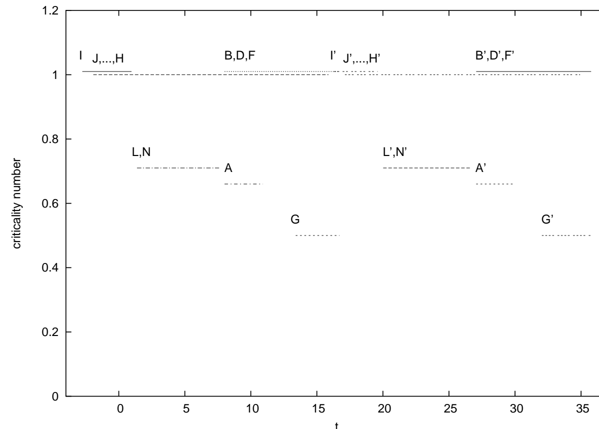
and *begin*(J) cannot be set earlier, because node J has a criticality of 1. This can be summed up by 3-tuple (6,0,0). Time slack of (G, B) can be summed up by (3,5,0) and time slack of (H, I) can be summed up by (0,0,6). The minimum sum of those numbers is 6. Therefore, j is effectively reduced to 19 and this is done by changing *begin* and *end* attributes as described in Alg. 9 in the next paragraph. Here, for each edge $e(u, v)$, first the slack as given in Equation 4.2 is subtracted from the amount of slack to removed. Second, slack in *begin*(v) is removed if necessary. If this is not enough, the same is done for *end*(u).

The resulting DAG is shown in Fig. 4.12. The *begin*(I) is reduced with 6. The *end*(F)=23-6 and *end*(G)=18-1. Now, the acyclic algorithm can be applied again and a time-criticality diagram can be made again (see Fig. 4.13). This plot shows that there is always a process with a criticality number of 1, which was the objective.

The criticality function for the *Cycle* has been determined. However, the *Cycle* is part of a KPN. The plan is to remove the *Cycle* and add *begin* and *end* attributes where edges were present. Any edge to a feed-forward node will get an *end* attribute, an edge from a feed-forward node to a non-*Cycle* node gets a *begin* attributes.

The nodes with edges to the feed-forward nodes have their *end* attribute set to this *begin* attribute. Similarly, the nodes with edges from feed-forward nodes have their

Figure 4.13: Double time criticality diagram, j=19.



begin attribute set to the *end* attribute of those nodes as used in Fig. 4.12 plus the fixed delay of 6 time units. E.g. node v with w(v)=8 has *begin*(v) set to 16+6.

Now that the nodes of the *Cycle* are removed from G, the remainder is acyclic. The acyclic algorithm can find a criticality function for those nodes.

4.4.3 Algorithm for Acyclic KPNs

In algorithm Alg. 7, the algorithm is broken up into 2 pieces. First the nodes of the *Cycle*s have their criticality function determined. Then the remaining graph is acyclic and it can be analyzed by the acyclic algorithm of Chapter 3.

Algorithm 7 Pseudo algorithm for finding criticality function for cyclic KPNs.

Let G' be graph G without feedback nodes and connecting edges
Set the *begin* attribute, of nodes with in-degree 0 in G', to 0
Set the *end* attribute, of nodes with out-degree 0 in G', to critical path length $L(G')$
DetermineCriticalityFunctionOfCycles(G)
Determine criticality function of remaining acyclic graph G

The part of the algorithm which determines the criticality function for the nodes of the *Cycle*s follows in Alg. 8. First a *Cycle* is selected. All ingoing and outgoing edges to/from the *Cycle* are put in set E_{in} and E_{out}. The nodes with ingoing edges to the *Cycle* (often) start processing at a different time. The same is true for nodes with outgoing edges from the *Cycle* which finish at a different time. The difference in this timing can be seen as a phase difference. Alg. 8 determines what the phase of the ingoing (parallel) data (streams) can be and what outgoing phase can be used by the remainder of G.

After Alg. 8 is used, the resulting criticality function ensures that data that enters the *Cycle* with a certain phase can be used directly in *Cycle* C. Second, when the nodes

of C are done executing, the data is delivered to the remainder of G with a phase that is effective without being too restrictive for the criticality function that has to be found. Last, the resulting criticality function ensures a delay that is minimal, which is equal to saying it is a valid criticality function.

Algorithm 8 *DetermineCriticalityFunctionOfCycles*, removes *Cycle*s from G and assigns criticality numbers to those nodes. Also assigns extra *begin*/*end* attributes to nodes which have edges to/from the *Cycle*s.

while (**exists** *Cycle*s **in** G) **do**
  select *Cycle* C, a path must exist from node with in-degree 0 to a feed-forward node of C,
    path does not contain nodes from other *Cycle*s
  let set E_{in} be defined by nodes with ingoing edges to C using G
  for $e(u, v)$ in E_{in}, **set** *begin*(v) to earliest start time of v and remove $e(u, v)$
  let set E_{out} be defined by nodes with outgoing edges from C using G'
  for $e(u, v)$ **in** E_{out}, **set** *end*(v) to latest finish time of v and remove $e(u, v)$
  let C' be the subset of feed-forward nodes of C
  for v **in** C', earliest finish time $(v) = max_{u \in pred(v)}(begin(u) + w(u) + w(v))$
  for $e(u, v)$ **in** E_{out}, *end*(u) $\leftarrow$ *end*(u) $- (min_{e(u,v) \in E_{out}}$*end*$(u) - eft(u))$
  let E_{fb} be the set of edges from feed-forward to feedback nodes, remove these edges from C
  for $e(u, v)$ **in** E_{fb}, **if** (*end*(u) $- j >$ *begin*(v)) **then**
    begin(v) $\leftarrow$ *end*(u) $- j$, j remains variable
  temporarily remove feedback nodes and edges to/from other *Cycle*s in C
  determine j by calculating critical path length $L(C)$
  RemoveSlack (by adjusting *begin* and *end* attributes of C)
  if (other *Cycle*s exist in C) **then**
    unremove feedback nodes from earlier
    Determine Criticality Function of Cycles, **do** recursive operation.
  calculate criticality function of C
  for $e(u, v)$ **in** E_{in}, *end*(u) $\leftarrow$ *begin*(v)
  for $e(u, v)$ **in** E_{out}, *begin*(v) $\leftarrow$ *end*(u) $+ (min_{e(u,v) \in E_{out}}$*end*$(u) - eft(u))$

The Alg. 9 changes the phase of the data from the edges which connect the feed-forward to the feedback nodes. It does it in such a way that the period of the execution of the nodes of a *Cycle* becomes minimal. The term *slack* indicates what amount of time is not used by: a) the *end* attribute of feed-forward nodes being set too late b) difference of the time when data becomes available to when it is used c) the *begin* attribute of feedback nodes being set too early. For each edge, this is summed up and the minimum is taken. This is the amount of time that can be spared each period, by changing the *end* and *begin* attributes. This effectively reduces j.

Algorithm 9 *RemoveSlack*, reduces period j in effect.

for $e(u, v)$ in E_{fb} do
  $slack_{end}$(u) = *end*(u) - earliest fi nish time(u)
  $slack_{begin}(v)$ = *begin*(v) - latest start time(v)
  $slack(e(u, v)) = (\textit{begin}(v) - \textit{end}(u) + j) + slack_{end}(u) + slack_{begin}(v))$
for $e(u, v)$ **in** E_{fb} **do**
  $toremove = \min slack$
  $toremove \leftarrow toremove - (\textit{begin}(v) - \textit{end}(u) + j)$
  if $(toremove \geq 0)$ **then**
    $toremove \leftarrow toremove - slack_{begin}(v)$
    if $toremove \geq 0$ **then**
      $\textit{begin}(v) \leftarrow \textit{begin}(v) + slack_{begin}(v)$
      $slack_{begin}(v) \leftarrow 0$
    else
      $\textit{begin}(v) \leftarrow \textit{begin}(v) + slack_{begin}(v) + toremove$
      $slack_{begin}(v) \leftarrow -toremove$
  if $(toremove \geq 0)$ **then**
    $toremove \leftarrow toremove - slack_{end}(u)$
    $\textit{end}(u) \leftarrow \textit{end}(u) - slack_{end}(u) - toremove$
    $slack_{end}(u) \leftarrow -toremove$

4.4.4 Example

In this example, the weighted cyclic graph G (see Fig. 4.14) will have its criticality function calculated. Graph G has two *Cycle*s (one in the other). Node E, I and M are feed-forward nodes and nodes G, H, J, K and O are feedback nodes for one *Cycle*. Node G, H and K form another *Cycle* in the feedback part of the first. Node G and K are the feed-forward nodes and H is the feedback node of this second *Cycle*.

To find the critical path length, first the feedback nodes from all *Cycle*s (i.e. G, H, J, K and O) are temporarily removed. Then the remaining DAG has a critical path length L of 24. i is determined by calculating the latest finish time and the earliest finish time of the feed-forward nodes. The latest finish time for node I is $t = L - w(R) - w(L) = 13$ and this is also the earliest time node I can finish execution.

Then the original graph G has its first *Cycle* isolated. The edges $e(I, O)$ and $e(M, O)$ are removed and feedback node H is temporarily not considered. The *begin*/*end* attributes are given in Fig. 4.15(a). The period j is 27 which determined by the path (O, K, G, E, I) with *begin*(O)= $i + 9 - j$ and *end*(I)=i and the sum of the weight of these nodes is 18. However, j can be lowered to 22 by setting *end*(M) to $i + 4$ (see Fig. 4.15(b). Now all *begin* and *end* attributes are known.

Because there is a second *Cycle* in the first *Cycle*, a recursive operation takes place. As if Fig. 4.15(b) would be the only existing KPN for *Cycle* (G, H, K), the criticality

Figure 4.14: Example of cyclic graph.

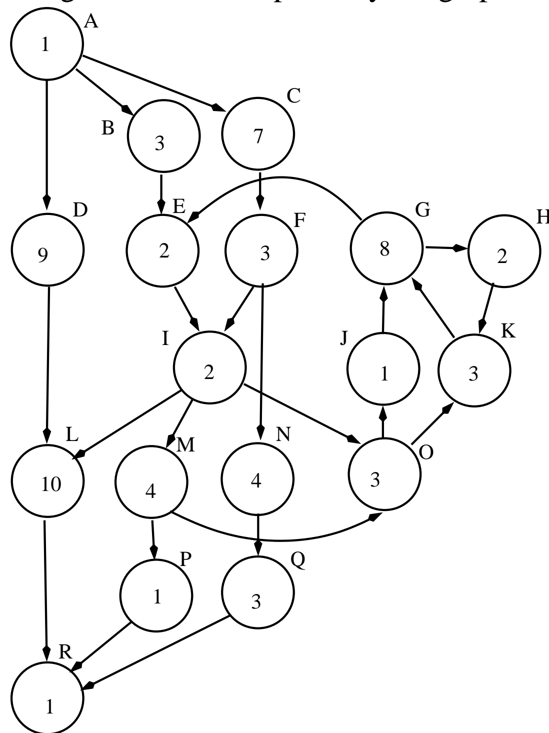
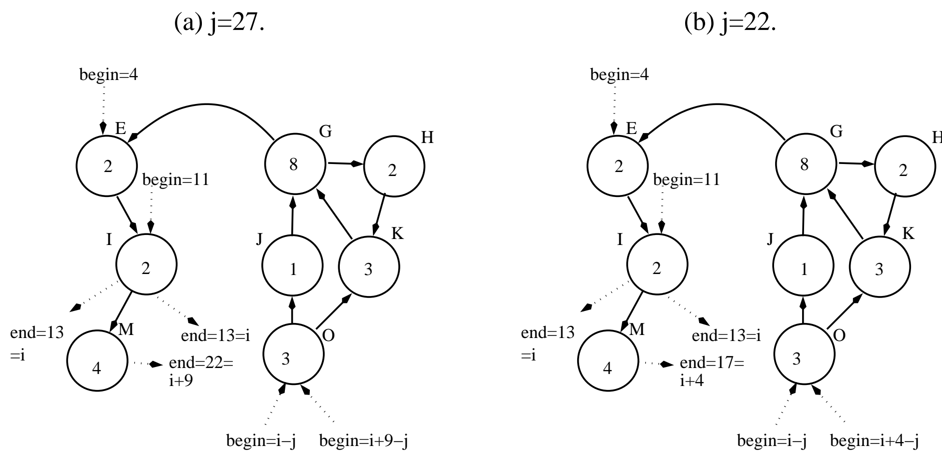


Figure 4.15: First *Cycle* with attributes.

(a) j=27. (b) j=22.



function for those nodes are calculated. Because the *begin* and *end* attributes are known, the critical path length L' is not needed. In Fig. 4.16 the second *Cycle* is given with the removed edge $e(G, H)$ and the *begin* and *end* attributes. The i' is 9 and j' is 13.

Since there is only one removed edge, j' cannot be lowered. The criticality numbers for nodes H, K and G are, given the *begin* and *end* attributes, all 1. Node G, H and K are removed. The *begin*(E) is set to 9, *end*(I)=1 and *end*(O)=-2. In Fig. 4.17 this result is given.

Figure 4.16: Second *Cycle* with attributes.

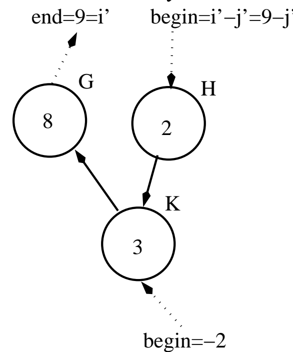
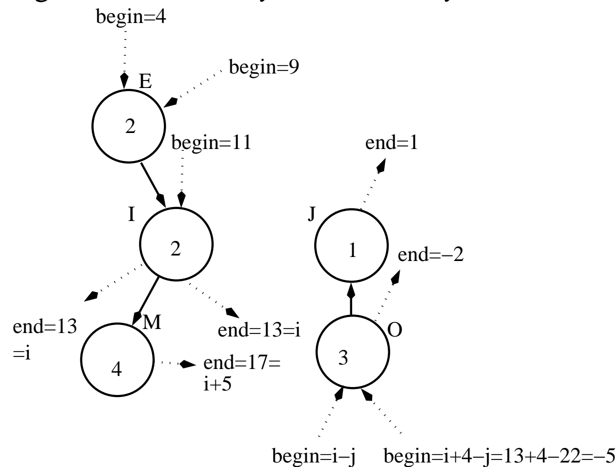


Figure 4.17: First *Cycle*, second *Cycle* removed.



In Fig. 4.17, sometimes two *begin* or *end* attributes are given. Later one would set the *begin* attribute to the largest value and the *end* attribute to the smallest value. Then some more criticality numbers can be calculated: c(E)=c(I)=c(M)=c(O)=1 and c(j)=1/3. Nodes E, G, H, I, J, K, M and O (which form the first *Cycle*) are removed from graph G. If edge (u, v) connected non-*Cycle* node u to feed-forward node v, *end*(u) is set to *begin*(v) if no smaller value was set earlier. The opposite is done for feed-forward to non-*Cycle* edges.

With the nodes of the first *Cycle* removed and the corresponding attributes set, Fig. 4.18 is the remaining acyclic KPN. The acyclic algorithm generates the fol-

lowing criticality numbers: c(A)=c(C)=c(F)=c(L)=c(L)=1, c(B)=0.375, c(D)=0.75, c(N)=c(Q)=0.583 and c(P)=0.167. The result, the complete graph G with *weight/criticality number* indicated inside the node is given in Fig. 4.19.

Figure 4.18: Graph G with the first *Cycle* removed.

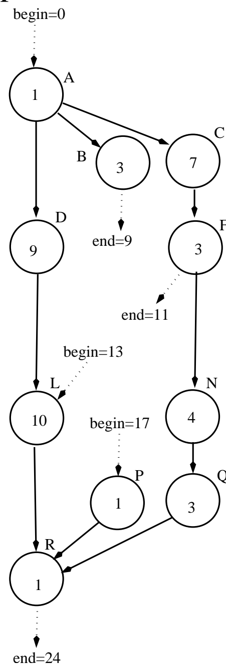
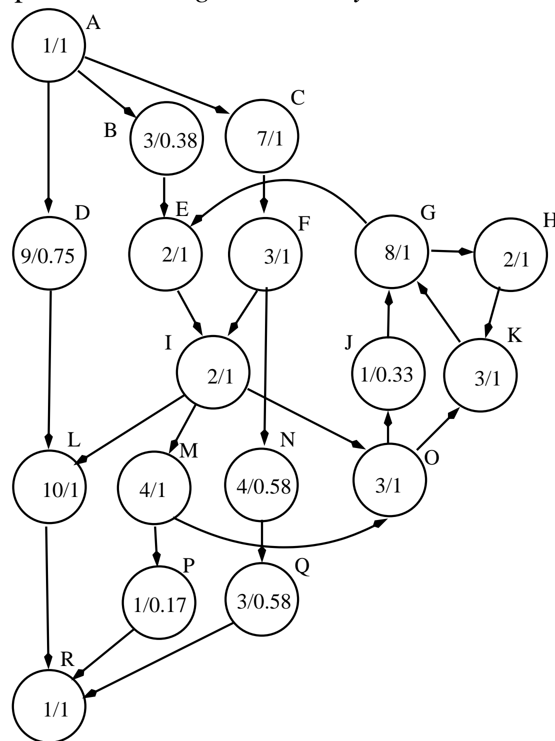


Figure 4.19: Graph G with *weight/criticality number* indicated inside node.



Applications of Criticality Numbers

5

5.1 Introduction

In the introduction of this thesis, it was mentioned that this research was started to help the designer answer certain questions about the *Application Modeling*, *Architecture Modeling*, *Mapping* and *Performance Analysis*. For each of these questions, an example is given of the way in which a criticality function can aid this process.

5.2 Visualization of Criticality Numbers

5.2.1 Introduction

A criticality function was introduced to show the performance of an application by determining the allowed slowdown of each process. However, a graph with for example 500 vertexes, each showing a number, is difficult to interpret. There are two other ways to visualize a graph with a criticality function. These are the time/criticality diagram and the *accumulated criticality function*.

5.2.2 Time Criticality Diagram

The time/criticality diagram plots a line for every process v. It starts at *begin*(v) and it ends at *end*(v). The y-axis gives c(v). A time/criticality diagram gives an overview of

the processes with the minimum amount of execution speed they require.

Figure 5.1: Time/criticality diagram of the example in Fig. 3.3.

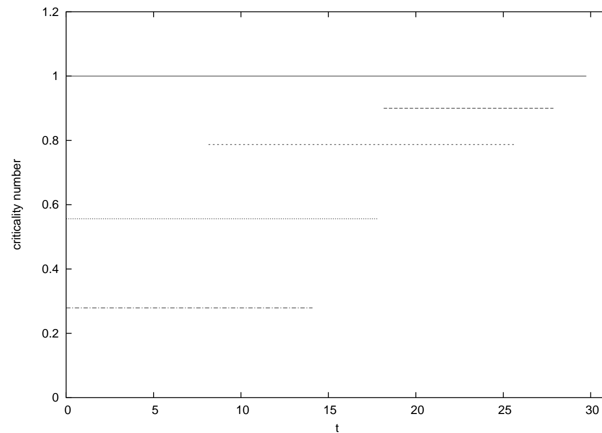
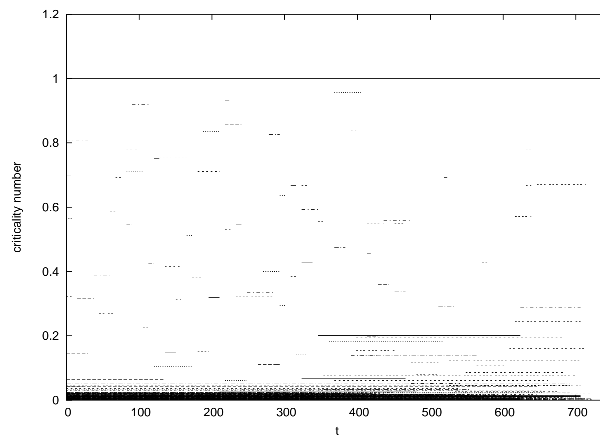


Figure 5.2: Example of more complex time/criticality diagram of a random graph with 500 nodes.



It can help an application modeler to identify where the processes are which processes to optimize. A process with a low criticality, does not need optimization. On the other hand, a single process which has a criticality of 1 while other processes have a much lower criticality is clearly a bottleneck. To conclude, a time/criticality diagram is a way to use performance analysis in system design (as described by YAPI[2]).

5.2.3 Accumulated Criticality Function

An accumulated criticality function cACC is defined as:

$$\text{cACC}(t) = \sum_{v \in G} c(v, t) \tag{5.1}$$

Function $c(v, t)$ is defined as:

$$c(v,t) = \begin{cases} c(v) & \text{if} \quad \textit{begin}(v) < t \leq \textit{end}(v); \\ 0 & \text{if} \quad \text{otherwise.} \end{cases} \tag{5.2}$$

This function sums all criticality numbers as indicated in a time/criticality diagram. It can be interpreted as a parallelism function. Let the average amount of parallelism of G be defined as the amount of processing divided by needed time. This would be

$$\text{average parallelism} = \sum_{v \in G} w(v)/L. \tag{5.3}$$

Then, this system has an average amount of parallelism equal to

$$\text{average parallelism} = \int_0^L \text{cACC}(t)\, dt/L \tag{5.4}$$

which is equal to the first summation because:

$$\int_{\textit{begin}(v)}^{\textit{end}(v)} c(v, t)\, dt = w(v). \tag{5.5}$$

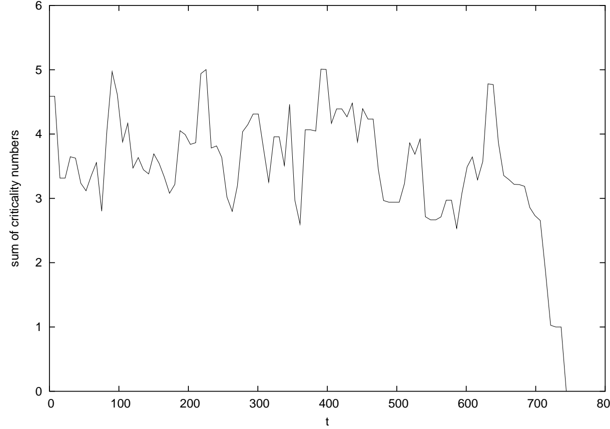
Because Eq. 5.4 is the standard method to average something and the units ("weight/time") are correct, cACC can be seen as a parallelism function. A limitation is, that it is not unique because the criticality function is not unique.

Fig. 5.3 shows that three full speed processors are insufficient to execute the application without decrease of performance. The amount to process (sum of weights) is too much. One could expect that the maxima (about 5) number of processors to be enough to execute the application. However, there are simple examples which contradict this. For example, a KPN has four processes with the same *begin* and *end* attributes. The criticality function is given as $c(v_1) = 1, c(v_2) = c(v_3) = 0.9$ and $c(v_4) = 0.2$. The function cACC=1+0.9+0.9+0.2=3. However, there is no schedule for these four processes with only 3 processors. Function caCC does not, per definition, indicate the actual number of processors needed to schedule an application on a platform.

5.3 Adaptation of DAG Scheduling for Cyclic Weighted Task Graphs

Contrary to scheduling a weighted DAG, scheduling a weighted cyclic graph has fewer solutions. This section provides an adaptation of a Highest Level First (HLF) list scheduling to schedule a cyclic weighted graph.

Figure 5.3: Function cACC of the criticality function also depicted by the time/criticality diagram in Fig. 5.2.



An HLF priority is determined by the length of the longest path in a DAG to an exit node. In the HLF, the path length indicates the minimum time needed to process the nodes on this path. So in the HLF, where path p is from v to an exit node, the priority is defined as:

$$\text{priority}_{HLF}(v) = \max \sum_{u \in p} w(u). \tag{5.6}$$

One consequence is that all exit nodes with the same weight have the same priority. For the criticality function algorithms, it is the goal to finish execution of exit nodes at a different time, as indicated by the *end* attribute. A modified priority is defined as:

$$\text{priority}_{HLF'}(v) = \max \sum_{u \in p} w(u) + L - \textit{end}(u_n). \tag{5.7}$$

Because path $p = (u_1, \ldots, u_n)$ and u_n is an exit node of the DAG, L-*end*(u_n) = 0 for DAGs. This priority can also be defined as:

$$\text{priority}'_{HLF'}(v) = \max \sum_{u \in p} w(u) - \textit{end}(u_n), \tag{5.8}$$

which does not change the ordering of the priorities, only the absolute value. The equation without L is used for cyclic KPNs so that the priority can be defined as:

$$\text{priority}_{KPN}(v) = \max \sum_{u \in p} w(u) - \textit{end}(u_n). \tag{5.9}$$

Path $p = (u_1, \ldots, u_n)$. Critical path length L does not need to be determined. Also, u_n can be set to an exit node of for example Fig. 4.12, which was a *Cycle*.

One extra bit of bookkeeping must be performed. The nodes of a *Cycle* would now get a fixed priority. This is not good because the priority should decrease for every following process that is executed. This means that if priority P has an ordering of :

$$P(v_1) > P(v_2) > \cdots > P(v_n) \tag{5.10}$$

then the process must be executed in the order $(v_1, v_2, \ldots, v_n)$. This would rule out multiple times execution of v_i. To correct this, every time a new *Cycle* is started, j must be added/substracted to correct this.

5.4 Executing a KPN on CAKE: Scheduling vs. Design Space Exploration

This section will describe the merging of scheduling and design space exploration on multiprocessors systems. Normally, scheduling is the mapping of a number of processes onto (a number of) processor(s). Generally there are too few processors or about enough to execute an application without a decrease in performance. For these problems, excellent scheduling mechanisms are available (especially for DAGs).

However, the CAKE platform is designed to be extended to have many processors. The design philosophy of CAKE is that an abundance of processors is available because relatively, memory will use most silicon. When an application must be scheduled on a platform with an abundance of processors (with various execution speeds), other choices must be made. Not *if* the application will execute without a decrease in performance, but *at what costs*. For example power consumption, bus bandwidth usage, memory usage, silicon area usage and core design/licensing costs can all be expressed as a function of execution speed. In other words, a process that needs to be executed faster will use more power, bus bandwidth, memory, use faster cores which are bigger and more expensive to design/license. Then, to schedule an application will then be the minimization of costs of using an architecture. It should identify which processor is needed. That can be seen as a design space exploration. Seen in this way, a criticality function provides a design space exploration.

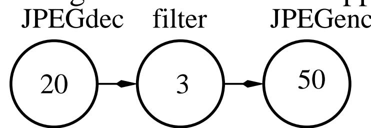
5.5 Reducing Memory Needs

Up to this point, the criticality algorithm was given to characterize the application. However, many criticality functions exist which influence the execution behavior. Some may not minimize execution speed, but save buffer memory for instance. It is assumed that process v with a criticality of $c(v)$ is executed $c(v)^{-1}$ times as fast as a process with a criticality of 1.

The next example (see Fig. 5.4) is about a (ficticious) KPN application. The graph consists of three nodes. The first converts a JPEG picture, the second filters the resulting

bitmap and the third converts the bitmap into a JPEG again. This application behaves very nice. The acyclic algorithm would assign a criticality number of 1 to each node.

Figure 5.4: Weighted DAG of KPN application



If this KPN processes more than one picture then:

- The first picture will be **decoded** in 20 time units, the second will be decoded in 40 time units, etc etc.
- The first picture will be **filtered** in 23 time units, the second will be filtered in 43 time units, etc etc.
- The first picture will be **encoded** in 73 time units, the second will be encoded in 123 time units, etc etc.

After a while, many images will be decoded and filtered, but not encoded again. The acyclic algorithm gives the fastest way to compute the images, especially for the first image. The memory use of this criticality function is not always desired.

By setting the criticality number of the decoder to 0.4, the filters' to 1 and the encoders' to 1, a pile-up of data in the FIFO from filter to encoder is avoided. The downside is that the first image is decoded at 103 time units, instead of 73 time units.

6 Implemented Parts

6.1 Introduction

Two things are implemented. First, it was shown that a trace of a KPN application can be made by changing the thread library of the CAKE system. The trace data can then be converted into a weighted graph. Second, the acyclic algorithm was implemented in C++ using the Boost Graph Library[7](freely available). This not only generates the criticality function, but also data to visualize this with the Dot and Gnuplot software (also freely available).

6.2 Tracer for Kahn Process Networks on Philips' CAKE Platform

The CAKE platform executes a Kahn process network with the aid of a thread library (which can be seen as a lightweight operating system). A process is scheduled on a processor. When a process wants to send or receive data it interrupts and the thread library will process that request. If a process blocks, e.g. because a FIFO is empty, another process from the queue is scheduled.

The thread library was altered to trace to generate the data for the weighted graphs. The context switch and the send/receive interrupts are timestamped. This is put in a memory structure. At the moment, only the context switch data is used. When a process

is switched, that data is sent to a file. This is an example of a small part of such data:

```
MPR in:Fc(s=4)   out:Fa(s=4)  tokencount = 4
15 0 138528 0 0
11 0 139259 0 1
9 0 139751 0 1
```

The first line shows that process MPR receives data from ingoing FIFO Fc and sends data to outgoing FIFO Fa. After the first line, the first column indicates the action. The number 15 stands for the schedule time, 11 stands for data sent to a FIFO, 10 stands for data received from a FIFO and 9 stands for a context switch to another process. The second and third column indicate time with the high_clock_counter and low_clock_counter. The fourth and fifth column indicate the number of tokens received and sent to FIFO FCC and FIFO Fa.

This trace shows that process MPR starts to execute at t=0,138528 and it stops executing at t=0,139751. At t=0,139259 it sends a token on an outgoing FIFO, called Fa. It can be seen that the fifth column, which corresponds to the outgoing FIFO to Fa, turns to 1.

This tracing is on top of the processor simulator. This means that if a processor system is available which implements all thread library calls, the tracing can take place in real time. Unfortunately, the thread library for Unix does not implement high_clock_counter and low_clock_counter at the moment. No further implementation details can be given here because the thread library is proprietary and so are the amendments.

6.3 Implementation of Criticality Function in C++

In Appendix A, the C++ source code of an implementation of the acyclic algorithm can be found. It was implemented with the Boost Graph Library (BGL), which can be found at `www.boost.org`[7]. The BGL is implemented similarly to the Standard Template Library (STL, which uses iterators, descriptors, etc).

The file *ascf.cpp* contains *main()*. There, a random directed graph is generated, random weights are assigned and the criticality function algorithm is applied. After that, a graph is generated in the dot graph format. Dot is part of the AT&T Graphviz graph drawing software[5].

The algorithms and functions are in *Critlib.hpp*. Not all functionality is demonstrated in the sample program *ascf.cpp*. Some of these functions are implemented in templates. A summation:

CritGraph The definition of graph with attributes as *begin*, *end*, criticality, etc.

RandomDigraph Takes a graph (without edges) and generates a (weakly connected) random directed graph. One thing that is not random about it, is that every vertex has at least one outgoing edge (except for one of course).

RandomLocalDigraph Is the same as RandomDigraph. This version has less edges from/to nodes which have a high difference in index number. This can be adjusted by the parameters *chance* and *div*. How this effects the graph can be observed from Fig. 6.1.

DotPrint2 Generates a graph in dot format on stdout. The name and two other properties are shown in the node.

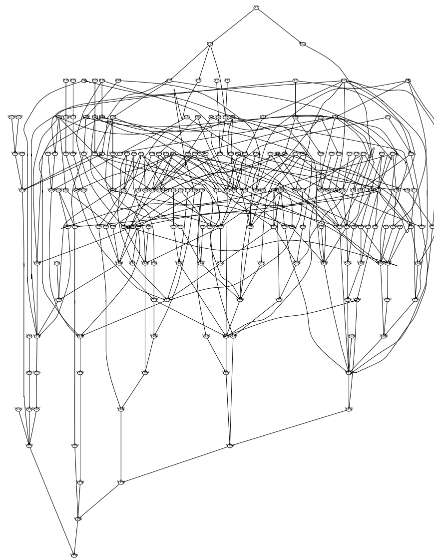
DotPrintAll Generates a graph in dot format. This exact format can be parsed by the DotReadAll function.

DotReadAll Can read dot formatted data, as done by DotPrintAll, which generates a CritGraph.

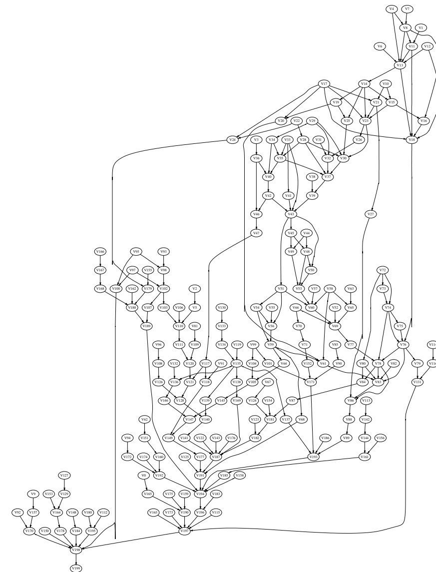
CriticalityDigraph Takes a weighted acyclic CritGraph and calculates the criticality number for each node.

Figure 6.1: Two graphs generated by RandomLocalDigraph, n=200. Visualized by Dot

(a) More global interconnects, div=1, chance=0.02.



(b) More local interconnects, div=200, chance=0.3.



Testing of the source code was done with GCC 2.95.4, standard 2.4.21 kernel on a Debian Linux 3.0 i386 system with 512 MB RAM. Memory usage on large graphs (n=6000) was limited to 120 MB. The executable is about 100KB in size.

For performance reasons, the implementation of sets was done with arrays, instead of sets such as $std :: set <>$. Although a lot of re-indexing is required, this was much

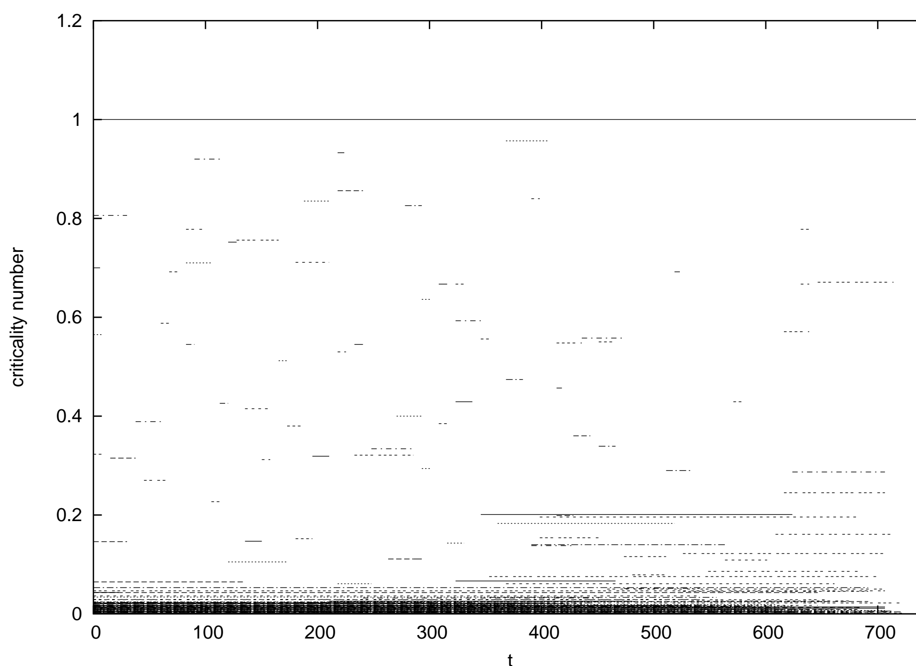
faster than the version with sets. To search for an element in a set takes $O(\log n)$ time, which is too much. Further optimization of this program should be done by optimizing the algorithm itself. The algorithms complexity slows down the program significantly when $n > 1000$. To observe the part of the algorithm which takes most time, each part of the program sends out characters on stderr. The slowdown occurs in parts of the code which have a high algorithmic complexity.

To generate a time/criticality diagram, the algorithm sends a c++ comment line to stdout. This comment line, such as:

```
//CriticalityDiagram:3171 3196 0.96
```

indicates that, at t=3171 up to t=3196, there is (a set of) process(es) with a criticality of 0,96. The *//* is there to keep dot from interpreting it as graph data. The data *3171 3196 0.96* was used by a Tcl/Tk script and Gnuplot to generate a time/criticality diagram, such as Fig. 6.2. The Tcl/Tk script is omitted for brevity.

Figure 6.2: Automatically generated time/criticality diagram. Visualized by Gnuplot.



7 Conclusions and Future Work

This thesis presents the criticality function for Kahn process network applications. It indicates which process is critical and by how much. A few performance goals for this function are given and an algorithm is provided to obtain such a function using application profiling data. Then it is explained how this function can be used to do YAPI system design. Furthermore, a few key implementation details are given and/or described. There is a description of the profiler and an implementation of (part of) the algorithm which finds a criticality function.

The result is that a criticality function is a tool that aids in all 4 tasks of a YAPI system design. These tasks are KPN application modeling, architecture modeling for this application, performance analysis and (evaluation of) the mapping of processes (i.e. scheduling). In contrast to the original intent of YAPI, this is not done through performance analysis of an application that executes on a system, but through (mostly) system independent application analysis. This allows the system designer to do this task without the need of a (predefined) system.

Much future work can be done on aspects of criticality functions. First, the algorithm with which the criticality function is generated can be improved in several ways. For example, our algorithm which generates a criticality function for acyclic KPNs removes a number of nodes per iteration. This allows the graph to no longer be weakly connected. The subgraphs can have their criticality function calculated separately which would result in fewer calculations.

Second, there may be other ways to generate a criticality function. These algorithms might be less complex, can reach different goals or have different use of (other)

resources such as communication.

Third, to really test and validate this method, a large Kahn process network application needs to be available. At the moment this is not the case. This thesis makes a number of assumptions which cannot be verified to be applicable. The model of cyclic graphs, which was not formally defined, cannot be proven at the moment.

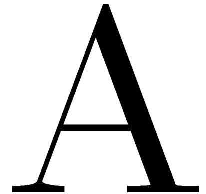
Fourth, the model that is used to generate a directed weighted graph from a simulation of a KPN is a simplified model of reality. It assumes that one process may not start to execute before all its predecessors are finished. In reality, the data is available earlier. This was not modeled to reduce modeling complexity.

Fifth, the criticality function algorithm uses a cost intuition that can be further formalized and rationalized. The exact cost function of executing a program and the minimization of this cost via the criticality function is unsolved. Also unsolved is the rationalization of a cost function. What is the cost price for executing at a certain speed? There are all sorts of ways to execute a process faster. The price/performance curve can be used to calculate real costs of the designed system in combination with an application.

Sixth, one aspect of YAPI is mapping/scheduling. One could expect to find a better scheduling method with a given criticality function. We do not have concrete ideas on how to explore this, however.

Last, this thesis does not completely address the intermediate step between processing on a homogeneous system with too few processors and a system with too many processors. Time was not available to analyze the cACC function. Although a few preliminary tests were performed, which indeed indicated that the average of cACC provides the minimal number of processors needed to execute with minimum execution time, no proof or reliable heuristics can be given now.

Source Code Acyclic Algorithm



A.1 Introduction

This appendix contains the C++ sourcecode of an implementation of the acyclic algorithm. The Boost Graph Library (BGL) was used to implement it. The BGL can be obtained from http://www.boost.org.

A.2 ascf.cpp

```
//*******************************************************************************
// (C) 2003 David Hofstee
//*******************************************************************************
// Important Notice: This source code is covered by copyright. You may browse
// the sourcecode at your convenience, similar as you may read a journal
// or a proceeding article in a public library. Using this source code for other than
// educational purposes may violate the copyright protection law.
//*******************************************************************************

#include <iostream>                     // for std::cout
#include <fstream>
#include <string>
#include <cstring>
#include <cstdlib>                      // for rand

#include <Critlib.hpp>  //includes definition CritGraph, RandomDigraph, DotPrint2, DotPrintAll
                        // CriticalityDigraph, definition of INT_TO_STR
                        // Also includes all boost headers!!!

int main(int argc ,char* argv[])
{
  //A CritGraph is a <listS,listS,bidirectionalS,VertexPropertyMap,no_property> graph,
  //see boost.org for extensive documentation.
  typedef property_map<CritGraph, vertex_index_t>::type Index;
  typedef property_map<CritGraph, vertex_weight_t>::type Weight;
  typedef property_map<CritGraph, vertex_criticality_t>::type Criticality;
  typedef property_map<CritGraph, vertex_begin_t>::type Begin;
  typedef property_map<CritGraph, vertex_end_t>::type End;
  typedef property_map<CritGraph, vertex_lcs_t>::type Lcs;
  typedef property_map<CritGraph, vertex_name_t>::type Name;

  typedef boost::graph_traits<CritGraph>::vertex_iterator vertex_iterator;

  int nodes = 30;
  int div = 30;
  double chance = 0.5;
  if(argc>2) {  //get arguments from commandline
    nodes = atoi(argv[1]);
    div = atoi(argv[2]);
    chance = double(atof(argv[3]));
  }
  CritGraph g3(nodes);
  RandomLocalDigraph(g3,chance,div);

  //get its attributes
  Weight weight3 = get(vertex_weight, g3);
  Index index3 = get(vertex_index, g3);
  Criticality criticality3 = get(vertex_criticality, g3);
  Begin begin3 = get(vertex_begin, g3);
  End end3 = get(vertex_end, g3);
  Lcs lcs3 = get(vertex_lcs, g3);
  Name vnames3 = get(vertex_name, g3);

  //give all vertices an index and name, this makes it possible to print...
  int nv = num_vertices(g3);
  vertex_iterator ib, iend;
  tie(ib,iend)=vertices(g3);
  for(int i=0;i<nv;i++) {
    index3[*ib]=i;
    string i_str;
    INT_TO_STR(i_str, i);
    vnames3[*ib]="V"+i_str;
    ib++;
  }

  //now give random weight between 1 and 10
  tie(ib,iend)=vertices(g3);
  for(;ib!=iend;ib++)
    weight3[*ib]=int((double(rand())/RAND_MAX)*10)+1;

  //determine criticality number
  CriticalityDigraph(g3);

  //DotPrint2(std::cout, g3, index3, vnames3, criticality3, weight3);
  DotPrintAll(std::cout, g3, index3, vnames3, criticality3, begin3, end3, weight3, lcs3);

  //exit
  return 0;
}
```

A.3 Critlib.hpp

```
//*******************************************************************************
// (C) 2003 David Hofstee
//*******************************************************************************
// Important Notice: This source code is covered by copyright. You may browse
// the sourcecode at your convenience, similar as you may read a journal
// or a proceeding article in a public library. Using this source code for other than
// educational purposes may violate the copyright protection law.
//*******************************************************************************

#include <iostream>                  // for std::cout
#include <fstream>
#include <utility>                   // for std::pair
#include <algorithm>                 // for std::for_each
#include <string>
#include <cstring>
#include <cstdlib>                    // for rand
#include <map>

#include <boost/config.hpp>
#include <boost/utility.hpp>                  // for boost::tie
#include <boost/graph/graph_traits.hpp>  // for boost::graph_traits
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/properties.hpp>     // for BOOST_DEF_PROPERTY
#include <boost/property_map.hpp>
#include <boost/graph/copy.hpp>              // for boost::copygraph...

#define  TESTING    1
#define  TESTCOUT   if(TESTING) std::cout
#define  TEST_CERR  if(TESTING) std::cerr
#define  INT_TO_STR(STR,INT) if(1){int ITS_int=INT; std::string ITS_neg=""; if(ITS_int<0){ITS_neg="-";\
 ITS_int=-ITS_int;} STR=""; if(ITS_int==0) STR="0"; const char* ITS_inttochar="0123456789"; \
 for(;ITS_int!=0;ITS_int=ITS_int/10) STR=ITS_inttochar[ITS_int%10]+STR; STR=ITS_neg+STR;}
#define  FORMAT_DOT_TEXT "size=\"7.5,10\"; ratio=fill;"

using namespace boost;

//************************************************************
// Definition of CritGraph
//************************************************************
//Defining possible properties, which are later added:
// - criticality, LCS
// - begin, end
// - weight
enum vertex_criticality_t { vertex_criticality = 1 };
enum vertex_begin_t { vertex_begin = 0 };
enum vertex_end_t { vertex_end = 0 };
enum vertex_weight_t { vertex_weight = 0 };
enum vertex_lcs_t { vertex_lcs = 0 };
namespace boost {
  BOOST_INSTALL_PROPERTY(vertex, criticality);
  BOOST_INSTALL_PROPERTY(vertex, begin);
  BOOST_INSTALL_PROPERTY(vertex, end);
  BOOST_INSTALL_PROPERTY(vertex, weight);
  BOOST_INSTALL_PROPERTY(vertex, lcs);
}
//Adding properties to VertexProperty and defining CritGraph.
namespace boost {
  typedef property < vertex_index_t, int > Index_Pr;
  typedef property < vertex_criticality_t, double, Index_Pr > Crit_Pr;
  typedef property < vertex_begin_t, double, Crit_Pr > Begin_Pr;
  typedef property < vertex_end_t, double, Begin_Pr > End_Pr;
  typedef property < vertex_weight_t, double, End_Pr > Weight_Pr;
  typedef property < vertex_lcs_t, double, Weight_Pr > Lcs_Pr;
  typedef property < vertex_name_t, std::string, Lcs_Pr > VertexProperty;
  typedef adjacency_list < boost::listS, boost::listS, boost::bidirectionalS, VertexProperty > CritGraph;
}


//************************************************************
// RandomDigraph(g&, double chance=0.5)
//
// RandomDigraph will take graph g (which should not have edges)
// and creates a random graph which is directed. Chance is the
// probability that an edge is created to a higher indexed node.
// chance should be set lower, 0,5 is high.
//************************************************************
template < class G >
void RandomDigraph(G& g, double chance=0.5) {
  typedef typename boost::graph_traits<G>::vertex_iterator vertex_iterator;
  typedef typename boost::graph_traits<G>::vertex_descriptor vertex_descriptor;

  //put all vertices in array
  int nv = num_vertices(g);
  std::vector<vertex_descriptor> vertex_descr_vec(nv);
```

```
  vertex_iterator ib, iend;
  boost::tie(ib,iend)=vertices(g);
  for(int i=0;i<nv;i++) {
    vertex_descr_vec[i]=static_cast<vertex_descriptor>(*ib);
    ib++;
  }

  //add edges
  for(int u=0;u<nv;u++) {  //for each edge...
    for(int v=u+1;v<nv;v++) {
      if( (double(rand())/RAND_MAX) < chance ) { //add e(u,v,g)
        add_edge(vertex_descr_vec[u],vertex_descr_vec[v],g);
      }
    }
    if( u<(nv-1) && (boost::degree(vertex_descr_vec[u],g)==0) ) {
      // u+[0...(nv-u-2)]+1 => [u+1...nv-1]
      int rand_vertex = u + int((double(rand())/RAND_MAX)*(double(nv-u-2)))+1;
      add_edge(vertex_descr_vec[u],vertex_descr_vec[rand_vertex],g);
    }
    if( u==(nv-1) && (boost::degree(vertex_descr_vec[u],g)==0) ) {
      int rand_vertex = int((double(rand())/RAND_MAX)*(double(nv-1)));
      add_edge(vertex_descr_vec[rand_vertex],vertex_descr_vec[u],g);
    }
    //      if(u==(nv-2)) add_edge(vertex_descr_vec[u],vertex_descr_vec[(nv-1)],g);
  }
}


//***********************************************************
// RandomLocalDigraph(g&, double chance=0.5)
//
// RandomLocalDigraph will take graph g (which should not have edges)
// and creates a random graph which is directed. Chance is the
// probability that an edge is created to a higher indexed node.
// Difference from RandomDigraph is that chances are higher if the
// index of the node is near the other one (locality).
// chance should be set lower, 0,5 is high.
//***********************************************************
template < class G >
void RandomLocalDigraph(G& g, double chance=0.5, double beta = 10.0) {
  typedef typename boost::graph_traits<G>::vertex_iterator vertex_iterator;
  typedef typename boost::graph_traits<G>::vertex_descriptor vertex_descriptor;

  //put all vertices in array
  int nv = num_vertices(g);
  std::vector<vertex_descriptor> vertex_descr_vec(nv);
  vertex_iterator ib, iend;
  boost::tie(ib,iend)=vertices(g);
  for(int i=0;i<nv;i++) {
    vertex_descr_vec[i]=static_cast<vertex_descriptor>(*ib);
    ib++;
  }

  //add vertices
  for(int u=0;u<nv;u++) {  //for each edge...
    for(int v=u+1;v<nv;v++) {
      if( (double(rand())/RAND_MAX) < (chance*(1-(beta*(double((v-u)*(v-u))/(double((nv-u)*(nv-u))))))) ) { //add e(u,v,g)
        add_edge(vertex_descr_vec[u],vertex_descr_vec[v],g);
      }
    }
    // ensure that there is at least one outgoing node
    if( u<(nv-1) && (boost::out_degree(vertex_descr_vec[u],g)==0) ) {
      // u+[0...(nv-u-2)]+1 => [u+1...nv-1]
      int rand_vertex = u + int((double(rand())/RAND_MAX)*(double(nv-u-2)))+1;
      add_edge(vertex_descr_vec[u],vertex_descr_vec[rand_vertex],g);
    }
    //the last one can not have an edge to another node, so we create an edge _to_ it.
    if( u==(nv-1) && (boost::degree(vertex_descr_vec[u],g)==0) ) {
      int rand_vertex = int((double(rand())/RAND_MAX)*(double(nv-1)));
      add_edge(vertex_descr_vec[rand_vertex],vertex_descr_vec[u],g);
    }
    //      if(u==(nv-2)) add_edge(vertex_descr_vec[u],vertex_descr_vec[(nv-1)],g);
  }
}


//***********************************************************
// DotPrint2
//
// Dotprint will take a graph g and make an indexed graph in the
// dot language. The label of each node will be
// "name(vertex)/p1(vertex)/p2(vertex)".
//***********************************************************
template < class G, class Index, class Name, class P1, class P2 >
void DotPrint2(std::ostream& out, const G& g,
               const Index& index, const Name& name, const P1& p1, const P2& p2) {
  //necessary typedefs
```

```
  typedef typename boost::graph_traits<G>::vertex_iterator vertex_iterator;
  typedef typename boost::graph_traits<G>::out_edge_iterator out_edge_iterator;
  typedef typename boost::graph_traits<G>::vertex_descriptor vertex_descriptor;
  // printing digraph, first its nodes, then the edges
  out << "digraph G {" << std::endl << FORMAT_DOT_TEXT << std::endl;
  std::pair<vertex_iterator, vertex_iterator> v;
  std::cout.precision(3);
  for(v = vertices(g);v.first!=v.second;v.first++) {
    out << index[*v.first]
        << " [label=\""
        <<name[*v.first] << "/"
        <<p1[*v.first] << "/"
        <<p2[*v.first] << "\"];" <<std::endl;
  }
  for(v = vertices(g);v.first!=v.second;v.first++) {
    out << "// node " <<name[*v.first] << std::endl;
    std::pair<out_edge_iterator, out_edge_iterator> ai;
    vertex_descriptor vert;
    for(ai = out_edges(*v.first, g);ai.first!=ai.second; ++ai.first) {
      vert = target(*ai.first, g);
      out << "\t"
          << index[*v.first]
          <<"->"
          << index[vert] << ";" << std::endl;
    }
  }
  out << "}" << std::endl;
}


//**********************************************************
// DotPrintAll
//
// Dotprint will take a graph g and make an indexed graph in the
// dot language. The label of each node will be:
// index(v) [label="name(v)",c=crit(v),b=begin(v),e=end(v),w=weight(v),lcs=lcs(v)]
//**********************************************************
template < class G, class Index, class Name, class Crit,
           class Begin, class End, class Weight, class Lcs >
void DotPrintAll(std::ostream& out, const G& g, const Index& index, const Name& name, const Crit& crit,
                 const Begin& begin, const End& end, const Weight& weight, const Lcs& lcs) {
  //necessary typedefs
  typedef typename boost::graph_traits<G>::vertex_iterator vertex_iterator;
  typedef typename boost::graph_traits<G>::out_edge_iterator out_edge_iterator;
  typedef typename boost::graph_traits<G>::vertex_descriptor vertex_descriptor;
  // printing digraph, first its nodes, then the edges
  out << "digraph G {" << std::endl << FORMAT_DOT_TEXT << std::endl;
  //  out << "digraph G {"<< std::endl;
  std::pair<vertex_iterator, vertex_iterator> v;
  for(v = vertices(g);v.first!=v.second;v.first++) {
    if(begin[*v.first]<0.00000000001) begin[*v.first]=0; //dot has problems with numbers in xe-y format
    out << index[*v.first] << " ["
        <<"label=\"" <<name[*v.first] << "\""
        << ",c=" << crit[*v.first]
        << ",b=" << begin[*v.first]
        << ",e=" <<end[*v.first]
        << ",w=" << weight[*v.first]
        << ",lcs=" << lcs[*v.first] << "]" <<std::endl;
  }
  for(v = vertices(g);v.first!=v.second;v.first++) {
    out << "// node " <<name[*v.first] << std::endl; //describe which node, eg: out << "//node A\n"
    out << "\t";                                     //describe its edges, eg:  out << "\t0->1;0->4;"
    std::pair<out_edge_iterator, out_edge_iterator> ai;
    vertex_descriptor vert;
    for(ai = out_edges(*v.first, g);ai.first!=ai.second; ++ai.first) {
      vert = target(*ai.first, g);
      out << index[*v.first] << "->" << index[vert] << " ; ";
    }
    out << std::endl;
  }
  out << "}" << std::endl;
}

//**********************************************************
// DotReadAll
//
// DotReadAll will create a graph g, reading from std::in. It only
// takes .dot files generated by DotPrintAll. It generates statemenst like:
// index(v) [label="name(v)",c=crit(v),b=begin(v),e=end(v),w=weight(v),lcs=lcs(v)]
//**********************************************************
//template < class boost::CritGraph >
void DotReadAll(std::string fname, boost::CritGraph& g) {
  using namespace boost;
  VertexProperty vp;
  typedef graph_traits<CritGraph>::vertex_descriptor vertex_descriptor;
  typedef property_map<CritGraph, vertex_index_t>::type Index;
  typedef graph_traits<CritGraph>::vertex_iterator vertex_iterator;
```

```
//variables in a vertex property map...
std::string str, name, tmp;
int v_index;
double crit, begin, end, weight, lcs;

//creating things I need... A comment line is the code that generated the
// text that I want to read ...
std::ifstream in (fname.c_str());
if (!in) {cerr << "DotReadAll: cannot open " <<fname << endl; exit(14);}

//remove comments from input stream
getline(in, str); //out << "digraph CritGraph {"<< std::endl;
while(str.find("//")==0 && !in.eof()) {
  getline(in, str);
}
getline(in, str); //out << "size = ".....

//read all vertices
getline(in,str);
while(str.find("//")!=0 && !in.eof()) {
  //e.g.: 6 [label="G",c=0,b=0,e=0,w=3,lcs=0]
  tmp = str.substr(0, str.find(" "));
  v_index = atoi(tmp.c_str());
  name    = str.substr((str.find('\"')+1),(str.find("\",")-str.find('\"')-1));
  tmp     = str.substr((str.find("c=")+2),(str.find(",b")-2-str.find("c=")));
  //      cout << tmp << " ";
  crit    = atof(tmp.c_str());
  tmp     = str.substr((str.find("b=")+2),(str.find(",e")-2-str.find("b=")));
  //      cout << tmp << " ";
  begin   = atof(tmp.c_str());
  tmp     = str.substr((str.find("e=")+2),(str.find(",w")-2-str.find("e=")));
  //      cout << tmp << " ";
  end     = atof(tmp.c_str());
  tmp     = str.substr((str.find("w=")+2),(str.find(",lcs")-2-str.find("w=")));
  //      cout << tmp << " ";
  weight = atof(tmp.c_str());
  tmp     = str.substr((str.find("lcs=")+4),(str.find("]")-4-str.find("lcs=")));
  //      cout << tmp << " "<< std::endl;
  lcs     = atof(tmp.c_str());
  //std::cout << "Index = "<< v_index << ", Name = "<< name
  //          <<" c="<< crit << " b="<< begin
  //          <<" e="<< end  << " w="<< weight
  //          <<" lcs="<< lcs <<std::endl;

  //this is a very badly documented part of Boost! In libs/graph/doc/transitive_closure.w
  // you can read what and why... eg: add_vertex(Index(i, Name('a' + i)), G);
  //VertexProperty(name, Lcs_Pr(lcs, Weight_Pr(weight, End_Pr(end, Begin_Pr(begin, Crit_Pr(crit, Index_Pr(i)))))))
  add_vertex(VertexProperty(name,
                            Lcs_Pr(lcs,
                                   Weight_Pr(weight,
                                             End_Pr(end,
                                                    Begin_Pr(begin,
                                                             Crit_Pr(crit,
                                                                     Index_Pr(v_index))))))),
             g); //add_vertex(property_map,g)
  //do again...
  getline(in,str);
}

//put all vertices in array
std::vector<vertex_descriptor> vertex_descr_vec(num_vertices(g));
vertex_iterator ib, iend;
vertex_descriptor vd;
Index index = get(vertex_index, g);
for(tie(ib,iend)=vertices(g);ib!=iend;++ib) {
  int i = index[*ib];
  vd = static_cast<vertex_descriptor>(*ib);
  vertex_descr_vec[i]=vd;
}

//Now add all edges...
std::string tmp_u,tmp_v;
int u,v;
while(str.find("}")!=0 && !in.eof()) {
  // eg: // node C
  //          2 -> 4 ; 2 -> 5 ; 2 -> 6 ;
  if(str.find("//")!=0) {
    while(str.find("->")!=string::npos) {
      //std::cout << str;
      tmp_u = str.substr(0,str.find("->"));
      tmp_v = str.substr((str.find("->")+2),str.find(";"));
      u = atoi(tmp_u.c_str());
      v = atoi(tmp_v.c_str());
      add_edge(vertex_descr_vec[u],vertex_descr_vec[v],g);
      str = str.substr((str.find(";")+1),string::npos);
      //std::cout << " so I added: "<< u <<"->"<< v << std::endl;
```

```
      }
    }
    //do again...
    getline(in,str);
  }
}


void CriticalityDigraph(CritGraph& g_orig) {
  typedef property_map<CritGraph, vertex_index_t>::type Index;
  typedef property_map<CritGraph, vertex_weight_t>::type Weight;
  typedef property_map<CritGraph, vertex_criticality_t>::type Criticality;
  typedef property_map<CritGraph, vertex_begin_t>::type Begin;
  typedef property_map<CritGraph, vertex_end_t>::type End;
  typedef property_map<CritGraph, vertex_lcs_t>::type Lcs;
  typedef property_map<CritGraph, vertex_name_t>::type Name;
  typedef boost::graph_traits<CritGraph>::vertex_iterator vertex_iterator;
  typedef boost::graph_traits<CritGraph>::vertex_descriptor vertex_descriptor;
  typedef boost::graph_traits<CritGraph>::in_edge_iterator in_edge_iterator;
  typedef boost::graph_traits<CritGraph>::out_edge_iterator out_edge_iterator;

  int nv;
  vertex_iterator ib, iend;
  std::set<vertex_descriptor> setB;
  std::set<vertex_descriptor> setU;
  double temp_end, temp_begin;
  vertex_descriptor v;
  in_edge_iterator eib, eiend;
  double L=0;
  std::set<vertex_descriptor>::iterator ib_setvd;
  std::set<vertex_descriptor>::iterator iend_setvd;
  int u_index;
  double c_max;
  double max_begin, max_end;
  double c;
  int b_max_index;
  vertex_descriptor u_max;
  std::set<vertex_descriptor>::iterator i_b;
  std::set<vertex_descriptor>::iterator i_u, i_u_temp;
  int loop_teller;

  //void copy_graph(const VertexListGraph& G, MutableGraph& G_copy,
  //     const bgl_named_params<P, T, R>& params = all defaults)
  CritGraph g(0);
  copy_graph(g_orig, g);

  //See if it works...
  Weight weight = get(vertex_weight, g);
  Index index = get(vertex_index, g);
  Criticality criticality = get(vertex_criticality, g);
  Begin begin = get(vertex_begin, g);
  End end = get(vertex_end, g);
  Lcs lcs = get(vertex_lcs, g);
  Name vnames = get(vertex_name, g);
  //DotPrintAll(std::cout, g, index, vnames, criticality, begin, end, weight, lcs);

  //Original needs to have property maps too
  Weight weight_orig = get(vertex_weight, g_orig);
  Index index_orig = get(vertex_index, g_orig);
  Criticality criticality_orig = get(vertex_criticality, g_orig);
  Begin begin_orig = get(vertex_begin, g_orig);
  End end_orig = get(vertex_end, g_orig);
  Lcs lcs_orig = get(vertex_lcs, g_orig);
  Name vnames_orig = get(vertex_name, g_orig);


  //put vertex_descriptors of g_orig in array, indexed. This way, the calculated
  // numbers can be put in the original graph.
  nv = num_vertices(g_orig);
  std::vector<vertex_descriptor> vertex_descr_vec(nv);
  boost::tie(ib,iend)=vertices(g_orig);
  for(int i=0;i<nv;i++) {
    vertex_descr_vec[index_orig[*ib]]= static_cast<vertex_descriptor>(*ib);
    ib++;
  }
  std::vector<vertex_descriptor> vertex_descr_vec2(nv);
  boost::tie(ib,iend)=vertices(g);
  for(int i=0;i<nv;i++) {
    vertex_descr_vec2[index_orig[*ib]]= static_cast<vertex_descriptor>(*ib);
    ib++;
  }

  //*****************************
  //INIT of ALGORITHM
  //*****************************
  //determine set B and U
  boost::tie(ib,iend)=vertices(g);
```

```
for(;ib!=iend;ib++) {
  if(boost::in_degree(*ib,g)==0)
    setB.insert(*ib);
  if(boost::out_degree(*ib,g)==0)
    setU.insert(*ib);
  begin[*ib]=0;
}

//determine critical path length L, ib:=(vertex) iterator begin, eib=edge iterator begin,...
for(int i=0;i<nv;i++) {
  temp_begin=0;
  v = vertex_descr_vec2[i];
  for(boost::tie(eib,eiend)=in_edges(v,g);eib!=eiend;eib++) {
    if(end[source(*eib,g)]>temp_begin)
      temp_begin=end[source(*eib,g)];
  }
  temp_end=temp_begin+weight[v];
  begin[v]=temp_begin;
  end[v]=temp_end;
}
//DotPrintAll(std::cout, g, index, vnames, criticality, begin, end, weight, lcs);

L=0;
ib_setvd    = setU.begin();
iend_setvd = setU.end();
for(;ib_setvd!=iend_setvd;ib_setvd++) {
  if(end[*ib_setvd]>L)
    L=end[*ib_setvd];
}
std::cout << "// Critical path length = " <<L << std::endl;
for(int i=0;i<nv;i++) {
  end[vertex_descr_vec2[i]]=L;
  begin[vertex_descr_vec2[i]]=0;
}

//******************************
//MAIN LOOP of ALGORITHM
//******************************
std::vector<int> now_index(nv);
std::vector<vertex_descriptor>** path = new std::vector<vertex_descriptor>*[num_vertices(g)];
std::vector<double>** pathl = new std::vector<double>*[num_vertices(g)];
while(num_vertices(g)!=0) {
  TEST_CERR << "~";
  //create table which translates the node's index into reduced index
  int t_now_index = 0;
  int num_ver = num_vertices(g);
  TEST_CERR << num_ver;
  for(boost::tie(ib,iend)=vertices(g);ib!=iend;ib++) {
    now_index[index[*ib]]=t_now_index;
    t_now_index++;
  }

  //create path and pathlength... path is defined differently (to save space)
  TEST_CERR << "!";
  int setBsize = setB.size();
  for(int i=0;i<num_ver;i++) {
    path[i]=new std::vector<vertex_descriptor>(setBsize);
  }
  for(int i=0;i<num_ver;i++) {
    pathl[i]=new  std::vector<double>(setBsize);
    for(int j = 0;j<setBsize;j++) {
      (*pathl[i])[j] = -1;
    }
  }

  //for each v in B do: pathl=w; path=this
  TEST_CERR << "@";
  ib_setvd = setB.begin();
  int t_index;
  for(int i=0;i<setBsize;i++) {
    t_index = index[*ib_setvd];
    t_index = now_index[t_index];
    (*pathl[t_index])[i]= weight[*ib_setvd];
    (*path[t_index])[i] = *ib_setvd; //end of path indicated with pointer to itself
    ib_setvd++;
  }
  TEST_CERR << setBsize;

  //calculate pathlength in topological order (which is, by non-coincidence, by index)
  TEST_CERR << "#";
  double v_weight;
  int v_index;
  int pred_index;
  vertex_descriptor pred_node;
  for(boost::tie(ib,iend)=vertices(g);ib!=iend;ib++) {
    v = *ib;
    v_weight = weight[v];
```

```
    v_index = now_index[index[v]];
    for(boost::tie(eib,eiend)=in_edges(v,g); eib!=eiend; eib++) {
      pred_node = source(*eib,g);
      pred_index = now_index[index[pred_node]];
      for(int j=0;j<setBsize;j++) {
        if( (*pathl[pred_index])[j]>0.0) {
          if( (*pathl[v_index])[j] < (*pathl[pred_index])[j]+v_weight) {
            (*pathl[v_index])[j]   = (*pathl[pred_index])[j]+v_weight;
            (*path[v_index])[j]    = pred_node;
          }
        }
      }
    }
  }

  //calculate largest criticality number
  TEST_CERR << "$";
  c_max = 0.0;
  loop_teller = 0;
  int setUsize = setU.size();
  double begin_b;
  //put begin/end time in array for faster lookup...
  double* begin_b_a = new double[setBsize];
  double* end_u_a   = new double[setUsize];
  i_b = setB.begin();
  for(int i=0;i<setBsize;i++) {
    begin_b_a[i]=begin[*i_b];
    i_b++;
  }
  i_u = setU.begin();
  for(int i=0;i<setU.size();i++) {
    end_u_a[i]=end[*i_u];
    i_u++;
  }
  //now for the real loop...
  i_b = setB.begin(); //i_b: setB iterator, i_u: setU iterator
  i_u_temp = setU.begin();
  for(int i=0;(i<setBsize && c_max!=1.0);i++) {
    //b_index = now_index[index[*i_b]];
    i_u = i_u_temp;
    begin_b = begin_b_a[i];
    for(int j=0;(j<setUsize && c_max!=1.0);j++) {
      u_index = now_index[index[*i_u]];
      if((*pathl[u_index])[i]>0) {
        c = double((*pathl[u_index])[i]/(end_u_a[j]-begin_b));
        if(c>c_max) {
          c_max = c;
          b_max_index = i;
          u_max= *i_u;
          max_end = end[*i_u];
          max_begin = begin[*i_b];
          loop_teller++;
        }
      }
      i_u++;
    }
    i_b++;
  }
  TEST_CERR << loop_teller;
  //spew out line for time/criticality diagram
  std::cout.precision(8);
  std::cout << "//CriticalityDiagram:" <<max_begin << " " <<max_end << " ";
  std::cout.precision(3);
  std::cout << c_max << std::endl;

  //put elements of largest path in list
  TEST_CERR << "%";
  std::list<vertex_descriptor> path_max(0);
  vertex_descriptor t_vd=u_max;
  path_max.insert(path_max.begin(),u_max);
  while(t_vd!=(*path[now_index[index[t_vd]]])[b_max_index]) {
    std::vector<vertex_descriptor> pathvector = (*path[now_index[index[t_vd]]]);
    t_vd = pathvector[b_max_index];
    path_max.insert(path_max.begin(),t_vd);
  }

  //assign criticality number to nodes, also in original graph
  TEST_CERR << "^";
  std::list<vertex_descriptor>::iterator icb, icend;
  icend=path_max.end();
  for(icb=path_max.begin();icb!=icend;icb++) {
    criticality[*icb]=c_max;
    criticality_orig[vertex_descr_vec[index[*icb]]]=c_max;
  }
  t_vd=u_max;
  begin[t_vd]=end[t_vd]-(weight[t_vd]/criticality[t_vd]);
  begin_orig[vertex_descr_vec[index[t_vd]]]=begin[t_vd];
```

```
    end_orig[vertex_descr_vec[index[t_vd]]]=end[t_vd];
    while(t_vd!=(*path[now_index[index[t_vd]]])[b_max_index]) {
      end[(*path[now_index[index[t_vd]]])[b_max_index]]=begin[t_vd];
      remove_edge((*path[now_index[index[t_vd]]])[b_max_index], t_vd, g); //remove edges from path_max...
      t_vd = (*path[now_index[index[t_vd]]])[b_max_index];
      begin[t_vd]=end[t_vd]-(weight[t_vd]/criticality[t_vd]);
      begin_orig[vertex_descr_vec[index[t_vd]]]=begin[t_vd];
      end_orig[vertex_descr_vec[index[t_vd]]]=end[t_vd];
    } //t_vd should now be set to the begin node, this is used later!

    //set begin/end attributes of adjacent vertices
    TEST_CERR << "&";
    icend=path_max.end();
    for(icb=path_max.begin();icb!=icend;icb++) {
      for(boost::tie(eib,eiend)=in_edges(*icb,g);eib!=eiend;eib++) {
        if(end[source(*eib,g)]>begin[*icb])
          end[source(*eib,g)] =begin[*icb];
      }
      out_edge_iterator boei, eoei;
      for(boost::tie(boei,eoei)=out_edges(*icb,g);boei!=eoei;boei++) {
        if(begin[target(*boei,g)]<end[*icb])
          begin[target(*boei,g)] =end[*icb];
      }
    }

    //remove nodes from path_max and add adjacent vertices in setB and setU
    TEST_CERR << "*";
    icend=path_max.end();
    TEST_CERR << path_max.size();
    for(icb=path_max.begin();icb!=icend;icb++) {
      //append adjacent nodes to B and U
      for(boost::tie(eib,eiend)=in_edges(*icb,g);eib!=eiend;eib++) {
        setU.insert(source(*eib,g));
      }
      out_edge_iterator boei, eoei;
      for(boost::tie(boei,eoei)=out_edges(*icb,g);boei!=eoei;boei++) {
        setB.insert(target(*boei,g));
      }
      //removing icb from B and U
      setB.erase(*icb);
      setU.erase(*icb);
      //removing edges and vertices from graph
      clear_vertex(*icb,g);
      remove_vertex(*icb,g);
    }
    path_max.clear();

    //delete path and pathlength
    TEST_CERR << "(";
    for(int i=0;i<num_ver;i++) {
      pathl[i]->~vector();
      path[i]->~vector();
    }

    //DotPrintAll(std::cout, g, index, vnames, criticality, begin, end, weight, lcs);

    TEST_CERR << ")" << std::endl;
  }
  std::cout << "//done_with_criticality_function" << std::endl;
  //  DotPrintAll(std::cout, g_orig, index_orig, vnames_orig, criticality_orig,
  //begin_orig, end_orig, weight_orig, lcs_orig);

  //template < class G, class Index, class Name, class P1, class P2 >
  //void DotPrint2(std::ostream& out, const G& g, const Index& index, const Name& name,
  //                const P1& p1, const P2& p2) {
  //DotPrint2(std::cout, g_orig, index_orig, vnames_orig, criticality_orig, weight_orig);

}
```

Bibliography

[1] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In *Communicating Process Architectures 2001*, 2001.

[2] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modelling for Signal Processing Systems. In *Proc. of the 37th Design Automation Conference*, pages 402–405, June 2000.

[3] David Hofstee and Ben H. H. Juurlink. Determining the Criticality of Processes in Kahn Process Networks for Design Space Exploration. In *Proceedings ProRISC 2002*, pages 292–297, November 2002.

[4] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *J. L. Rosenfeld, editor, Information Processing 74: Proc. IFIP Congress 74, North-Holland*, pages 471–475, August 1974.

[5] Authors: see website. World Wide Web, `http://www.research.att.com/sw/tools/graphviz/`.

[6] P. Stravers and J. Hoogerbrugge. Homogeneous Multiprocessing and the Future of Silicon Design Paradigms. In *Proc. of the 2001 International Symposium on VLSI Technology, Systems, and Applications.*, pages 184–187, April 2001.

[7] Boost People /people/people.htm. Boost. World Wide Web, http://www.boost.org.

[8] D. Zhu, R. Melhem, and B. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. In *22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 84–94, December 2001.