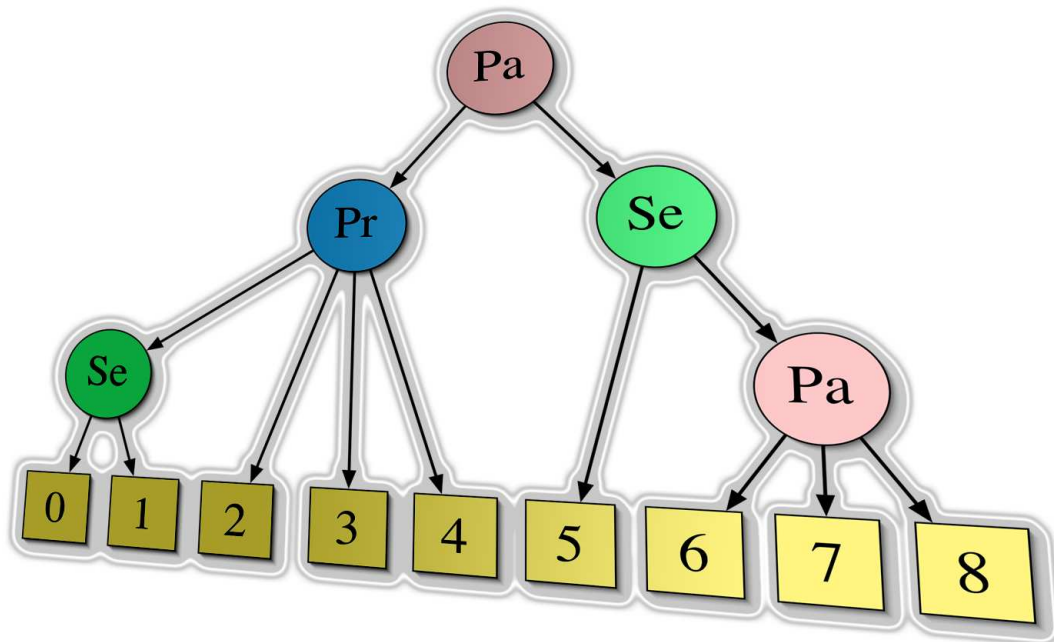


Computing the Modular Decomposition of a Graph

Lukas Hämmerle



Student Thesis SA-2004-03

Winter Term 2003/2004

February 20th 2004

Tutor: Prof. Dr. Thomas Erlebach

Supervisor: Prof. Dr. Thomas Erlebach

Abstract: This document describes how to efficiently compute the modular decomposition of a graph using the module factorizing permutation order and then applying several computation steps to this order. Although the implemented algorithm described in the following does not run in linear time $\mathcal{O}(n + m)$ it is quite fast and can compute the decomposition of graphs with several thousand nodes within few minutes.

Keywords: graph, graph algorithms, graph decompositions, factorizing permutation order, modular decomposition

Conventions and Preliminaries

In this document short forms and acronyms are used as follows.

Acronym	Meaning
MD	Modular Decomposition
MDT	Modular Decomposition Tree
FT	Fracture Tree
PG	Prime Graph
MFPO	Module Factorizing Permutation Order
AS	Autonomous System

The term *graph* will denote a simple, loop-free and undirected graph. The described implementation internally works with bidirected graphs but in fact calculates the MD for undirected graphs.

Let $G = (V, E)$ be a graph with *vertex set* V and *edge set* E . The number of nodes of G is $n = |V|$ and the number of edges of G is $m = |E|$. $N(v)$ will be the set of *neighbors* of node v and $\overline{N}(v)$ the set of *non-neighbors* of v . In the following G denotes the graph whose MD is calculated. \overline{G} denotes the complement graph of G . $G(X)$ will be the subgraph of G that is induced by a set of vertices X .

The MD of G consists of the MDT and a set of PGs, which store the adjacency relations between the child nodes of every prime node in the MDT.

Contents

1	Introduction	4
1.1	Goals	4
1.2	Use and Application	4
1.3	Theory	4
2	Algorithm and Implementation	6
2.1	MFPO	6
2.2	Fracture Tree Scans	9
2.2.1	MFPO Parenthesizing (Scan 1)	9
2.2.2	Building the FT (Scan 2)	10
2.2.3	Module Recognition (Scan 3)	10
2.2.4	Dummy Node Deletion (Scan 4)	11
2.2.5	Recovering Merged Modules (Scan 5)	11
2.2.6	Weak Module Deletion (Scan 6)	13
2.2.7	Final Output of MD	13
2.3	Problems	13
2.4	Timing Analysis	14
3	Analysis of AS Graphs	17
4	Conclusions and Extensions	19
5	Acknowledgments	21
A	File Structure	23
B	Compilation	24
C	Command line Options and Usage	25
C.1	Input and Output Files	25
C.2	Options	26
D	File Format	27
E	Other Examples	28

Chapter 1

Introduction

1.1 Goals

- Implement an efficient algorithm (possibly of $\mathcal{O}(n+m)$) to compute the MD of an undirected, simple graph.
- Analyze some large Internet graphs using the implemented algorithm.

1.2 Use and Application

Some applications of the MD are listed in [DGM01] that proposes another algorithm to compute the MD of a graph than the one used in the following. These include: graph orienting so that the resulting digraph is a poset relation ([G67, G80, MS99]), recognition of poset relations, comparability graphs, maximum independent sets, chromatic numbers and some problems on partial orders ([MR84, HM87]). See [M85, MA85, MR84] for surveys of applications of the MD.

1.3 Theory

Modules

A *module* of a graph $G = (V, E)$ is a set $X \subseteq V$ of vertices where all vertices in X have the same neighbors in $V \setminus X$. V and the singletons of V satisfy the above definition of a module. They are called *trivial modules*.

As seen in Figure 1.1 there can exist subsets of V that are modules which are not disjoint to other modules. A module with subset of vertices X of G overlaps another module with a subset of vertices Y of G if $X \cap Y \neq \{\}$, $X \setminus Y \neq \{\}$ and $Y \setminus X \neq \{\}$. Overlapping modules are called *weak modules* (e.g. $WM1, WM2, WM3$).

Strong modules (e.g. $SM4$ or $SM5$) are modules that do not overlap any other module. They can contain other strong modules. For example strong module $SM5$ contains the trivial strong module a and the non-trivial strong module $SM4$. A graph can have exponentially many modules but only $\mathcal{O}(n)$ strong modules.

Two strong modules M_1 with subset of vertices X_1 and M_2 with subset of vertices X_2 are *adjacent* to each other if every vertex of X_1 is adjacent to all vertices of X_2 and vice versa. Two modules are non-adjacent to each other if there is no vertex of X_1 that is adjacent to any other vertex of X_2 .

If X is a set of vertices that represents a strong module M , a *maximum strong module* of M is a subset of X that represents a strong module that is not contained by any other strong module than M .

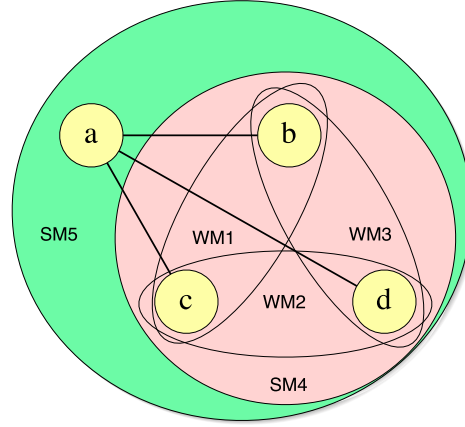


Figure 1.1: Weak and strong modules

Modular Decomposition

The MD of a graph G consists of the MDT T_{MD} whose inner nodes induce sets of strong modules of G . The leaf nodes of T_{MD} correspond to the vertices of G and are trivial modules by themselves. Any inner node of the MDT is of one of the three types *prime*, *series* and *parallel*. If there is a subset X of vertices of G that corresponds to a strong module M . Then M is of type:

- Prime** Both, $G(X)$ and $\overline{G}(X)$ are connected.
- Series** All maximum strong modules contained by M are **adjacent** to each other in $G(X)$. $\overline{G}(X)$ is not connected.
- Parallel** All maximum strong modules contained by M are **non-adjacent** to each other in $G(X)$. $G(X)$ is not connected.

In Figure 2.5 an example graph is drawn and in Figure 2.11 its corresponding MDT.

The MD of a graph is unique and fully defines a graph G that was decomposed. To reconstruct the graph G using the MD one needs additional information about the structure of the prime nodes in the MDT. Thus for every prime node an additional graph, called *prime graph*, has to be saved. A PG contains all the children nodes of the corresponding prime node in the MDT. These children nodes can be of any type (series, parallel, prime). The important information of the PGs is contained in the adjacency relations between the child nodes. A PG must not contain any non-trivial module.

See Figure 2.12 for the example PG of the MD in Figure 2.11.

Except for prime nodes, there must not be a child node in the MDT whose father is of the same type.

Chapter 2

Algorithm and Implementation

To compute the MD of a graph we use algorithms described in [HMP03], [CHM02] and [HPV99]. The calculation process can be divided into two phases:

1. Compute the MFPO of G with a partition refining algorithm
2. Compute the MDT and the PGs of G by applying six scans to the MFPO and its derived data structures.

In the following it is explained step-by-step how the algorithm works and how it was implemented.

C++ was used as programming language in combination with *Algorithmic Solutions*¹ LEDA library that provides a large set of powerful data structures and functions. All the provided functions and data structures are well documented, almost bug free (see Chapter 4) and efficiently implemented.

2.1 MFPO

A *Module Factorizing Permutation Order* is a permutation order where every strong module occurs as a factor. For a given graph there may exist up to $n!$ factorizing permutations.

Input and Output

The implementation starts by reading a specified graph and other command line arguments. See Appendix C for detailed description of the command line options.

Graph G first is checked. If it is not simple and loop-free the program exits with an error. In a next step G is converted to a bidirected graph, where $\forall(x, y) \in E : (y, x) \in E$. This way a directed graph can be made kind of undirected and thus conform to the problem.

Output of the following partition refining algorithm is an order called MFPO. There are exponentially many valid MFPO for a given graph and in fact the implementation will compute a random MFPO per default (there is a deterministic mode though, see Appendix C).

Refining

The refining algorithm uses a list of parts as data structure. A *part* contains a subset of vertices of G . P_i is the part containing the pivot node p_i that is used to split up another part P_j in its neighbors and non-neighbors of p_i .

There are mainly two important rules to compute the MFPO.

¹See LEDA product description <http://www.algorithmic-solutions.com/enleda.htm>

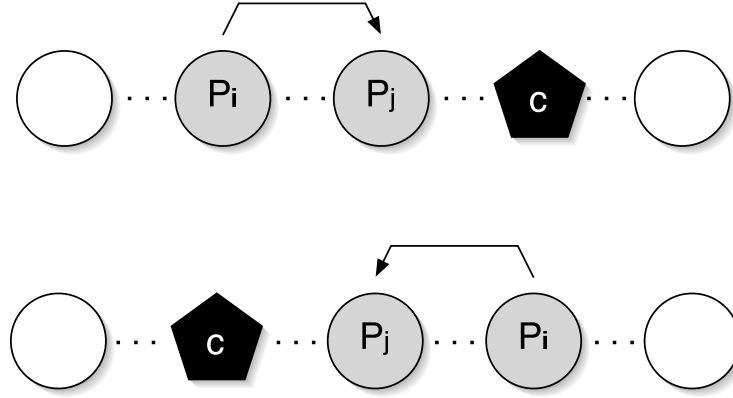


Figure 2.1: Condition for pivot rule case i

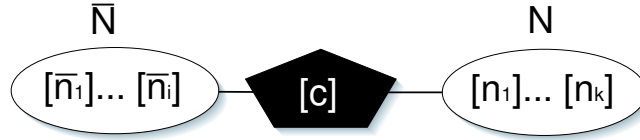


Figure 2.2: Result of applied center rule

- Center rule:

At the beginning of a recursion there is only one part P containing a set of vertices. The center rule picks an arbitrary vertex as center node c . Splitting up P according to Figure 2.2 gives the three parts, $\overline{N}(c)$ - c - $N(c)$.

- Pivot rule:

- Case i: $P_i \prec P_j \prec c$ or $c \prec P_j \prec P_i$ (see Figure 2.1)

P_j is split up with new part $N(p_i) \cap P_j$ before new part $\overline{N}(p_i) \cap P_j$ as can be seen in Figure 2.3.

- Case ii: Otherwise

P_j is split up with new part $\overline{N}(p_i) \cap P_j$ before new part $N(p_i) \cap P_j$ as shown in Figure 2.4.

The refining used in the implementation works as described in algorithm 1.

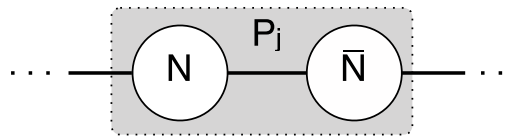


Figure 2.3: Result of applied pivot rule; case i

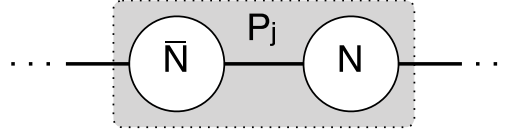


Figure 2.4: Result of applied pivot rule; case ii

Algorithm 1 Simple partition refining scheme from [HPV99]

```

begin
| Pick a center node  $c$ 
| Create  $\mathcal{O}$  using the center rule with  $c$ 
| repeat
| | Select a part  $P_i$  with a pivot  $p_i$ 
| | foreach part  $P_j \neq P_i$  do extend  $\mathcal{O}$  using pivot rule
| until the partition can not be extended further
end

```

How can one know what pivot rule to apply (1) and how can one know when the partition can't be extended further (2) in Algorithm 1?

1. There are two variables in the implementation; *PivotPassed* is only *true* when P_i in the list of parts lies behind part P_j that is split up. Equivalently for the variable *CenterPassed*.
2. At the beginning of a recursion (after the center rule was applied) all three parts are unmarked. From then on every part that was P_i is marked. Any newly split parts that were created by a pivot rule are unmarked. Marked parts can't be pivot parts but they can get split up (and therefore become unmarked again) by another pivot part. The implementation stops refining if every part is marked. Then it is checked if any part contains more than one node and if so, algorithm 1 recursively is called on that part.

After this phase, the algorithm has computed a valid MFPO where all the strong modules do occur as factors in consecutive order.

The parts and also the nodes within the parts are contained in lists. These lists are always traversed in the same direction. Every vertex in a part will be pivot p_i before the succeeding part is selected as P_i .

Example

To better illustrate the calculation of the MFPO and the following steps let's use an example (Figure 2.5, same example as used in [HMP03]). This example was run with command `'./cmd Graphs/InGraph.gw Graphs/OutGraph.gw -d -s -v'` that uses the deterministic mode and shows every calculation step.

The above command will compute the following MFPO for the example:

[1] [0] [2] [4] [3] [6] [7] [8] [5]

Any MFPO of this example can be generated if the child nodes of the MDT in Figure 2.11 are permuted. A depth-first search algorithm that would output only the leaf nodes then would deliver a valid MFPO.

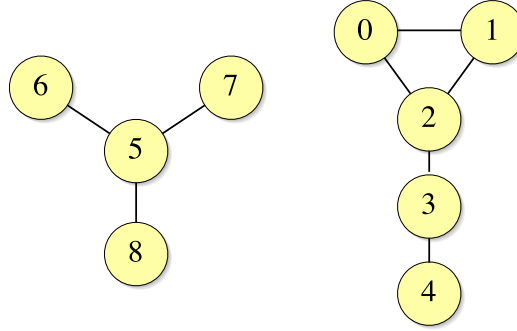


Figure 2.5: Example graph

2.2 Fracture Tree Scans

Cutters

Given a set $S \subset V$ of vertices, a vertex $x \notin S$ is a *cutter* of S if S is not a module of the induced graph $G = (S \cup x, E')$. Thus there must be at least one vertex v in S that has a different adjacency relation with x than some other vertex of $S \setminus v$.

In the following six scans we will make use of cutters to find the borders of modules.

2.2.1 MFPO Parenthesizing (Scan 1)

In this scan we are calculating the first cutter fc and the last cutter lc of a given pair (x, x') of consecutive nodes in the MFPO. Using these cutters a parenthesized version of the MFPO can be generated that is used in later steps to build a FT.

Since in the MFPO any strong module of G already is represented as a factor, the following scans have to find the borders of the strong modules. A set of vertices X represents a module only if there doesn't exist any cutter for X . Therefore cutters are found for nodes that are not part of the same module. The algorithm checks all consecutive pairs of nodes in the MFPO in order to find their cutters.

The implementation finds the left cutter of a pair of nodes (x, x') by checking the adjacency relation of every node v to the pair (x, x') beginning at the head of the list. Vice versa for the right cutter. Node v is a cutter of (x, x') if $v \in N(x) \wedge v \notin N(x')$ or $v \notin N(x) \wedge v \in N(x')$. If fc does exist, a pair of parentheses is inserted in the MFPO. The left parenthesis is inserted right before fc and the right parentheses is inserted right after x . The nodes within these newly inserted parentheses build the *left fracture*. If lc exists there must be a *right fracture* that is enclosed by parentheses from x' until lc . The cutters for every consecutive pair in the MFPO are also saved in an array for reuse in later calculation steps.

The final result of this scan is called *Dyck word*. It has as many opening parentheses as closing ones and between two nodes all the closing parentheses were inserted before the opening ones. But a pair of parentheses of a fracture f_1 may overlap the parentheses of another fracture f_2 . So they don't need to be correctly nested. Moreover a module and a fracture can not overlap.

The dyck word for the example looks as follows:

[2] [0] [4] [3] [2] [4] [3] [0] [3] [8] [8] [3]
 (([1] [0] ((([2]) [4]) [3])) (([6] [7] [8]) [5]))

Above every parenthesis the index of node x of the pair (x, x') is shown.

This calculation step can be improved to run in $\mathcal{O}(n + m)$. See Chapter 4 for a description.

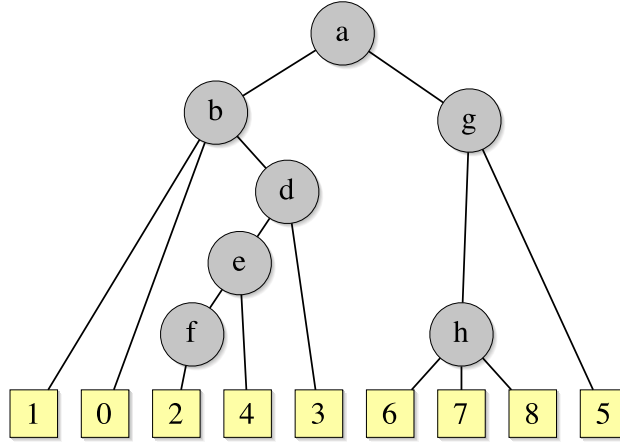


Figure 2.6: Fracture tree generated from MFPO

2.2.2 Building the FT (Scan 2)

As the name of this scan lets assume the goal of this calculation step is to create the FT that is used in the following scans to calculate the MDT. The Dyck word calculated in the first scan corresponds to a depth-first search of the FT, and therefore we have to do the reverse transformation in this scan.

The algorithm travels from the left to the right through the parenthesized MFPO from the previous scan. We build the FT according to the following rules. If we encounter:

- $($: Add a new child vertex c_i to the current node and make c_i the current node v_c
- $)$: Make the father of the current node to be the current node v_c .
- **[Node]** : Add a leaf node to current node v_c .

At the end of the dyck word the current node v_c should be again the root node of FT. This tree probably is quite deep and consists of many nodes that do not exist in the final MDT.

In the implementation of this algorithm a template graph is used to save the FT. All the leaf nodes store a pointer to the corresponding node of G . Vice versa the nodes of G contain pointers to the leaf nodes of FT. The inner nodes hold no information yet.

For the example of Figure 2.5 the FT is drawn in Figure 2.6. Notice that this is only one of many possible FTs.

2.2.3 Module Recognition (Scan 3)

The goal of scan 3 is to find inner nodes that represent modules and to mark the other non-leaf nodes as *dummy nodes*.

The resulting example FT is depicted in Figure 2.7. The three identified dummy nodes are d , e and f . The four numbers above the nodes stand for $(fv(v), lv(v), fc(v), lc(v))$ of a node v where fv stands for *first vertex* and lv for *last vertex*. The first vertex fv depicts the leftmost located leaf vertex of an inner node in the MFPO. Likewise, the last vertex lv is the rightmost leaf vertex of an inner node.

$fc(v)$ and $lc(v)$ denote the first and last cutter of an inner node v in the FT. Remember that every node in the FT represents a possible strong module. This scan identifies modules by searching the dummy nodes that do not represent a module.

There are two types of nodes that will be marked as dummy nodes:

- Nodes with only one child (node f in Figure 2.7)

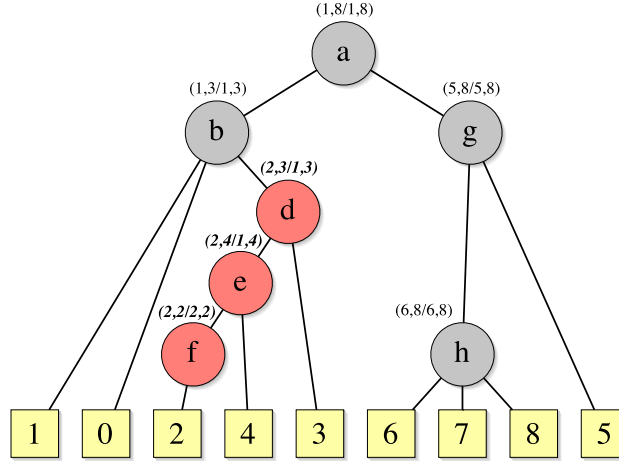


Figure 2.7: FT after module recognition step

- Nodes that do not represent a module (nodes d and e in Figure 2.7)

First category nodes are easy to find. They are marked so that we just have to check them for deletion in the following scan.

To identify dummy nodes the algorithm recursively does a bottom-up search that returns for every node the quadruple (fv, lv, fc, lc) . A leaf node will return its index number for fv , lv and nil for fc , lc since there is no cutter for a single node. The cutters for two consecutive vertices in the MFPO were already calculated in the first scan and can be reused.

So a parent node v first gets all the return values from its children. From these returned values the leftmost $fv(v)$, $fc(v)$ and the rightmost $lv(v)$, $lc(v)$ are calculated and then used to decide if v is a dummy node. This is the case if one of the following conditions is true.

- $fc(v)$ left of $fv(v)$ in the MFPO
- $lc(v)$ right of $lv(v)$ in the MFPO

The identified dummy nodes are marked by storing a pointer to themselves as node information while the inner nodes representing true modules store nil as node information.

2.2.4 Dummy Node Deletion (Scan 4)

The goal of this scan is to delete the found dummy nodes. It checks every node of V and deletes it if it was marked as dummy node in scan 3. This operation is of $\mathcal{O}(n)$.

The function that deletes a dummy node must connect the children to the parent node of the dummy. The edges leading to the children are inserted right before the edge of the dummy node that is deleted after all the children are connected.

After this first wave of dummy nodes is deleted, the resulting FT looks like in Figure 2.8.

2.2.5 Recovering Merged Modules (Scan 5)

As can be seen in Figure 2.8, the structure of the FT now resembles the final MDT quite well. But the strong module consisting of nodes 1,0 still is missing.

This scan identifies merging of strong series and parallel modules and recovers them by inserting new inner nodes in the FT. Moreover the types of all the inner nodes are identified in this scan.

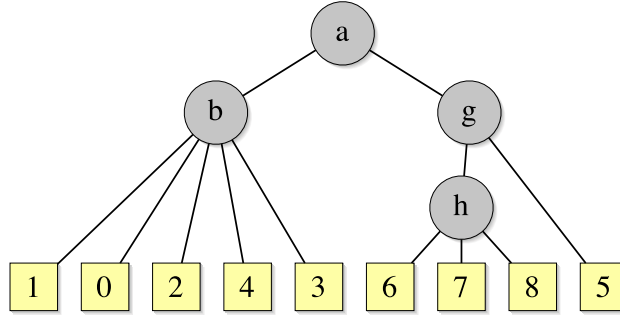


Figure 2.8: FT after dummy node deletion

Twins

Two vertices are twins if no vertex cuts them, respectively there does not exist a cutter for these two vertices. If they are adjacent they are *true twins*, if not they are *false twins*.

Recovering

The algorithm does again a bottom-up search that returns for every node v the tuple $(fv(v), lv(v))$. If the recursive function treats node v , it calls itself according to the MFPO with every child node c_i as new current node. The child nodes c_i, c_{i+1} , whose $lv(c_i), fv(c_{i+1})$ are consecutive vertices in the MFPO, return their values and the current node v uses them to check the twin relation. c_i, c_{i+1} with $(fv(c_i), lv(c_i), fc(c_i), lc(c_i))$ and $(fv(c_{i+1}), lv(c_{i+1}), fc(c_{i+1}), lc(c_{i+1}))$ are:

- **Twins:**
 For pair $p = (lv(c_i), fv(c_{i+1}))$;
 $fc(p)$ does not exist or is right/equal to $fv(c_i)$ and
 $lc(p)$ does not exist or is left/equal to $lv(c_{i+1})$
 The vertices c_i, c_{i+1} are:
 - **True twins:** If $lv(c_i) \in N(fv(c_{i+1}))$ in G
 - **false twins:** Otherwise
- **No twins:** Otherwise

Every pair of consecutive nodes in the MFPO is used only once.

In this calculation step the type of each inner node is identified.

- If all children of the current node v are **true twins**, v is a **series** node.
- If all children of the current node v are **false twins**, v is a **parallel** node.
- If two children of the current node v are **no twins** at all, v is a **prime** node.

The implementation checks the return values from the last two children for a twin relation. If there are some strong modules among the children of a prime node v , the representing vertices are merged under a new node as in the example of Figure 2.9, where the leaf nodes 1,0 are merged. The nodes that are to be merged appear consecutively according to the MFPO. There exists a strong module that has to be recovered if the current prime node has children that are twins. The merging results in a new series or a new parallel node. Only merging of series and parallel nodes can happen.

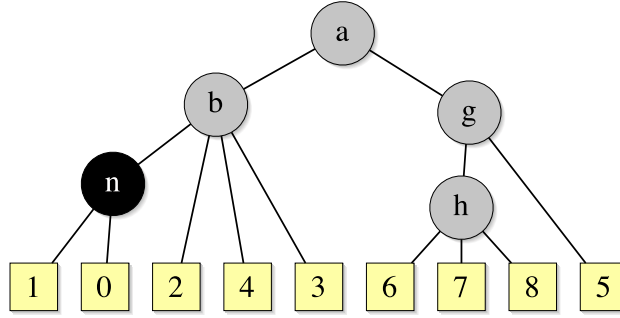


Figure 2.9: FT after one module was recovered

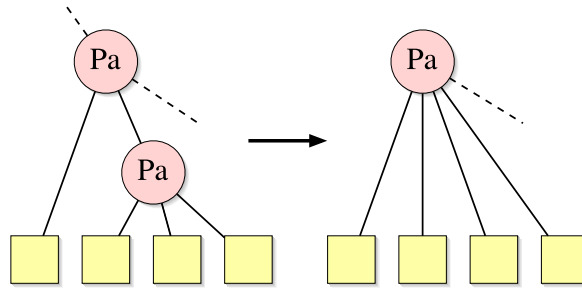


Figure 2.10: Example of a weak module deletion

2.2.6 Weak Module Deletion (Scan 6)

In this last step every inner series or parallel node v of the FT is checked. If v has the same type as its father it is deleted and its children are connected to the father node of v . In our example graph there is no inner node of the same type as its father. See Figure 2.10 for an imaginary example where the described situation occurs.

2.2.7 Final Output of MD

At this point we know the MDT of the original graph G . An additional step in the implementation will construct a PG for every prime node in the MDT. These PGs are saved together with the MDT as formatted *GraphWin* files. Check Appendix D to read what format is used to save these graphs.

The final result of the example graph is drawn in Figures 2.11 (MDT) and 2.12 (PG).

2.3 Problems

One of the problems during development of this implementation was understanding of the proposed algorithm of [CHM02]. Unfortunately the authors presented only a few examples and implementation instructions that could have helped to program their proposed algorithm.

Another problem is to verify the calculated results. Verifying the MD of simple example graphs (by eye) is not sufficient to prove the correctness of an implementation, since these examples most certainly do not cover all possible faults. Also it is not sufficient to declare an implementation as correct if it delivers always the same results, even if there exists randomness during calculation but this gives at least some credibility.

An additional verification method that is used in the implementation, is to add some functions that do a raw plausibility check of the calculated MD against some necessary rules. Not only

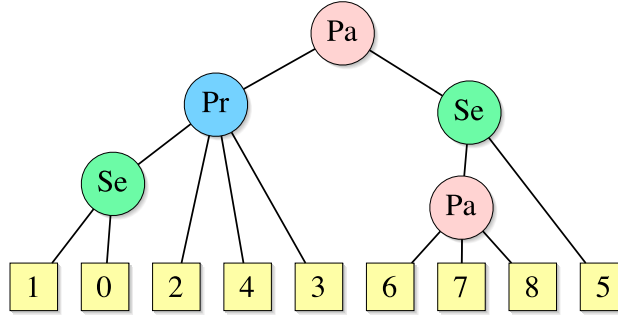


Figure 2.11: Modular Decomposition of example graph in Figure 2.5

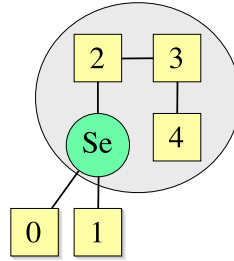


Figure 2.12: Prime graph of single prime node in Figure 2.11

necessary but also sufficient checks, e.g. verifying if a module is a strong module, seem to be difficult to find without computing the MD again.

Some other simple algorithm that computes the MD by enumerating all subsets of V and checking which of them are strong modules then was programmed by Prof. Thomas Erlebach. But this brute-force implementation is highly inefficient and can't be used with large graphs. To compute the MD of a graph with 30 nodes it takes about 11 hours of computation. But even if one had an implementation that can be trusted and that is fast enough, it is necessary to have a tool that compares the two resulting MDs.

In Chapter 4 there is another promising method of verification proposed.

2.4 Timing Analysis

To analyze the calculation time of the implementations it is important to use large graphs that take some time to calculate the MD. Therefore two AS graphs of 1997 with 3015 nodes/5158 edges and 2000 with 6474 nodes/12574 edges were used (see Chapter 3 for detailed information on AS graphs). Since the AS graph 2000 approximately has twice as many nodes and edges as the AS graph 1997, an $\mathcal{O}(n + m)$ algorithm should need about double the time for the calculation.

	AS 1997 [s]	AS 2000 [s]	Comparison
Partition Refining	18.84	114.50	607%
Parenthesizing	7.15	42.16	589%
Building Fracture Tree	0.02	0.04	200%
Finding dummy nodes	0.01	0.02	200%
Removing dummy nodes	0.01	0.02	200%
Strong module recovering	0.01	0.02	200%
Weak module deletion	0.01	0.01	100%
Total calculation time ²	26.04	156.75	601%

Table 2.1: Time analysis of AS graphs

In Table 2.1 and in Figures 2.13 and 2.14 we can get an idea which calculation steps are of $\mathcal{O}(n + m)$. Only the first two of the seven calculation steps really contribute to the running time remarkably. The partition refining step and the parenthesizing step are also the non-linear calculation steps of this implementation. See Chapter 4 for improvement suggestions.

²Calculation was done on a SunBlade 100, 500MHz, 512MB RAM. Values contain pure calculation time without overhead of reading/saving/analyzing graph. Averaged values of 10 runs.

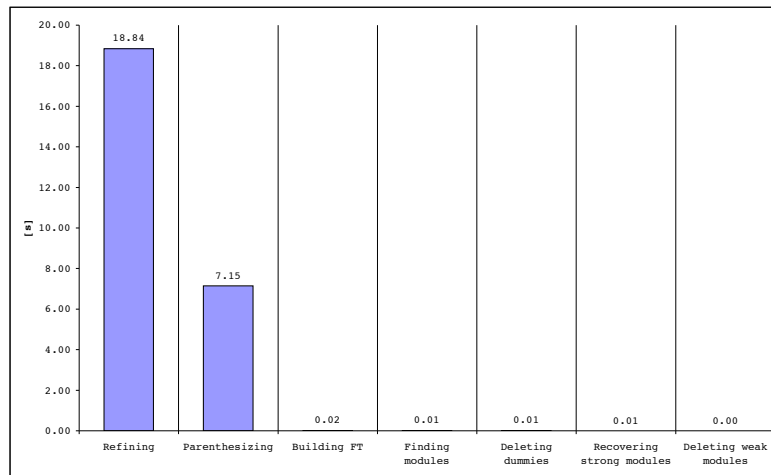


Figure 2.13: Distribution of calculation times AS 1997 graph

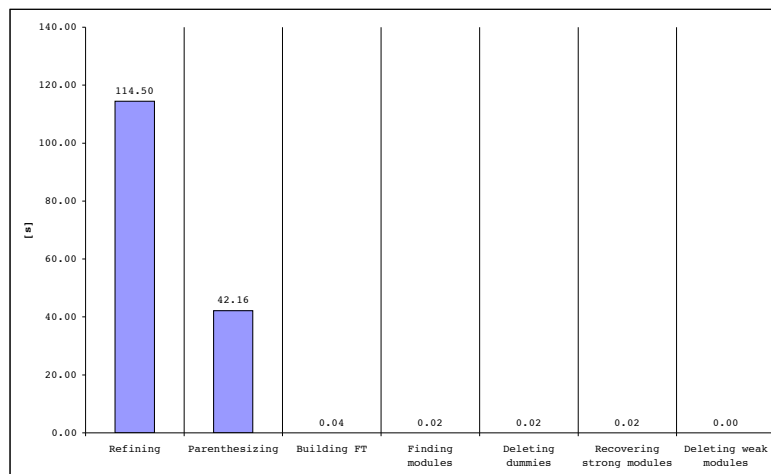


Figure 2.14: Distribution of calculation times AS 2000 graph

Chapter 3

Analysis of AS Graphs

Another task of this project was to use the implemented algorithm to analyze network graphs that represent the structure of the Internet. Since nowadays there are millions of hosts connected to the Internet it is nearly impossible to construct a network graph where every node corresponds to a network host. But it is rather easy to get a graph whose nodes represent single AS. An AS is a connected group of one or more Internet networks run by one or more network operators which has a *single* and *clearly defined* routing policy¹.

Two AS graphs were studied. They contain the adjacency relations between all the AS of the Internet as of November 8, 1997 and January 2, 2000.

The general structure of the MDT of these two AS graphs is shown in Figure 3.1. In both graphs the root node is of type prime, which was quite likely. Since the depth of both MDT is quite low with 4 and 5 levels, most of the leaf nodes are directly attached to the root node in the MDT. One difference in structure - except the size - between the 1997 and the 2000 AS graph is the additional level of the AS graph 1997. In that graph there is a path “*PR - Pa - Se - Pa - x*”, where x stands for a single AS. The longest path in the 2000 AS graph is only “*PR - Pa - Se - x*”

Overall the MDT's structure of these graphs is not very diversified. Fortunately the root node is not of parallel type which would imply that the Internet is not connected.

The many parallel nodes result from the tree-like structure of the Internet where many customers (big companies, universities, governments and smaller ISPs) are attached to the same backbone ISP. These customers form together a parallel module. Obviously there are quite a few AS that use the same backbone provider and are connected to another customer too. That's the reason why there exist some series nodes. For another graphical example of an AS-like graph see Appendix E.

¹Autonomous Systems http://www.apnic.net/info/faq/as_faq.html

Data	1997	2000
Total vertices in MDT	3283	7058
- Leaf vertices (AS)	3015	6474
- Inner nodes (Modules)	268	584
Depth of FT	5	4
Series nodes	3	4
- Minimum children	2	2
- Average children	2	2
- Maximum children	2	2
Parallel nodes	264	579
- Minimum children	2	2
- Average children	6.36	6.07
- Maximum children	169	268
Prime nodes	1	1
- Children	1596	3534

Table 3.1: AS graph analysis

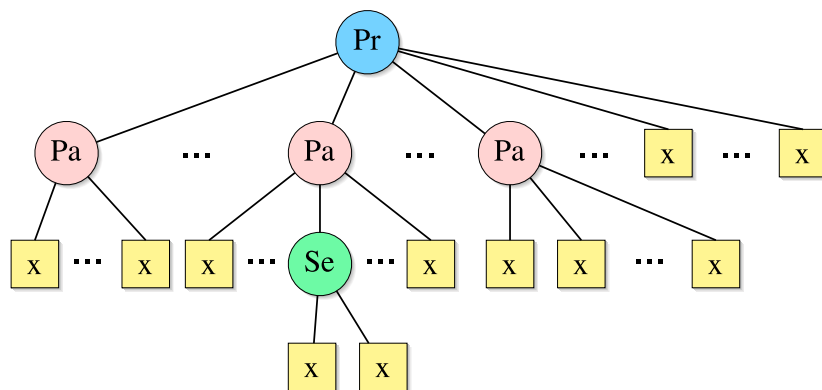


Figure 3.1: Structure of AS graph Jan. 2000. [x] corresponds to a single AS

Chapter 4

Conclusions and Extensions

With the program described in the preceding chapters a tool was created to compute the modular decomposition of an undirected graph fast and efficiently. It can be applied not only to small graphs of few nodes but also to large network graphs with thousands of nodes. The calculation time on a modern PC will stay in the order of some minutes. A highly inefficient brute-force program that simply enumerates all subsets of the vertices needed about 11 hours to compute the MD of a graph with 30 nodes while the implemented solution does the same in a few seconds. Although the implemented program does not have $\mathcal{O}(n + m)$ it still runs fast enough to compute the MD of large graphs.

An extended implementation could aim at better efficiency. There are two calculation steps that don't run in linear time (see Section 2.4):

1. Partition refining
2. Parenthesizing of MFPO

The data structures to implement the linear time partition refining (1) algorithm proposed in [HPV99] already exist since originally it was planned to implement this faster version after the simple but slow version worked. But time was too short in the end, thus it was decided to remain with this implementation.

The parenthesizing step (2) can be improved by a bucket sorting algorithm that first sorts the neighbors of two vertices x and x' according to their position in the MFPO. Then these two adjacency lists could be processed in parallel in order to find the first difference/cutter. This would then allow to find the first and last cutter of the pair (x, x') in less time than in the existing solution that checks every node (beginning at start/end of MFPO) adjacency relation to x and x' until a cutter is found.

It is possible to merge some of the scans to save some calculation time. E.g. scans 3 and 4 could be done in one run. And so probably can scans 5 and 6.

To better verify the results of this MD implementation, one could add a feature to reconstruct the original graph using its modular decomposition. This way it would be possible to compare the input graph G with the reconstructed G' . If they match and if you can assure that there are no PGs that contain any non-trivial module, the calculated MD most certainly is correct.

Another nice-to-have feature would be a tree drawing function that draws the FT in a better way. The existing solution starts placing the children of a node v just below v . If there already is a node the first child is placed at the first non occupied position on that level. This implies that the FT grows to the right while the root node is on the upper left corner. A better solution would try to place the root node in the upper middle and then distribute the children evenly without overlapping on the next level. To improve the readability of the FT the leaf nodes could be placed at the lowest level of the tree in the order of the MFPO.

Bugs

- There is a bug in LEDA concerning the structure *GraphWin* that is used to display graphs: When a *GraphWin* variable that has never been used (displayed in a window) goes out of scope, the destructor may do something harmful, which results in a *BusError* during run time. To fix this bug the header file “*graphwin.h*” was modified to make the *GraphWin* variable “*call_entry_counter*” public. This way the variable can be set to ‘0’ every time a *GraphWin* object is created but not shown in a window. That way the described error does not occur.

Chapter 5

Acknowledgments

This project was my second semester project at the Swiss Federal Institute of Technology Zurich (ETHZ). At this place I especially want to thank my tutor and supervisor Prof. Thomas Erlebach who supported me very well during this project. Whenever I had questions he could help me quickly what I appreciated very much and what one should not take for granted.

We also have to thank Michel Habib and Fabien de Montgolfier who could help us quickly to clarify a problem in Lemma 1 of paper [HMP03].

Last but not least I want to thank my co-workers in the “*Dungeon*” (ETZ C96) who made the time down here much more fun than this room looks like :)

Bibliography

- [CHM02] C. Capelle, M. Habib, F. de Montgolfier, *Graph Decompositions and Factorizing Permutations*, Discrete Mathematics and Theoretical Computer Science 5, pages 55-70, 2002.
- [CPS85] D. G. Corneil, Y. Perl, L.K. Stewart, *A linear recognition algorithm for cographs*, SIAM J. Comput. 3 (1985), pages 926-934.
- [DGM01] E. Dahlhaus, J. Gustedt, R. McConnell, Efficient and Practical Algorithms for Sequential Modular Decomposition, Journal of Algorithms 41 (2001), pages 360-387, 2000.
- [G67] T. Gallai, *Transitiv orientierbare Graphen*, Acta Math. Acad. Sci. Hungar. 18 (1967), pages 25-66.
- [G80] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980
- [HM87] M. Habib and R. H. Möhring, *On some complexity properties of N -free posets and posets of bounded decomposition diameter*, Discrete Math. 63 (1987), pages 157-182.
- [HMP03] M. Habib, F. de Montgolfier, C. Paul, *A simple linear-time modular decomposition algorithm for graphs, using order extending*, Submission to SODA 2004 (see also *LIRMM 03007 Technical Report 2003*), 2003.
- [HPV99] M. Habib, C. Paul, L. Viennot, *Partition refinement techniques: an interesting algorithmic toolkit*, International Journal of Foundations of Computer Science 10 (1999), no. 2, pages 189-196, 1999.
- [MS99] R. M. McConnell and J.P. Spinrad, *Modular decomposition and transitive orientation*, Discrete Math. 201, Nos. 1-3 (1999), pages 189-241
- [MR84] R. H. Möhring, F.J. Radermacher, *Substitution decomposition for discrete structures and connections with combinatorial optimization*, Ann. Discrete Math. 19 (1984), pages 257-356.
- [M85] R. H. Möhring, *Algorithmic aspects of comparability graphs and interval graphs*, Graphs and Order (I. Rival, Ed.), pages 41-101, Reidel, Boston, 1985.
- [MA85] R. H. Möhring, *Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and Boolean functions*, Ann. Oper. Res. 4 (1985), pages 195-225.

Appendix A

File Structure

The algorithm described in this document is programmed in C++ using a special library called LEDA by Algorithmic-Solutions. See Appendix B for details on compilation of shared/static library files.

The source code consists of four C++ Files. Datastructures.h is included in all the other .h files:

Filename	Function
ComputeModularDecomposition.c	Contains the <i>main()</i> routine and some debugging/auxiliary functions. Controls the program flow.
Datastructures.c / .h	Most of the used data structures are contained in this file. Although the implemented algorithm to compute the MFPO does not run in linear time the data structures are prepared to implement the linear time version of the algorithm. Includes all the needed LEDA libraries.
PartitionRefining.c / .h	Algorithm to calculate the MFPO. Uses the non-linear time algorithm described in [HPV99]
Decomposition.c / .h	Functions to implement the six FT scans and to show the GraphWin of the FT.
graphwin.h	Modified LEDA header file for the GraphWin class. See section “Bugs” in Chapter 4.

Appendix B

Compilation

The make file looks like that:

```
XLIB = -L/usr/openwin/lib
LEDADIR = $(LEDAROOT)
CC = g++ -O
CFLAGS = -Wall -ansi -pedantic -Dunix -g

CLIBS = DataStructures.c PartitionRefining.c Decomposition.c
OBJS   =      DataStructures.o PartitionRefining.o Decomposition.o \
ComputeModularDecomposition.o

cmd:    $(OBJS)
$(CC) $(CFLAGS) -o cmd $(OBJS) -L$(LEDADIR) $(XLIB) \
-lW -lP -lG -lL -lX11 -lm -lsocket -lnsl

DataStructures.o: DataStructures.c DataStructures.h DataStructures.c
PartitionRefining.o: PartitionRefining.c DataStructures.h DataStructures.c
Decomposition.o: Decomposition.c DataStructures.h DataStructures.c

.c.o:
$(CC) $(CFLAGS) -I$(LEDADIR)/incl -c $*.c

all:
$(CC) $(CFLAGS) -I$(LEDADIR)/incl -o cmd $(CLIBS) \
ComputeModularDecomposition.c -L$(LEDADIR) -L~/Source/ $(XLIB) \
-lW -lP -lG -lL -lX11 -lm -lsocket -lnsl
```

In the source code directory use the command "*make*" to compile and link the binary "*cmd*" (stands for "*C*ompute *M*odular *D*ecomposition"). This will compile only the files that were modified since the last compilation.

Use "*make all*" to compile and link all files together. With shared libraries this task should take less than a minute on a SunBlade 100 and the resulting binary file should be approximately 3 Mbytes in size.

Appendix C

Command line Options and Usage

To see all the command line options you can use, change to the directory where the binary “*cmd*” is stored.

Then run the program with “*./cmd*” or “*./cmd -h*” and you should see the following output:

```
"Compute Modular Decomposition of a Graph"
Version: 1, February 2004
Lukas Haemmerle <haemmluk@ee.ethz.ch>, D-ITET, ETH Zurich
Supervision: Prof. Thomas Erlebach <erlebach@tik.ee.ethz.ch>, TIK, ETH Zurich
```

Usage:

```
cmd [-h]
cmd In.graph Out.graph [Options]
```

Options are:

```
-a : Analyze and show statistics about calculated MD
-c : Check for plausibility
-d : Use deterministic calculation of MFPO
-f : Show FT node index together with index of InputGraph.gw nodes
-i : Show node index instead of node type
-m : Only compute Module Factorizing Partition Order (MFPO)
-s : Show modular decomposition tree
-t : Show used CPU time for calculation steps
-v : High verbosity
```

C.1 Input and Output Files

If you want to compute the MD you always have to specify an input file as first argument and an output file as second argument. The input file must be a valid LEDA graph file that then will be read by the program. The formatted MDT is saved in the file that you specify as second argument. All (if any) PGs are saved within the same directory as the output graph, named like the output graph but with “_PN#*Index number*#. #*Suffix*#”. E.g. an output graph named “OutGraph.gw” could produce a PG called “OutGraph_PN3.gw” where “PN” stands for prime node and 3 is the index number of this prime node in MDT that is saved in “OutGraph.gw”. The output file’s suffix typically should be “.gw” or “.graph”.

C.2 Options

- **-a:**
After the computation of the MD some additional statistics will be printed. This includes number of total nodes, leaf nodes, inner nodes, prime/series/parallel nodes and its minimum/average/maximum children.
- **-c :**
Checks the MD against some rules that it must fulfill, conditions that are necessary but not sufficient. These conditions are: No node must have more than one parent, there must not be children nodes that have the same type as their parents except for prime nodes and most important of all: It is validated that all inner nodes represent modules. Unfortunately it is non-trivial to check if these modules are strong. That's why this check is a necessary condition for a valid MD but not a sufficient one.
- **-d :**
During the calculation of the MFPO there is a function that randomly chooses a new center node. With this option enabled always the first node in a part is chosen as center. This may help to debug a problem but in general it is not necessary to use this option. Every valid MFPO should lead to a valid MD.
- **-f :**
If you enable the option -s to show the FT/MD the leaf nodes are by default labeled with the index number of the input graph. This option will also display the FT/MD index for the leaf nodes.
- **-i :**
Instead of displaying the type of the nodes as label you will see the index numbers of these nodes. You still can see what type the inner nodes are because of the color.
- **-m :**
Just compute the MFPO and print it. Useful for debugging together with the option -v.
- **-s :**
Graphically display the MDT when the calculation has finished. To view the PGs in the displayed window, first click on 'done' in the upper right corner and then select a prime node. A new window will open showing the PG of the node you selected.
If this option is used together with -v not only the MDT at the end is displayed but also every step of the six scans described above.
You will be asked to confirm this option when you try to use it on a graph that has more than 30 nodes. In general it will take some minutes to display a graph with some thousand nodes but it is possible. You possibly will see some graphic errors (produced by LEDA) though.
- **-t :**
For every important calculation step the used cpu-time is displayed.
- **-v**
Mainly for debugging. A lot of information is displayed for every calculation step. Useful for people who have read this documentation and know how the MD is calculated. One has to confirm this option when using it with graphs that consist of more than 30 nodes. It's not recommended to use this feature on large graphs with more than 1000 nodes since the time to output all the information will slow down calculation dramatically.

Appendix D

File Format

The program saves at least one file. If run with the following command `'cmd Ingraph.gw Outgraph.gw'` there will be created a file `'Outgraph.gw'` that contains the MDT. Depending on how many prime nodes there were in the MDT, additional PGs are saved. They will be named `'Outgraph_PN#'` where `#` stands for the index number of the prime node in MDT. The internal structure of all these output files is the same. Basically it is the structure used by the leda graph format with one extension. Every node holds additional information about its type. Node information is stored as string.

- A leaf node of the MDT contains the index number of the corresponding node in `'Ingraph.gw'`.
- An inner node of MDT will store its type information (`'Pa','Se','Pr'`) and then after a white space the index number of the corresponding node in the MDT.
- There is no edge information stored in MDT.

While the MDT graph is a directed graph where the edges start at a father node and end at a child node in the MDT, the PGs are bidirected graphs since internally all the computations are done on bidirected graphs.

The output file of the example graph discussed in the previous chapters will look like this:

```
LEDA.GRAPH
string
string
14
|{Parallel 0}|
|{Prime 1}|
|{1}|
|{0}|
|{2}|
|{4}|
|{3}|
|{Series 10}|
|{Parallel 11}|
|{6}|
|{7}|
|{8}|
|{5}|
# Edges and graphwin data
...
```

Appendix E

Other Examples

Since it is nearly impossible to present the full AS graph, a small example graph was constructed that is of similar form as these large graphs. In Figure E.1 a graph is drawn that shows an “Internet-like” structure.

As you can see in Figure E.2 the resulting MD has a comparable structure to that of the analyzed AS graphs of Chapter 3. There is a prime node as root node together with some examples of parallel, series and parallel-series nodes as they exist (with exception of the series node directly connected to the prime node) in the big AS graph. In Figure E.3 the corresponding PG is shown.

Nodes 16 and 15 together build a series module that is connected to a parallel module, as it is the case a few times in the real AS graph. Nodes 17 and 18 in contrast build a series module that is directly connected to the prime node, which does not occur in the real AS graphs.

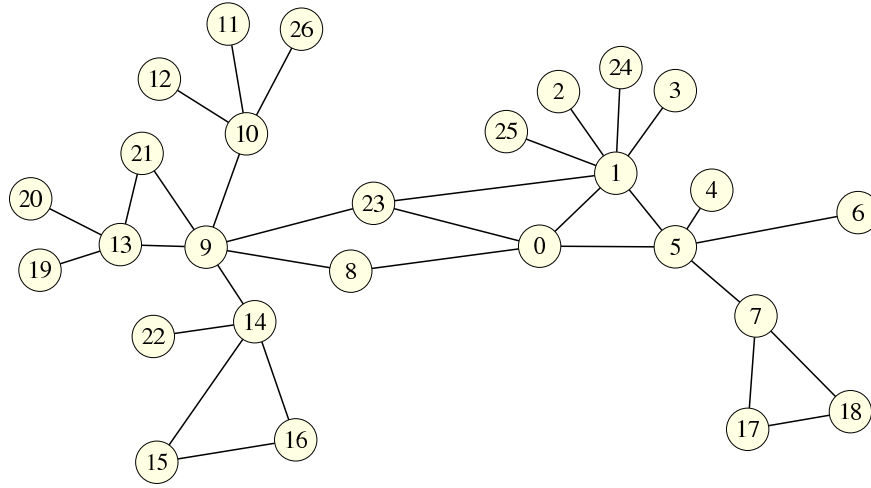


Figure E.1: Example AS-like graph with Internet characteristic

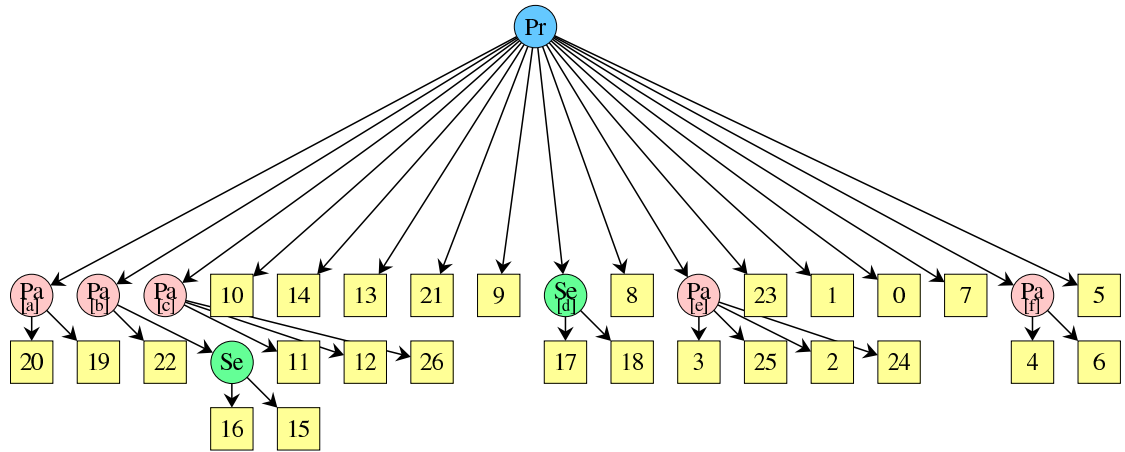


Figure E.2: MDT of example graph of Figure E.1

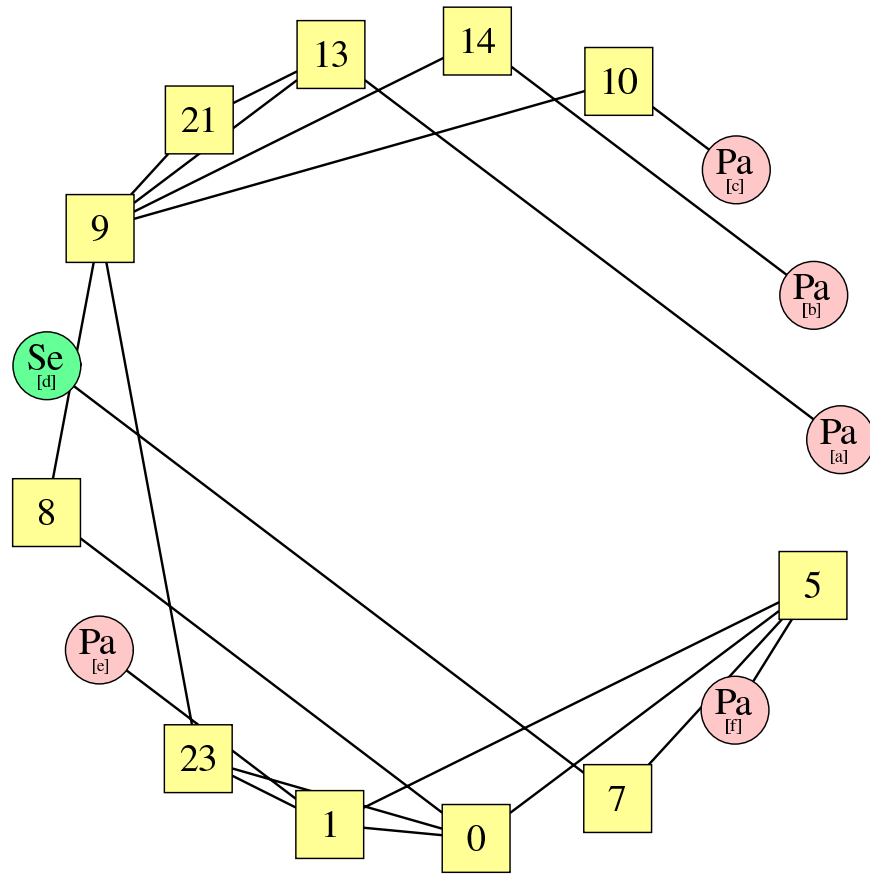


Figure E.3: Prime graph of MD in Figure E.2