19. Semi-structured databases

The use of the internet and the development of the theory of databases mutually affect each other. The contents of web sites are usually stored by databases, while the web sites and the references between them can also be considered a database which has no fixed schema in the usual sense. The contents of the sites and the references between sites are described by the sites themselves, therefore we can only speak of semi-structured data, which can be best characterized by directed labeled graphs. In case of semi-structured data, recursive methods are used more often for giving data structures and queries than in case of classical relational databases. Different problems of databases, e.g. restrictions, dependencies, queries, distributed storage, authorities, uncertainty handling, must all be generalized according to this. Semi-structuredness also raises new questions. Since queries not always form a closed system like they do in case of classical databases, that is, the applicability of queries one after another depends on the type of the result obtained, therefore the problem of checking types becomes more emphasized.

The theoretical establishment of relational databases is closely related to finite modelling theory, while in case of semi-structured databases, automata, especially tree automata are most important.

19.1. Semi-structured data and XML

By semi-structured data we mean a directed rooted labeled graph. The root is a special node of the graph with no entering edges. The nodes of the graph are objects distinguished from each other using labels. The objects are either atomic or complex. Complex objects are connected to one or more objects by directed edges. Values are assigned to atomic objects. Two different models are used: either the vertices or the edges are labeled. The latter one is more general, since an edge-labeled graph can be assigned to all vertex-labeled graphs in such a way that the label assigned to the edge is the label assigned to its endpoint. This way we obtain a directed labeled graph for which all inward directed edges from a vertex have the same label. Using this transformation, all concepts, definitions and statements concerning edge-labeled graphs can be rewritten for vertex-labeled graphs.

The following method is used to gain a vertex-labeled graph from an edge-labeled graph. If edge (u, v) has label c, then remove this edge, and introduce a new vertex w with

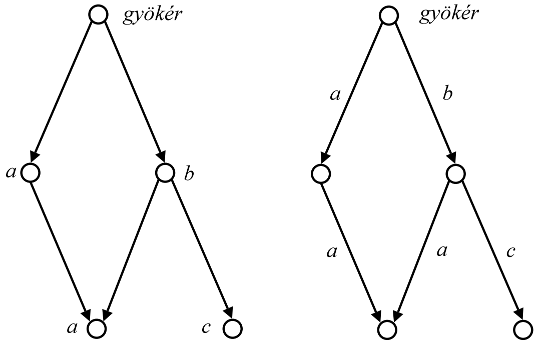

Figure 19.1. Edge-labeled graph assigned to a vertex-labeled graph.

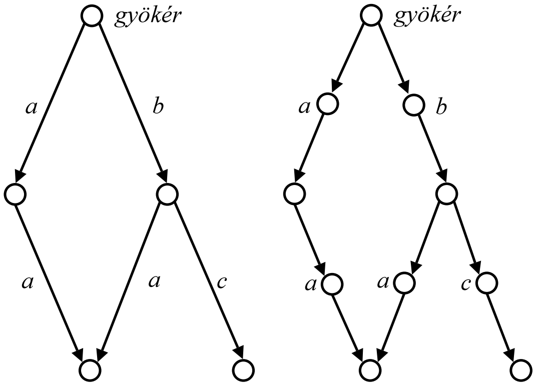

Figure 19.2. An edge-labeled graph and the corresponding vertex-labeled graph.

label c, then add edges (u, w) and (w, v). This way we can obtain a vertex-labeled graph of $m + n$ nodes and $2m$ edges from an edge-labeled graph of n vertices and m edges. Therefore all algorithms and cost bounds concerning vertex-labeled graphs can be rewritten for edge-labeled graphs.

Since most books used in practice use vertex-labeled graphs, we will also use vertex-labeled graphs in this chapter.

The ***XML*** (e**X**tensible **M**arkup **L**anguage) language was originally designed to describe embedded ordered labeled elements, therefore it can be used to represent trees of semi-structured data. In a wider sense of the XML language, references between the elements can also be given, thus arbitrary semi-structured data can be described using the XML language.

The medusa.inf.elte.hu/forbidden site written in XML language is as follows. We can obtain the vertex-labeled graph of Figure 19.3 naturally from the structural characteristics of the code.

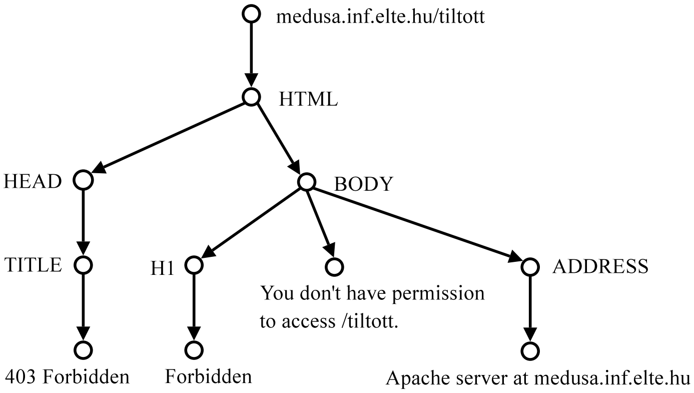
medusa.inf.elte.hu/tiltott
HTML
HEAD
BODY
TITLE
H1
You don't have permission to access /tiltott.
ADDRESS
403 Forbidden
Forbidden
Apache server at medusa.inf.elte.hu

Figure 19.3. The graph corresponding to the XML file „forbidden".

```
<HTML>
        <HEAD>
                <TITLE>403 Forbidden</TITLE>
        </HEAD>
        <BODY>
                <H1>Forbidden</H1>
                You don't have permission to access /forbidden.
                <ADDRESS>Apache Server at medusa.inf.elte.hu </ADDRESS>
        </BODY>
</HTML>
```

Exercises

19.1-1 Give a vertex-labeled graph that represents the structure and formatting of this chapter.

19.1-2 How many different directed vertex-labeled graphs exist with n vertices, m edges and k possible labels? How many of these graphs are not isomorphic? What values can be obtained for $n = 5$, $m = 7$ and $k = 2$?

19.1-3 Consider a tree in which all children of a given node are labeled with different numbers. Prove that the nodes can be labeled with pairs (a_v, b_v), where a_v and b_v are natural numbers, in such a way that

a. $a_v < b_v$ for every node v.

b. If u is a descendant of v, then $a_v < a_u < b_u < b_v$.

c. If u and v are siblings and $number(u) < number(v)$, then $b_u < a_v$.

19.2. Schemas and simulations

In case of relational databases, schemas play an important role in coding and querying data, query optimization and storing methods that increase efficiency. When working with semi-

R	A	B	C
	0	0	0
	1	2	2

Q	C	D
	0	0
	0	2
	1	1

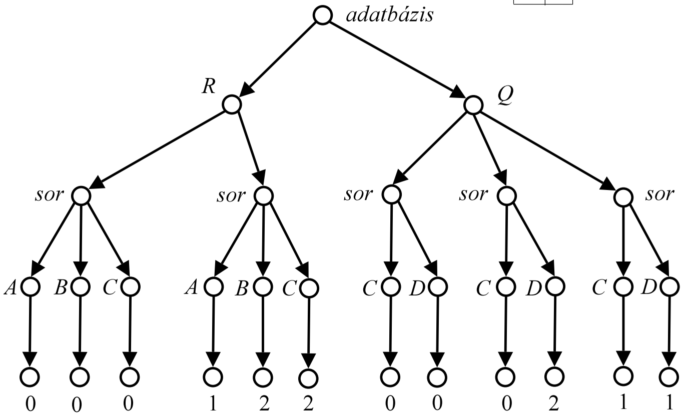

Figure 19.4. A relational database in the semi-structured model.

structured databases, the schema must be obtained from the graph. The schema restricts the possible label strings belonging to the paths of the graph.

Figure 19.4 shows the relational schemas with relations $R(A, B, C)$ and $Q(C, D)$, respectively, and the corresponding semi-structured description. The labels of the leaves of the tree are the components of the tuples. The directed paths leading from the root to the values contain the label strings *database.R.tuple.A, database.R.tuple.B, database.R.tuple.C, database.Q.tuple.C, database.Q.tuple.D*. This can be considered the schema of the semi-structured database. Note that the schema is also a graph, as it can be seen on Figure 19.5. The disjoint union of the two graphs is also a graph, on which a simulation mapping can be defined as follows. This way we create a connection between the original graph and the graph corresponding to the schema.

Definition 19.1 *Let $G = (V, E, A, label())$ be a vertex-labeled directed graph, where V denotes the set of nodes, E the set of edges, A the set of labels, and $label(v)$ is the label belonging to node v. Denote by $E^{-1}(v) = \{u \mid (u, v) \in E\}$ the set of the start nodes of the edges leading to node v. A binary relation s ($s \subseteq V \times V$) is a* ***simulation****, if, for $s(u, v)$,*
i) $label(u) = label(v)$ and
ii) for all $u' \in E^{-1}(u)$ there exists a $v' \in E^{-1}(v)$ such that $s(u', v')$
Node v simulates node u*, if there exists a simulation s such that $s(u, v)$.* ***Node u and node v are similar****, $u \approx v$, if u simulates v and v simulates u.*

It is easy to see that the empty relation is a simulation, that the union of simulations is a simulation, that there always exists a maximal simulation and that similarity is an equivalence relation. We can write E instead of E^{-1} in the above definition, since that only means that the direction of the edges of the graph is reversed.

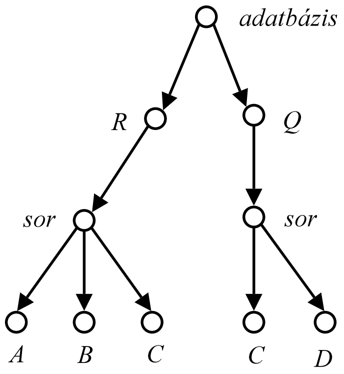

Figure 19.5. The schema of the semi-structured database given in Figure 19.4.

We say that graph D simulates graph S if there exists a mapping $f : V_S \mapsto V_D$ such that the relation $(v, f(v))$ is a simulation on the set $V_S \times V_D$.

Two different schemas are used, a lower bound and an upper bound. If the data graph D simulates the schema graph S, then ***S* is a lower bound of *D***. Note that this means that all label strings belonging to the directed paths in S appear in D at some directed path. If S simulates D, then ***S* is an upper bound of *D***. In this case, the label strings of D also appear in S.

In case of semi-structured databases, the schemas which are greatest lower bounds or lowest upper bounds play an important role.

A map between graphs S and D that preserves edges is called a morphism. Note that f is a morphism if and only if D simulates S. To determine whether a morphism from D to S exists is an NP-complete problem. We will see below, however, that the calculation of a maximal simulation is a PTIME problem.

Denote by $sim(v)$ the nodes that simulate v. The calculation of the maximal simulation is equivalent to the determination of all sets $sim(v)$ for $v \in V$. First, our naive calculation will be based on the definition.

```
Naive-Maximal-Simulation($G$)

for all $v \in V$
    do $sim(v) \leftarrow \{u \in V \mid label(u) = label(v)\}$
while $\exists\, u, v, w \in V :\ v \in E^{-1}(u)\ \wedge\ w \in sim(u)\ \wedge\ E^{-1}(w) \cap sim(v) = \emptyset$
      do $sim(u) \leftarrow sim(u) \setminus \{w\}$
return $\{sim(u) \mid u \in V\}$
```

Proposition 19.2 *The algorithm* Naive-Maximal-Simulation *computes the maximal simulation in $O(m^2 n^3)$ time if $m \geq n$.*

Proof. Let us start with the elements of $sim(u)$. If an element w of $sim(u)$ does not simulate u by definition according to edge (v, u), then we remove w from set $sim(u)$. In this case, we say that we improved set $sim(u)$ according to edge (v, u). If set $sim(u)$ cannot be improved according to any of the edges, then all elements of *sim*(u) simulate u. To complete the proof, notice that the **while** cycle consists of at most n^2 iterations. ■

The efficiency of the algorithm can be improved using special data structures. First, introduce a set *sim-candidate(u)*, which contains *sim*(u), and of the elements of whom we want to find out whether they simulate u.

```
Improved-Maximal-Simulation($G$)
for all $v \in V$
    do $sim\text{-}candidate(u) \leftarrow V$
       if $E^{-1}(v) = \emptyset$
          then $sim(v) \leftarrow \{u \in V \mid label(u) = label(v)\}$
          else $sim(v) \leftarrow \{u \in V \mid label(u) = label(v) \ \wedge \ E^{-1}(u) \neq \emptyset\}$
while $\exists\, v \in V \ : \ sim(v) \neq sim\text{-}candidate(v)$
      do $removal\text{-}candidate \leftarrow E(sim\text{-}candidate(v)) \setminus E(sim(v))$
         for all $u \in E(v)$
             do $sim(u) \leftarrow sim(u) \setminus removal\text{-}candidate$
         $sim\text{-}candidate(v) \leftarrow sim(v)$
return $\{sim(u) \mid u \in V\}$
```

The **while** cycle of the improved algorithm possesses the following invariant characteristics.

I_1: $\forall\, v \in V : \ sim(v) \subseteq sim\text{-}candidate(v)$.
I_2: $\forall\, u, v, w \in V : (v \in E^{-1}(u) \ \wedge \ w \in sim(u)) \ \Rightarrow \ (E^{-1}(w) \cap sim\text{-}candidate(v) \neq \emptyset)$.

When improving the set *sim*(u) according to edge (v, u), we check whether an element $w \in sim(u)$ has parents in *sim*(v). It is sufficient to check that for the elements of *sim-candidate*(v) instead of *sim*(v) because of I_2. Once an element $w' \in sim\text{-}candidate(v) \setminus sim(v)$ was chosen, it is removed from set *sim-candidate*(v).

We can further improve the algorithm if we do not compute the set *removal-candidate* in the iterations of the **while** cycle but refresh the set dynamically.

```
EFFICIENT-MAXIMAL-SIMULATION($G$)
for all $v \in V$
    do $sim\text{-}candidate(v) \leftarrow V$
       if $E^{-1}(v) = \emptyset$
          then $sim(v) \leftarrow \{u \in V \mid label(u) = label(v)\}$
          else $sim(v) \leftarrow \{u \in V \mid label(u) = label(v) \ \wedge \ E^{-1}(u) \neq \emptyset\}$
    $removal\text{-}candidate(v) \leftarrow E(V) \setminus E(sim(v))$
 while $\exists \, v \in V \ : \ removal\text{-}candidate(v) \neq \emptyset$
       do for all $u \in E(v)$
            do for all $w \in removal\text{-}candidate(v)$
                 do if $w \in sim(u)$
                       then $sim(u) \leftarrow sim(u) \setminus \{w\}$
                            for all $w'' \in E(w)$
                                do if $E^{-1}(w'') \cap sim(u) = \emptyset$
                                      then $removal\text{-}candidate(u) \leftarrow removal\text{-}candidate(u) \cup \{w''\}$
            $sim\text{-}candidate(v) \leftarrow sim(v)$
            $removal\text{-}candidate(v) \leftarrow \emptyset$
return $\{sim(u) \mid u \in V\}$
```

The above algorithm possesses the following invariant characteristic with respect to the **while** cycle.

I_3: $\forall \, v \in V$: $removal\text{-}candidate(v) = E(sim\text{-}candidate(v)) \setminus E(sim(v))$.

Use an $n \times n$ array as a *counter* for the realization of the algorithm. Let the value $counter[w'', u]$ be the nonnegative integer $|E^{-1}(w'') \cap sim(u)|$. The initial values of the counter are set in $O(mn)$ time. When element w is removed from set $sim(u)$, the values $counter[w'', u]$ must be decreased for all children w'' of w. By this we ensure that the innermost **if** condition can be checked in constant time. At the beginning of the algorithm, the initial values of the sets $sim(v)$ are set in $O(n^2)$ time if $m \geq n$. The setting of sets $removal\text{-}candidate(v)$ takes altogether $O(mn)$ time. For arbitrary nodes v and w, if $w \in removal\text{-}candidate(v)$ is true in the i-th iteration of the **while** cycle, then it will be false in the j-th iteration for $j > i$. Since $w \in removal\text{-}candidate(v)$ implies $w \notin E(sim(v))$, the value of $sim\text{-}candidate(v)$ in the j-th iteration is a subset of the value of $sim(v)$ in the i-th iteration, and we know that invariant I_3 holds. Therefore $w \in sim(u)$ can be checked in $\sum_v \sum_w |E(v)| = O(mn)$ time. $w \in sim(u)$ is true at most once for all nodes w and u, since once the condition holds, we remove w from set $sim(u)$. This implies that the computation of the outer **if** condition of the **while** cycle takes $\sum_v \sum_w (1 + |E(v)|) = O(mn)$ time.

Thus we have proved the following proposition.

Proposition 19.3 *The algorithm* EFFECTIVE-MAXIMAL-SIMULATION *computes the maximal simulation in $O(mn)$ time if $m \geq n$.*

If the inverse of a simulation is also a simulation, then it is called a bisimulation. The empty relation is a bisimulation, and there always exist a maximal bisimulation. The maximal bisimulation can be computed more efficiently than the simulation. The maximal bisimulation can be computed in $O(m \lg n)$ time using the PT algorithm. In case of edge-labeled graphs, the cost is $O(m\lg(m+n))$.

We will see that bisimulations play an important role in indexing semi-structured databases, since the quotient graph of a graph with respect to a bisimulation contains the same label strings as the original graph. Note that in practice, instead of simulations, the so-called ***DTD*** descriptions are also used as schemas. DTD consists of data type definitions formulated in regular language.

Exercises

19.2-1 Show that simulation does not imply bisimulation.

19.2-2 Define the operation *turn-tree* for a directed, not necessarily acyclic, vertex-labeled graph G the following way. The result of the operation is a not necessarily finite graph G', the vertices of which are the directed paths of G starting from the root, and the labels of the paths are the corresponding label strings. Connect node p_1 with node p_2 by an edge if p_1 can be obtained from p_2 by deletion of its endpoint. Prove that G and *turn-tree*(G) are similar with respect to the bisimulation.

19.3. Queries and indexes

The information stored in semi-structured databases can be retrieved using queries. For this, we have to fix the form of the questions, so we give a ***query language***, and then define the meaning of questions, that is, the ***query evaluation*** with respect to a semi-structured database. For efficient evaluation we usually use indexes. The main idea of ***indexing*** is that we reduce the data stored in the database according to some similarity principle, that is, we create an index that reflects the structure of the original data. The original query is executed in the index, then using the result we find the data corresponding to the index values in the original database. The size of the index is usually much smaller than that of the original database, therefore queries can be executed faster. Note that the inverted list type index used in case of classical databases can be integrated with the schema type indexes introduced below. This is especially advantageous when searching XML documents using keywords.

First we will get acquainted with the query language consisting of regular expressions and the index types used with it.

Definition 19.4 *Given a directed vertex-labeled graph $G = (V, E, root, \Sigma, label, id, value)$, where V denotes the set of vertices, $E \subseteq V \times V$ the set of edges and Σ the set of labels. Σ contains two special labels, ROOT and VALUE. The label of vertex v is $label(v)$, and the identifier of vertex v is $id(v)$. The root is a node with label ROOT, and from which all nodes can be reached via directed paths. If v is a leaf, that is, if it has no outgoing edges, then its label is VALUE, and $value(v)$ is the value corresponding to leaf v. Under the term path we always mean a directed path, that is, a sequence of nodes $n_0, \ldots, n_p$ such that there is an edge from n_i to n_{i+1} if $0 \le i \le p-1$. A sequence of labels $l_0, \ldots, l_p$ is called a* ***label sequence*** *or* ***simple expression****.* ***Path $n_0, \ldots, n_p$ fits to the label sequence $l_0, \ldots, l_p$*** *if $label(n_i) = l_i$ for all $0 \le i \le p$.*

We define regular expressions recursively.

Definition 19.5 *Let $R ::= \varepsilon \mid \Sigma \mid \_ \mid R.R \mid R|R \mid (R) \mid R? \mid R^*$, where* ***R*** *is a* ***regular expression,***

*and ε is the empty expression, _ denotes an arbitrary label, . denotes succession, | is the logical OR operation, ? is the optional choice, and * means finite repetition. Denote by* ***L(R)*** *the regular language consisting of the label sequences determined by R.* ***Node n fits to a label sequence*** *if there exists a path from the root to node n such that fits to the label sequence.* ***Node*** *n* ***fits to the regular expression*** *R if there exists a label sequence in the language $L(R)$, to which node n fits. The* ***result of the query on graph G*** *determined by the regular expression R is the set $R(G)$ of nodes that fit to expression R.*

Since we are always looking for paths starting from the root when evaluating regular expressions, the first element of the label sequence is always ROOT, which can therefore be omitted.

Note that the set of languages $L(R)$ corresponding to regular expressions is closed under intersection, and the problem whether $L(R) = \emptyset$ is decidable.

The result of the queries can be computed using the nondeterministic automaton A_R corresponding to the regular expression R. The algorithm given recursively is as follows.

```
NAIVE-EVALUATION($G, A_R$)
Visited ← $\emptyset$                          ▷ If we were in node $u$ in state $s$,
                                              then $(u, s)$ was put in set Visited.
TRAVERSE (root($G$), starting-state($A_R$))
```

```
TRAVERSE($u, s$)
if $(u, s) \in$ Visited
  then return result[$u, s$]
Visited ← Visited $\cup \{(u, s)\}$
result[$u, s$] ← $\emptyset$
for all $s \xrightarrow{\epsilon} s'$          ▷ If we get to state $s'$ from state $s$ by reading sign $\epsilon$.
   do if s’ ∈ final-state($A_R$)
        then result[$u, s$] ← $\{u\} \cup$ result[$u, s$]
      result[$u, s$] ← result[$u, s$] $\cup$ TRAVERSE($u, s'$)
for all $s \xrightarrow{label(u)} s'$        ▷ If we get to state $s'$ from state $s$ by reading sign label($u$).
   do if $s' \in$ final-state($A_R$)
        then result[$u, s$] ← $\{u\} \cup$ result[$u, s$]
   for all $v$, where $(u, v) \in E(G)$          ▷ Continue the traversal for the
                                              children of node $u$ recursively.
      do result[$u, s$] ← result[$u, s$] $\cup$ TRAVERSE($v, s'$)
return result[$u, s$]
```

Proposition 19.6 *Given a regular query R and a graph G, the calculation cost of $R(G)$ is a polynomial of the number of edges of G and the number of different states of the finite nondeterministic automaton corresponding to R.*

Proof. The sketch of the proof is the following. Let A_R be the finite nondeterministic automaton corresponding to R. Denote by $|A_R|$ the number of states of A_R. Consider the breadth-first traversal corresponding to the algorithm TRAVERSE of graph G with m edges, starting from the root. During the traversal we get to a new state of the automaton according to the label of the node, and we store the state reached at the node for each node. If the final state of the automaton is acceptance, then the node is a result. During the traversal, we sometimes have to step back on an edge to ensure we continue to places we have not seen yet. It can be proved that during a traversal every edge is used at most once in every state, so this is the number of steps performed by that automaton. This means $O(|A_R|m)$ steps altogether, which completes the proof. ■

Two nodes of graph G are ***indistinguishable with regular expressions*** if there is no regular R for which one of the nodes is among the results and the other node is not. Of course, if two nodes cannot be distinguished, then their labels are the same. Let us categorize the nodes in such a way that nodes with the same label are in the same class. This way we produce a partition P of the set of nodes, which is called the ***basic partition***. It can also be seen easily that if two nodes are indistinguishable, then it is also true for the parents. This implies that the set of label sequences corresponding to paths from the root to the indistinguishable nodes is the same. Let $L(n) = \{l_0, \ldots, l_p \mid n$ fits to the label sequence $l_0, \ldots, l_p\}$ for all nodes n. Nodes n_1 and n_2 are indistinguishable if and only if $L(n_1) = L(n_2)$. If the nodes are assigned to classes in such a way that the nodes having the same value $L(n)$ are arranged to the same class, then we get a refinement P' of partition P. For this new partition, if a node n is among the results of a regular query R, then all nodes from the equivalence class of n are also among the results of the query.

Definition 19.7 *Given a graph $G = (V, E, root, \Sigma, label, id, value)$ and a partition P of V that is a refinement of the basic partition, that is, for which the nodes belonging to the same equivalence class have the same label. Then the graph $I(G) = (P, E', root', \Sigma, label', id', value')$ is called an* ***index****. The nodes of the index graph are the equivalence classes of partition P, and $(I, J) \in E'$ if and only if there exist $i \in I$ and $j \in J$ such that $(i, j) \in E$. If $I \in P$, then $id'(I)$ is the identifier of index node I, $label'(I) = label(n)$, where $n \in I$, and $root'$ is the equivalence class of partition P that contains the root of G. If $label(I) = VALUE$, then $label'(I) = \{value(n) \mid n \in I\}$.*

Given a partition P of set V, denote by ***class(n)*** the equivalence class of P that contains n for $n \in V$. In case of indexes, the notation $I(n)$ can also be used instead of $class(n)$.

Note that basically the indexes can be identified with the different partitions of the nodes, so partitions can also be called indexes without causing confusion. Those indexes will be good that are of small size and for which the result of queries is the same on the graph and on the index. Indexes are usually given by an equivalence relation on the nodes, and the partition corresponding to the index consists of the equivalence classes.

Definition 19.8 *Let P be the partition for which $n, m \in I$ for a class I if and only if $L(n) = L(m)$. Then the index $I(G)$ corresponding to P is called a* ***naive index****.*

In case of naive indexes, the same language $L(n)$ is assigned to all elements n of class I in partition P, which will be denoted by $L(I)$.

Proposition 19.9 *Let I be a node of the naive index and R a regular expression. Then $I \cap R(G) = \emptyset$ or $I \subseteq R(G)$.*

Proof. Let $n \in I \cap R(G)$ and $m \in I$. Then there exists a label sequence $l_0, \ldots, l_p$ in $L(R)$ to which n fits, that is, $l_0, \ldots, l_p \in L(n)$. Since $L(n) = L(m)$, m also fits to this label sequence, so $m \in I \cap R(G)$. ■

```
NAIVE-INDEX-EVALUATION($G, R$)
let $I_G$ be the naive index of $G$
$Q \leftarrow \emptyset$
for all $I \in$ NAIVE-EVALUATION($I_G, A_R$)
    do $Q \leftarrow Q \cup I$
return $Q$
```

Proposition 19.10 *Set Q produced by the algorithm* NAIVE-INDEX-EVALUATION *is equal to $R(G)$.*

Proof. Because of the previous proposition either all elements of a class I are among the results of a query or none of them. ■

Using naive indexes we can evaluate queries, but, according to the following proposition, not efficiently enough. The proposition was proved by Stockmeyer and Meyer in 1973.

Proposition 19.11 *The creation of the naive index I_G needed in the algorithm* NAIVE-INDEX-EVALUATION *is PSPACE-complete.*

The other problem with using naive indexes is that the sets $L(I)$ are not necessary disjoint for different I, which might cause redundancy in storing.

Because of the above we will try to find a refinement of the partition corresponding to the naive index, which can be created efficiently and can still be used to produce $R(G)$.

Definition 19.12 *Index $I(G)$ is **safe** if for any $n \in V$ and label sequence $l_0, \ldots, l_p$ such that n fits to the label sequence $l_0, \ldots, l_p$ in graph G, class(n) fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$. Index $I(G)$ is **exact** if for any class I of the index and label sequence $l_0, \ldots, l_p$ such that I fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$, arbitrary node $n \in I$ fits to the label sequence $l_0, \ldots, l_p$ in graph G.*

Safety means that the nodes belonging to the result we obtain by evaluation using the index contain the result of the regular query, that is, $R(G) \subseteq R(I(G))$, while exactness means that the evaluation using the index does not provide false results, that is, $R(I(G)) \subseteq R(G)$. Using the definitions of exactness and of the edges of the index the following proposition follows.

Proposition 19.13 *1. Every index is safe.*
2. The naive index is safe and exact.

If I is a set of nodes of G, then the language $L(I)$, to the label strings of which the elements of I fit, was defined using graph G. If we wish to indicate this, we use the notation $L(I, G)$. However, $L(I)$ can also be defined using graph $I(G)$, in which I is a node. In this case, we can use the notation $L(I, I(G))$ instead of $L(I)$, which denotes all label sequences to which node I fits in graph $I(G)$. $L(I, G) = L(I, I(G))$ for safe and exact indexes, so in this case we can write $L(I)$ for simplicity. Then $L(I)$ can be computed using $I(G)$, since the size of $I(G)$ is usually smaller than that of G.

Arbitrary index graph can be queried using the algorithm Naive-Evaluation. After that join the index nodes obtained. If we use an exact index, then the result will be the same as the result we would have obtained by querying the original graph.

```
Index-Evaluation($G, I(G), A_R$)
let $I(G)$ be the index of $G$
$Q \leftarrow \emptyset$
for all $I \in$ Naive-Evaluation($I(G), A_R$)
    do $Q \leftarrow Q \cup I$
return $Q$
```

First, we will define a safe and exact index that can be created efficiently, and is based on the similarity of nodes. We obtain the 1-index this way. Its size can be decreased if we only require similarity locally. The A(k)-index obtained this way lacks exactness, therefore using the algorithm Index-Evaluation we can get results that do not belong to the result of the regular query R, so we have to test our results to ensure exactness.

Definition 19.14 *Let $\approx$ be an equivalence relation on set V such that, for $u \approx v$,*
i) $label(u) = label(v)$,
ii) if there is an edge from node u' to node u, then there exists a node v' for which there is an edge from node v' to node v and $u' \approx v'$.
iii) if there is an edge from node v' to node v, then there exists a node u' for which there is an edge from node u' to node u and $u' \approx v'$.
The above equivalence relation is called a ***bisimulation****. Nodes u and v of a graph are* ***bisimilar*** *if and only if there exists a bisimulation $\approx$ such that $u \approx v$.*

Definition 19.15 *Let P be the partition consisting of the equivalence classes of a bisimulation. The index defined using partition P is called* ***1-index****.*

Proposition 19.16 *The 1-index is a refinement of the naive index. If the labels of the ingoing edges of the nodes in graph G are different, that is, $label(x) \neq label(x')$ for $x \neq x'$ and $(x, y), (x', y) \in E$, then $L(u) = L(v)$ if and only if u and v are bisimilar.*

Proof. $label(u) = label(v)$ if $u \approx v$. Let node u fit to the label sequence $l_0, \dots, l_p$, and let u' be the node corresponding to label l_{p-1}. Then there exists a v' such that $u' \approx v'$ and $(u', u), (v', v) \in E$. u' fits to the label sequence $l_0, \dots, l_{p-1}$, so, by induction, v' also fits to the label sequence $l_0, \dots, l_{p-1}$, therefore v fits to the label sequence $l_0, \dots, l_p$. So, if two nodes are in the same class according to the 1-index, then they are in the same class according to the naive index as well.

To prove the second statement of the proposition, it is enough to show that the naive index corresponds to a bisimulation. Let u and v be in the same class according to the naive index. Then $label(u) = label(v)$. If $(u', u) \in E$, then there exists a label sequence $l_0, \dots, l_p$ such that the last two nodes corresponding to the labels are u' and u. Since we assumed that the labels of the parents are different, $L(u) = L' \cup L''$, where L' and L'' are disjoint, and $L' = \{l_0, \dots, l_p \mid u'$ fits to the sequence $l_0, \dots, l_{p-1}$, and $l_p = label(u)\}$, while $L'' = L(u) \setminus L'$. Since $L(u) = L(v)$, there exists a v' such that $(v', v) \in E$ and $label(u') = label(v')$. $L' =$

$\{l_0, \dots, l_p \mid v'$ fits to the sequence $l_0, \dots, l_{p-1}$, and $l_p = label(v)\}$ because of the different labels of the parents, so $L(u') = L(v')$, and $u' \approx v'$ by induction, therefore $u \approx v$. ■

Proposition 19.17 *The 1-index is safe and exact.*

Proof. If x_p fits to the label sequence $l_0, \dots, l_p$ in graph G because of nodes $x_0, \dots, x_p$, then, by the definition of the index graph, there exists an edge from $class(x_i)$ to $class(x_{i+1})$, $0 \le i \le p-1$, that is, $class(x_p)$ fits to the label sequence $l_0, \dots, l_p$ in graph $I(G)$. To prove exactness, assume that I_p fits to the label sequence $l_0, \dots, l_p$ in graph $I(G)$ because of $I_0, \dots, I_p$. Then there are $u' \in I_{p-1}$, $u \in I_p$ such that $u' \approx v'$ and $(v', v) \in E$, that is, $v' \in I_{p-1}$. We can see by induction that v' fits to the label sequence $l_0, \dots, l_{p-1}$ because of nodes $x_0, \dots, x_{p-2}, v'$, but then v fits to the label sequence $l_0, \dots, l_p$ because of nodes $x_0, \dots, x_{p-2}, v', v$ in graph G. ■

If we consider the bisimulation in case of which all nodes are assigned to different partitions, then the graph $I(G)$ corresponding to this 1-index is the same as graph G. Therefore the size of $I(G)$ is at most the size of G, and we also have to store the elements of I for the nodes I of $I(G)$, which means we have to store all nodes of G. For faster evaluation of queries we need to find the smallest 1-index, that is, the coarsest 1-index. It can be checked that x and y are in the same class according to the coarsest 1-index if and only if x and y are bisimilar.

```
1-Index-Evaluation($G, R$)
let $I_1$ be the coarsest 1-index of $G$
return Index-Evaluation($G, I_1, A_R$)
```

In the first step of the algorithm, the coarsest 1-index has to be given. This can be reduced to finding the coarsest stable partition, what we will discuss in the next section of this chapter. Thus using the efficient version of the PT-algorithm, the coarsest 1-index can be found with computation cost $O(m \lg n)$ and space requirement $O(m+n)$, where n and m denote the number of nodes and edges of graph G, respectively.

Since graph I_1 is safe and exact, it is sufficient to evaluate the query in graph I_1, that is, to find the index nodes that fit to the regular expression R. Using Proposition 19.6, the cost of this is a polynomial of the size of graph I_1.

The size of I_1 can be estimated using the following parameters. Let p be the number of different labels in graph G, and k ***the diameter of graph*** G, that is, the length of the longest directed path. (No node can appear twice in the directed path.) If the graph is a tree, then the diameter is the depth of the tree. We often create websites that form a tree of depth d, then we add a navigation bar consisting of q elements to each page, that is, we connect each node of the graph to q chosen pages. It can be proved that in this case the diameter of the graph is at most $d + q(d-1)$. In practice, d and q are usually very small compared to the size of the graph. The proof of the following proposition can be found in the paper of Milo and Suciu.

Proposition 19.18 *Let the number of different labels in graph G be at most p, and let the diameter of G be less than k. Then the size of the 1-index I_1 defined by an arbitrary bisimulation can be bounded from above with a bound that only depends on k and p but does not depend on the size of G.*

Exercises

19.3-1 Show that the index corresponding to the maximal simulation is between the 1-index and the naive index with respect to refinement. Give an example that shows that both inclusions are proper.

19.3-2 Denote by $I_s(G)$ the index corresponding to the maximal simulation. Does $I_s(I_s(G)) = I_s(G)$ hold?

19.3-3 Represent graph G and the state transition graph of the automaton corresponding to the regular expression R with relational databases. Give an algorithm in a relational query language, for example in PL/SQL, that computes $R(G)$.

19.4. Stable partitions and the PT-algorithm

Most index structures used for efficient evaluation of queries of semi-structured databases are based on a partition of the nodes of a graph. The problem of creating indexes can often be reduced to finding the coarsest stable partition.

Definition 19.19 *Let E be a binary relation on the finite set V, that is, $E \subseteq V \times V$. Then V is the set of* ***nodes****, and E is the set of* ***edges****. For arbitrary $S \subseteq V$, let $E(S) = \{y \mid \exists x \in S, (x, y) \in E\}$ and $E^{-1}(S) = \{x \mid \exists\, y \in S, (x, y) \in E\}$. We say that B* ***is stable with respect to*** *S for arbitrary $S \subseteq V$ and $B \subseteq V$, if $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. Let P be a partition of V, that is, a decomposition of V into disjoint sets, or in other words, blocks. Then P* ***is stable with respect to*** *S, if all blocks of P are stable with respect to S. P* ***is stable with respect to partition*** *P', if all blocks of P are stable with respect to all blocks of P'. If P is stable with respect to all of its blocks, then partition P is* ***stable****. Let P and Q be two partitions of V. Q* ***is a refinement of*** *P, or P* ***is coarser than*** *Q, if every block of P is the union of some blocks of Q. Given V, E and P,* ***the coarsest stable partition*** *is the coarsest stable refinement of P, that is, the stable refinement of P that is coarser than any other stable refinement of P.*

Note that stability is sometimes defined the following way. B is stable with respect to S if $B \subseteq E(S)$ or $B \cap E(S) = \emptyset$. This is not a major difference, only the direction of the edges is reversed. So in this case stability is defined with respect to the binary relation E^{-1} instead of E, where $(x, y) \in E^{-1}$ if and only if $(y, x) \in E$, since $(E^{-1})^{-1}(S) = \{x \mid \exists\, y \in S, (x, y) \in E^{-1})\} = \{x \mid \exists\, y \in S, (y, x) \in E\} = E(S)$.

Let $|V| = n$ and $|E| = m$. We will prove that there always exists a unique solution of the problem of finding the coarsest stable partition, and there is an algorithm that finds the solution in $O(m \lg n)$ time with space requirement $O(m + n)$. This algorithm was published by R. Paige and R. E. Tarjan in 1987, therefore it will be called the ***PT-algorithm***.

The main idea of the algorithm is that if a block is not stable, then it can be split into two in such a way that the two parts obtained are stable. First we will show a naive method. Then, using the properties of the split operation, we will increase its efficiency by continuing the procedure with the smallest part.

Definition 19.20 *Let E be a binary relation on V, $S \subseteq V$ and Q a partition of V. Furthermore, let* ***split***(S, Q) *be the refinement of Q which is obtained by splitting all blocks B of Q*

that are not disjoint from $E^{-1}(S)$, that is, $B \cap E^{-1}(S) \neq \emptyset$ and $B \setminus E^{-1}(S) \neq \emptyset$. In this case, add blocks $B \cap E^{-1}(S)$ and $B \setminus E^{-1}(S)$ to the partition instead of B. S is a ***splitter*** *of Q if $split(S, Q) \neq Q$.*

Note that Q is not stable with respect to S if and only if S is a splitter of Q.

Stability and splitting have the following properties, the proofs are left to the Reader.

Proposition 19.21 *Let S and T be two subsets of V, while P and Q two partitions of V. Then*
1. Stability is preserved under refinement, that is, if Q is a refinement of P, and P is stable with respect to S, then Q is also stable with respect to S.
2. Stability is preserved under unification, that is, if P is stable with respect to both S and T, then P is stable with respect to $S \cup T$.
3. The split operation is monotonic in its second argument, that is, if P is a refinement of Q, then $split(S, P)$ is a refinement of $split(S, Q)$.
4. The split operation is commutative in the following sense. For arbitrary S, T and P, $split(S, split(T, P)) = split(T, split(S, P))$, and the coarsest partition of P that is stable with respect to both S and T is $split(S, split(T, P))$.

In the naive algorithm, we refine partition Q starting from partition P, until Q is stable with respect to all of its blocks. In the refining step, we seek a splitter S of Q that is a union of some blocks of Q. Note that finding a splitter among the blocks of Q would be sufficient, but this more general way will help us in improving the algorithm.

```
NAIVE-PT($V, E, P$)
1  $Q \leftarrow P$
2  while $Q$ is not stable
3        do let $S$ be a splitter of $Q$ that is the union of some blocks of $Q$
4           $Q \leftarrow split(S, Q)$
5  return $Q$
```

Note that the same set S cannot be used twice during the execution of the algorithm, since stability is preserved under refinement, and the refined partition obtained in step 4 is stable with respect to S. The union of the sets S used can neither be used later, since stability is also preserved under unification. It is also obvious that a stable partition is stable with respect to any S that is a union of some blocks of the partition. The following propositions can be proved easily using these properties.

Proposition 19.22 *In any step of the algorithm* Naive-PT*, the coarsest stable refinement of P is a refinement of the actual partition stored in Q.*

Proof. The proof is by induction on the number of times the cycle is executed. The case $Q = P$ is trivial. Suppose that the statement holds for Q before using the splitter S. Let R be the coarsest stable refinement of P. Since S consists of blocks of Q, and, by induction, R is a refinement of Q, therefore S is the union of some blocks of R. R is stable with respect to all of its blocks and the union of any of its blocks, thus R is stable with respect to S, that is, $R = split(S, R)$. On the other hand, using that the split operation is monotonic, $split(S, R)$ is a refinement of $split(S, Q)$, which is the actual value of Q. ■

Proposition 19.23 *The algorithm* Naive-PT *determines the unique coarsest stable refinement of P, while executing the cycle at most $n - 1$ times.*

Proof. The number of blocks of Q is obviously at least 1 and at most n. Using the split operation, at least one block of Q is divided into two, so the number of blocks increases. This implies that the cycle is executed at most $n-1$ times. Q is a stable refinement of P when the algorithm terminates, and, using the previous proposition, the coarsest stable refinement of P is a refinement of Q. This can only happen if Q is the coarsest stable refinement of P. ■

Proposition 19.24 *If we store the set $E^{-1}(\{x\})$ for all elements x of V, then the cost of the algorithm* Naive-PT *is at most $O(mn)$.*

Proof. We can assume, without restricting the validity of the proof, that there are no sinks in the graph, that is, every node has outgoing edges. Then $1 \le |E(\{x\})|$ for arbitrary x in V. Consider a partition P, and split all blocks B of P. Let B' be the set of the nodes of B that have at least one outgoing edge. Then $B' = B \cap E^{-1}(V)$. Now let $B'' = B \setminus E^{-1}(V)$, that is, the set of sinks of B. Set B'' is stable with respect to arbitrary S, since $B'' \cap E^{-1}(S) = \emptyset$, so B'' does not have to be split during the algorithm. Therefore, it is enough to examine partition P' consisting of blocks B' instead of P, that is, a partition of set $V' = E^{-1}(V)$. By adding blocks B'' to the coarsest stable refinement of P' we obviously get the coarsest stable refinement of P. This means that there is a preparation phase before the algorithm in which P' is obtained, and a processing phase after the algorithm in which blocks B'' are added to the coarsest stable refinement obtained by the algorithm. The cost of preparation and processing can be estimated the following way. V' has at most m elements. If, for all x in V we have $E^{-1}(\{x\})$, then the preparation and processing requires $O(m + n)$ time.

From now on we will assume that $1 \le |E(\{x\})|$ holds for arbitrary x in V, which implies that $n \le m$. Since we store sets $E^{-1}(\{x\})$, we can find a splitter among the blocks of partition Q in $O(m)$ time. This, combined with the previous proposition, means that the algorithm can be performed in $O(mn)$ time. ■

The algorithm can be executed more efficiently using a better way of finding splitter sets. The main idea of the improved algorithm is that we work with two partitions besides P, Q and a partition X that is a refinement of Q in every step such that Q is stable with respect to all blocks of X. At the start, let $Q = P$ and let X be the partition consisting only one block, set V. The refining step of the algorithm is repeated until $Q = X$.

```
PT($V, E, P$)

$Q \leftarrow P$
$X \leftarrow \{V\}$
while $X \neq Q$
        do let $S$ be a block of $X$ that is not a block of $Q$,
           and $B$ a block of $Q$ in $S$ for which $|B| \le |S|/2$
           $X \leftarrow (X \setminus \{S\}) \cup \{B, S \setminus B\}$
           $Q \leftarrow split(S \setminus B, split(B, Q))$
return $Q$
```

Proposition 19.25 *The result of the* PT-*algorithm is the same as that of algorithm* Naive-PT.

Proof. At the start, Q is a stable refinement of P with respect to the blocks of X. In step 5, a block of X is split, thus we obtain a refinement of X. In step 6, by refining Q using splits we ensure that Q is stable with respect to two new blocks of X. The properties of stability mentioned in Proposition 19.21 and the correctness of algorithm Naive-PT imply that the PT-algorithm also determines the unique coarsest stable refinement of P. ■

In some cases one of the two splits of step 6 can be omitted. A sufficient condition is that E is a function of x.

Proposition 19.26 *If* $|E(\{x\})| = 1$ *for all* x *in* V*, then step 6 of the* PT-*algorithm can be exchanged with* $Q \leftarrow \mathit{split}(B, Q)$.

Proof. Suppose that Q is stable with respect to a set S which is the union of some blocks of Q. Let B be a block of Q that is a subset of S. It is enough to prove that $\mathit{split}(B, Q)$ is stable with respect to $(S \setminus B)$. Let B_1 be a block of $\mathit{split}(B, Q)$. Since the result of a split according to B is a stable partition with respect to B, either $B_1 \subseteq E^{-1}(B)$ or $B_1 \subseteq E^{-1}(S) \setminus E^{-1}(B)$. Using $|E(\{x\})| = 1$, we get $B_1 \cap E^{-1}(S \setminus B) = \emptyset$ in the first case, and $B_1 \subseteq E^{-1}(S \setminus B)$ in the second case, which means that we obtained a stable partition with respect to $(S \setminus B)$. ■

Note that the stability of a partition with respect to S and B generally does not imply that it is also stable with respect to $(S \setminus B)$. If this is true, then the execution cost of the algorithm can be reduced, since the only splits needed are the ones according to B because of the reduced sizes.

The two splits of step 6 can cut a block into four parts in the general case. According to the following proposition, one of the two parts gained by the first split of a block remains unchanged at the second split, so the two splits can result in at most three parts. Using this, the efficiency of the algorithm can be improved even in the general case.

Proposition 19.27 *Let* Q *be a stable partition with respect to* S*, where* S *is the union of some blocks of* Q*, and let* B *be a block of* Q *that is a subset of* S*. Furthermore, let* D *be a block of* Q *that is cut into two (proper) parts* D_1 *and* D_2 *by the operation* $\mathit{split}(B, Q)$ *in such a way that none of these is the empty set. Suppose that block* D_1 *is further divided into the nonempty sets* D_{11} *and* D_{12} *by* $\mathit{split}(S \setminus B, \mathit{split}(B, Q))$*. Then*
1. $D_1 = D \cap E^{-1}(B)$ *and* $D_2 = D \setminus D_1$ *if and only if* $D \cap E^{-1}(B) \neq \emptyset$ *and* $D \setminus E^{-1}(B) \neq \emptyset$*.*
2. $D_{11} = D_1 \cap E^{-1}(S \setminus B)$ *and* $D_{12} = D_1 \setminus D_{11}$ *if and only if* $D_1 \cap E^{-1}(S \setminus B) \neq \emptyset$ *and* $D_1 \setminus E^{-1}(S \setminus B) \neq \emptyset$*.*
3. The operation $\mathit{split}(S \setminus B, \mathit{split}(B, Q))$ *leaves block* D_2 *unchanged.*
4. $D_{12} = D_1 \cap (E^{-1}(B) \setminus E^{-1}(S \setminus B))$*.*

Proof. The first two statements follow using the definition of the split operation. To prove the third statement, suppose that D_2 was obtained from D by a proper decomposition. Then $D \cap E^{-1}(B) \neq \emptyset$, and since $B \subseteq S$, $D \cap E^{-1}(S) \neq \emptyset$. All blocks of partition Q, including D, are stable with respect to S, which implies $D \subseteq E^{-1}(S)$. Since $D_2 \subseteq D$, $D_2 \subseteq E^{-1}(S) \setminus E^{-1}(B) = E^{-1}(S \setminus B)$ using the first statement, so D_2 is stable with respect to the set $S \setminus B$, therefore a split according to $S \setminus B$ does not divide block D_2. Finally, the fourth statement follows from $D_1 \subseteq E^{-1}(B)$ and $D_{12} = D_1 \setminus E^{-1}(S \setminus B)$. ■

Denote by $counter(x, S)$ the number of nodes in S that can be reached from x, that is, $counter(x, S) = |S \cap E(\{x\})|$. Note that if $B \subseteq S$, then $E^{-1}(B) \setminus E^{-1}(S \setminus B) = \{x \in E^{-1}(B) \mid counter(x, B) = counter(x, S)\}$.

Since sizes are always halved, an arbitrary x in V can appear in at most $\lg n + 1$ different sets B that were used for refinement in the PT-algorithm. In the following, we will give an execution of the PT algorithm in which the determination of the refinement according to block B in steps 5 and 6 of the algorithm costs $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$. Summing this for all blocks B used in the algorithm and for all elements of these blocks, we get that the complexity of the algorithm Efficient-PT is at most $O(m \lg n)$. To give such a realization of the algorithm, we have to choose good data structures for the representation of our data.

- Attach node x to all edges (x, y) of set E, and attach the list $\{(x, y) \mid (x, y) \in E\}$ to all nodes y. Then the cost of reading set $E^{-1}(\{y\})$ is proportional to the size of $E^{-1}(\{y\})$.

- Let partition Q be a refinement of partition X. Represent the blocks of the two partitions by records. A block S of partition X is ***simple*** if it consists of one block of Q, otherwise it is ***compound***.

- Let C be the list of all compound blocks in partition X. At start, let $C = \{V\}$, since V is the union of the blocks of P. If P consists of only one block, then P is its own coarsest stable refinement, so no further computation is needed.

- For any block S of partition P, let $Q\text{-}blocks(S)$ be the double-chained list of the blocks of partition Q the union of which is set S. Furthermore, store the values $counter(x, S)$ for all x in set $E^{-1}(S)$ to which one pointer points from all edges (x, y) such that y is an element of S. At start, the value assigned to all nodes x is $counter(x, V) = |E(\{x\})|$, and make a pointer to all nodes (x, y) that points to the value $counter(x, V)$.

- For any block B of partition Q, let $X\text{-}block(B)$ be the block of partition X in which B appears. Furthermore, let $size(B)$ be the cardinality of B, and $elements(B)$ the double-chained list of the elements of B. Attach a pointer to all elements that points to the block of Q in which this element appears. Using double chaining any element can be deleted in $O(1)$ time.

Using the proof of Proposition 19.24, we can suppose that $n \le m$ without restricting the validity. It can be proved that in this case the space requirement for the construction of such data structures is $O(m)$.

```
Efficient-PT($V, E, P$)
if $|P| = 1$
   then return $P$
$Q \leftarrow P$
$X \leftarrow \{V\}$
$C \leftarrow \{V\}$                            ▷ $C$ is the list of the compound blocks of $X$.
while $C \neq \emptyset$
      do let $S$ be an element of $C$
         let $B$ be the smaller of the first two elements of $S$
         $C \leftarrow C \setminus \{S\}$
         $X \leftarrow (X \setminus \{S\}) \cup \{\{B\}, S \setminus \{B\}\}$
         $S \leftarrow S \setminus \{B\}$
         if $|S| > 1$
            then $C \leftarrow C \cup \{S\}$
         Generate set $E^{-1}(B)$ by reading the edges $(x, y)$ of set $E$ for which $y$
         is an element of $B$, and for all elements $x$ of this set, compute the value
         $counter(x, B)$.
         Find blocks $D_1 = D \cap E^{-1}(B)$ and $D_2 = D \setminus D_1$ for all blocks
         $D$ of $Q$ by reading set $E^{-1}(B)$
         By reading all edges $(x, y)$ of set $E$ for which $y$ is an element of $B$,
         create set $E^{-1}(B) \setminus E^{-1}(S \setminus B)$ checking the condition
         $counter(x, B) = counter(x, S)$
         Reading set $E^{-1}(B) \setminus E^{-1}(S \setminus B)$, for all blocks $D$ of $Q$,
         determine the sets $D_{12} = D_1 \cap (E^{-1}(B) \setminus E^{-1}(S \setminus B))$
         and $D_{11} = D_1 \setminus D_{12}$
         for all blocks $D$ of $Q$ for which $D_{11} \neq \emptyset$, $D_{12} \neq \emptyset$ and $D_2 \neq \emptyset$
             do if $D$ is a simple block of $X$
                   then $C \leftarrow C \cup \{D\}$
                $Q \leftarrow (Q \setminus \{D\}) \cup \{D_{11}, D_{12}, D_2\}$
         Compute the value $counter(x, S)$ by reading
         the edges $(x, y)$ of $E$ for which $y$ is an element of $B$.
return $Q$
```

Proposition 19.28 *The algorithm* Efficient-PT *determines the coarsest stable refinement of P. The computation cost of the algorithm is $O(m \lg n)$, and its space requirement is $O(m + n)$.*

Proof. The correctness of algorithm follows from the correctness of the PT-algorithm and Proposition 19.27. Because of the data structures used, the computation cost of the steps of the cycle is proportional to the number of edges examined and the number of elements of block B, which is $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$ altogether. Sum this for all blocks B used during the refinement and all elements of these blocks. Since the size of B is at most half the size of S, arbitrary x in set V can be in at most $\lg n + 1$ different sets B. Therefore, the total computation cost of the algorithm is $O(m \lg n)$. It can be proved easily that a space of $O(m + n)$ size is enough for the storage of the data structures used in the algorithm and their maintenance. ■

Note that the algorithm could be further improved by contracting some of its steps but that would only decrease computation cost by a constant factor.

Let $G^{-1} = (V, E^{-1})$ be the graph that can be obtained from G by changing the direction of all edges of G. Consider a 1-index in graph G determined by the bisimulation $\approx$. Let I and J be two classes of the bisimulation, that is, two nodes of $I(G)$. Using the definition of bisimulation, $J \subseteq E(I)$ or $E(I) \cap J = \emptyset$. Since $E(I) = (E^{-1})^{-1}(I)$, this means that J is stable with respect to I in graph G^{-1}. So the coarsest 1-index of G is the coarsest stable refinement of the basic partition of graph G^{-1}.

Corollary 19.29 *The coarsest 1-index can be determined using the algorithm* Efficient-PT. *The computation cost of the algorithm is at most $O(m \lg n)$, and its space requirement is at most $O(m + n)$.*

Exercises

19.4-1 Prove Proposition 29.21.

19.4-2 Partition P is size-stable with respect to set S if $|E(\{x\}) \cap S| = |E(\{y\}) \cap S|$ for arbitrary elements x, y of a block B of P. A partition is size-stable if it is size-stable with respect to all its blocks. Prove that the coarsest size-stable refinement of an arbitrary partition can be computed in $O(m \lg n)$ time.

19.4-3 The 1-index is minimal if no two nodes I and J with the same label can be contracted, since there exists a node K for which $I \cup J$ is not stable with respect to K. Give an example that shows that the minimal 1-index is not unique, therefore it is not the same as the coarsest 1-index.

19.4-4 Prove that in case of an acyclic graph, the minimal 1-index is unique and it is the same as the coarsest 1-index.

19.5. A(k)-indexes

In case of 1-indexes, nodes of the same class fit to the same label sequences starting from the root. This means that the nodes of a class cannot be distinguished by their ancestors. Modifying this condition in such a way that indistinguishability is required only locally, that is, nodes of the same class cannot be distinguished by at most k generations of ancestors, we obtain an index that is coarser and consists of less classes than the 1-index. So the size of the index decreases, which also decreases the cost of the evaluation of queries. The 1-index was safe and exact, which we would like to preserve, since these guarantee that the result we get when evaluating the queries according to the index is the result we would have obtained by evaluating the query according to the original graph. The A(k)-index is also safe, but it is not exact, so this has to be ensured by modification of the evaluation algorithm.

Definition 19.30 *The* ***k-bisimulation*** $\approx^k$ *is anequivalence relation on the nodes V of a graph defined recursively as*
i) $u \approx^0 v$ if and only if label(u) = label(v),
ii) $u \approx^k v$ if and only if $u \approx^{k-1} v$ and if there is an edge from node u' to node u, then there is a node v' from which there is an edge to node v and $u' \approx^{k-1} v'$, also, if there is an edge from node v' to node v, then there is a node u' from which there is an edge to node u and

$u' \approx^{k-1} v'$.
*In case $u \approx^k v$ u and v are **k-bisimilar**. The classes of the partition according to the **A**(k)**-index** are the equivalence classes of the k-bisimulation.*

The „A” in the notation refers to the word „approximative”.

Note that the partition belonging to $k = 0$ is the basic partition, and by increasing k we refine this, until the coarsest 1-index is reached.

Denote by $L(u, k, G)$ the label sequences of length at most k to which u fits in graph G. The following properties of the A(k)-index can be easily checked.

Proposition 19.31
1. If u and v are k-bisimilar, then $L(u, k, G) = L(v, k, G)$.
2. If I is a node of the A(k)-index and $u \in I$, then $L(I, k, I(G)) = L(u, k, G)$.
3. The A(k)-index is exact in case of simple expressions of length at most k.
4. The A(k)-index is safe.
5. The $(k + 1)$-bisimulation is a (not necessarily proper) refinement of the k-bisimulation.

The A(k)-index compares the k-distance half-neighbourhoods of the nodes which contain the root, so the equivalence of the nodes is not affected by modifications outside this neighbourhood, as the following proposition shows.

Proposition 19.32 *Suppose that the shortest paths from node v to nodes x and y contain more than k edges. Then adding or deleting an edge from u to v does not change the k-bisimilarity of x and y.*

We use a modified version of the PT-algorithm for creating the A(k)-index. Generally, we can examine the problem of approximation of the coarsest stable refinement.

Definition 19.33 *Let P be a partition of V in the directed graph $G = (V, E)$, and let $P_0, P_1, \ldots, P_k$ be a sequence of partitions such that $P_0 = P$ and P_{i+1} is the coarsest refinement of P_i that is stable with respect to P_i. In this case, partition P_k is the k**-step approximation of the coarsest stable refinement of** P.*

Note that every term of sequence P_i is a refinement of P, and if $P_k = P_{k-1}$, then P_k is the coarsest stable refinement of P. It can be checked easily that an arbitrary approximation of the coarsest stable refinement of P can be computed greedily, similarly to the PT-algorithm. That is, if a block B of P_i is not stable with respect to a block S of P_{i-1}, then split B according to S, and consider the partition *split*(S, P_i) instead of P_i.

```
Naive-Approximation($V, E, P, k$)

$P_0 \leftarrow P$
for $i \leftarrow 1$ to $k$
    do $P_i \leftarrow P_{i-1}$
        for all $S \in P_{i-1}$such that $split(S, P_i) \neq P_i$
            do $P_i \leftarrow split(S, P_i)$
return $P_k$
```

Note that the algorithm Naive-Approximation could also be improved similarly to the PT-algorithm.

Algorithm Naive-Approximation can be used to compute the A(k)-index, all we have to notice is that the partition belonging to the A(k)-index is stable with respect to the partition belonging to the A($k-1$)-index in graph G^{-1}. It can be shown that the computation cost of the A(k)-index obtained this way is $O(km)$, where m is the number of edges in graph G.

```
A(k)-Index-Evaluation($G, A_R, k$)

1 let $I_k$ be the A($k$)-index of $G$
2 $Q \leftarrow$ Index-Evaluation($G, I_k, A_R$)
3 for all $u \in Q$
4      do if $L(u) \cap L(A_R) = \emptyset$
5           then $Q \leftarrow Q \setminus \{u\}$
6 return $Q$
```

The A(k)-index is safe, but it is only exact for simple expressions of length at most k, so in step 4, we have to check for all elements u of set Q whether it satisfies query R, and we have to delete those from the result that do not fit to query R. We can determine using a finite nondeterministic automaton whether a given node satisfies expression R as in Proposition 19.6, but the automaton has to run in the other way. The number of these checks can be reduced according to the following proposition, the proof of which is left to the Reader.

Proposition 19.34 *Suppose that in the graph I_k belonging to the A(k)-index, index node I fits to a label sequence that ends with $s = l_0, \ldots, l_p$, $p \leq k-1$. If all label sequences of the form s'.s that start from the root satisfy expression R in graph G, then all elements of I satisfy expression R.*

Exercises

19.5-1 Denote by $A_k(G)$ the A(k)-index of G. Determine whether $A_k(A_l(G)) = A_{k+l}(G)$.
19.5-2 Prove Proposition 19.31.
19.5-3 Prove Proposition 19.32.
19.5-4 Prove Proposition 19.34.
19.5-5 Prove that the algorithm Naive-approximation generates the coarsest k-step stable approximation.
19.5-6 Let $A = \{A_0, A_1, \ldots, A_k\}$ be a set of indexes, the elements of which are A(0)-, A(1)-, …, A(k)-indexes, respectively. A is *minimal*, if by uniting any two elements of A_i, A_i is not stable with respect to A_{i-1}, $1 \leq i \leq k$. Prove that for arbitrary graph, there exists a unique minimal A the elements of which are coarsest A(i)-indexes, $0 \leq i \leq k$.

19.6. D(k)- and M(k)-indexes

When using A(k)-indexes, the value of k must be chosen appropriately. If k is too large, the size of the index will be too big, and if k is too small, the result obtained has to be checked too many times in order to preserve exactness. Nodes of the same class are similar locally, that is, they cannot be distinguished by their k distance neighbourhoods, or, more precisely, by the paths of length at most k leading to them. The same k is used for all nodes, even though there are less important nodes. For instance, some nodes appear very rarely in results of queries in practice, and only the label sequences of the paths passing through them are examined. There is no reason for using a better refinement on the less important nodes. This suggests the idea of using the dynamic D(k)-index, which assigns different values k to the nodes according to queries. Suppose that a set of queries is given. If there is an $R.a.b$ and an $R.a.b.c$ query among them, where R and R' are regular queries, then a partition according to at least 1-bisimulation in case of nodes with label b, and according to at least 2-bisimulation in case of nodes with label c is needed.

Definition 19.35 *Let $I(G)$ be the index graph belonging to graph G, and to all index node I assign a nonnegative integer $k(I)$. Suppose that the nodes of block I are $k(I)$-bisimilar. Let the values $k(I)$ satisfy the following condition: if there is an edge from I to J in graph $I(G)$, then $k(I) \geq k(J) - 1$. The index $I(G)$ having this property is called a* ***D(k)-index****.*

The „D" in the notation refers to the word „dynamic". Note that the A(k)-index is a special case of the D(k)-index, since in case of A(k)-indexes, the elements belonging to any index node are exactly k-bisimilar.

Since classification according to labels, that is, the basic partition is an A(0)-index, and in case of finite graphs, the 1-index is the same as an A(k)-index for some k, these are also special cases of the D(k)-index. The D(k)-index, just like any other index, is safe, so it is sufficient to evaluate the queries on them. Results must be checked to ensure exactness. The following proposition states that exactness is guaranteed for some queries, therefore checking can be omitted in case of such queries.

Proposition 19.36 *Let $I_1, I_2, \ldots, I_s$ be a directed path in the D(k)-index, and suppose that $k(I_j) \geq j - 1$ if $1 \leq j \leq s$. Then all elements of I_s fit to the label sequence label(I_1), label$(I_2), \ldots$, label(I_s).*

Proof. The proof is by induction on s. The case $s = 1$ is trivial. By the inductive assumption, all elements of I_{s-1} fit to the label sequence label(I_1), label$(I_2), \ldots$, label(I_{s-1}). Since there is an edge from node I_{s-1} to node I_s in graph $I(G)$, there exist $u \in I_s$ and $v \in I_{s-1}$ such that there is an edge from v to u in graph G. This means that u fits to the label sequence label(I_1), label$(I_2), \ldots$, label(I_s) of length $s - 1$. The elements of I_s are at least $(s - 1)$-bisimilar, therefore all elements of I_s fit to this label sequence. ■

Corollary 19.37 *The D(k)-index is exact with respect to label sequence $l_0, \ldots, l_m$ if $k(I) \geq m$ for all nodes I of the index graph that fit to this label sequence.*

When creating the D(k)-index, we will refine the basic partition, that is, the A(0)-index. We will assign initial values to the classes consisting of nodes with the same label. Suppose we use t different values. Let K_0 be the set of these values, and denote the elements of K_0 by $k_1 > k_2 > \ldots > k_t$. If the elements of K_0 do not satisfy the condition given in the D(k)-index, then we increase them using the algorithm Weight-Changer, starting with the greatest value, in such a way that they satisfy the condition. Thus, the classes consisting of nodes with the same label will have good k values. After this, we refine the classes by splitting them, until all elements of a class are k-bisimilar, and assign this k to all terms of the split. During this process the edges of the index graph must be refreshed according to the partition obtained by refinement.

```
Weight-Changer($G$, $K_0$)

$K \leftarrow \emptyset$
$K_1 \leftarrow K_0$
while $K_1 \neq \emptyset$
      do for all $I$, where $I$ is a node of the A(0)-index and $k(I) = max(K_1)$
            do for all $J$, where $J$ is a node of the A(0)-index and there is an edge from $J$ to $I$
                  $k(J) \leftarrow max(k(J), max(K_1) - 1)$
      $K \leftarrow K \cup \{max(K_1)\}$
      $K_1 \leftarrow \{k(A) \mid A$ is a node of the A(0)-index $\} \setminus K$
return $K$
```

It can be checked easily that the computation cost of the algorithm Weight-Changer is $O(m)$, where m is the number of edges of the A(0)-index.

```
D($k$)-Index-Creator($G$, $K_0$)

 1  let $I(G)$ be the A(0)-index belonging to graph $G$, let $V_I$ be the set of nodes of $I(G)$,
    let $E_I$ be the set of edges of $I(G)$
 2  $K \leftarrow$ Weight-Changer($G$, $K_0$)        ▷ Changing the initial weights
                                                       according to the condition of the D($k$)-index.
 3  for $k \leftarrow 1$ to $max(K)$
 4     do for all $I \in V_I$
 5           do if $k(I) \geq k$
 6                 then for all $J$, where $(J, I) \in E_I$
 7                         do $V_I \leftarrow (V_I \setminus \{I\}) \cup \{I \cap E(J), I \setminus E(J)\}$
 8                            $k(I \cap E(J)) \leftarrow k(I)$
 9                            $k(I \setminus E(J)) \leftarrow k(I)$
10                            $E_I \leftarrow \{(A, B) \mid A, B \in V_I, \exists\, a \in A, \exists\, b \in B, (a, b) \in E\}$
11                            $I(G) \leftarrow (V_I, E_I)$
12  return $I(G)$
```

In step 7, a split operation is performed. This ensures that the classes consisting of $(k-1)$-bisimilar elements are split into equivalence classes according to k-bisimilarity. It can be proved that the computation cost of the algorithm D(k)-Index-Creator is at most $O(km)$, where m is the number of edges of graph G, and $k = max(K_0)$.

In some cases, the D(k)-index results in a partition that is too fine, and it is not efficient enough for use because of its huge size. Over-refinement can originate in the following. The algorithm D(k)-Index-Creator assigns the same value k to the nodes with the same label, although some of these nodes might be less important with respect to queries, or appear more often in results of queries of length much less than k, so less fineness would be enough for these nodes. Based on the value k assigned to a node, the algorithm Weight-Changer will not decrease the value assigned to the parent node if it is greater than $k - 1$. Thus, if these parents are not very significant nodes considering frequent queries, then this can cause over-refinement. In order to avoid over-refinement, we introduce the M(k)-index and the M*(k)-index, where the „M” refers to the word „mixed”, and the „*” shows that not one index is given but a finite hierarchy of gradually refined indexes. The M(k)-index is a D(k)-index the creation algorithm of which not necessarily assigns nodes with the same label to the same k-bisimilarity classes.

Let us first examine how a D(k)-index $I(G) = (V_I, E_I)$ must be modified if the initial weight k_I of index node I is increased. If $k(I) \geq k_I$, then $I(G)$ does not change. Otherwise, to ensure that the conditions of the D(k)-index on weights are satisfied, the weights on the ancestors of I must be increased recursively until the weight assigned to the parents is at least $k_I - 1$. Then, by splitting according to the parents, the fineness of the index nodes obtained will be at least k_I, that is, the elements belonging to them will be at least k_I-bisimilar. This will be achieved using the algorithm Weight-Increaser.

```
Weight-Increaser($I, k_I, I(G)$)

if $k(I) \geq k_I$
  then return $I(G)$
for all $(J, I) \in E_I$
    do $I(G) \leftarrow$ Weight-Increaser($J, k_I - 1, I(G)$)
for all $(J, I) \in E_I$
    do $V_I \leftarrow (V_I \setminus \{I\}) \cup \{I \cap E(J), I \setminus E(J)\}$
       $E_I \leftarrow \{(A, B) \mid A, B \in V_I, \exists\, a \in A, \exists\, b \in B, (a, b) \in E\}$
       $I(G) \leftarrow (V_I, E_I)$
return $I(G)$
```

The following proposition can be easily proved, and with the help of this we will be able to achieve the appropriate fineness in one step, so we will not have to increase step by step anymore.

Proposition 19.38 *$u \approx^k v$ if and only if $u \approx^0 v$, and if there is an edge from node u' to node u, then there is a node v', from which there is an edge to node v and $u' \approx^{k-1} v'$, and, conversely, if there is an edge from node v' to node v, then there is a node u', from which there is an edge to node u and $u' \approx^{k-1} v'$.*

Denote by FRE the set of simple expressions, that is, the label sequences determined by the frequent regular queries. We want to achieve a fineness of the index that ensures that it is exact on the queries belonging to FRE. For this, we have to determine the significant nodes, and modify the algorithm D(k)-Index-Creator in such a way that the not significant nodes and their ancestors are always deleted at the refining split.

Let $R \in FRE$ be a frequent simple query. Denote by S and T the set of nodes that fit to R in the index graph and data graph, respectively, that is $S = R(I(G))$ and $T = R(G)$. Denote by $k(I)$ the fineness of index node I in the index graph $I(G)$, then the nodes belonging to I are at most $k(I)$-bisimilar.

```
Refine($R, S, T$)
for all $I \in S$
    do $I(G) \leftarrow$ Refine-Index-Node($I$, length($R$), $I \cap T$)
while $\exists I \in V_I$ such that $k(I) <$ length($R$) and $I$ fits to $R$
    do $I(G) \leftarrow$ Weight-Increaser($I$, length($R$), $I(G)$)
return $I(G)$
```

The refinement of the index nodes will be done using the following algorithm. First, we refine the significant parents of index node I recursively. Then we split I according to its significant parents in such a way that the fineness of the new parts is k. The split parts of I are kept in set H. Lastly, we unite those that do not contain significant nodes, and keep the original fineness of I for this united set.

```
Refine-Index-Node($I$, $k$, significant-nodes)
if $k(I) \geq k$
  then return $I(G)$
for all $(J, I) \in E_I$
    do significant-parents $\leftarrow E^{-1}$(significant-nodes) $\cap J$
       if significant-parents $\neq \emptyset$
         then Refine-Index-Node($J$, $k - 1$, significant-parents)
k-previous $\leftarrow k(I)$
$H \leftarrow \{I\}$
for all $(J, I) \in E_I$
    do if $E^{-1}$(significant-parents) $\cap J \neq \emptyset$
         then for all $F \in H$
                  do $V_I \leftarrow (V_I \setminus \{F\}) \cup \{F \cap E(J), F \setminus E(J)\}$
                     $E_I \leftarrow \{(A, B) \mid A, B \in V_I, \exists a \in A, \exists b \in B, (a, b) \in E\}$
                     $k(F \cap E(J)) \leftarrow k$
                     $k(F \setminus E(J)) \leftarrow k$
                     $I(G) \leftarrow (V_I, E_I)$
                     $H \leftarrow (H \setminus \{F\}) \cup \{F \cap E(J), F \setminus E(J)\}$
remaining $\leftarrow \emptyset$
for all $F \in H$
    do if significant-nodes $\cap F = \emptyset$
         then remaining $\leftarrow$ remaining $\cup F$
              $V_I \leftarrow (V_I \setminus \{F\})$
$V_I \leftarrow V_I \cup \{$remaining$\}$
$E_I \leftarrow \{(A, B) \mid A, B \in V_I, \exists a \in A, \exists b \in B, (a, b) \in E\}$
$k$(remaining) $\leftarrow$ k-previous
$I(G) \leftarrow (V_I, E_I)$
return $I(G)$
```

The algorithm Refine refines the index graph $I(G)$ according to a frequent simple expression in such a way that it splits an index node into not necessarily equally fine parts, and thus avoids over-refinement. If we start from the A(0)-index, and create the refinement for all frequent queries, then we get an index graph that is exact with respect to frequent queries. This is called the M(k)-index. The set *FRE* of frequent queries might change during the process, so the index must be modified dynamically.

Definition 19.39 *The* ***M(k)-index*** *is a D(k)-index created using the following* M(k)-Index-Creator *algorithm.*

```
M(k)-Index-Creator($G$, $FRE$)

$I(G) \leftarrow$ the A(0) index belonging to graph $G$
$V_I \leftarrow$ the nodes of $I(G)$
for all $I \in V_I$
    do $k(I) \leftarrow 0$
$E_I \leftarrow$ the set of edges of $I(G)$
for all $R \in FRE$
    do $I(G) \leftarrow$ Refine($R$, $R(I(G))$, $R(G)$)
return $I(G)$
```

The M(k)-index is exact with respect to frequent queries. In case of a not frequent query, we can do the following. The M(k)-index is also a D(k)-index, therefore if an index node fits to a simple expression R in the index graph $I(G)$, and the fineness of the index node is at least the length of R, then all elements of the index node fit to the query R in graph G. If the fineness of the index node is less, then for all of its elements, we have to check according to Naive-Evaluation whether it is a solution in graph G.

When using the M(k)-index, over-refinement is the least if the lengths of the frequent simple queries are the same. If there are big differences between the lengths of frequent queries, then the index we get might be too fine for the short queries. Create the sequence of gradually finer indexes with which we can get from the A(0)-index to the M(k)-index in such a way that, in every step, the fineness of parts obtained by splitting an index node is greater by at most one than that of the original index node. If the whole sequence of indexes is known, then we do not have to use the finest and therefore largest index for the evaluation of a simple query, but one whose fineness corresponds to the length of the query.

Definition 19.40 *The* ***M*(k)-index*** *is a sequence of indexes $I_0, I_1, \ldots, I_k$ such that*

1. *Index I_i is an M(i)-index, where $i = 0, 1, \ldots, k$.*
2. *The fineness of all index nodes in I_i is at most i, where $i = 0, 1, \ldots, k$.*
3. *I_{i+1} is a refinement of I_i, where $i = 0, 1, \ldots, k-1$.*
4. *If node J of index I_i is split in index I_{i+1}, and J' is a set obtained by this split, that is, $J' \subseteq J$, then $k(J) \leq k(J') \leq k(J) + 1$.*
5. *Let J be a node of index I_i, and $k(J) < i$. Then $k(J) = k(J')$ for $i < i'$ and for all J' index nodes of $I_{i'}$ such that $J' \subseteq J$.*

It follows from the definition that in case of M*(k)-indexes I_0 is the A(0)-index. The last property says that if the refinement of an index node stops, then its fineness will not change anymore. The M*(*k*)-index possesses the good characteristics of the M(*k*)-index, and its structure is also similar: according to frequent queries the index is further refined if it is necessary to make it exact on frequent queries, but now we store and refresh the coarser indexes as well, not only the finest.

When representing the M*(*k*)-index, we can make use of the fact that if an index node is not split anymore, then we do not need to store this node in the new indexes, it is enough to refer to it. Similarly, edges between such nodes do not have to be stored in the sequence of indexes repeatedly, it is enough to refer to them. Creation of the M*(*k*)-index can be done similarly to the M(*k*)-Index-Creator algorithm. The detailed description of the algorithm can be found in the paper of He and Yang.

With the help of the M*(*k*)-index, we can use several strategies for the evaluation of queries. Let *R* be a frequent simple query.

The simplest strategy is to use the index the fineness of which is the same as the length of the query.

```
M*(k)-Index-Naive-Evaluation($G$, $FRE$, $R$)
$\{I_0, I_1 ,\ldots, I_k\} \leftarrow$ the M*(k)-index corresponding to graph $G$
$h \leftarrow$ length($R$)
return Index-Evaluation($G$, $I_h$, $A_R$)
```

The evaluation can also be done by gradually evaluating the longer prefixes of the query according to the index the fineness of which is the same as the length of the prefix. For the evaluation of a prefix, consider the partitions of the nodes found during the evaluation of the previous prefix in the next index and from these, seek edges labeled with the following symbol. Let $R = l_0, l_1, \ldots, l_h$ be a simple frequent query, that is, *length*(*R*) = *h*.

```
M*(k)-Index-Evaluation-Top-to-Bottom($G$, $FRE$, $R$)
{$I_0, I_1, \ldots, I_k$} ← the M*($k$)-index corresponding to graph $G$
$R_0 \leftarrow l_0$
$H_0 \leftarrow \emptyset$
for all $C \in E_{I_0}(\text{root}(I_0))$                ▷ The children of the root in graph $I_0$.
    do if label($C$) = $l_0$
          then $H_0 \leftarrow H_0 \cup \{C\}$
for $j \leftarrow 1$ to length($R$)
    do $H_j \leftarrow \emptyset$
       $R_j \leftarrow R_{j-1}.l_j$
       $H_{j-1} \leftarrow$ M*($k$)-Index-Evaluation-Top-to-Bottom($G$,$FRE$, $R_{j-1}$)
       for all $A \in H_{j-1}$                          ▷ Node $A$ is a node of graph $I_{j-1}$.
           do if $A = \cup B_m$, where $B_m \in V_{I_j}$       ▷ The partition of node $A$ in graph $I_j$.
                 then for minden $B_m$
                          do for all $C \in E_{I_j}(B_m)$       ▷ For all children of
                                                                $B_m$ in graph $I_j$.
                                 do if label($C$) = $l_j$
                                       then $H_j \leftarrow H_j \cup \{C\}$
return $H_h$
```

Our strategy could also be that we first find a subsequence of the label sequence corresponding to the simple query that contains few nodes, that is, its selectivity is large. Then find the fitting nodes in the index corresponding to the length of the subsequence, and using the sequence of indexes see how these nodes are split into new nodes in the finer index corresponding to the length of the query. Finally, starting from these nodes, find the nodes that fit to the remaining part of the original query. The detailed form of the algorithm M*(k)-Index-Prefiltered-Evaluation is left to the Reader.

Exercises

19.6-1 Find the detailed form of the algorithm M*(k)-Index-Prefiltered-Evaluation . What is the cost of the algorithm?

19.6-2 Prove Proposition 19.38.

19.6-3 Prove that the computation cost of the algorithm Weight-Changer is $O(m)$, where m is the number of edges of the A(0)-index.

19.7. Branching queries

With the help of regular queries we can select the nodes of a graph that are reached from the root by a path the labels of which fit to a given regular pattern. A natural generalization is to add more conditions that the nodes of the path leading to the node have to satisfy. For example, we might require that the node can be reached by a label sequence from a node with a given label. Or, that a node with a given label can be reached from another node by a path with a given label sequence. We can take more of these conditions, or use their negation or composition. To check whether the required conditions hold, we have to step not only forward according to the direction of the edges, but sometimes also backward. In the following, we will give the description of the language of branching queries, and introduce the forward-backward indexes. The forward-backward index which is safe and exact with respect to all branching queries is called FB-index. Just like the 1-index, this is also usually too large, therefore we often use an FB(f,b,d)-index instead, which is exact if the length of successive forward steps is at most f, the length of successive backward steps is at most b, and the depth of the composition of conditions is at most d. In practice, values f, b and d are usually small. In case of queries for which the value of one of these parameters is greater than the corresponding value of the index, a checking step must be added, that is, we evaluate the query on the index, and only keep those nodes of the resulted index nodes that satisfy the query.

If there is a directed edge from node n to node m, then this can be denoted by n/m or $m\backslash n$. If node m can be reached from node n by a directed path, then we can denote that by $n//m$ or $m\backslash\backslash n$. (Until now we used . instead of /, so // represents the regular expression _* or * in short.)

From now on, a label sequence is a sequence in which separators are the forward signs (/, //) and the backward signs (\, \\). A sequence of nodes fit to a label sequence if the relation of successive nodes is determined by the corresponding separator, and the labels of the nodes come according to the label sequence.

There are only forward signs in ***forward label sequences***, and only backward signs in ***backward label sequences***.

Branching queries are defined by the following grammar .

```
branching_query              ::=   forward_label sequence [ or_expression ] forward_sign branching_expression
                                   | forward_label_sequence [ or_expression ]
                                   | forward_label_sequence
or_expression                ::=   and_expression or or_expression
                                   | and_expressnion
and_expression               ::=   branching_condition and and_expression
                                   | not_branching_condition and and_expression
                                   | branching_condition
                                   | not_branching_condition
not_branching_condition      ::=   not branching_condition
branching_condition          ::=   condition_label_sequence [ or_expression ] branching_condition
                                   | condition_label_sequence [ or_expression ]
                                   | condition_label_sequence
condition_label_sequence     ::=   forward_sign label_sequence
                                   | backward_sign label_sequence
```

In branching queries, a condition on a node with a given label holds if there exists a label sequence that fits to the condition. For example, the *root//a/b*[*\c//d* and not *\e/f*]*/g* query seeks nodes with label g such that the node can be reached from the root in such a way that the labels of the last two nodes are a and b, furthermore, there exists a parent of the node with label b whose label is c, and among the descendants of the node with label c there is one with label d, but it has no children with label e that has a parent with label f.

If we omit all conditions written between signs [ ] from a branching query, then we get the ***main query*** corresponding to the branching query. In our previous example, this is the query *root//a/b/g*. The main query always corresponds to a forward label sequence.

A directed graph can be assigned naturally to branching queries. Assign nodes with the same label to the label sequence of the query, in case of separators / and \, connect the successive nodes with a directed edge according to the separator, and in case of separators // and \\, draw the directed edge and label it with label // or \\. Finally, the logic connectives are assigned to the starting edge of the corresponding condition as a label. Thus, it might happen that an edge has two labels, for example // and „and". Note that the graph obtained cannot contain a directed cycle because of the definition of the grammar.

A simple degree of complexity of the query can be defined using the tree obtained. Assign 0 to the nodes of the main query and to the nodes from which there is a directed path to a node of the main query. Then assign 1 to the nodes that can be reached from the nodes with sign 0 on a directed path and have no sign yet. Assign 2 to the nodes from which a node with sign 1 can be reached and have no sign yet. Assign 3 to the nodes that can be reached from nodes with sign 2 and have no sign yet, etc. Assign $2k + 1$ to the nodes that can be reached from nodes with sign $2k$ and have no sign yet, then assign $2k+2$ to the nodes from which nodes with sign $2k + 1$ can be reached and have no sign yet. The value of the greatest sign in the query is called the ***depth of the tree***. The depth of the tree shows how many times the direction changes during the evaluation of the query, that is, we have to seek children or parents according to the direction of the edges. The same query could have been given in different ways by composing the conditions differently, but it can be proved that the value defined above does not depend on that, that is why the complexity of a query was not defined as the number of conditions composed.

The 1-index assigns the nodes into classes according to incoming paths, using bisimulations. The concept of stability used for computations was ***descendant-stability***. A set A of the nodes of a graph is ***descendant-stable*** with respect to a set B of nodes if $A \subseteq E(B)$ or $A \cap E(B) = \emptyset$, where $E(B)$ is the set of nodes that can be reached by edges from B. A partition is stable if any two elements of the partition are descendant-stable with respect to each other. The 1-index is the coarsest descendant-stable partition that assigns nodes with the same label to same classes, which can be computed using the PT-algorithm. In case of branching queries, we also have to go backwards on directed edges, so we will need the concept of ***ancestor-stability*** as well. A set A of nodes of a graph is ***ancestor-stable*** with respect to a set B of the nodes if $A \subseteq E^{-1}(B)$ or $A \cap E^{-1}(B) = \emptyset$, where $E^{-1}(B)$ denotes the nodes from which a node of B can be reached.

Definition 19.41 *The* ***FB-index*** *is the coarsest refinement of the basic partition that is ancestor-stable and descendant-stable.*

Note that if the direction of the edges of the graph is reversed, then an ancestor-stable partition becomes a descendant-stable partition and vice versa, therefore the PT-algorithm and its improvements can be used to compute the coarsest ancestor-stable partition. We will use this in the following algorithm. We start with classes of nodes with the same label, compute the 1-index corresponding to this partition, then reverse the direction of the edges, and refine this by computing the 1-index corresponding to this. When the algorithm stops, we get a refinement of the initial partition that is ancestor-stable and descendant-stable at the same time. This way we obtain the coarsest such partition. The proof of this is left to the Reader.

```
FB-Index-Creator$(V, E)$
$P \leftarrow A(0)$                          ▷ Start with classes of nodes with the same label.
while $P$ changes
    do $P \leftarrow PT(V, E^{-1}, P)$               ▷ Compute the 1-index.
       $P \leftarrow PT(V, E, P)$                  ▷ Reverse the direction of edges, and
                                            ▷ compute the 1-index.
return $P$
```

The following corollary follows simply from the two stabilities.

Corollary 19.42 *The FB-index is safe and exact with respect to branching queries.*

The complexity of the algorithm can be computed from the complexity of the PT-algorithm. Since P is always the refinement of the previous partition, in the worst case refinement is done one by one, that is, we always take one element of a class and create a new class consisting of that element. So in the worst case, the cycle is repeated $O(n)$ times. Therefore, the cost of the algorithm is at most $O(mn \lg n)$.

The partition gained by executing the cycle only once is called the ***F+B-index***, the partition obtained by repeating the cycle twice is the ***F+B+F+B-index***, etc.

The following proposition can be proved easily.

Proposition 19.43 *The F+B+F+B+$\cdots$+F+B-index, where F+B appears d times, is safe and exact with respect to the branching queries of depth at most d.*

Nodes of the same class according to the FB-index cannot be distinguished by branching queries. This restriction is usually too strong, therefore the size of the FB-index is usually much smaller than the size of the original graph. Very long branching queries are seldom used in practice, so we only require local equivalence, similarly to the A(k)-index, but now we will describe it with two parameters depending on what we want to restrict: the length of the directed paths or the length of the paths with reversed direction. We can also restrict the depth of the query. We can introduce the FB(f, b, d)-index, with which such restricted branching queries can be evaluated exactly. We can also evaluate branching queries that do not satisfy the restrictions, but then the result must be checked.

```
FB$(f, b, d)$-Index-Creator$(V, E, f, b, d)$
$P \leftarrow A(0)$                          ▷ start with classes of nodes with the same label.
for $i \leftarrow 1$ to $d$
    do $P \leftarrow$ Naive-Approximation$(V, E^{-1}, P, f)$      ▷ Compute the A($f$)-index.
       $P \leftarrow$ Naive-Approximation$(V, E, P, b)$ ▷ Reverse the direction of the edges, and
                                            ▷ compute the A($b$)-index.
return $P$
```

The cost of the algorithm, based on the computation cost of the A(k)-index, is at most $O(dm \max(f, b))$, which is much better than the computation cost of the FB-index, and the index graph obtained is also usually much smaller.

The following proposition obviously holds for the index obtained.

Proposition 19.44 *The FB(f, b, d)-index is safe and exact for the branching queries in which the length of forward-sequences is at most f, the length of backward-sequences is at most b, and the depth of the tree corresponding to the query is at most d.*

As a special case we get that the FB(∞, ∞, ∞)-index is the FB-index, the FB(∞, ∞, d)-index is the F+B+$\cdots$+F+B-index, where F+B appears d times, the FB(∞, 0, 1)-index is the 1-index, and the FB(k, 0, 1)-index is the A(k)-index.

Exercises

19.7-1 Prove that the algorithm FB-Index-Creator produces the coarsest ancestor-stable and descendant-stable refinement of the basic partition.
19.7-2 Prove Proposition 19.44.

19.8. Index refresh

In database management we usually have three important aspects in mind. We want space requirement to be as small as possible, queries to be as fast as possible, and insertion, deletion and modification of the database to be as quick as possible. Generally, a result that is good with respect to one of these aspects is worse with respect to another aspect. By adding indexes of typical queries to the database, space requirement increases, but in return we can evaluate queries on indexes which makes them faster. In case of dynamic databases that are often modified we have to keep in mind that not only the original data but also the index has to be modified accordingly. The most costly method which is trivially exact is that we create the index again after every modification to the database. It is worth seeking procedures to get the modified indexes by smaller modifications to those indexes we already have.

Sometimes we index the index or its modification as well. The index of an index is also an index of the original graph, although formally it consists of classes of index nodes, but we can unite the elements of the index nodes belonging to the same class. It is easy to see that by that we get a partition of the nodes of the graph, that is, an index.

In the following, we will discuss those modifications of semi-structured databases when a new graph is attached to the root and when a new edges is added to the graph, since these are the ones we need when creating a new website or a new reference.

Suppose that $I(G)$ is the 1-index of graph G. Let H be a graph that has no common node with G. Denote by $I(H)$ the 1-index of H. Let $F = G + H$ be the graph obtained by uniting the roots of G and H. We want to create $I(G+H)$ using $I(G)$ and $I(H)$. The following proposition will help us.

Proposition 19.45 *Let $I(G)$ be the 1-index of graph G, and let J be an arbitrary refinement of $I(G)$. Then $I(J) = I(G)$.*

Proof. Let u and v be two nodes of G. We have to show that u and v are bisimilar in G with respect to the 1-index if and only if $J(u)$ and $J(v)$ are bisimilar in the index graph $I(G)$ with respect to the 1-index of $I(G)$. Let u and v be bisimilar in G with respect to the 1-index. We will prove that there is a bisimulation according to which $J(u)$ and $J(v)$ are bisimilar in $I(G)$. Since the 1-index is the partition corresponding to the coarsest bisimulation, the given bisimulation is a refinement of the bisimulation corresponding to the 1-index, so $J(u)$

and $J(v)$ are also bisimilar with respect to the bisimulation corresponding to the 1-index of $I(G)$. Let $J(a) \approx' J(b)$ if and only if a and b are bisimilar in G with respect to the 1-index. Note that since J is a refinement of $I(G)$, all elements of $J(a)$ and $J(b)$ are bisimilar in G if $J(a) \approx' J(b)$. To show that the relation $\approx'$ is a bisimulation, let $J(u')$ be a parent of $J(u)$, where u' is a parent of u_1, and u_1 is an element of $J(u)$. Then u_1, u and v are bisimilar in G, so there is a parent v' of v for which u' and v' are bisimilar in G. Therefore $J(v')$ is a parent of $J(v)$, and $J(u') \approx' J(v')$. Since bisimulation is symmetric, relation $\approx'$ is also symmetric. We have proved the first part of the proposition.

Let $J(u)$ and $J(v)$ be bisimilar in $I(G)$ with respect to the 1-index of $I(G)$. It is sufficient to show that there is a bisimulation on the nodes of G according to which u and v are bisimilar. Let $a \approx' b$ if and only if $J(a) \approx J(b)$ with respect to the 1-index of $I(G)$. To prove bisimilarity, let u' be a parent of U. Then $J(u')$ is also a parent of $J(u)$. Since $J(u)$ and $J(v)$ are bisimilar if $u \approx' v$, there is a parent $J(v'')$ of $J(v)$ for which $J(u')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$, and v'' is a parent of an element v_1 of $J(v)$. Since v and v_1 are bisimilar, there is a parent v' of v such that v' and v'' are bisimilar. Using the first part of the proof, it follows that $J(v')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$. Since bisimilarity is transitive, $J(u')$ and $J(v')$ are bisimilar with respect to the 1-index of $I(G)$, so $u' \approx' v'$. Since relation $\approx'$ is symmetric by definition, we get a bisimulation. ■

As a consequence of this proposition, $I(G + H)$ can be created with the following algorithm for disjoint G and H.

```
Graphaddition-1-Index($G$, $H$)

$P_G \leftarrow A_G(0)$                          ▷ $P_G$ is the basic partition according to labels.
$P_H \leftarrow A_H(0)$                          ▷ $P_H$ is the basic partition according to labels.
$I_1 \leftarrow PT(V_G, E_G^{-1}, P_G)$          ▷ $I_1$ is the 1-index of $G$.
$I_2 \leftarrow PT(V_H, E_H^{-1}, P_H)$          ▷ $I_2$ is the 1-index of $H$.
$J \leftarrow I_1 + I_2$                         ▷ The 1-indexes are joined at the roots.
$P_J \leftarrow A_J(0)$                          ▷ $P_J$ is the basic partition according to labels.
$I \leftarrow PT(V_J, E_J^{-1}, PJ)$             ▷ $I$ is the 1-index of $J$.
return $I$
```

To compute the cost of the algorithm, suppose that the 1-index $I(G)$ of G is given. Then the cost of the creation of $I(G + H)$ is $O(m_H \lg n_H + (m_{I(H)} + m_{I(G)}) \lg(n_{I(G)} + n_{I(H)}))$, where n and m denote the number of nodes and edges of the graph, respectively.

To prove that the algorithm works, we only have to notice that $I(G)+I(H)$ is a refinement of $I(G+H)$ if G and H are disjoint. This also implies that index $I(G)+I(H)$ is safe and exact, so we can use this as well if we do not want to find the minimal index. This is especially useful if new graphs are added to our graph many times. In this case we use the ***lazy method***, that is, instead of computing the minimal index for every pair, we simply sum the indexes of the addends and then minimize only once.

Proposition 19.46 *Let $I(G_i)$ be the 1-index of graph G_i, $i = 1, \ldots, k$, and let the graphs be disjoint. Then $I(G_1 + \cdots + G_k) = I(I(G_1) + \cdots + I(G_k))$ for the 1-index $I(G_1 + \cdots + G_k)$ of the union of the graphs joined at the roots.*

In the following we will examine what happens to the index if a new edge is added to the graph. Even an operation like this can have significant effects. It is not difficult to construct a graph that contains two identical subgraphs at a distant of 2 from the root which cannot be contracted because of a missing edge. If we add this critical edge to the graph, then the two subgraphs can be contracted, and therefore the size of the index graph decreases to about the half of its original size.

Suppose we added a new edge to graph G from u to v. Denote the new graph by G', that is, $G' = G + (u, v)$. Let partition $I(G)$ be the 1-index of G. If there was an edge from $I(u)$ to $I(v)$ in $I(G)$, then the index graph does not have to be modified, since there is a parent of the elements of $I(v)$, that is, of all elements bisimilar to v, in $I(u)$ whose elements are bisimilar to u. Therefore $I(G') = I(G)$.

If there was no edge from $I(u)$ to $I(v)$, then we have to add this edge, but this might cause that $I(v)$ will no longer be stable with respect to $I(u)$. Let Q be the partition we get from $I(G)$ by splitting $I(v)$ in such a way that v is in one part and the other elements of $I(v)$ are in the other, and leaving all other classes of the partition unchanged. Q defines its edges the usual way, that is, if there is an edge from an element of a class to an element of another class, then we connect the two classes with an edge directed the same way.

Let partition X be the original $I(G)$. Then Q is a refinement of X, and Q is stable with respect to X according to G'. Note that the same invariant property appeared in the PT-algorithm for partitions X and Q. Using Proposition 19.45 it is enough to find a refinement of $I(G')$. If we can find an arbitrary stable refinement of the basic partition of G', then, since the 1-index is the coarsest stable partition, this will be a refinement of $I(G')$. X is a refinement of the basic partition, that is, the partition according to labels, and so is Q. So if Q is stable, then we are done. If it is not, then we can stabilize it using the PT-algorithm by starting with the above partitions X and Q. First we have to examine those classes of the partition that contain a children of v, because these might lost their stability with respect to the two new classes gained by the split. The PT-algorithm stabilizes these by splitting them, but because of this we now have to check their children, since they might have lost stability because of the split, etc. We can obtain a stable refinement using this stability-propagator method. Since we only walk through the nodes that can be reached from v, this might not be the coarsest stable refinement. We have shown that the following algorithm computes the 1-index of the graph $G + (u, v)$.

```
Edgeaddition-1-Index($G$, $(u, v)$)

1  $P_G \leftarrow A_G(0)$                          ▷ $P_G$ is the basic partition according to labels.
2  $I \leftarrow PT(V_G, E_G^{-1}, P_G)$              ▷ $I$ is the 1-index of $G$.
3  $G' \leftarrow G + (u, v)$                       ▷ Add edge $(u, v)$.
4  if $(I(u), I(v)) \in E_I$                        ▷ If there was an edge from $I(u)$ to $I(v)$,
                                                      then no modification is needed.
5     then return $I$
6  $I' \leftarrow \{v\}$                            ▷ Split $I(v)$.
7  $I'' \leftarrow I(v) \setminus \{v\}$
8  $X \leftarrow I$                                 ▷ $X$ is the old partition.
9  $E_I \leftarrow E_I \cup \{(I(u), I(v))\}$       ▷ Add an edge from $I(u)$ to $I(v)$.
10 $Q \leftarrow (I \setminus \{I(v)\}) \cup \{I', I''\}$   ▷ Replace $I(v)$ with $I'$ and $I''$.
11 $E \leftarrow E_Q$                               ▷ Determine the edges of $Q$.
12 $J \leftarrow PT(V_{G'}, E_{G'}^{-1}, P_{G'}, X, Q)$    ▷ Execute the PT-algorithm starting with $X$ and $Q$.
13 $J \leftarrow PT(V_J, E_J^{-1}, P_J)$            ▷ $J$ is the coarsest stable refinement.
14 return $J$
```

Step 13 can be omitted in practice, since the stable refinement obtained in step 12 is a good enough approximation of the coarsest stable partition, there is only 5% difference between them in size.

In the following we will discuss how FB-indexes and A(k)-indexes can be refreshed. The difference between FB-indexes and 1-indexes is that in the FB-index, two nodes are in the same similarity class if not only the incoming but also the outgoing paths have the same label sequences. We saw that in order to create the FB-index we have to execute the PT-algorithm twice, using it on the graph with the edges reversed at the second time. The FB-index can be refreshed similarly to the 1-index. The following proposition can be proved similarly to Proposition 19.45, therefore we leave it to the Reader.

Proposition 19.47 *Let $I(G)$ be the FB-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the FB-index of J. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the FB-index of $G + H$ can be created using the following algorithm for disjoint G and H.

```
Graphaddition-FB-Index($G$, $H$)

$I_1 \leftarrow$ FB-Index-Creator$(V_G, E_G)$      ▷ $I_1$ is the FB-index of $G$.
$I_2 \leftarrow$ FB-Index-Creator$(V_H, E_H)$      ▷ $I_2$ is the FB-index of $H$.
$J \leftarrow I_1 + I_2$                           ▷ Join the FB-indexes at their roots.
$I \leftarrow$ FB-Index-Creator$(V_J, E_J)$        ▷ $I$ is the FB-index of $J$.
return $I$
```

When adding edge (u, v), we must keep in mind that stability can be lost in both directions, so not only $I(v)$ but also $I(u)$ has to be split into $\{v\}$, $(I \setminus \{v\})$ and $\{u\}$, $(I(u) \setminus \{u\})$, respectively. Let X be the partition before the modification, and Q the partition obtained after the splits. We start the PT-algorithm with X and Q in step 3 of the algorithm FB-Index-Creator. When stabilizing, we will now walk through all descendants of v and all ancestors of u.

```
Edgeaddition-FB-Index($G$, $(u, v)$)

$I \leftarrow$ FB-index-creator$(V_G, E_G)$               ▷ $I$ is the FB-index of $G$.
$G' \leftarrow G + (u, v)$                                ▷ Add edge $(u, v)$.
if $(I(u), I(v)) \in E_I$                                 ▷ If there was an edge from $I(u)$ to $I(v)$,
                                                            then no modification is needed.
  then return $I$
$I_1 \leftarrow \{v\}$                                    ▷ Split $I(v)$.
$I_2 \leftarrow I(v) \setminus \{v\}$
$I_3 \leftarrow \{u\}$                                    ▷ Split $I(u)$.
$I_4 \leftarrow I(u) \setminus \{u\}$
$X \leftarrow I$                                          ▷ $X$ is the old partition.
$E_I \leftarrow E_I \cup \{(I(u), I(v))\}$                ▷ Add an edge form $I(u)$ to $I(v)$.
$Q \leftarrow (I \setminus \{I(v), I(u)\}) \cup \{I_1, I_2, I_3, I_4\}$   ▷ Replace $I(v)$ with $I_1$ and $I_2$,
                                                            $I(u)$ with $I_3$ and $I_4$.
$E \leftarrow E_Q$                                        ▷ Determine the edges of $Q$.
$J \leftarrow$ FB-Index-Creator$(V_{G'}, E_{G'}, X, Q)$   ▷ Start the PT-algorithm with $X$ and $Q$
                                                            in the algorithm FB-Index-Creator.
$J \leftarrow$ FB-Index-Creator$(V_J, E_J)$               ▷ $J$ is the coarsest ancestor-stable
                                                            and descendant-stable refinement.
return $J$
```

Refreshing the A(k)-index after adding an edge is different than what we have seen. There is no problem with adding a graph though, since the following proposition holds, the proof of which is left to the Reader.

Proposition 19.48 *Let $I(G)$ be the A(k)-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the A(k)-index of $I(J)$. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the A(k)-index of $G + H$ can be created using the following algorithm for disjoint G and H.

```
Graphaddition-A($k$)-Index$(G, H)$

$P_G \leftarrow A_G(0)$                                       ▷ $P_G$ is the basic partition according to labels.
$I_1 \leftarrow$ Naive-Approximation$(V_G, E_G^{-1}, P_G, k)$   ▷ $I_1$ is the A($k$)-index of $G$.
$P_H \leftarrow A_H(0)$                                       ▷ $P_H$ is the basic partition according to labels.
$I_2 \leftarrow$ Naive-Approximation$(V_H, E_H^{-1}, P_H, k)$   ▷ $I_1$ is the A($k$)-index of $H$.
$J \leftarrow I_1 + I_2$                                      ▷ Join the A($k$)-indexes.
$P_J \leftarrow A_J(0)$                                       ▷ $P_J$ is the basic partition according to labels.
$I \leftarrow$ Naive-Approximation$(V_J, E_J^{-1}, P_J, k)$     ▷ $I$ is the A($k$)-index of $J$.
return $I$
```

If we add a new edge (u, v) to the graph, then, as earlier, first we split $I(v)$ into two parts, one of which is $I' = \{v\}$, then we have to repair the lost k-stabilities walking through the descendants of v, but only within a distant of k. What causes the problem is that the A(k)-index contains information only about k-bisimilarity, it tells us nothing about $(k-1)$-bisimilarity. For example, let v_1 be a child of v, and let $k = 1$. When stabilizing according to the 1-index, v_1 has to be detached from its class if there is an element in this class that is not a children of v. This condition is too strong in case of the A(1)-index, and therefore it causes too many unnecessary splits. In this case, v_1 should only be detached if there is an element in its class that has no 0-bisimilar parent, that is, that has the same label as v. Because of this, if we refreshed the A(k)-index the above way when adding a new edge, we would get a very bad approximation of the A(k)-index belonging to the modification, so we use a different method. The main idea is to store all A(i)-indexes not only the A(k)-index, where $i = 1, \ldots, k$. Yi et al. give an algorithm based on this idea, and creates the A(k)-index belonging to the modification. The given algorithms can also be used for the deletion of edges with minor modifications, in case of 1-indexes and A(k)-indexes.

Exercises

19.8-1 Prove Proposition 19.47.

19.8-2 Give an algorithm for the modification of the index when an edge is deleted from the data graph. Examine different indexes. What is the cost of the algorithm?

19.8-3 Give algorithms for the modification of the D(k)-index when the data graph is modified.

Problems

19-1. Implication problem regarding constraints

Let R and Q be regular expressions, x and y two nodes. Let predicate $R(x, y)$ mean that y can be reached from x by a label sequence that fits to R. Denote by $R \subseteq Q$ the constraint $\forall x(R(\text{root}, x) \rightarrow Q(\text{root}, x))$. $R = Q$ if $R \subseteq Q$ and $Q \subseteq R$. Let C be a finite set of constraints, and c a constraint.

a. Prove that the implication problem $C \models c$ is a 2-EXPSPACE problem.

b. Denote by $R \subseteq Q@u$ the constraint $\forall v(R(u, v) \rightarrow Q(u, v))$. Prove that the implication problem is undecidable with respect to this class.

19-2. Transformational distance of trees

Let the *transformational distance* of vertex-labeled trees be the minimal number of basic operations with which a tree can be transformed to the other. We can use three basic operations: addition of a new node, deletion of a node, and renaming of a label.

a. Prove that the transformational distance of trees T and T' can be computed in $O(n_T n_{T'} d_T d_{T'})$ time, with storage cost of $O(n_T n_{T'})$, where n_T is the number of nodes of the tree and d_T is the depth of the tree.

b. Let S and S' be two trees. Give an algorithm that generates all pairs (T, T'), where T and T' simulates graphs S and S', respectively, and the transformational distance of T and T' is less then a given integer n. (This operation is called *approximate join.*)

19-3. Queries of distributed databases

A distributed database is a vertex-labeled directed graph the nodes of which are distributed in m partitions (servers). The edges between different partitions are *cross references*. Communication is by message broadcasting between the servers. An algorithm that evaluates a query is *efficient*, if the number of communication steps is constant, that is, it does not depend on the data and the query, and the size of the data transmitted during communication only depends on the size of the result of the query and the number of cross references. Prove that an efficient algorithm can be given for the regular query of distributed databases in which the number of communication steps is 4, and the size of data transmitted is $O(n^2) + O(k)$, where n is the size of the result of the query, and k is the number of cross references. (*Hint:* Try to modify the algorithm Naive-Evaluation for this purpose.)

Chapter notes

This chapter examined those fields of the world of semi-structured databases where the morphisms of graphs could be used. Thus we discussed the creation of schemas and indexes from the algorithmic point of view. The world of semi-structured databases and XML is much broader than that. A short summary of the development, current issues and the possible future development of semi-structured databases can be found in the paper of Vianu [22].

The paper of M. Henzinger, T. Henzinger and Kopke [10] discusses the computation of the maximal simulation. They extend the concept of simulation to infinite graphs that can be represented efficiently (these are called effective graphs), and prove that for such graphs, it can be determined whether two nodes are similar. In their paper, Corneil and Gotlieb [6] deal with quotient graphs and the determination of isomorphism of graphs. Arenas and Libkin [1] extend normal forms used in the relational model to XML documents. They show that arbitrary DTD can be rewritten without loss as XNF, a normal form they introduced.

Buneman, Fernandez and Suciu [4] introduce a query language, the UnQL, based on structural recursion, where the data model used is defined by bisimulation. Gottlob, Koch and Pichler [8] examine the classes of the query language XPath with respect to complexity and parallelization. For an overview of complexity problems we recommend the classical work of Garey and Johnson [7] and the paper of Stockmeyer and Meyer [20].

The PT-algorithm was first published in the paper of Paige and Tarjan [18]. The 1-index based on bisimulations is discussed in detail by Milo and Suciu [17], where they also introduce the 2-index, and as a generalization of this, the T-index

The A(k)-index was introduced by Kaushik, Shenoy, Bohannon and Gudes [11]. The D(k)-index first appeared in the work of Chen, Lim and Ong [5]. The M(k)-index and the M*(k)-index, based on frequent queries, are the results of He and Yang [9]. FB-indexes of branching queries were first examined by Kaushik, Bohannon, Naughton and Korth [13].

The algorithms of the modifications of 1-indexes, FB-indexes and A(k)-indexes were summarized by Kaushik, Bohannon, Naughton and Shenoy [14]. The methods discussed here are improved and generalized in the work of Yi, He, Stanoi and Yang [23]. Polyzotis and Garafalakis use a probability model for the study of the selectivity of queries [19].

Kaushik, Krishnamurthy, Naughton and Ramakrishnan [12] suggest the combined use of structural indexes and inverted lists.

The book of Tucker [21] and the encyclopedia edited by Khosrow-Pour [15] deal with the use of XML in practice.

The theory of XML has no literature in Hungarian yet, but several books discuss its practical use [2, 3, 16].

Bibliography

[1] M. Arenas, L. Libkin. A normal form for XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems*, 2002, pp. 85–96. 947

[2] C. F. Bates. *XML in Theory and Praxis*. John Wiley & Sons, 2003. 948

[3] N. Bradley. *The XML Companion (3. edition)*. Addison-Wesley, 2004. 948

[4] P. Buneman, M. Fernandez, D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The International Journal on Very Large Data Bases*, 9(1):76–110, 2000. 947

[5] Q. Chen, A. Lim, K. W. Ong. An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 134–144. 947

[6] D. G. Corneil, C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, 1970. 947

[7] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 947

[8] G. Gottlob, C. Koch, R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd Symposium on Principles of Database Systems*, 2003, pp. 179–190. 947

[9] H. He, J. Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 683–694. 947

[10] M. R. Henzinger, T. A. Henzinger, P. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1995, pp. 453–462. 947

[11] R. Kaushik, R. Krishnamurthy, J. F. Naughton, R. Ramakrishnan. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*. 947

[12] R. Kaushik, R. Shenoy, P. F. Bohannon, E. Gudes. On the integration of structure indexes and inverted lists. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 779–790. 948

[13] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pp. 133–144. 947

[14] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth, P. Shenoy. Updates for structure indexes. In *Proceedings of Very Large Data Bases*, 2002, pp. 239–250. 947

[15] M. Khosrow-Pour (szerkesztő). *Encyclopedia of Information Science and Technology, Vol. 1, Vol. 2, Vol. 3, Vol. 4, Vol. 5*. Idea Group Inc., 2005. 948

[16] B. McLaughlin. *Java and XML*. O'Reilly, 2000. 948

[17] T. Milo, D. Suciu. Index structures for path expressions. Lecture Notes in Computer Science, Vol. 1540. Springer-Verlag, 1999, pp. 277–295. 947

[18] R. Paige, R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. 947

[19] N. Polyzotis, M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data*, 2002, pp. 358–369. 947

[20] L. Stockmeyer, A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM Press, 1973, pp. 1–9. 947

[21] A. Tucker. *Handbook of Computer Science*. Chapman & Hall/CRC, 2004. 948

[22] V. Vianu. A Web Odyssey: from Codd to XML. In *Proceedings of the 20th Symposium on Principles of Database Systems*, 2001, pp. 1–5. 947

[23] K. Yi, H. He, I. Stanoi, J. Yang. Incremental maintenance of XML structural indexes. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 491–502. 947

Subject Index

Name index

Contents