Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Ondřej Vošta

Automatická konstrukce schématu pro množinu XML dokumentů

Katedra softwarového inženýrství
Vedoucí diplomové práce: *RNDr. Irena Mlýnková*
Studijní program: *Informatika*

Na tomto místě bych rád poděkoval vedoucí své diplomové práce RNDr. Ireně Mlýnkové za cenné rady a náměty, které nezanedbatelným dílem přispěly k dokončení této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 15. prosince 2005

Ondřej Vošta

Obsah

Seznam diagramů

Název práce: *Automatická konstrukce schématu pro množinu XML dokumentů*
Autor: *Ondřej Vošta*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Irena Mlýnková*
E-mail vedoucí: *mlynkova@ksi.mff.cuni.cz*
Abstrakt:

Jazyk XML je stále důležitější formát pro uchování a výměnu dat. Pro korektní výměnu dat, jejich efektivní ukládání a zpracování je nutné znát jejich strukturu. Stále však existuje a je vytvářeno mnoho dokumentů bez popisu jejich struktury.

Tato práce se zabývá možností automatického generování schémat popisujících strukturu na základě dané množiny dokumentů. Prezentovaný algoritmus rozděluje elementy dokumentů do skupin podle podobnosti. Pro každou skupinu je vygenerován regulární výraz co nejlépe popisující strukturu vstupních elementů s využitím možností jazyka XML Schema.
Klíčová slova: *XML, XML Schema, odvozování schématu, odvozování regulárních výrazů z pozitivních příkladů, rozdělování XML elementů podle podobnosti*

Title: *Automatic construction of schema for given set of XML documents*
Author: *Ondřej Vošta*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Irena Mlýnková*
Supervisor's e-mail address: *mlynkova@ksi.mff.cuni.cz*
Abstract:

XML is still more and more important format for storing and exchanging data. In the face of this tendency, there are still lots of documents, and new ones are created, without any decribtion of their structure. However, for correct exchanging and efficient storing and quering of data, there is a weighty need for some structure description.

This thesis is focusing on a possibility to automatically generate such description from the set of given documents. The presented algorithm clusters document elements into groups according to the similarity of their structure. For each group is then infered a regular expression describes the structure of input elements using advantages of XML Schema language. Finally, all partial schemas are joined to the XML Schema document.
Keywords: *XML, XML Schema, infering schema, infering regular expressions from positive samples, clustering XML elements*

1 Úvod

Uchování dat, jejich výměna a efektivní zpracování jsou klíčovou oblastí moderní doby. Je přirozené a dnes i samozřejmé využívat pro správu dat a komunikaci možností jež nabízí výpočetní technika a počítačové sítě (zejména celosvětová sít internet). Pro prezentaci, sdílení a výměnu informací bylo v historii počítačů navrženo mnoho metod a jazyků. Za všechny uveďme patrně nejznámější jazyk HTML[1] . Tento jazyk slouží k prezentaci informací především prostřednictvím webového serveru a protokolu HTTP[2] . Postupem času se stala zřejmá tendence využití tohoto jazyka zejména k popisu vizuální prezentace dokumentu. Tím se ale začaly vytrácet už tak skromné sémantické prvky struktury dokumentu. S rozvojem výpočetní techniky, zejména sítě internet, a jejím rozšiřováním mezi stále větší množství uživatelů začalo enormně narůstat množství dostupných dokumentů v tomto jazyce. Navíc požadavky na vizuální formu dokumentu rostly společně s možnostmi zobrazovacích zařízení uživatelů a tak se stávají dokumenty v tomto jazyce stále složitější, stále méně přehledné a objem užitečných informací v poměru k celkovému objemu dokumentu stále klesá. Nejdůležitějším nedostatkem jazyka HTML pro potřeby elektronické komunikace a správy informací je pevná množina vyjadřovacích prostředků a tím nízká rozšiřitelnost a flexibilita pro konkrétní použití.

V reakci na tyto potřeby a nedostatky vznikl jazyk XML[3] . Jazyk XML je stejně jako jazyk HTML podmnožinou obecného značkovacího jazyka SGML[4] . Narozdíl od jazyka SGML je XML navržen s ohledem na jednoduchost. Z uživatelského hlediska je snadné se tomuto jazyku naučit a také elektronické zpracování je snadné a efektivní. XML dokumenty je tak možné zpracovávat i v mobilních zařízeních a dalších zařízeních s velmi omezenou výpočetní kapacitou. Na druhé straně oproti jazyku HTML se XML soustřeďuje zejména na popis struktury informací a jejich sémantického významu. Protože navrhnout společný jazyk pro významový popis informací ze všech oborů lidské činnosti by bylo zhola nemožné, je jazyk XML vytvořen jako rozšiřitelný standard. Každý si tak může vytvořit prostředky pro popis informací ze svého oboru (problémové domény). Avšak aby dva subjekty mohly mezi sebou komunikovat a vyměňovat si informace, je nezbytně nutné, aby oba dodržovaly společnou strukturu dokumentů a společnou množinu výrazových prostředků. Za tímto účelem bylo vytvořeno mnoho standardů pro popis schématu dokumentů. Jedním z nich je jazyk XML Schema. Schéma dokumentu nejen umožňuje komunikaci mezi subjekty, ale zároveň umožňuje mnohem efektivnější ukládání těchto dokumentů,

[1] HyperText Markup Language - http://www.w3.org/TR/REC-html40/
[2] HyperText Transfer Protocol
[3] eXtensible Markup Language - http://www.w3.org/TR/REC-xml/
[4] Standardized General Markup Language

případně jejich kompresi anebo vyhledávání informací v bázích dat založených na dokumentech v jazyce XML.

Ne vždy však existují k dokumentům schémata. V jiné situaci sice máme schéma dokumentů, ale toto schéma nám z různých důvodů nevyhovuje. Samozřejmě, abychom mohli psát dokumenty podle určitého schématu, musíme toto schéma nejdříve vytvořit. Ve všech těchto případech je pak potřeba, aby člověk nějakým způsobem schéma dokumentu změnil, zpětně odhadl nebo jinak vytvořil. Zvláště u složitých dokumentů to může být značně náročný úkol. A tak je nasnadě vytvořit algoritmy, které tuto činnost usnadní, ne-li ji plně zautomatizují. Přirozeným jazykem, jak popsat počítači požadované vlastnosti dokumentů, které chceme vytvářet či zpracovávat, je poskytnout několik příkladů. A tak, abychom usnadnili reverzní získávání, případně vytváření zcela nových schémat, potřebujeme vytvořit metody, jak z dokumentů zpětně odhadnout jejich strukturu. Důležité je, že máme k dispozici pouze dokumenty, které odpovídají budoucímu schématu a nemáme žádné dokumenty, které naopak budoucímu schématu nesmějí odpovídat. Takovéto odvozování schématu je otázka vhodného zobecnění vstupních příkladů. Pokud zobecníme příklady málo, nebude nám vytvořené schéma dostačovat. Pokud příklady zobecníme příliš bude výsledné schéma postrádat důležité detaily struktury dokumentů.

1.1 Cíl práce

Cílem této diplomové práce je navrhnout algoritmus pro reverzní získání schématu z příkladů dokumentů a toto schéma popsat v jazyce XML Schema. Základem práce je navrhnout vhodný algoritmus pro odvozování regulárních výrazů popisujících strukturu dokumentu s využitím možností jazyka XML Schema. Druhým dílčím cílem je využít dalších vlastností jazyka XML Schema tak, aby bylo vygenerováno co nejkvalitnější schéma dokumentů. Poslední částí práce je ukázková implementace navržených algoritmů. Při implementaci je využito již implementovaných knihoven funkcí pro práci s XML dokumenty.

1.2 Přehled práce

V úvodu je popsána motivace pro vytvoření práce a jsou navrženy cíle, kterých chceme dosáhnout. Ve druhé kapitole jsou definovány základní pojmy používané ve zbytku práce. V této kapitole jsou také stručně popsány základní technologie, které jsou v práci použity a je na ně odkazováno. Zejména se jedná o jazyk XML, jazyk pro popis struktury XML dokumentů XML Schema a základy z teorie jazyků, automatů a gramatik. Ve třetí kapitole jsou ukázány jednotlivé úhly pohledu jak lze řešení v této oblasti charakterizovat. Ve čtvrté kapitole pak popíšeme existující metody a přístupy k řešení této problematiky. V páté kapitole navrhneme a podrobně popíšeme algoritmy pro zpracování

XML dokumentů. Šestá kapitola popisuje architekturu a implementaci navrženého řešení. Sedmá kapitola obsahuje experimenty a praktické ukázky práce navrženého řešení. Poslední osmá kapitola obsahuje závěrečné hodnocení řešení a možnosti dalšího vývoje.

2 Použité technologie

2.1 XML - eXtensible Markup Language

Jazyk XML je obecný standardizovaný jazyk pro popis strukturovaných dat. Tento jazyk byl standardizován W3 konzorciem[5] a je podmnožinou obecného značkovacího jazyka SGML. Výhodou tohoto jazyka je jednoduchost, přísná syntaxe a flexibilita. Je určen pro publikování a výměnu dat, ale i pro jejich ukládání a vyhledávání. Jazyk XML je v podstatě metajazyk. Předepisuje pouze syntaxi. Jednotlivé značky už jsou závislé na konkrétních datech a použití.

Elementy

Základním stavebním kamenem jazyka jsou elementy. Každý element se skládá většinou ze dvou značek - otevírací a uzavírací, z atributů a z obsahu. Obsahem elementů mohou být jak data tak další elementy. Značky elementu jsou uzavřeny mezi znaky < a >. Koncová značka začíná znakem / a pokračuje stejným názvem jako otevírací značka. Prázdný, tedy nepárový element (element, který nemá otevírací a uzavírací značku, ale pouze jedinou značku) má znak / před uzavíracím znakem >.

```
<osoba>
  <jméno>Jan</jméno>
  <příjmení>Novák</příjmení>
  <zaměstnanec />
</osoba>
```

Každý element je buď párový, pak má obě značky (otevírací i uzavírací) a je mezi nimi celý uzavřen, anebo je nepárový, pak má pouze jedinou značku a nemá žádný obsah. Dvojice otevíracích a uzavíracích značek musí tvořit správné uzávorkování. Každý XML dokument musí mít právě jeden kořenový element, tedy celý dokument musí být uzavřen v jediném elementu, jehož potomky jsou všechny ostatní elementy. O správně uzávorkovaném dokumentu s jedním kořenovým elementem říkáme, že je správně strukturovaný (well-formed).

Pokud navíc dodáme dokument spolu s popisem jeho struktury, říkáme že je dokument validní. Respektive dokument je validní vůči danému schématu, pokud mu svou strukturou a obsahem vyhovuje. Struktura dokumentu popisuje přípustné názvy elementů, přípustný obsah, atributy a povolené vnořování elementů do sebe. K popisu struktury dokumentů je možné využít například jazyků jako je DTD či XML Schema.

[5] World Wide Web consortium - http://www.w3.org

Atributy

Každý element může obsahovat atributy. Atributy jsou součástí značky elementu. V případě párových elementů jsou součástí jejich otevírací značky. Atributy mají tvar název="hodnota", kde hodnota atributu je vždy uzavřena do uvozovek.

```
<osoba id="1684">
  <jméno>Jan</jméno>
  <příjmení>Novák</příjmení>
  <zaměstnanec plat="50000"/>
</osoba>
```

Schéma dokumentu definuje nejen povolené atributy, ale i jejich volitelnost a v případě některých jazyků (jako například XML Schema) i datový typ atributu.

Instrukce pro zpracování

Instrukce pro zpracování jsou instrukce pro nadřazený program, který zpracovává dokument. Pro data v dokumentu nemají žádný význam, pouze ovlivňují způsob zpracování dokumentu. Jsou to příkazy ve tvaru <?identifikátor data?>. Asi nejdůležitější instrukcí pro zpracování je deklarace XML dokumentu. Tato instrukce by měla být umístěna na začátku dokumentu, před všemi ostatními výrazy jazyka. Má tvar <?xml version="1.0" encoding="UTF-8" standalone="no"?>. První atribut určuje použitou verzi jazyka (v dnešní době pouze verze 1.0), druhý atribut encoding obsahuje informaci o znakové sadě a poslední atribut standalone určuje, zda může být dokument zpracováván samostatně, nebo vyžaduje externí definice.

Entity

Entity jsou zástupné artefakty jazyka, které jsou při zpracování nahrazeny svým obsahem. Pokud například v našem firemním dokumentu stále používáme dlouhý oficiální název naší společnosti, můžeme si nadefinovat entitu logo jako "Velmi dlouhý název společnosti spol. s r.o.", a pak v dokumentu používat pouze &logo;. Tato entita pak bude při zpracování dokumentu na každém místě, kde se vyskytuje, nahrazena definovaným textem. Entity tedy mají tvar &název;. Definice entity pak vypadá <!ENTITY název "text">. V XML jsou 3 předdefinované entity pro znaky, které mají speciální význam. Jsou to < pro znak <, > pro znak > a & pro znak &.

Komentáře

Komentář je text uzavřený mezi znaky <!-- a -->. Vše co je mezi těmito dvěma značkami nepodléhá zpracování dokumentu.

CDATA

Uvnitř sekce CDATA neprobíhá zpracování dokumentu a obsah této sekce je bez zpracování poslán na výstup. Uvnitř této sekce pozbývají znaky <, > a & svůj speciální význam. Sekce je ohraničena značkami <![CDATA[a]]>. Tato sekce je vhodná pro případ, kdy potřebujeme do dokumentu vložit větší blok textu, v němž nechceme provádět zpracování (obsahuje speciální znaky, ale ve zcela jiném významu).

Podrobný popis jazyka XML je možné nalézt na stránkách konzorcia W3 [1].

2.2 XML Schema

XML Schema je jazyk pro popis struktury XML dokumentů. Je možné popsat platné elementy, pro každý element povolené atributy (včetně volitelnosti a datového typu) a vztahy rodič-potomek. Umožňuje vytvářet uživatelské datové typy, určit pořadí podelementů i počet opakování podelementů. Umožňuje specifikovat, zda má element obsah a jakého je typu. Pro atributy je možné nastavit implicitní hodnoty. Jazyk XML Schema je stejně jako jazyk XML standardizován w3 konzorciem.

Jmenné prostory

Jeden dokument může obsahovat značky z několika schémat. Aby se tvůrci schémat nemuseli složitě domlouvat na jménech a nedocházelo ke kolizím názvů elementů a atributů, byly vytvořeny tzv. jmenné prostory. Každé schéma tak definuje elementy v určitém jmenném prostoru. V rámci jednoho jmenného prostoru už musí být všechny názvy unikátní (názvy atributů samozřejmě pouze v kontextu svého elementu). Každý jmenný prostor je jednoznačně identifikován svým URI[6] . Jmenný prostor elementu a všech jeho podelementů nadefinujeme pomocí atributu xmlns:prefix="URI jmenného prostoru". Pro použití elementu z definovaného jmenného prostoru pak píšeme <prefix:lokální-název-elementu .....>, pro atribut píšeme prefix:lokální-název-atributu="hodnota". V každém dokumentu je možné definovat implicitní jmenný prostor. Tento je s prázdným prefixem a není potřeba prefix před elementy ani atributy psát. Implicitní jmenný prostor definujeme takto: xmlns="URI implicitního jmenného prostoru".

Připojení schématu k dokumentu

Připojení schématu k dokumentu provedeme uvedením atributu schemaLocation v definici elementu, kde je definován jmenný prostor. Obsah atributu schemaLocation je mezerou oddělený seznam dvojic URI-jmenného-prostoru umístění-schématu. Položky dvojice jsou znovu odděleny znakem mezera. Pokud

[6] Uniform Resource Identifier

připojované schéma nemá definován jmenný prostor, připojíme jej specifikováním atributu `noNamespaceSchemaLocation`, jehož obsahem je umístění schématu. Ke každému dokumentu můžeme připojit pouze jediné schéma bez definovaného jmenného prostoru (jinak by mohlo docházet ke kolizím jmen v dokumentu). Všechny názvy ze schématu bez definovaného jmenného prostoru jsou uváděny bez prefixu, tedy jakoby se jednalo o implicitní jmenný prostor.

Syntaxe

Jazyk XML Schema je konkrétním použitím jazyka XML. Pro popis schématu XML dokumentu tedy není potřeba zavádět další jazyk. Samozřejmě existuje pro jazyk XML Schema definice struktury pro XML dokumenty jazyka XML Schema. Protože se jedná o XML dokument, má každé schéma právě jeden kořenový element a tím je element `schema`. Stejně jako všechny ostatní elementy XML Schema dokumentu, je i tento element ze jmenného prostoru s URI `http://www.w3.org/2001/XMLSchema`.

Elementy

Pomocí značky `element` definujeme povolené elementy v dokumentu. Pokud je značka `element` přímým potomkem elementu `schema`, pak definuje globální element. Globální element je viditelný z celého schématu, a je ho také možné použít jako kořenový element dokumentu. Pokud je značka `element` uvedena v rámci definice komplexního typu, jedná se o lokální element. Narozdíl od DTD umožňuje jazyk XML Schema definovat několik různých kořenových elementů. Atribut `name` určuje název elementu. Nejdůležitější definicí v rámci elementu je definice jeho typu. Typ může být buď jednoduchý nebo komplexní. Typ můžeme definovat buď pomocí atributu `type` nebo jako vnořený element.

```
<xs:element name="osoba">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="jméno" type="TJméno" />
      <xs:element name="příjmení" type="TJméno" />
      <xs:element name="zaměstnanec" type="TZaměstnanec" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Atributy

Každému elementu můžeme definovat množinu povolených atributů. Jednotlivé atributy definujeme pomocí značky `attribute`. Název atributu určíme atributem `name`. Výskyt atributu můžeme upravit uvedením atributu `use`. Ten nabývá hodnot optional (volitelný atribut), prohibited (zakázaný atribut), required (povinný atribut). Stejně jako u elementů musíme specifikovat datový typ atributu. To provedeme stejně jako u elementu buď uvedením atributu `type` nebo specifikováním typu jako vnořeného elementu. Atributem `ref` je možné se odkázat na globálně definovaný atribut a atribut `default` určuje implicitní hodnotu atributu.

```
<xs:complexType name="TZaměstnanec" >
  <xs:attribute name="plat" type="xs:positiveInteger" />
</xs:complexType>
```

Jednoduché typy

Jednoduchý datový typ nesmí obsahovat elementy ani atributy. Základním typem v XML Schema je typ string. XML Schema dále umožňuje odvozování typů pomocí omezení nadřazeného typu. Jednoduché typy je možné rozdělit na vestavěné a uživatelsky definované.

Komplexní typ

Komplexní typ definujeme elementem complexType. Komplexní typ obsahuje množinu definic atributů a definici jednoduchého nebo složeného obsahu. Jednoduchý obsah je omezení jednoduchého typu nebo rozšíření jednoduchého typu o atributy. Složený obsah je omezením nebo rozšířením komplexního typu.

- **Sequence** je datový typ určující sekvenci položek v pevném pořadí, tedy všechny položky se musí vyskytovat (v závislosti na jejich vlastním určení počtu výskytů) a to v pořadí přesně jak je určeno elementem `sequence`. Obsahem `sequence` může být `element`, `choice`, `sequence` nebo `all`
- **Choice** je datový typ pro exkulzivní výběr položky z množiny. Výsledkem je tedy právě jedna položka z dané množiny. Obsahem `choice` může být `element`, `choice`, `sequence` nebo `all`
- **All** je datový typ pro výskyt všech elementů z množiny maximálně jednou, ale v libovolném pořadí. Tento element je rozšířením oproti DTD a může výrazně přispět ke zpřehlednění některých schémat. Obsahem `all` může být pouze `element`. Toto omezení je přijato k zajištění deterministického datového modelu.

Například typu

```
<xs:all>
  <xs:element name="A" type="TA" />
  <xs:element name="B" type="TB" />
</xs:all>
```

odpovídají sekvence elementů

```
<A /><B />  i <B /><A />
```

Další možnosti XML Schema

Další vlastnosti jazyka XML Schema jako jsou substituce, skupiny, unikátní hodnoty atributů, klíče a cizí klíče, značka pro libovolný element či atribut, notace a další, nejsou z hlediska naší práce s XML Schema zajímavé a proto se zde o nich nebudeme zmiňovat. Podrobná specifikace jazyka je k nalezení na stránkách w3 konzorcia [2].

Srovnání s DTD

V tomto odstavci jsou vyzdvihnuta důležitá vylepšení jazyka XML Schema oproti DTD. Z hlediska této práce je asi nejdůležitějším rozšířením zavedení typu `all` pro výskyt elementů v libovolném pořadí. Z hlediska regulárních jazyků se nejedná o rozšíření jejich vyjadřovací síly, ale z uživatelského hlediska je to prvek umožňující zjednodušení zápisu některých schémat. Další důležité rozšíření je možnost definice různé struktury (typu) pro elementy stejného jména za předpokladu, že se nacházejí zcela jinde ve struktuře dokumentu. Posledním rozšířením výnamným z hlediska této práce je možnost definovat více kořenových elementů v rámci schématu (každý dokument pak má jako kořenový element libovolný z nich, avšak vždy právě jeden).

2.3 Automaty a gramatiky

Některé metody automatického generování schémat stavějí na teorii jazyků. Proto zde uvedeme základní definice z této oblasti.

Abeceda, slovo a jazyk

Abecedou můžeme nazvat libovolnou konečnou množinu symbolů. Mějme tedy abecedu Σ. Slovo nad touto abecedou je libovolná sekvence symbolů z abecedy. Délka slova je počet symbolů v sekvenci. Speciálním symbolem λ označíme prázdné slovo. Tedy slovo, které neobsahuje žádné symboly, slovo délky 0. Jazyk je libovolná, potenciálně nekonečná, množina slov nad danou abecedou. Symbolem Σ^* označíme množinu všech slov nad danou abecedou. Symbolem Σ^+ označíme množinu všech neprázdných slov nad danou abecedou.

Gramatika

Gramatika G je čtveřice G=(S,T,V,R), kde S je počáteční symbol, T je množina terminálů (abeceda), V je množina neterminálů a R je sada přepisovacích pravidel. Pravidla z R pak mají podobu

$$(V \cup T)^* \rightarrow (V \cup T)^* \text{ , kde na levé straně pravidla je alespoň 1 neterminál}$$

Aplikování pravidla na řetězec terminálů a neterminálů znamená, že nahradíme jeden výskyt levé části pravidla jeho pravou částí. Gramatika G generuje slovo w, pokud existuje posloupnost pravidel taková, že začneme počátečním symbolem S a postupným aplikováním pravidel z posloupnosti vytvoříme slovo w.

Gramatiky můžeme rozdělit podle složitosti pravidel do několika tříd.

- **Obecné gramatiky** (třída L_0) jsou libovolné gramatiky odpovídající výše uvedené definici.
- **Kontextové gramatiky** (třída L_1) jsou gramatiky, jejichž pravidla odpovídají následujícímu omezení

  $$uXw \to uYw \text{ , kde } u, w \in (T \cup V)^*, X \in VaY \in (T \cup V)^+$$

  Tedy neterminál X v kontextu slov u, w přepíšeme na neprázdnou posloupnost terminálů a neterminálů. Pokud se počáteční neterminál S nevyskytuje na pravé straně žádného pravidla, může gramatika ještě obsahovat speciální pravidlo pro přepis neterminálu S na prázdné slovo.
- **Bezkontextové gramatiky** (třída L_2) jsou gramatiky, jejichž pravidla mají na levé straně vždy právě jeden neterminál.
- **Regulární gramatiky** (třída L_3) jsou gramatiky, jejichž pravidla odpovídají následujícímu omezení:

  $$X \to uY \text{, kde } X \in V, u \in T^* \text{ a } Y \in V \text{ nebo } Y = \lambda$$

  Tedy na levé straně je vždy právě jeden neterminál a na pravé straně je posloupnost terminálů následovaná volitelně jedním neterminálem.

Konečný automat

Konečný automat A je pětice $A = (\Sigma, Q, q_0, F, d)$, kde Σ je abeceda, Q je množina stavů, q_0 je počáteční stav, F je množina koncových stavů a d je přechodová funkce. Přechodová funkce d je $d : Qx\Sigma \to Q$, tedy funkce projektující stav a symbol na nový stav. Konečný automat se při své práci nachází vždy v právě jednom stavu (na počátku se nachází v počátečním stavu q_0). Jeden krok automatu vypadá takto: automat si přečte 1 symbol ze vstupu a na základě aktuálního stavu a přečteného symbolu změní pomocí přechodové funkce svůj stav. Jakmile automat dosáhne koncového stavu, ohlásí to na výstup. Řekneme, že automat A akceptuje slovo $w = s_1...s_n$, jestliže existuje posloupnost stavů $q_1...q_n$ taková, že $d(q_0, s_1) = q_1, ..., d(q_{n-1}, s_n) = q_n$ a stav q_n je koncový. Automat A akceptuje jazyk L, jestliže akceptuje všechna slova jazyka a neakceptuje žádné slovo, které v jazyce L není. Konečné automaty akceptují právě třídu regulárních jazyků.

Cesta v automatu (analogicky i v gramatice) je posloupnost $q_{a(0)}$, $e_{a(0)}$, $\ldots$, $e_{a(k)}$, $q_{a(k+1)}$, kde $q_{a(i)}$ jsou stavy automatu a $e_{a(i)} \equiv$ d($q_{a(i)}$,$s_{a(i+1)}$)=$q_{a(i+1)}$) jsou přechody automatu ($s_{a(i)}$ jsou symboly z abecedy Σ). Slovo odpovídající cestě je právě sekvence symbolů $s_{a(i)}$. Pokud máme deterministický automat a je dán počáteční stav cesty, pak je cesta jednoznačně určena posloupností symbolů.

Regulární výraz

Regulární výraz je řetězec nad abecedou $\Sigma \cup \{?, +, *, |, (,)\}$, tedy zavedeme do abecedy další speciální symboly. Symbol `*` znamená libovolné opakování předcházející skupiny symbolů. Tedy výrazu `a*` odpovídají slova `a`, `aa`, `aaaaaaa`, ale i prázdné slovo λ. Symbol `?` značí volitelný výskyt předcházející skupiny, například výrazu `xy?` odpovídají slova `x` a `xy`. Symbol `+` značí opakování předchozí skupiny, ale narozdíl od symbolu `*` se musí skupina vyskytovat alespoň jednou. Například výrazu `u+v` odpovídají slova `uv`, `uuuuv` ale ne slovo `v`. Symbol `|` značí výběr jedné skupiny. Například výrazu `p|q` odpovídají slova `p` a `q`. Nakonec symboly `(` a `)` uzavírají skupinu znaků. Například výrazu `(ab)+` odpovídají slova `ab`, `ababab` nebo například `ababababab`. Regulární výrazy je možné spojovat za sebe, spojovat pomocí symbolu `|` nebo vnořovat do sebe. Příkladem složitějšího regulárního výrazu může být výraz `((x|y|z1?)+u?vw*)*`. Regulární výrazy popisují regulární jazyky. Jsou tedy ekvivalentní gramatikám třídy L_3 a konečným automatům.

Rozšířený konečný automat

Rozšířený konečný automat je obdoba konečného automatu s tím rozdílem, že funkce d je definována jako $d : Qx(\text{regulární výrazy nad } \Sigma) \rightarrow Q$. Tedy přechod automatu není podmíněn jediným symbolem, ale regulárním výrazem nad příslušnou abecedou.

Prefixový strom

Mějme vstupní sekvence terminálů. Prefixový automat (strom) je pak takový deterministický konečný automat, který akceptuje přesně vstupní sekvence a cesty v automatu začínající v počátečním stavu a odpovídající dvěma různým sekvencím mají společné právě ty stavy, které odpovídají společnému prefixu sekvencí a žádné jiné.

3 Požadavky různých aplikací

Aplikací, které by mohly využít automatické nebo alespoň částečně automatické generování schémat XML dokumentů, je nepřeberné množství. Jednou z oblastí, kde je možné takové algoritmy využít, je design případně redesign schémat dokumentů. Tedy situace, kdy je člověk postaven před úkol vytvořit společné schéma pro dokumenty, které budou poté psát další lidé. Zvláště u velmi (co se týká struktury) rozsáhlých dokumentů, může být pojmutí a vytvoření celé šíře struktury dokumentů pro člověka velmi náročný úkol. Velkým usnadněním by jistě bylo, kdyby mohl designér sepsat několik vzorových dokumentů, které by program zpracoval a navrhl schéma. Schéma by pak designér prošel, a buď by ho ručně upravil, nebo měnil vzorové dokumenty, dokud by nebyl spokojen.

Jiným druhem využití automatického generování struktury dokumentů může být případ, kdy sice existuje k daným dokumentům popis schématu, ale toto schéma je příliš rozsáhlé a vytvořené aby pokrývalo mnohem větší šíři dokumentů. Úkolem tedy je vytvořit nové schéma, jako podmnožinu stávajícího, které bude s větší přesností popisovat danou podmnožinu dokumentů. Tedy všude kde je potřeba vytvářet pohledy a poddokumenty. Na druhou stranu se můžeme v praxi setkat s přesně opačným problémem. Máme několik různých zdrojů dokumentů a ty potřebujeme sjednotit pod jediné obecnější schéma.

Aplikací, které potřebují automaticky generovat schémata XML dokumentů, je velké množství. A samozřejmě platí, že každá aplikace má trochu jiné požadavky na použitý algoritmus. Některé aplikace potřebují výsledky ihned, jiné mohou čekat na složitější výpočet. Některé aplikace potřebují jako výsledek hotové a kompletní schéma, u jiných aplikací stačí určité nekompletní řešení. U algoritmů tedy můžeme sledovat tyto vlastnosti ([3]):

- **Komplexní struktura dokumentů:** Pokud je struktura vstupních dokumentů jednoduchá, je automatické generování snadné a výsledky jsou dobré. Avšak většinou je potřeba generovat schéma k dokumentům, jejichž struktura je v jistém smyslu komplexní.
- **Rychlost:** Pokud máme on-line aplikaci, kde uživatel na výsledek čeká, je potřeba jej dodat v co možná nejkratším čase. Naopak pokud navrhujeme nové schéma pro budoucí dokumenty nebo zpracováváme archiv, nehraje čas až takovou roli.
- **Kompletnost výsledku:** V některých případech, převážně pokud je výsledek programu pouze mezivýsledkem a předává se k dalšímu zpracování například člověkem, není nezbytně nutné dodávat kompletní schéma, ale pouze část, kostra nebo jiné částečné řešení je dostatečné. Většinou však potřebujeme kompletní výsledek a nejlépe korektní DTD, XML

Schema nebo jiný dokument popisující schéma (validní ve smyslu, že odpovídá všem nárokům kladeným na daný typ dokumentu příslušnou normou nebo doporučením, kterým se řídí práce s tímto typem dokumentu).

- **Podrobnosti struktury:** Určitý typ aplikací potřebuje prostě "nějaké schéma" a i triviální řešení je dostatečně dobré. Vždy k dokumentům existují 2 triviální schémata. První je schéma, které akceptuje přesně vstupní dokumenty (bez jakéhokoliv zobecnění). Protipólem je pak schéma, které jednoduše akceptuje jakýkoliv dokument (případně pouze nad danou množinou elementů). Většinou je však potřeba najít rozumný kompromis mezi těmito dvěma póly. Schéma dostatečně zobecněné, aby akceptovalo všechny dokumenty, které je potřeba a zároveň schéma, které dostatečně popisuje dokumenty a zachovává detaily jejich struktury.
- **Čitelnost:** Tato vlastnost určuje jistou kompaktnost a výstižnost schématu. Někdy je potřeba vytvořit maximálně detailní schéma s minimem zjednodušení. Někdy je vhodnější vytvořit obecnější schéma s menším počtem detailů, které ale bude malé a výstižné. Například pokud bude výsledek číst člověk, je pro něj mnohem užitečnější, pokud je schéma vytvořeno podle druhé varianty.
- **Změna vstupních dat:** V některých případech je možné, abychom na základě výsledku algoritmu změnili strukturu vstupních dokumentů. Taková situace nastává například při návrhu nového schématu. Jindy máme naopak na vstupu dokumenty tak jak jsou a musíme výsledek absolutně přizpůsobit těmto dokumentům.
- **Interaktivní zpracování:** Specifický typ aplikací nám umožňuje komunikovat s uživatelem. Můžeme tak uživateli předložit mezivýsledky a umožnit mu vhodným způsobem zasáhnout do procesu zpracování. V ostatních aplikacích je naopak nutné zpracovat vstupy zcela automaticky, v dávkovém režimu a bez vnějšího zásahu.

Je jistě zřejmé, že mnohé vlastnosti se navzájem ovlivňují. Nejdůležitější je asi vztah mezi komplexní strukturou dokumentů, kompletností výsledku, podrobností struktury a čitelností. Tedy pokud máme na vstupu dokumenty se složitou komplexní strukturou a chceme obdržet kompletní výsledek, pak není možné dostat kompaktní a plně podrobnou strukturu.

Zároveň existují případy, kdy je vygenerování schématu snadné a ostatní vlastnosti nehrají roli. Pokud se spokojíme s jedním z triviálních řešení co se týká podrobnosti struktury, pak vygenerování takového schématu je rychlé, výsledek bude kompletní a dobře čitelný pro uživatele a není potřeba měnit strukturu vstupních dokumentů. Podobně pokud je struktura dokumentů na vstupu jednoduchá, je i výsledek jednoduchý, čitelný, kompletní a není potřeba měnit strukturu vstupních dokumentů.

4 Existující řešení

4.1 Rozdělení

Z hlediska přístupu a metodiky je možné existující řešení problému automatického generování schémat rozdělit do několika skupin. Asi nejzajímavější je rozdělení podle povahy výsledku a způsobu jeho konstrukce, které je zde uvedeno. Mimo to je ještě v literatuře možné nalézt skupinu algoritmů - tzv. merging states algorithms, které pro reprezentaci hledaného jazyka využívají konečný automat a zobecňování jazyka provádějí slučováním stavů tohoto automatu (zmíněno například v [1]).

- **Heuristická řešení:** Tato řešení jsou obvykle založena na empirických zkušenostech s tvorbou schémat "ručně" - člověkem. Jejich zřejmou výhodou je motivace a řešení z praxe, takže se na praxi soustřeďují a snaží se podávat co nejlepší výsledky na praktických aplikacích. Heuristická řešení můžeme ještě rozdělit na dvě podskupiny.
- **Odvozování gramatiky:** Tato řešení jsou postavena na teoretických základech a jako taková poskytují jistou garanci výsledků.

Heuristická řešení jsou heuristická v tom smyslu, že výsledky, které vytvářejí nepatří do žádné dané třídy gramatik. Obvykle jsou založeny na zobecňování triviálního schématu pomocí různých více či méně empirických pravidel, která by pravděpodobně použil člověk postavený před úkol "vhodně zobecnit dané schéma". Příkladem takového pravidla může být například: "pokud se ve schématu vyskytuje posloupnost více jak 3 výskytů stejného elementu pak je nanejvýš pravděpodobné, že se zde může vyskytovat libovolný počet opakování tohoto elementu". Heuristická řešení můžeme ještě rozdělit na dvě skupiny. První skupina přímo zobecňuje triviální schéma tak dlouho, dokud nedosáhne vhodného výsledku. Druhá skupina z triviálního schématu vygeneruje velké množství možných řešení. Tato řešení pak ohodnotí a vybere to nejlepší. První skupina přirozeně odpovídá lidskému postupu při řešení tohoto problému a je mnohem častější. Druhá metoda trpí výrazně většími nároky, protože obvykle generuje velké množství potenciálních řešení a pak každé jednotlivě hodnotí. Navíc musí metoda hodnocení zohlednit dva protichůdné faktory - obecnost schématu a jeho složitost, což je už z principu hodnocení velmi špatně definovatelné. Naopak rozhodování první skupiny je mnohem náročnější a chybný krok může způsobit, že se už nedostaneme k optimálnímu řešení.

Metody využívající odvozování gramatiky jsou postaveny na dobrých teoretických základech teorie jazyků. Ačkoliv gramatika popisující jakýkoliv XML dokument (XML jazyk) patří do třídy bezkontextových gramatik v Chomského hierarchii [12], lze tento problém zredukovat na odvozování několika oddělených regulárních jazyků. Toto zjednodušení je vysvětleno v sekci XML jazyky. I přes toto zjednodušení, je takový problém neřešitelný. V sekci Identifikovatelnost jazyků je definována identifikovatelnost jazyka[6] a uvedeme Goldovu větu [6], podle které nejsou regulární jazyky identifikovatelné pouze pomocí pozitivních příkladů. Tedy pomocí množiny instancí daného regulárního jazyka. Metody odvozující gramatiku tedy využívají dalších znalostí o požadovaném výsledku pro vyřešení tohoto problému. Tyto "další" znalosti mohou být různých druhů. Například může uživatel zadat maximální počet uzlů konečného automatu akceptujícího daný jazyk, algoritmus se může uživatele dotazovat na (ne)vhodnost výsledku anebo je výsledná třída jazyků omezena z regulárních na některou jejich identifikovatelnou podtřídu.

4.2 Heuristické metody

Jak již bylo předznamenáno výše, tyto metody jsou založeny na empirických pravidlech pro konstrukci, případně ohodnocení, výsledného schématu. Z povahy těchto algoritmů je dáno, že jejich výsledky nepatří do žádné třídy jazyků, narozdíl od metod založených na odvozování gramatiky. Metody obvykle triviálně zachytí strukturu daných vzorů dokumentů a tyto schémata různě kombinují dohromady a zobecňují, právě na základě daných pravidel.

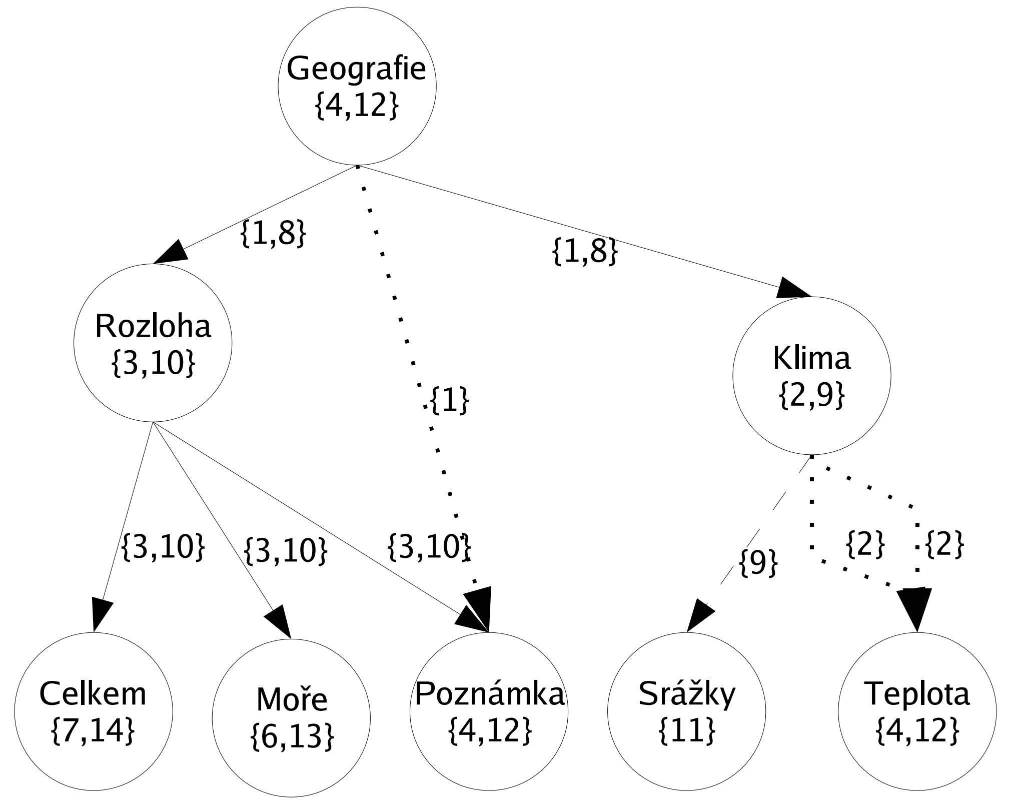
4.2.1 Metody přímo konstruující výsledek

4.2.1.1 DTD-Miner

Tato metoda, uvedená v [7], zpracovává vstupy ve třech fázích. V první fázi zkonstruuje pro každý vstupní dokument strom popisující strukturu daného dokumentu. V následující fázi spojí stromy všech vstupních dokumentů do jediné struktury, která popisuje strukturu všech vstupních dokumentů současně a přirozeně odpovídá požadovanému výsledku (schématu množiny příkladů, DTD). V poslední fázi se za pomoci optimalizačních pravidel generuje výsledné schéma v podobě DTD.

V první fázi zkonstruujeme pro každý vstupní dokument strom dokumentu. Uzly stromu reprezentují značky v dokumentu (výskyty jednotlivých elementů), hrany pak reprezentují vztah značek rodič-dítě, tedy vztah zanoření. Navíc ještě každému uzlu přiřadíme globálně jednoznačný identifikátor. Tento identifikátor musí být jednoznačný v rámci všech stromů vstupních dokumentů.

Pro interpretaci struktury všech dokumentů je použit orientovaný acyklický graf, který je nazýván spanning graf (termín je převzaný z původního



1: Příklad zkonstruovaného spanning grafu

textu. Překlad do češtiny by měl být kostra grafu, ale v tomto kontextu se nejedná o kostru grafu jak je definována v teorii grafů).

Definice Spanning graf: *Spanning graf je graf* $G = (V, E)$*, kde* $V = \{N :$ *v některém vstupním dokumentu existuje element s názvem* $N\}$ *a* $(A, B) \in E$*, pokud v některém dokumentu existuje element s názvem* B*, který je přímým potomkem elementu s názvem* A*.*

Každý uzel reprezentuje právě jeden druh elementu vyskytujícího se v některém vstupním dokumentu. Každý uzel má tedy jednoznačně přiřazený název elementu, který reprezentuje. Hrana mezi uzly pak reprezentuje relaci vnoření v některém vstupním dokumentu. Ke každému uzlu navíc udržujeme seznam uzlů ze stromů dokumentů, které se na tento uzel promítají (jsou daného druhu). Dále u každého uzlu udržujeme seznam atributů vyskytujících se u daného elementu (spolu s atributem udržujeme také informaci, jestli je atribut volitelný, případně také datový typ atributu). U hran navíc udržujeme seznam rodičovských uzlů, z nichž vedly hrany ve stromech dokumentů, které se spojily v tuto výslednou hranu celkového grafu.

```
procedure SlitíDokumentu(dokument,spangraf)
  return LCS(dokument.root,spangraf.root)
end procedure

procedure LCS(uzelDokumentu,uzelGrafu)
  uzelGrafu.seznamUzlů.přidej(uzelDokumentu)
  uzelGrafu.seznamAtributů.přidej(uzelDokumentu.atributy)
  seznam1 = uzelDokumentu.potomci
  seznam2 = uzelGrafu.potomci
  LCS_1,...,LCS_k = SpolečnéPodsekvence(seznam1, seznam2)
  docPozice = uzelDokumentu.prvníPotomek
  grafPozice = uzelGrafu.prvníPotomek
  for i=1 to
k while docPozice ≠ od LCS_i.prvníUzel
      vložPotomkaPřed(uzelGrafu, docPozice, grafPozice)
      docPozice = docPozice.následník
    end while
    foreach dvojici (potomekDokumentu, potomekGrafu) z LCS_i
      hrana[uzelGrafu, potomekGrafu].seznamRodičů.přidej(uzelDokumentu)
      LCS(potomekDokumentu, potomekGrafu)
      docPozice = potomekDokumentu
      grafPozice = potomekGrafu
    end for
    docPozice = docPozice.následník
    grafPozice = grafPozice.následník
  end for
  while docPozice existuje
    vložPotomkaNakonec(uzelGrafu, docPozice)
    docPozice = docPozice.následník
  end while
end procedure
```

2: Algoritmus konstrukce spanning grafu

Konstrukce grafu

V diagramu `2: Algoritmus konstrukce spanning grafu` je uveden krok inkrementálního algoritmu pro konstrukci výše popsaného grafu. Do prázdného grafu jsou postupně "přilévány" stromy jednotlivých vstupních dokumentů. Přilévání stromu dokumentu probíhá rekurzivně. Jako první jsou slity kořen stromu a kořen konstruovaného grafu. Pro slití uzlů je nutno provést aktualizaci informací o uzlu (přidání do seznamu uzlů ze stromů dokumentů, aktualizace množiny atributů). Dále je potřeba slít sekvence poduzlů tak, aby výsledek co nejlépe zachovával pořadí podelementů a zároveň byl výsledný řetězec co nejkratší. Pro slévání sekvencí podelementů použijeme algoritmus pro hledání nejdelších společných podsekvencí. Uzly ze společných podsekvencí pak rekurzivně sléváme. U společných podsekvencí ještě aktualizujeme informace na společných hranách - přidáme do seznamu rodičů identifikátor uzlu. Příklad jednoduchého zkonstruovaného grafu uvádíme na obrázku `1: Příklad zkonstruovaného spanning grafu`. Graf vznikl ze dvou vstupních dokumentů. Tečkované jsou hrany, které se vyskytovaly pouze v prvním dokumentu, čárkované jsou hrany, které se vyskytovaly pouze ve druhém dokumentu a plné jsou hrany, které se vyskytovaly v obou vstupních dokumentech.

V poslední fázi, na základě spanning grafu a několika optimalizačních pravidel, zkonstruujeme výsledné DTD. Graf, jak jsme jej zkonstruovali v předchozím odstavci, už přirozeně odpovídá požadovanému výsledku. Uzly odpovídají elementům, tedy každému uzlu bude odpovídat jedna značka `<!ELEMENT` ve výsledném DTD. Uzly obsahují seznam atributů. Hrany vedoucí z uzlu pak reprezentují sekvenci podelementů, tedy odpovídají regulárnímu výrazu pro povolené podelementy v DTD. Zatím se však jedná pouze o sekvence elementů, které nepopisují ani příslušné sekvence vstupních dokumentů, a ani pro svou přesnost nejsou příliš použitelné jako schéma. Sekvence je tedy třeba optimalizovat pomocí následujících pravidel.

- **Volitelnost:** Pokud je seznam na hraně podmnožinou seznamu uzlů rodiče, pak je element označen jako volitelný. Na obrázku zkonstruovaného grafu se jedná například o podelement `Poznámka` elementu `Geografie`. Na hraně obsahuje seznam pouze uzel 1, seznam u rodiče obsahuje uzly 1 a 8. Podelement `Poznámka` je tedy označen jako volitelný. Jak je na obrázku vidět, podelement poznámka obsahoval pouze první dokument, druhý už nikoliv.
- **Opakování:** Pokud jsou sousedící vícenásobné hrany se stejným podelementem, pak je možné tyto hrany spojit v jednu, sjednotit jejich seznamy a cílový podelement opakovat. Na obrázku je taková vícenásobná hrana mezi elementem `Klima` a podelementem `Teplota`. Navíc, konkrétně na tuto hranu v obrázku, je možné aplikovat i první pravidlo, proto bude podelement `Teplota` opakován 0-krát nebo vícekrát.
- **Skupiny:** Provedeme rozdělení hran do skupin podle seznamů na hranách. Potom je vždy u sousedních skupin provedeno porovnání sekvence cílových elementů. Pokud je nalezena společná podsekvence, pak je možné příslušné části spojit, sloučit množiny na hranách a skupinu je možné opakovat (na příslušné místo je zaveden metaznak * nebo +).

Tento algoritmus produkuje velmi jednoduché regulární výrazy a navíc někdy výsledné schéma příliš zobecňuje. Například vůbec nezavádí operátor výběru `|`. Na místě, kde by se měl takový operátor zavést, například A|B, tento algoritmus uvede oba elementy jako nepovinné, tedy A?B?. Druhý výraz je samozřejmě mnohem volnější. Nejenže jím generovaný jazyk obsahuje slova A a B jako u prvního výrazu, ale navíc také prázdné slovo a slovo AB. Druhým problémem tohoto algoritmu je zmiňovaná acykličnost grafu reprezentujícího strukturu všech vstupních dokumentů. Nechť jsou na vstupu 2 dokumenty, takové že v prvním bude element A obsahovat element B a v druhém naopak, tedy element B bude obsahovat element A. Pokud jsou takové dokumenty vloženy na vstup tohoto algoritmu, pak zkonstruovaný graf musí obsahovat cyklus. Nicméně cyklus nepůsobí příliš problémy. Při přilévání potenciálních dalších dokumentů se algoritmus nemůže zacyklit, protože přilévaný strom dokumentu je konečný a žádné cykly neobsahuje. A při generování DTD z grafu se soustředíme vždy jen na jeden vrchol a z něho vystupující hrany. Takže když bychom brali jako vstup výše uvedené 2 dokumenty, výsledkem bude rekurzivní schéma, dovolující vnořování elementů A a B do libovolné hloubky. To nemuselo být záměrem původních dokumentů, ale je to dáno omezeními DTD.

4.2.1.2 Fred

Metoda Fred (uvedená v [8]) patří ke klasickým heuristickým metodám. Hledaný jazyk reprezentuje pomocí rozšířených gramatických pravidel (pravou stranu pravidel tvoří regulární výraz). Nejzajímavější částí tohoto řešení je množství pravidel jak zjednodušit regulární výraz bez změny jazyka, který generuje.

Nejdříve jsou z předložených příkladů dokumentů, respektive z jejich struktury, vybudována gramatická pravidla typu

$$ELEMENT \rightarrow DEFINICE$$

Kde DEFINICE je sekvence podelementů obsažených v ELEMENT. Poté jsou tato pravidla kombinována, zobecňována a redukována podle daných pravidel.

Pro každý element vyskytující se v předložených příkladech dokumentů jsou vybrána gramatická pravidla, kde je tento element na levé straně pravidla. Tato pravidla jsou zkombinována tak, že pravé strany pravidel jsou spojeny s pomocí exkluzivního výběru `|` do jediného pravidla.

Zobecňování

Takto vytvořené pravidlo popisuje právě vstupní dokumenty, ale to je pro většinu aplikací naprosto nepoužitelné. Proto je potřeba toto pravidlo zobecnit. Tedy upravit jej tak, že bude popisovat více možných instancí dokumentů. Tyto instance nejsou obsaženy ve vstupním vzorku, a tak nejsou vlastnosti těchto nových dokumentů přímo zřejmé. Nejčastějším příkladem takového zobecnění je nahrazení opakující se posloupnosti regulárním výrazem s metaznakem *.

Redukce

Zobecňování gramatiky má vedlejší efekt, že výsledný výraz je úspornější co do zápisu, výstižnější, a tím i čitelnější a pochopitelnější. Čitelnost, pochopitelnost a výstižnost jsou pro výsledek velmi důležité - zvláště pokud bude výsledné schéma používáno člověkem pro tvorbu nových dokumentů. Proto je na místě, za pomoci dalších pravidel tyto kvality ještě rozvinout.

- **Přebytečné metaznaky:** Pokud jsou ve výrazu metaznaky, které nijak neovlivňují výsledný jazyk, můžeme je odstranit. Takové situace jsou prázdné závorky, vícenásobné závorky, závorky okolo podvýrazů s operátory s vyšší prioritou. Přebytečné jsou také metaznaky, které je možné nahradit kratší sekvencí metaznaků. Jedná se o typ výrazů:

$$(a+)? \rightarrow a*$$
$$(a|b+)* \rightarrow (a|b)*$$

- **Spojení podvýrazů se stejným operátorem:** Konkrétně s operátory & a `|`. Jde o typy výrazů:

$$(a\&b)\&c \rightarrow a\&b\&c$$
$$(a|b)|c \rightarrow a|b|c$$

- **Opakující se atom:** Pokud se ve výrazu opakuje tentýž atom (tedy element), je možné tyto podvýrazy spojit do jediného. Takové typy výrazů jsou:

$$aaa \rightarrow a+$$
$$aa? \rightarrow a+$$
$$a|a+ \rightarrow a+$$
$$ab|ab \rightarrow ab$$

- **Volitelné podvýrazy:** Pokud se podvýrazy spojené operátorem `|` liší nejvýše na jednom místě, pak je možné výrazy spojit pomocí volitelného podvýrazu:

$$a+b|ab \rightarrow a*b$$
$$abc|ab|ac \rightarrow abc?|ac$$

  Je nutné poznamenat, že u druhého příkladu již není možné zbylé výrazy spojit, protože se liší již na dvou místech. Pokud by byl výraz zredukován na ab?c?, výraz by umožnil existenci osamoceného elementu a. Ale to v původním výrazu povoleno nebylo.

4.2.1.3 Heuristika sk-Ant

Tato metoda (prezentovaná v [5]) patří do skupiny algoritmů, které pro zobecňování jazyka používají spojování stavů automatu (tzv. merging states algorithms). Tato metoda je založená na výrazně jiné myšlence než většina ostatních. Metoda nepoužívá žádná empirická pravidla pro zobecňování, ale prohledává prostor všech zobecnění PTA (Prefix Tree Automaton) pomocí slučování stavů a hledá nejlepší řešení. Algoritmus nekonstruuje všechna možná řešení, ale prostorem prochází - konstruuje jen několik málo řešení, která postupně zlepšuje. Protože prozkoumání celého prostoru (všech možných řešení) by bylo kvůli rozsahu nemožné, používá metoda optimalizační algoritmy k dosažení co nejlepšího řešení. Algoritmus spojuje slučovací podmínku sk-string s heuristickou optimalizací ACO (Ant Colony Optimisation). Pro hodnocení kvality schématu a pro rozhodování optimalizačního algoritmu se používá metrika MML (Minimum Message Length) založená na maximalizování podmíněné pravděpodobnosti abstrakce vzhledem k daným příkladům.

Definice Probabilistic Finite State Automaton (PFSA): *Mějme abecedu Σ, pak pravděpodobnostní konečný automat PFSA (Probabilistic Finite State Automaton) je čtveřice $PFSA = (\Sigma, Q, P, q_1)$. Kde Σ je abeceda, Q je množina stavů, q_1 je počáteční stav a P je přechodová matice definující pravděpodobnost $p_{i,j}(a)$, že automat přejde ze stavu q_i do stavu q_j po písmenu a. A $p_{i,f}$ určuje pravděpodobnost, že stav q_i je koncový. Deterministický PFSA, je takový automat, že pro každý uzel q_i a znak* a *existuje nejvýše jeden uzel takový, že $p_{i,j}(a) \neq 0$.*

Poznamenejme pouze, že narozdíl od obyčejných konečných automatů, je zde determinismus skutečným omezením, tedy k nedeterministickému PFSA nelze vždy najít deterministický PFSA. V našem případě se omezíme pouze na tyto deterministické automaty.

Minimum Message Length

Princip MML říká, že nejlepší je taková abstrakce daných příkladů, která má nejvyšší podmíněnou pravděpodobnost. Předpokládejme, že máme teorii T s pravděpodobností p(T), která se snaží popsat příklady D. Pak podmíněná pravděpodobnost T vzhledem k D, označíme p(T|D) je:

$$p(T|D) = \frac{p(TD)}{p(D)} = \frac{p(T)p(D|T)}{p(D)}$$

Aplikováním Bayesovy věty dostáváme, že abychom maximalizovali p(T|D), musíme maximalizovat součin p(T)p(D|T). Na základě vlastností logaritmu a pravděpodobnosti je to ekvivalentní minimalizaci -log(p(T))-log(p(D|T)). Z informační teorie víme, že optimální kód pro symbol s pravděpodobností p je -log(p). Tedy maximalizování podmíněné pravděpodobnosti p(T|D) je ekvivalentní minimalizaci kódu abstrakce a příkladů. Abychom mohli použít MML jako míru kvality schématu, musíme stanovit metodu pro zakódování PFSA (respektive pro získání délky kódu PFSA, tedy kvality schématu).

$$MML(A) = \sum_{j=1}^{N} \log_2 \frac{(t_j - 1)!}{(m_j - 1)! \prod_{i=1}^{m_j} (n_{ij} - 1)!} + M(\log_2 V + 1) + M' \log_2 N - \log_2 (N-1)!$$

Kde N je počet stavů PFSA, V je mohutnost abecedy zvětšená o 1, t_j je počet průchodů j-tým stavem, m_j je počet hran vedoucích z j-tého stavu (zvětšený o 1 u koncových stavů), m'_j je počet hran vedoucích z j-tého stavu bez ohledu na koncové stavy, $n_{i,j}$ je počet průchodů po hraně mezi i-tým a j-tým stavem. M je součet m_j, M' je součet m'_j(tedy počet všech hran).

```
procedure skstring
  řetězce = k-stringy ze stavu s₁ seřazené podle pravděpodobnosti
  celkem = 0
  foreach řetězec z řetězce
    celkem = celkem + řetězec.pravděpodobnost
    if d(s₂,řetězec)=φ then
      return false
    end if
    if celkem > s then
      řetězce = k-stringy ze stavu s₂ seřazené podle pravděpodobnosti
      celkem = 0
      foreach řetězec z posloupnosti řetězce
        celkem = celkem + řetězec.pravděpodobnost
        if d(s₁, řetězec)=φ then
          return false
        end if
        if celkem ≥  s then
          return true
        end if
      end for
      return true
    end if
  end for
  return false
end procedure
```

3: Algoritmus sk-string

Sk-string

Tato metoda určuje kritérium pro ekvivalenci stavů. Základem je rozšíření `k-tail` heuristiky uvedené Biermannem a Feldmanem v [9], která je vlastně zjednodušením Nerodovy ekvivalence. Podle Nerodovy relace jsou dva stavy ekvivalentní, jestliže jsou nerozlišitelné podle řetězců, které je mohou následovat. Heuristika `k-tails` zjednodušuje toto kritérium tak, že bere v úvahu pouze řetězce maximálně délky k. Metoda `sk-string` se liší od `k-tails` použitím pravděpodobnostního automatu a bere v úvahu pouze `s` procent nejvíce pravděpodobných `k-stringů`. Rozdíl mezi `k-tail` a `k-string` je, že `k-string` nemusí končit v koncovém stavu, ale pak má délku přesně `k`. Pravděpodobnost řetězce je součin pravděpodobností na přechodech mezi stavy, které projdeme, když akceptujeme řetězec daným automatem. Zde je uvedena varianta AND této metody (`3: Algoritmus sk-string`). Existuje ještě několik dalších variant (OR, LAX, STRICT), ale ty zde nebudeme uvádět. Algoritmus v cyklu projde `s` nejpravděpodobnějších `k-stringů` vedoucích z prvního stavu a zjistí zda vedou příslušné `k-stringy` i z druhého stavu. Poté projde `s` nejpravděpodobnějších `k-stringů` vedoucích z druhého stavu a ověří zda vedou i z prvního stavu (AND varianta).

```
procedure ACO
  nejlepší = PTA
  iterace = 0
  while iterace < maximum iterací
    mravenci = udělejMravence(početMravenců, PTA, výběrPřesunu)
    početZivých = početMravenců
    zlepšení = false
    while početZivých > 0
      foreach mravenec z mravenci
        mravenec.udělejKrok()
        if mravenec.zemřel then
          if mml(mravenec.řešení) < mml(nejlepší) then
            nejlepší = mravenec.řešení
            zlepšení = true
          end if
          početZivých = početZivých - 1
        end if
      end for
    end while
    umístiFeromony(mravenci)
    if not zlepšení then
      iterace = iterace + 1
    else
      iterace = 0
    end if
  end while
  return nejlepší
end procedure
```

4: Algoritmus Ant Colony Optimisation

Ant Colony Optimisation

Tato optimalizační technika přiřazuje dvojicím stavů pravděpodobnost, že jejich sloučení je správná cesta a při prohledávání používá pozitivní odezvu. Kostra je uvedena v diagramu `4: Algoritmus Ant Colony Optimisation`. Provádí několik iterací, aby měla odezva účinek. K prohledávání zavádí objekty nazvané

Mravenec. Mravenec de facto reprezentuje stav nebo pozici při prohledávání prostoru řešení. V každé iteraci prochází mravenec prohledávaným prostorem a pro rozhodování používá pouze nějakou primitivní heuristiku. Když se mravenec přesouvá, tak za sebou nechává "odezvu" - feromony - jejichž množství závisí na kvalitě nalezeného řešení. Prohledávaným prostorem se pohybuje několik mravenců. V některých implementacích, včetně této, je pokládání feromonů opožděno až na konec iterace, kdy všichni mravenci dokončí svou cestu. Velikost odezvy, resp. množství feromonů, je přímo závislé na kvalitě nalezeného řešení. Takže přesuny na lepší řešení mají větší šanci, že budou opakovány některým mravencem v dalších iteracích. Ačkoliv každý jednotlivý mravenec většinou nemůže najít příliš dobré řešení (najde pouze některé lokální optimum), pokud spolupracuje několik mravenců v několika iteracích, mohou najít mnohem lepší řešení. V našem případě budou mravenci hledat nejlepší zobecnění automatu, resp. dvojice stavů, které sloučíme.

Funkce `výběrPřesunu`(`5: Algoritmus ACO - VýběrPřesunu`) slouží k přizpůsobení tohoto obecného algoritmu konkrétním účelům. Je to funkce, která určuje další pohyb mravence.

```
procedure VýběrPřesunu //standardní verze
  možnosti = prázdná množina
  foreach dvojice ze seznamDvojic
    h = (-změnaMML(A,dvojice) + MML(A)) / MML(A)
    p = feromony[dvojice]
    hodnota = p^α + h^β
    možnosti.přidej((hodnota,dvojice))
  end for
  return náhodný výběr z možnosti s ohledem na pravděpodobnost
end procedure
```

5: Algoritmus ACO - VýběrPřesunu

Poslední nepopsaný detail algoritmu je aktualizace feromonů. Feromony se aktualizují na konci každé iterace. Množství feromonů je závislé na kvalitě nalezeného řešení a je přizpůsobeno na základě úspěchů ostatních mravenců. Tedy množství feromonů pro mravence `a` je

$$p = průměrnéMML/MML(a.řešení)$$

kde průměrné MML je průměr z řešení nalezených jednotlivými mravenci. Tato hodnota feromonů je dána každé dvojici stavů, které bylo potřeba sloučit k dosažení daného výsledku. Pokud se více stavů sloučilo do jediného, je potřeba přiřadit feromony každé kombinaci dvojic zúčastněných stavů.

Sk-ANT

Spojením obou předchozích metod dostaneme hybridní metodu využívající výhod obou přístupů. Tato hybridní metoda je vlastně modifikací Ant Colony Optimisation heuristiky, kde jsou mravenci řízeni pomocí sk-string metody. Jedinou změnou je tedy změna funkce `VýběrPřesunu`, která určuje další pohyb každého jednotlivého mravence. Stejně jako v původní metodě postupně procházíme všechny dvojice stavů a přiřazujeme jim pravděpodobnost, že jejich sloučení je správná cesta. V nové metodě navíc přidáme dvojici pouze pokud odpovídá kritériu výběru metody `sk-string`, tedy `k-stringy` do celkové pravděpodobnosti `s` jsou z obou stavů shodné. Navíc pokud je toto kritérium příliš přísné, tak jej postupně zeslabujeme.

```
procedure VýběrPřesunu //sk-ANT verze
  možnosti = prázdná množina
  while možnosti.počet=0
    foreach dvojice ze seznamDvojic
      if skKritérium(dvojice) then
        h = (-změnaMML(A,dvojice) + MML(A)) / MML(A)
        p = feromony[dvojice]
        hodnota = p^α + h^β
        možnosti.přidej((hodnota,dvojice))
      end if
    end for
    if možnosti.počet=0 then
      skKritérium.zeslabit
    end if
  end while
  return náhodný výběr s ohledem na pravděpodobnost z možnosti
end procedure
```

6: Algoritmus ACO - přesun pomocí sk-string kritéria

4.2.2 Výběrové metody

Obecně mají všechny tyto metody dvě fáze. Nejdříve ze zadaných příkladů generují velké množství potenciálních schémat. Toto generování se provádí na základě sady heuristických pravidel. Vstupem této fáze jsou tedy vzorové dokumenty a výstupem množina potenciálních schémat. Druhá fáze pak ohodnotí všechna nalezená schémata z předchozí fáze a vybere schéma s nejlepším hodnocením. Metoda hodnocení schémat tak určuje vlastnosti výsledného schématu. V zásadě se jedná o kompromis mezi konkrétností a obecnosti schématu. Jako příklad je zde uvedena metoda XTRACT popsaná v [10].

4.2.2.1 Xtract

4.2.2.1.1 Generování

Tuto fázi je možné ještě rozdělit na dvě - zobecňování a faktorizaci. Během zobecňování jsou ve vstupních dokumentech hledány určité vzory a ty jsou pak nahrazeny odpovídajícími regulárními výrazy. Během této fáze jsou do původně jednoduchých příkladů přidávány zástupné znaky jako je *,+,? a jsou generovány stále obecnější a obecnější schémata. Během faktorizace je zápis již vygenerovaných schémat optimalizován pomocí hledání společných podvýrazů. Vstupem generování jsou všechny sekvence podelementů daného elementu, ale zpracovává se každá sekvence zvlášť. Výstupem generování pak nejsou schémata, která by nutně pokrývala všechny vstupní řetězce, ale každý vygenerovaný regulární výraz pokrývá alespoň ten jeden vstupní řetězec, ze kterého vznikl. V poslední fázi, kterou je generování schématu, tedy není cílem nalézt schéma s nejnižším hodnocením, ale podmnožinu schémat, která je nejlevnější, ale pokrývá všechny vstupní řetězce (tato jednotlivá podschémata jsou pak spojena do jediného výsledku pomocí operátoru OR).

Zobecňování

Aby bylo možné vybrat nejlepší schéma, je třeba vytvořit širokou paletu schémat. Pokud by byl výběr pouze ze schémat triviálně akceptujících vstupní instance, bylo by snažení zbytečné. Proto je potřeba postupně zobecňovat triviální schéma a generovat tak schémata s různou mírou kompromisu mezi obecností a konkrétností. Kostra zobecňujícího algoritmu je v diagramu `7: XTRACT - algoritmus zobecnování`. Procedura `Zobecňuj` stojí na 2 podprogramech - nahrazování opakujících se sekvencí (NajdiOpakování) a nahrazení výběrů (NajdiVýběrovéVzory). Tyto podprogramy jsou opakovaně volány s různými parametry pro dosažení co nejrozmanitějšího prostoru kandidátů.

Podprogram `NajdiOpakování`(`8: XTRACT - opakování`) dostává jako vstup výraz který má zobecnit, tedy sekvenci znaků z abecedy Σ, a parametr r, který určuje minimální počet opakování, který je potřeba aby podvýraz byl prohlášen za opakující se sekvenci. Případná nalezená opakující se sekvence je nahrazena atomickým symbolem. To umožňuje nadále pracovat se změněným výrazem jako s původním - pouze se rozšíří množina symbolů. V dalších opakováních algoritmu se tak již nahrazené sekvence, resp jejich zástupné symboly mohou vyskytnout v dalších opakujících se sekvencích. Například pro sekvenci abababcababc a `r=2` je v první obrátce cyklu zaveden symbol `A=(ab)+`, tedy výraz bude `AcAc` a v další obrátce je zaveden symbol `B=(Ac)+`. Nakonec tedy bude upravený výraz `((ab)+c)+`.

```
procedure Zobecňuj(I)
  foreach sekvenci
s z I přidej s do S_G
    foreach r z množiny 2,3,4
      s'=NajdiOpakování(s,r)
      foreach d z množiny 0.1*|s'|,0.5*|s'|,|s'|
        s"=NajdiVýběrovéVzory(s',d)
        přidej s" do S_G
      end for
    end for
  end for
end procedure
```

7: XTRACT - algoritmus zobecnování

```
procedure NajdiOpakování(s,r)
  do
    nechť x je podsekvence s s maximálním počtem spojitých opakování
    if počet opakování je menší jak r then return
s nahraď všechny spojité výskyty x v s novým symbolem A_i=(x)*
  while (s neobsahuje vhodnou spojitou sekvenci)
  return
s end procedure
```

8: XTRACT - opakování

Podprogram `NajdiVýběrovéVzory` (`9: XTRACT - výběrové vzory`) hledá sekvence typu $(a_1|\ldots|a_m)^*$ na základě shlukování stejných symbolů v určité části řetězce. Podprogram dostává jako parametr vstupní sekvenci symbolů (schéma které má zobecnit) a parametr d určující jak moc může být shluk stejných symbolů "roztáhlý". Většinu práce provádí podprogram `Rozděl`, který dostává stejné parametry. Zbytek podprogramu `NajdiVýběrovéVzory` už pouze nahrazuje shluky příslušnými regulárními výrazy.

Podprogram `Rozděl` prochází vstupní řetězec a zvětšuje shluk tak dlouho, dokud se v dosahu (ve vzdálenosti menší než d) vyskytuje symbol, který je již ve shluku obsažen. Pokud už není možné shluk rozšířit, tak se ukončí a vytváří se další shluk.

Tedy například pro sekvenci abcbcca a d=2. Na začátku obsahuje první shluk pouze první znak `a` algoritmus se pokusí shluk rozšířit. Ve vzdálenosti 2 se však znak `a` znovu nevyskytuje (na druhé a třetí pozici jsou znaky `b,c`). Proto je shluk ukončen a na následujícím znaku je započat nový. Nový shluk nejdříve obsahuje pouze druhý znak, tedy `b`. Na čtvrté pozici se vyskytuje symbol `b` znovu, proto můžeme shluk rozšířit a zahrnout do něj i `c`. Na páté a šesté pozici jsou znovu znaky `c`, které jsou již ve shluku obsaženy, a proto je možné jej rozšířit až sem. Na sedmé pozici je symbol `a`, který není ve shluku a za tímto znakem už řetězec končí. Symbol `a` proto již není zahrnut do aktuálního shluku, ale je zahrnut do dalšího. Výstupem tedy budou 3 shluky: a,bcbcc,a.

```
procedure NajdiVýběrovéVzory(s,d)
  s_1,...,s_n=Rozděl(s,d)
  foreach každou sekvenci s_i z s_1,...,s_n
    nechť a_1,...,a_m je množina symbolů z s_i
    if m>1 then
      nahraď s_i novým symbolem A_i = (a_1|...|a_m)*
    end if
  end for
end procedure

procedure MůžuRozšířit(s,d,s_i)
  if libovolný symbol z s_i se vyskytuje v s vpravo od s_i
    maximálně ve vzdálenosti d
  then
    return true
  else
    return false
  end if
end procedure

procedure Rozděl(s,d)
  i=start=konec=1
  s_i=s[start,konec]
  while konec<|s|
    while konec<|s| a MůžuRozšířit(s,d,s_i)
      konec=konec+1
      s_i=s[start,konec]
    end while
    if konec<|s| then
      i=i+1
      start=konec+1
      konec=konec+1
      s_i=s[start,konec]
    end if
  end while
  return s_1,...,s_i
end procedure
```

9: XTRACT - výběrové vzory

Faktorizace

V tomto kroku je provedena faktorizace výběru podmnožiny kandidátů na základě společných podvýrazů. Například pro podmnožinu kandidátů ac, ad, bc, bd je možné faktorizací odvodit (a|b)(c|d). Faktorizací se již nezobecňují kandidující schémata, ale kratším zápisem můžeme dosáhnout menší MDL hodnoty schématu. Faktorizovaná schémata jsou zcela přirozená a běžná v reálných schématech na místech, kde popisujeme objekt pomocí několika nezávislých voleb. Například článek může být prezentován na konferenci, vydán v časopise nebo může být součástí knihy. Zároveň může být z oboru chemie, informatiky, matematiky nebo fyziky. Přirozeně pak bude mít schéma podobu `(konference|časopis|kniha)(chemie|informatiky|matematika|fyzika)`.

Ideálně by bylo třeba faktorizovat všechny podmnožiny kandidátů vzniklých při zobecňování. Avšak to by bylo samozřejmě velmi nepraktické (nebo spíš nepoužitelné), protože množina kandidátů může být rozsáhlá. Proto je potřeba navrhnout heuristickou strategii pro výběr podmnožin. Intuitivně budou vybráni kandidáti, kteří

- sdílejí společné předpony nebo přípony
- mají co nejmenší překryv s ostatními schématy ve výběru

Detailní algoritmus pro výběr podmnožin ani detaily algoritmu pro faktorizaci nebudou v této práci uvedeny - tyto algoritmy již příliš nezasahují do oblasti zájmu této práce a je možné je najít v referencích [10] a [11].

4.2.2.1.2 Výběr

Toto je nejdůležitější část algoritmu. Úkolem v této fázi je vybrat ze schémat vygenerovaných v předchozí části to nejlepší. Respektive schéma, které je nejblíže ideálnímu. Zde je uvedena jedna z možností hodnocení schémat nazvaná MDL princip (Minimum Description Length). Každé schéma je ohodnoceno počtem bitů potřebných k zakódování vstupní kolekce dokumentů na základě tohoto schématu a počtem bitů potřebných k vyjádření samotného schématu. Schéma s nejmenším hodnocením pak odpovídá představě skoro optimálního schématu. Skoro optimální schéma je dostatečně obecné, jednoduché a výstižné a zároveň popisuje dostatek detailů (tedy není zase příliš obecné). Skoro optimální schéma je tedy takové, které je vyvážené z hlediska přesnosti a obecnosti. Příliš obecné schéma sice bude zakódováno malým počtem bitů, ale bude potřebovat mnoho bitů na zakódování jednotlivých instancí. Naproti tomu příliš konkrétní schéma spotřebuje málo bitů na popis instancí, ale zakódování schématu si vyžádá velmi dlouhý kód.

Kódování schématu

Nechť je Σ množina podelementů, $\mathcal{M}$ je množina metaznaků $\mathcal{M} = \{|, *, +, ?, (,)\}$. Schéma je vyjádřeno pomocí regulárního výrazu, tedy řetězce nad abecedou $\Sigma \cup \mathcal{M}$. Počet bitů nutných k zakódování schématu nad abecedou $\Sigma \cup \mathcal{M}$ délky n (počet znaků z $\Sigma \cup \mathcal{M}$) je $n * \lceil log(|\Sigma| + |\mathcal{M}|) \rceil$. Jednoduše tedy délka schématu je délka jeho regulárního výrazu v bitech.

Kódování instancí

Nejdříve je instance schématu zakódována s použitím schématu pomocí sekvence čísel. Délka sekvence v bitech je pak hledaná délka kódu. Nechť je sekvence čísel kódující řetězec `s` pomocí schématu `D` označena `seq(s,D)`. Nyní je `seq(s,D)` rekurzivně definováno podle komponent schématu pomocí následujících pravidel.

- $seq(s, D) = e$ jestliže `D=s`, tedy schéma je sekvence terminálů (písmen z abecedy Σ), která je stejná jako `s`. Symbol e reprezentuje prázdný kód.
- $seq(s_1...s_k, D_1...D_k) = seq(s_1, D_1)...seq(s_k, D_k)$ jestliže `D` je konkatenací schémat D_1 až D_k a `s` můžeme napsat jako konkatenaci $s_1 \ldots s_k$
- $seq(s, D_1|...|D_k) = iseq(s, D_i)$, tedy pokud je schéma `D` exkluzivní výběr z k možností, pak se kód skládá z indexu `i`, určujícího výběr a kódu `s` podle tohoto výběru. Je potřeba poznamenat, že na index `i` potřebujeme $log\lceil k \rceil$ bitů.
- $seq(s_1...s_k, D*) = kseq(s_1, D)...seq(s_k, D)$, tedy pokud se schéma skládá z opakování podbloku `D` a `s` je konkatenace instancí `D`, pak se kód skládá z počtu opakování a kódů jednotlivých instancí `D`. V tomto případě neexistuje jednoduchý způsob určení počtu bitů pro reprezentaci počtu opakování `k`. Nejdříve je počet bitů nutných k interpretaci `k` zakódován unárně, tedy zapíšeme $log\lceil k \rceil$ jedniček následovaných nulou. Poté pomocí $log\lceil k \rceil$ bitů zakódujeme samotné číslo `k`.

Mezi výše uvedenými pravidly nejsou uvedena pravidla pro metaznaky + a ?. Avšak tyto znaky je možné zpracovat stejným způsobem jako *. Pro + bude `k>0` a pro ? bude `k` buď 0 nebo 1. Pro všechna pravidla mimo prvního je potřeba ověřit, zda daný podřetězec `s` odpovídá podschématu `D` a rozdělit `s` na jednotlivé komponenty (tam kde je to potřeba). Toho je možné docílit v čase $O(|D| * |s|)$ pomocí nedeterministického konečného automatu. V některých případech může být více možností jak rozdělit `s` na komponenty. V takovém případě by bylo ideální projít všechny možnosti a vybrat tu s nejlepším hodnocením (nejkratším kódem). To by ale výrazně zhoršilo složitost výpočtu, a proto není ve zkoumaném řešení implementováno.

Nyní je pro každé schéma spočtena délka jeho kódu a délky kódů vstupních řetězců jako instancí schématu. Hledané schéma je pak to, které má nejmenší součet těchto hodnot.

4.2.2.1.3 Generování schématu

V poslední fázi je tedy k dispozici množina I vstupních řetězců, množina J částečných schémat (ve smyslu že pokrývají část vstupních řetězců), je spočtena délka kódu každého částečného schématu a je spočtena délka kódu vstupních řetězců pomocí těchto podschémat. Nyní je potřeba vybrat podmnožinu částečných schémat, která bude pokrývat všechny vstupní řetězce a bude mít minimální hodnotu výrazu

$$\min_{F \subset J} \{ \sum_{j \in F} c(j) + \sum_{i \in I} \min_{j \in F} d(j,i) \}$$

Toto je vlastně problém známý jako Problém přiřazení zdrojů ("Facility location problem"), kde je dána množina producentů, množina klientů, cena za použití producenta a cena za obsloužení klienta producentem. V tomto případě jsou částečná schémata producenti a vzorové řetězce klienti. Po nalezení nejlepší podmnožiny F jednoduše všechna částečná schémata z F spojíme operátorem OR a máme hledané schéma.

4.3 Metody odvozující gramatiku

Na strukturu XML dokumentu je možné nahlédnout jako na gramatiku. Dokument odpovídající danému schématu je pak slovo nad abecedou elementů dokumentu, odpovídající dané gramatice. XML dokument vyžaduje správné párování otevíracích a uzavíracích elementů (elementy bez uzavírací značky, `<element />`, je možné z hlediska hledání schématu nahradit párovým elementem s prázdným obsahem). Gramatika schopná popsat strukturu XML dokumentu tedy patří do třídy bezkontextových gramatik v Chomského hierarchii (viz. [12]). Avšak strukturu každého elementu zvlášť (tedy pouze výskyt podelementů uvnitř daného elementu - vztah elementů rodič-dítě) je možné popsat regulárním jazykem. Taktéž všechny jazyky na popis struktury XML dokumentů (DTD, XML Schema, ...) umožňují zápis regulárními výrazy (popisují regulární jazyky). Toto zjednodušení pro XML jazyky popsané Berstelem a Boassonem [1] ještě detailně popíšeme v sekci `XML jazyky`. Gold ve své práci [6] ukázal metodu učení se jazyka z pozitivních příkladů. Nadefinoval identifikovatelnost jazyků a dokázal větu, z které plyne, že regulární jazyky nejsou v tomto smyslu identifikovatelné z pozitivních příkladů. Tyto závěry popíšeme v sekci `Identifikovatelnost jazyků`.

4.3.1 XML jazyky

Dobře formovaný XML dokument je dobře uzávorkovaný dokument. Pokud navíc dokument odpovídá zadanému schématu, říkáme že je validní. Na schéma dokumentu můžeme nahlížet jako na gramatiku. Nadefinujme si tedy třídu gramatik (jazyků) pro schémata XML dokumentů.

Definice XML gramatiky: *Mějme množinu prvků (značek) A. A' je množina uzavíracích značek $A' = \{a' | a \in A\}$. Množina terminálů gramatiky je pak $T = A \cup A'$, množina neterminálů $V = \{X_a | a \in A\}$. Jeden z neterminálů označíme jako axiom (počáteční neterminál). Pro všechna $a \in A$ jsou $R_a \subset V*$ regulární množiny. Gramatika pak obsahuje pravidla $X_a \rightarrow ara'$, kde $r \in R_a$. Můžeme též psát $X_a \rightarrow aR_aa'$.*

Ještě nadefinujme tzv. Dyckův jazyk D_A pro množinu elementů A (pokud je množina A zřejmá z kontextu označujeme Dyckův jazyk pouze D).

Definice Dyckův jazyk: *Dyckův jazyk je jazyk generovaný gramatikou s množinou terminálů $T = A \cup A'$, počátečním neterminálem X a pravidly*

$$X \rightarrow X_{a1}|...|X_{an}$$
$$X_a \rightarrow a(X_{a1}|...|X_{an}) * a'$$

Nutno poznamenat, že tento jazyk, až na speciální případ kdy $|A| = 1$, není XML jazykem. Tento jazyk obsahuje všechna dobře uzávorkovaná slova nad abecedou terminálů T. Pro libovolný element $a \in A$ ještě nadefinujeme jazyk $D_a = D_A \cap a(A \cup A')a'$. D_a už je XML jazyk a dokonce největší nad množinou A s axiomem X_a.

Každé slovo $w \in D_a$ je možné jednoznačně dekomponovat na $au_{a1}...u_{an}a'$, kde je každé $u_{ai} \in D_{ai}$. Sekvenci $a_1...a_n$ nazveme stopa(trace). Množinu $F_a(L) = \{v | uvw \in L \& v \in D_a$ pro libovolná slova u,w} nazveme faktor `L` podle `a`. Množina $S_a(L)$ je množina stop všech slov z $F_a(L)$ a nazýváme ji vzor(surface).

Mějme množinu $S = \{S_a | a \in A\}$. Definujme XML gramatiku G:

$$T = A \cup A'$$
$$V = \{X_a | a \in A\}$$
$$R_a = \{X_{a1}...X_{an} | a_1...a_n \in S_a\}$$

Jazyk vygenerovaný pomocí G a vybraného axiomu X_{a0} je jediný XML jazyk odpovídající S jako množině vzorů. Tedy pokud se podaří odvodit pro daný (neznámý) jazyk regulární množiny S_a pro všechna $a \in A$, pak už je jednoznačně určená i gramatiku a potažmo celý jazyk. Tedy pro XML jazyky není nutné odvozovat přímo bezkontextový jazyk, ale je možné úlohu zjednodušit a odvozovat regulární množiny (resp. regulární jazyky).

4.3.2 Identifikovatelnost jazyků

Gold ve své práci [6] definoval model učení z pozitivních příkladů.

Definice Model učení: *Máme odvozovací stroj (IM). Tento stroj dostává na vstupu nekonečnou posloupnost příkladů, resp. postupně jednotlivé vzorky z posloupnosti. Na výstupu pak po každém vzorku dává hypotézu pro popis hledaného jazyka. Celkem nezáleží jakou má popis jazyka formu (gramatika, konečný automat, ...). Formálně tedy*

Definice Enumerace jazyka: *Funkce $E : N \rightarrow L$, kde N jsou přirozená čísla a L je množina slov jazyka, je enumerace jazyka. Enumerace přiřazuje každému slovu číslo a naopak.*

Vstupem IM jsou vzorky $E(1), ..., E(N)$, kde E je enumerace jazyka L. A výstupem IM je posloupnost D(i) popisů hledaného jazyka ze vzorků E(j), $j \leq i$.

Definice Identifikovatelnost: *Třída jazyků L je identifikovatelná, pokud existuje IM takový, že $\forall E : \exists n_0 : \forall n > n_0 : D(n) = D(n_0)$ a navíc $D(n_0)$ je popis hledaného jazyka L. Tedy pro všechny posloupnosti E enumerující jazyky z třídy L existuje konečné číslo n_0 takové, že od vzorku $E(n_0)$ už IM nezmění svůj odhad na popis jazyka a navíc tento odhad správně popisuje jazyk enumerovaný pomocí vybraného E.*

Gold dokázal větu o identifikovatelnosti jazyků.

Věta Gold: *Třída jazyků, která obsahuje všechny konečné jazyky a alespoň jeden nekonečný, není identifikovatelná pouze z pozitivních příkladů.*

Precizní důkaz této věty nebudeme uvádět, případní zájemci mohou najít detail v [6]. Jednoduše je možné nahlédnout správnost tvrzení následující úvahou: Pokud třída jazyků obsahuje všechny konečné jazyky a alespoň jeden nekonečný, tak po libovolném počtu vstupů nejsme schopni rozhodnout, zda je již výsledek hotový nebo bude potřeba popis jazyka ještě zobecnit (resp. upravit popis tak, aby akceptoval další slova). Jinak řečeno, v procesu zobecňování není u takové třídy jazyků zarážka, která zabrání přílišnému zobecnění (stavu kdy akceptujeme více slov než požadovaný jazyk).

Takovou neidentifikovatelnou třídou jazyků jsou i regulární jazyky, které jsou potřeba k odvození XML gramatiky. Proto je potřeba využít dalších informací, které umožní správně odvodit strukturu předložených dokumentů. Mezi takové informace patří zejména:

- **Interakce s uživatelem:** Odvozovací stroj využívající interakce s uživatelem. Jednoduše po každém kroku předloží uživateli výsledek a ten rozhodne zda je takový výsledek uspokojující, nebo předloží IM další příklad slova z jazyka. Tato "dodatečná informace" je samozřejmě nejlepší jakou můžeme dostat (ve smyslu že uživatel bude vždy s výsledkem spokojený), ale klade určité nároky na uživatele a zabraňuje plně automatickému dávkovému zpracování.

- **Maximální počet uzlů:** Jako výsledný popis jazyka nechť je konečný automat. Uživatel při spuštění IM zadá maximální počet uzlů automatu akceptujícího daný jazyk. IM pak zobecňuje automat (sléváním uzlů) tak dlouho, až dosáhne tohoto maximálního počtu uzlů.
- **Identifikovatelná podtřída:** IM pracuje s informací, že výsledný jazyk patří do některé podtřídy regulárních jazyků, která je identifikovatelná. Toto je nejčastější forma dodatečné informace využívaná v současných algoritmech.

4.3.3 Funkcí rozlišitelné jazyky

Tato metoda je uvedena v [4]. Definujeme rozlišovací funkci, třídu jazyků rozlišitelných funkcí a konečný automat rozlišitelný funkcí. A uvádí algoritmus pro odvozovací stroj pro danou funkcí rozlišitelnou třídu jazyků.

Definice Rozlišovací funkce: *Mapování $f : T^* \to F$, kde T je množina terminálů a F je konečná množina nazveme rozlišovací funkce pokud platí* $\forall u, w, z \in T* : f(w) = f(z) \Rightarrow f(wu) = f(zu)$.

Definice Funkcí rozlišitelný jazyk: *Jazyk L je rozlišitelný funkcí f (píšeme že L je f-rozlišitelný), jestliže* $\forall u, v, w, z : f(w) = f(z) \Rightarrow ((wu \in L \& wv \in L) \Rightarrow (zu \in L \Leftrightarrow zv \in L))$

Definice Funkci rozlišitelný automat: *Automat $A = (Q, T, d, q_0, Q_F)$ je f-rozlišitelný, jestliže*

- *A je deterministický*
- *Pokud $d * (q_0, x) = d * (q_0, y)$ pak $f(x) = f(y)$*
- *Pokud $q_1 \neq q_2, q_1, q_2 \in Q_F$ nebo $d(q_1, a) = d(q_2, a)$ pak $f(q_1) \neq f(q_2)$*

Tedy pokud vedou z počátečního stavu dvě cesty po dvou slovech do téhož stavu, pak se hodnota rozlišovací funkce pro tato dvě slova rovná. Z toho také plyne, že f(q) pro $q \in Q$ je dobře definované - pro q vezmeme libovolné slovo x tak, že $d * (q_0, x) = q$ a $f(q) = f(x)$.

Mějme na vstupu množinu $I = \{w_1, ..., w_m\}$ příkladů hledaného jazyka (množinu vzorků ze vstupních dokumentů), $w_i = a_{i,1} ... a_{i,n(i)}$. Z té je sestrojen jednoduchý nedeterministický automat akceptující přesně slova z I, tedy množina stavů bude odpovídat písmenům $a_{i,j}$(plus počáteční stavy $a_{i,0}$), koncové stavy budou poslední písmena slov (tedy $a_{i,n(i)}$), přechodová hrana s písmenem $a_{i,j}$ bude mezi stavy $a_{i,j-1}$ a $a_{i,j}$. Pro stav $q_{i,j}$ nadefinujeme příponu FS a předponu HS

Definice Předpona HS a přípona FS:

$$FS(q_{i,j}) = a_{i,j}...a_{i,n(i)}$$
$$HS(q_{i,j}) = a_{i,1}...a_{i,j-1}$$

Protože existuje pouze jediné slovo vedoucí do stavu q, konkrétně HS(q), je možné nadefinovat f(q)=f(HS(q)).

Definice Relace shodných stavů: *Nechť $\simeq_f$ je relace definovaná takto:*

$$q_{i,j} \simeq_f q_{k,l} \text{ pokud}$$
$$HS(q_{i,j}) = HS(q_{k,l}) \text{ nebo}$$
$$FS(q_{i,j}) = FS(q_{k,l}) \text{ a } f(q_{i,j}) = f(q_{k,l})$$

Relace $\simeq_f$ není ekvivalence. Proto dodefinujeme relaci $\equiv_f$ jako tranzitivní uzávěr $\simeq_f$. Z `A` vytvoříme nový automat `A'` tak, že stavy `A` sloučíme podle tříd ekvivalence $\equiv_f$. Výsledný automat `A'` je f-rozlišitelný a jazyk, který akceptuje je nejmenší f-rozlišitelný jazyk obsahující `I`.

Rozlišovací funkce mohou být velmi různorodé. Například funkce

$$f(x) = \text{"sufix x délky k"}$$

vede ke třídě jazyků nazývané k-reversible. Nebo funkce

$$f(x) = \text{"množina terminálů vyskytujících se v x"}$$

vede ke třídě terminály rozlišitelných jazyků.

4.3.4 Metoda k,h contextual

Ještě uvedeme jednu metodu velmi podobnou předcházející, která ale uvádí velmi zajímavé rozšíření pro interakci s uživatelem. Tuto metodu prezentovala Helena Ahonen v práci `[3]`. Znovu prezentuje identifikovatelnou podtřídu regulárních jazyků, jako dodatečnou informaci potřebnou k identifikování třídy jazyků. Tato metoda vychází z předpokladu, že kontext jazyka je omezený. Tedy pokud jsou 2 dostatečně velké podřetězce stejné, elementy následující po nich nerozliší na kterém místě se nacházejí. Nejdříve definujme k-kontextové jazyky, poté je rozšíříme na k,h-kontextové. Ukažme odvozovací algoritmus a nakonec popíšeme rozšíření umožňující interakci s uživatelem.

Definice k-kontextový jazyk: *Jazyk L je k-kontextový, jestliže pro stavy p_0,q_0 a slovo x délky k, $d^*(p_0, x) = p$ a $d^*(q_0, x) = q$, pak $p = q$.*

Tedy pokud ze dvou různých stavů dojdeme po slově délky k do nějakých stavů, pak jsou tyto stavy shodné. Jinak řečeno, pokud jsou v automatu dvě shodné cesty (slova) délky `k`, pak končí ve stejném stavu.

Definice k,h-kontextový jazyk: *Jazyk L je k,h-kontextový, jestliže pro libovolné dva stavy p_k,q_k a slovo $a_1 \ldots a_k$ existují stavy p_0,q_0 tak, že $d(p_0, a_1) = p_1 ... d(p_{k-1}, a_k) = p_k$ a $d(q_0, a_1) = q_1 ... d(q_{k-1}, a_k) = q_k$ pak $p_i = q_i$ pro $0 \leq h \leq i \leq k$.*

Je to zpřísnění předešlé podmínky, resp. větší zobecnění jazyka (slučuje se více stavů). Při zobecňování se slučují nejenom koncové stavy řetězců, ale také celý "ocas".

```
procedure khcontext
  foreach element E
    zkonstruuj prefixový automat A z řetězců podelementů všech výskytů E
    do
      foreach dvojici stavů p,q
        pokud p,q splňují podmínku, zobecni A sloučením stavů p,q
      end for
    while existují stavy které můžeme sloučit
    zkonvertuj A na regulární výraz
  end for
end procedure
```

10: Algoritmus odvozování k,h-kontextových jazyků

Toto je obecný algoritmus odvozovacích strojů založených na slučování stavů. De facto je také stejný jako algoritmus uvedený v předchozí sekci, který slučoval na základě tříd ekvivalence. Podmínkou pro sloučení stavů je v tomto případě slovo délky `k` které vede do různých stavů.

Pokud je automat (potažmo jazyk) `k-kontextový`, pak každá cesta délky k obsahující stejnou sekvenci znaků končí ve stejném stavu. Díky takto omezenému kontextu je možné každý `k-kontextový` jazyk definovat pomocí konečné množiny řetězců délky $k + 1$. Tato množina obsahuje všechny podřetězce slov z jazyka délky nejvýše $k + 1$. Tyto podřetězce nazýváme `k-gramy`. Každý `k-kontextový` jazyk má jednoznačnou množinu `k-gramů` a naopak množina `k-gramů` jednoznačně definuje jazyk. Tyto `k-gramy` tak můžeme předložit uživateli. Ten může některé odebrat, jiné naopak přidat a algoritmus je schopen z takto upravené množiny znovu vygenerovat jazyk. Reprezentace jazyka pomocí seznamu možných podřetězců je pro uživatele přirozená a snadno pochopitelná.

Tato metoda poskytuje konkrétní třídu identifikovatelných jazyků. Nesporným přínosem je snadno pochopitelná forma interakce s uživatelem. Ovšem předpoklad omezeného kontextu jazyků je dosti diskutabilní a odtržený od praktických aplikací.

4.3.1 Alergia

V závěru výčtu existujících metod pro odvozování regulárních jazyků představíme ještě jednu značně odlišnou metodu. Metoda Alergia (uvedená v `[14]`) odvozuje tzv. SRL (Stochastic Regular Languages) jazyky, neboli regulární jazyky obohacené o pravděpodobnosti výskytu. Pomocí slučování stavů odvozuje pravděpodobnostní konečný automat tak, aby byl kompatibilní s očekávaným rozložením pravděpodobnosti v jazyce. Očekávané rozložení je dáno instancemi ve vstupní množině příkladů.

Definice Stochastic Finite Automaton: *Mějme abecedu A, pravděpodobnostní konečný automat SFA pak je čtveřice SFA=(A,Q,P,q_1). Kde* `A` *je abeceda,* `Q` *je množina stavů, q_1 je počáteční stav a* `P` *je matice přechodu definující pravděpodobnost $p_{i,j}(a)$, že automat přejde ze stavu q_i do stavu q_j po písmenu* `a`*. A $p_{i,f}$ určuje pravděpodobnost, že stav q_i je koncový.*

Definice Stochastic Regular Language: *Označme $p(w)$ pravděpodobnost, že automat vygeneruje slovo w. Pak jazyk generovaný automatem je $L = \{w \in A^* | p(w) \neq 0\}$.*

Pokud automat neobsahuje žádné zbytečné uzly, pak generuje pravděpodobnostní rozložení pro A*. Dva automaty jsou ekvivalentní, pokud generují stejné pravděpodobnostní rozložení nad A*. Tedy nestačí, že generují stejný jazyk, ale musí také generovat slova se stejnou pravděpodobností. Algoritmus Alergia se omezuje pouze na deterministické SFA, tedy takové, že pro každý uzel q_i a znak `a` existuje nejvýše jeden uzel takový, že $p_{i,j}(a) \neq 0$.

```
procedure Alergia(S)
  A = pravděpodobnostní prefixový automat z množiny příkladů
S for j=2 to počet uzlů
A for i=1 to j-1
      if Kompatibilní(i,j) then
        sluč(A,i,j)
        převeďNaDeterministický(A)
        break i-cyklus
      end if
    end for
  end for
end procedure
```

11: Algoritmus Alergia

Kostra algoritmu je uvedena v `11: Algoritmus Alergia`. Nejdříve je postaven pravděpodobnostní prefixový automat. Tedy vstupní slova jsou postupně přidávána do automatu a počítá se relativní četnost průchodu hranami a kon-

```
procedure Kompatibilní(i,j)
  //jestli je dostatečně podobná pravděpodobnost koncových stavů
  if Rozdílné(n_i, f_i(*), n_j, f_j(*)) then
    return false
  end if
  foreach znak a z abecedy
    if Rozdílné(n_i, f_i(a), n_j, f_j(a)) then
      return false
    end if
    if not Kompatibilní(d(i,a), d(j,a)) then
      return false
    end if
  end for
  return true
end procedure

procedure Rozdílné(n,f,n',f')
  return |f/n - f'/n'| > sqrt(0.5*log(2/alpha))*(1/sqrt(n)+1/sqrt(n'))
end procedure
```

12: Alergia - Kompatibilita stavů

covými uzly. Z těchto četností jsou pak vypočteny příslušné pravděpodobnosti. Poté algoritmus porovnává dvojice stavů. Pokud jsou 2 stavy ekvivalentní, pak mají stejné pravděpodobnostní rozložení pro hrany vedoucí ze stavu, tedy pro všechny znaky a, $p_i(a) = p_j(a)$, a navíc cílové uzly musí být také ekvivalentní. Protože experimentální data, tedy v tomto případě i data v příkladech, podléhají jisté pravděpodobnostní fluktuaci, ekvivalence stavů je akceptována v určitém rozsahu - mluvíme o kompatibilitě stavů.Rozsah kompatibility pro proměnnou p a očekávanou frekvenci výskytů f z n je stanoven podle vzorce

$$|p - f/n| < sqrt(1/(2n)log(2/\alpha)) \text{ s pravděpodobností větší než 1-}\alpha$$

Kde p je roven f/n druhého uzlu. Toto využijeme jako kritérium pro ekvivalenci uzlů. Stavy jsou tedy ekvivalentní pokud

$$|p_i(a) - (p_j(a)| < sqrt(1/(2n)log(2/\alpha))$$

Pokud jsou hrany kompatibilní, pravděpodobnost se příliš neliší, je možné rekurzivně testovat kompatibilitu cílových uzlů. Tato rekurze provede pouze konečný počet iterací díky pořadí v jakém jsou uzly porovnávány. Pokud jsou porovnávány stavy q_i a q_j, pak q_j je vždy kořenem podstromu automatu, tedy v podautomatu dosažitelném z tohoto stavu nejsou cykly a generovaný jazyk je konečný. Další otázkou je determinismus automatu. Po sloučení stavů by potenciálně mohl vzniknout nedeterministický automat, ale vzhledem k tomu, že cílové uzly slučovaných uzlů musí být také kompatibilní, jsou sloučeny také, takže nedeterministický automat vzniknout nemůže. Nakonec, po sloučení uzlů, jsou všechny relativní četnosti přepočítány.

Tento algoritmus, stejně jako mnoho v této sekci, je postavený na silných teoretických základech a jeho výsledky jsou precizní v tom smyslu, že jsou zařaditelné do určité kategorie jazyků, jsou predikovatelné, avšak základy na kterých je algoritmus postaven ne vždy vystihují praktické příklady použití.

5 Návrh řešení

5.1 Shrnutí analýzy

Všechny algoritmy uvedené v předchozí kapitole používají jako výstupní formát jazyk DTD. Tento jazyk je ve svých možnostech velmi omezený. Zde prezentovaný algoritmus využívá jako cílový formát jazyk XML Schema a snaží se využít jeho možností.

První inovací je rozdělení elementů podle strukturální podobnosti. Ne vždy mají všechny elementy se stejným názvem stejný sémantický význam a stejnou strukturu. Toto rozšíření umožňuje takové elementy podle jejich struktury rozlišit.

Další inovací je zavedení permutačních operátorů, respektive typu `<xs:all ... >` jazyka XML Schema. Tento typ umožňuje zjednodušit zápis regulárních výrazů popisujících sekvence, kde se vyskytují elementy v libovolném pořadí.

Poslední inovací je možnost definovat více kořenových elementů dokumentu. V určitých případech může být rozumné a žádoucí definovat schéma pro dokumenty mající různé kořenové elementy.

Uvedený algoritmus vychází zejména z metod `Heuristika sk-Ant` a `Xtract`. Z první metody je využita myšlenka heuristické optimalizace ACO, z druhého je použita a upravena myšlenka principu MDL pro hodnocení nalezeného schématu. Heuristická optimalizace umožní spojit několik přístupů dohromady a princip MDL je umožní jednotnou formou hodnotit.

5.2 Přehled řešení

V první podkapitole je představen algoritmus jako celek. Jsou nastíněny jeho jednotlivé části a ukázáno jejich provázání. Poté jsou podrobně popsány jeho stěžejní části. V kapitole `Rozdělení elementů` je popsáno rozdělení elementů na skupiny podle podobnosti. V následující kapitole `Odvozování regulárních výrazů` je popsáno odvozování struktury elementů. Znovu je nejdříve prezentován celkový přehled a v podkapitolách jsou popsány detaily stěžejních částí.

5.3 Hlavní část programu

Hlavní část programu se skládá ze čtyř kroků, které odpovídají přípravné fázi, dvěma po sobě následujícím funkcím algoritmu a koncové fázi. Program je naznačený ve schématu `13: Hlavní část prezentovaného algoritmu`. Nejdříve jsou načteny vstupní dokumenty a jsou převedeny do podoby stromů.

Definice Strom dokumentu: *Strom dokumentu je dvojice $S = (V, E)$, kde V je množina vrcholů, která odpovídá množině instancí elementů v dokumentu.*

E je množina hran taková, že $(a, b) \in E$ právě když má element a přímého potomka b.

Pak vytvoříme orientovaný graf závislostí elementů.

Definice Graf závislostí: *Graf závislostí je dvojice $G = (V, E)$, kde V je množina vrcholů, které odpovídají názvům elementů ve všech vstupních dokumentech. E je množina hran taková, že $(A, B) \in E$ právě když existuje na vstupu dokument, kde element s názvem A obsahuje přímého potomka s názvem B.*

Následují dvě stěžejní fáze algoritmu. Pro každý název elementu (uzel grafu závislosti) provedeme rozdělení instancí do skupin podle podobnosti. Pro každou skupinu pak nalezneme vhodné schéma vyjadřující obsah elementu, resp. strukturu jeho podřízených elementů. Nakonec jsou všechna nalezená schémata zpracována do společného schématu v jazyce XML Schema. Dvěma stěžejním sekcím jsou věnovány následující kapitoly.

```
stromyDokumentu = nactiDokumenty()
grafZavislosti = vytvorGrafZavislosti(stromyDokumentu)
rozdeleni = provedRozdeleniElementu(grafZavislosti, stromyDokumentu)
schemata = new Seznam
foreach skupina z rozdeleni
  schemata.pridej(vytvorSchema(skupina))
end for
vysledek = vytvorXMLSchema(schemata, stromyDokumentu)
```

13: Hlavní část prezentovaného algoritmu

5.4 Rozdělení elementů

Jazyk XML Schema dovoluje pro elementy se stejným názvem, které jsou však na různých místech struktury dokumentu, definovat různé schéma jejich obsahu. Jinými slovy element `název` může mít jiný sémantický význam (a jinou strukturu), pokud se nachází uvnitř elementu `kniha` a jiný, pokud se nachází uvnitř elementu `softwareový-projekt`. Prvním krokem prezentovaného algoritmu je tedy rozdělení elementů do skupin podle podobnosti. Aby bylo možné provést toto rozdělení, je třeba analyzovat jak na sebe elementy navzájem odkazují. Proto prvním krokem je vytvoření grafu závislostí.

Konstrukce grafu probíhá postupně procházením všech dokumentů jak je ukázáno v diagramu `14: Konstrukce grafu závislostí`. Nejdříve je vytvořen prázdný graf. Pak pro každý dokument vytvoříme uzel pro jeho kořen (pokud neexistoval) a vytvoříme pro kořen dokumentu zavislosti. Přidávání závislostí pak probíhá rekurzivně. Vytvoříme hrany z aktuálního uzlu do uzlů všech podelementů a rekurzivně pokračujeme pro jednotlivé podelementy. Protože vstupem jsou stromy je tento algoritmus konečný.

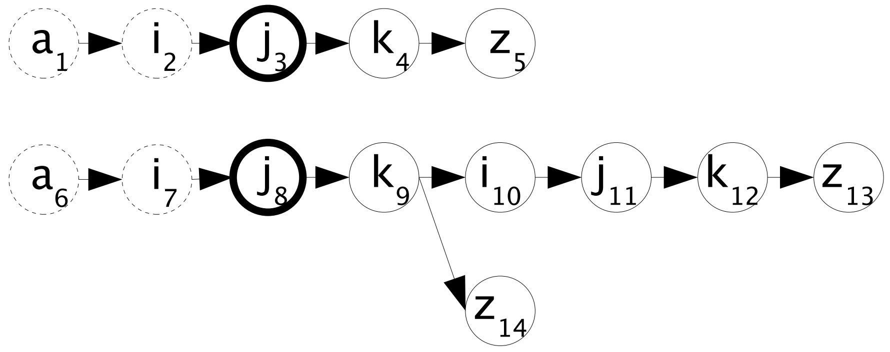
```
procedure vytvorGrafZavislosti(stromyDokumentu)
  graf = vytvorPrazdnyGraf()
  foreach dokument v stromyDokumentu
    if not graf.existujeUzelProElement(dokument.koren.nazev) then
      graf.vytvorUzelProElement(dokument.koren.nazev)
    end if
    pridejZavislostielementu(graf, dokument.koren)
  end for
end procedure

procedure pridejZavislostiElementu(graf, element)
  uzel = graf.najdiUzelProElement(element.nazev)
  foreach podelement v element.seznamDeti
    if not graf.existujeUzelProElement(podelement.nazev) then
      graf.vytvorUzelProElement(podelement.nazev)
    end if
    cilovyUzel = graf.najdiUzelProElement(podelement.nazev)
    graf.vytvorHranu(uzel, cilovyUzel)
    pridejZavislostiElementu(podelement)
  end for
end procedure
```

14: Konstrukce grafu závislostí

Vzdálenost elementů

Pro samotné rozdělení elementů do skupin je třeba definovat podobnost elementů. Struktura elementu je dána podstromem, v jehož kořeni je daný element. Definujeme tedy metriku, která nám pro každou dvojici stromů dává jejich vzdálenost založenou na podobnosti. Vzdálenost stromů definujeme jako editační vzdálenost, tedy počet operací potřebných k přetransformování jednoho stromu v druhý. Uvažované operace jsou odebrání listu a přidání listu (samozřejmě včetně hrany). Vzdálenost stromů budeme ještě počítat relativně k jejich velikosti, takže počet editačních operací podělíme součtem velikostí obou stromů.

Taková definice vzdálenosti stromů však nebude vhodná pro rekurzivní stromy, tedy stromy, kde uzel může být (nepřímým) potomkem sám sobě(což je častý případ v reálných dokumentech, viz `[15]`). Na obrázku `15: Editační vzdálenost rekurzivních stromů` jsou takové 2 stromy ukázány. Jejich schéma odpovídá rekurzivní struktuře, kdy element $\boldsymbol{a}$ obsahuje $\boldsymbol{i}$, ten obsahuje $\boldsymbol{j}$, ten obsahuje $\boldsymbol{k}$ a ten obsahuje buď $\boldsymbol{z}$ nebo znovu $\boldsymbol{i}$. Podle definované metriky by vzdálenost tlustou čarou označených vrcholů ***j*** byla 4. Avšak podstromy vznikly z téže definice, a tak by (ideálně) měla být vzdálenost 0.



15: Editační vzdálenost rekurzivních stromů

Modifikujme tedy tuto metriku takto: pokud se porovnávané stromy v podstromu neshodují (v obrázku uzel i_{10}), hledáme v podstromu znovu kořen druhého ze stromů (protože v obrázku porovnáváme vzdálenost uzlů `j`, budeme hledat uzel `j3`) a zkoušíme podobnost znovu. Nakonec vybereme nejlepší variantu.

Stejný problém nastane, pokud budeme porovnávat vrcholy `a`. U vrcholu `i10` se stromy neshodují. Kořen druhého stromu (tentokrát `a1`) také není možné nalézt v podstromu. Avšak uzel `i` už se v prvním podstromě nacházel. Můžeme tedy zkusit porovnávat stromy znovu z tohoto bodu (tedy vrcholy `i10` a `i2`).

Počet operací spočteme rekurzivním průchodem stromů jak je naznačeno v diagramech `16: Vzdálenost stromů (1)` a `17: Vzdálenost stromů (2)`. Začneme u kořenů stromů a procházíme stromy do hloubky a postupujeme po společných hranách. Jakmile nenalezneme společné hrany, pak zkoušíme hledat v příslušném podstromě všechny výskyty kořene druhého stromu a v cestě ke kořeni druhého stromu hledat výskyt nespárovaného podelementu (viz. diagram `17: Vzdálenost stromů (2)`). V diagramu `19: Editační vzdálenost stromů` jsou ukázány 2 podstromy. Rozdílné hrany jsou v obrázku vyznačeny tečkovaně. Tedy při porovnávání podelementů elementů `B`, narazíme na rozdílné hrany vedoucí do elementu `D`, resp. `E`. Postrom `E` obsahuje další hranu, a tak je počet editačních operací roven 3.

Pořadí dělení

Elementy budeme rozdělovat v pořadí podle grafu závislosti. Máme dvě možnosti - postupovat od uzlů s nejmenším výstupním uzlem nebo od uzlů s nejmenším vstupním uzlem. První varianta odpovídá postupu "od listů" dokumentů. Tato metoda má výhodu, že nemusíme zkoumat celé podstromy uzlů, ale již rozdělené uzly/elementy můžeme považovat za listy (typicky tedy

```
procedure vzdalenostStromu(strom1, strom2)
  koren1 = strom1.koren
  koren2 = strom2.koren
  vzdalenost = vzdalenostUzlu(koren1, koren2, koren1, koren2)
  return vzdalenost / (strom1.velikost + strom2.velikost)
end procedure
```

16: Vzdálenost stromů (1)

porovnáváme stromy hloubky 1). Tím snižujeme celkovou výpočetní náročnost algoritmu. Avšak pokud prohlásíme dva elementy za rozdílné, a některé jejich předky (stejného jména) i přes tuto skutečnost sloučíme, musíme se ve výpočtu vrátit a rozdílné elementy sloučit (respektive přerozdělit jejich množinu). Jiným řešením problému by bylo zakázat sloučení předků, jejichž potomci téhož jména jsou rozděleni. Druhá varianta postupu v grafu závislostí odpovídá postupu "od kořenů". V této variantě je nutno porovnávat celé, potenciálně hluboké podstromy avšak nikdy není třeba se vracet.

Dělící algoritmus

Pro samotné rozdělení elementů je použit spojovací (agglomerative), jednoduchý (Partitional Clustering) algoritmus (detaily těchto algoritmů se zabývá [13]). Tedy algoritmus, který začíná s maximálně oddělenými skupinami a postupně slučuje vzory do skupin, a který vytváří jedno rozdělení, nikoliv jejich hierarchii. Konkrétně je použit upravený algoritmus Mutual Neighborhood Clustering. Algoritmus je parametrizován třemi parametry. Minimální vzdáleností, maximální vzdáleností a faktorem `K`. Definujeme vlastnost vzájemného sousedstv takto:

Definice Vzájemné sousedství: *Mějme vzory* `A` *a* `B`*. Nechť* `A` *je* `i`*-tý nejbližší soused* `B` *a* `B` *je* `j`*-tý nejbližší soused* `A`*. Pak Hodnotu vzájemného sosedství definujeme jako* $MNV(A, B) = i + j$

Algoritmus pak umístí 2 prvky `A` a `B` do stejné skupiny, pokud

- vzdálenost `A` a `B` není větší jak maximální vzdálenost
- vzdálenost `A` a `B` je menší jak minimální vzdálenost
- nebo $MNV(A, B) \leq K$

Celý algoritmus je uvedený v diagramu `21: Rozdělení elementů do skupin`. Na začátku rozdělíme elementy do skupin podle cesty k nim. Za určitých okolností by bylo možné definovat pro dva elementy stejného jména pod jedním rodičem dvě různá schémata. Takové dva dokumenty by však musely být přítomné v jiných částech schématu rodiče. Avšak takové schéma by bylo velmi matoucí (viz. příklad `20: Schéma s dvěma elementy stejného jména`), instance ta-

```
procedure vzdalenostUzlu(uzel1, uzel2, koren1, koren2)
  seznam1 = setridPotomky(uzel1) //setridim poduzly podle nazvu
  seznam2 = setridPotomky(uzel2)
  vzdalenost = 0
  while seznam1.nejsemNakonci  a seznam2.nejsemNakonci
    potomek1 = seznam1.aktualniPrvek
    potomek2 = seznam2.aktualniPrvek
    if potomek1.nazev = potomek2.nazev then
    v = vzdalenostUzlu(potomek1, potomek2, koren1, koren2)
      vzdalenost = vzdalenost +
v seznam1.dalsiPrvek
      seznam2.dalsiPrvek
    else if potomek1.nazev < potomek2.nazev then
      hodnotaPodstrom = Podstrom(potomek1, koren1, koren2)
      hodnotaVzhuru = Vzhuru((potomek1, uzel2, koren1, koren2)
      min=minimum(hodnotaPodstrom, hodnotaVzhuru)
      vzdalenost = vzdalenost + min
      seznam1.dalsiPrvek
    else
      hodnotaPodstrom = Podstrom(potomek2, koren2, koren1)
      hodnotaVzhuru = Vzhuru((potomek2, uzel1, koren2, koren1)
      min=minimum(hodnotaPodstrom, hodnotaVzhuru)
      vzdalenost = vzdalenost + min
      seznam2.dalsiPrvek
    end if
  end while
  while seznam1.nejsemNakonci
    potomek1 = seznam1.aktualniPrvek
    hodnotaPodstrom = Podstrom(potomek1, koren1, koren2)
    hodnotaVzhuru = Vzhuru((potomek1, uzel2, koren1, koren2)
    min=minimum(hodnotaPodstrom, hodnotaVzhuru)
    vzdalenost = vzdalenost + min
    seznam1.dalsiPrvek
  end while
  while seznam2.nejsemNakonci
    potomek2 = seznam2.aktualniPrvek
    hodnotaPodstrom = Podstrom(potomek2, koren2, koren1)
    hodnotaVzhuru = Vzhuru((potomek2, uzel1, koren2, koren1)
    min=minimum(hodnotaPodstrom, hodnotaVzhuru)
    vzdalenost = vzdalenost + min
    seznam2.dalsiPrvek
  end while
  return vzdalenost
end procedure
```

17: Vzdálenost stromů (2)

kového schématu by byly již téměř nepochopitelné a praktická použitelnost je velmi diskutabilní. Proto tuto možnost nebudeme uvažovat a elementy stej-

```
procedure Podstrom(potomek, koren1, koren2)
  podstrom = najdiVPodstrome(potomek, koren2)
  min = nedefinovano
  foreach uzel in podstrom
    v = vzdalenost(uzel, koren2, koren1, koren2)
    if v<min then min=v
  end for
  return min
end procedure

procedure Vzhuru(potomek1, rodic2, koren1, koren2)
  vzhuru = najdiVCesteOdKorene(rodic2, potomek1)
  min = nedefinovano
  foreach uzel in vzhuru
    v = vzdalenost(potomek1, uzel, koren1, koren2)
    if v<min then min=v
  end for
  return min
end procedure
```

18: Vzdálenost stromů (3)

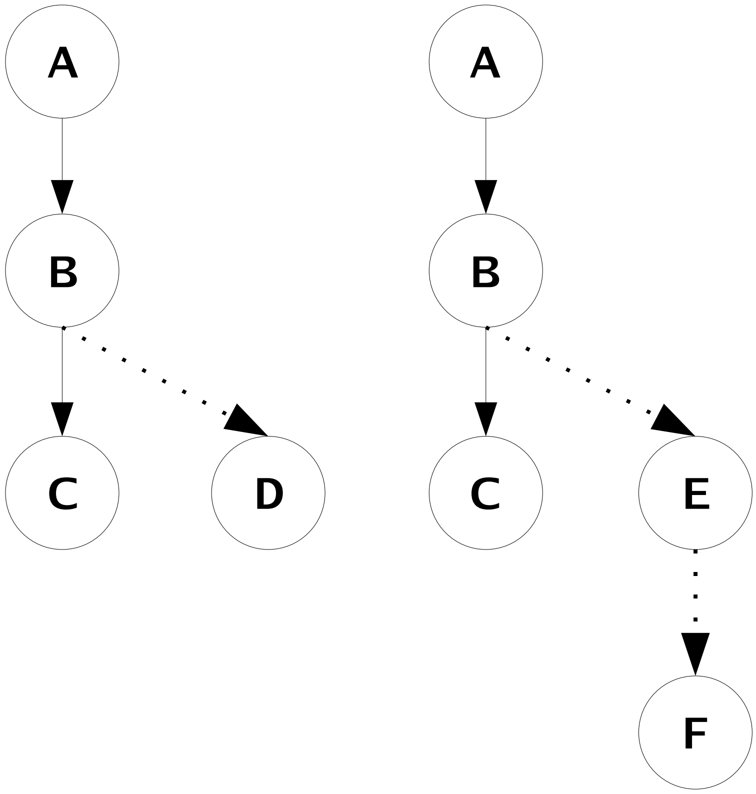
ného jména stejného rodiče budeme implicitně zařazovat do stejné skupiny (přiřadíme jim ve výsledku společné schéma).

Po úvodním rozdělení spočteme pro každou skupinu jejího reprezentanta a pro každou dvojici skupin spočteme jejich vzdálenost, respektive vzdálenost jejich reprezentantů. Z matice vzdáleností je spočtena matice sousednosti.

5.5 Odvozování regulárních výrazů

Cílem algoritmu je pro dané vstupní sekvence entit (pro účely tohoto podprogramu můžeme mluvit o entitách jazyka daného vstupními sekvencemi) odvodit regulární výraz, který postihuje minimálně všechny vstupní sekvence a je vhodně zobecněný, aby byl výsledný výraz jednoduchý, ale ne příliš zobecněný. Pro tvorbu regulárních výrazů uvažujeme běžné operátory konkatenace, exkluzivního výběru a iterace. Navíc zavádíme, ne zcela běžný operátor permutace, který zajišťuje, že se podvýrazy mohou vyskytovat v libovolném pořadí, avšak každý maximálně jednou. Tento operátor nezvyšuje vyjadřovací sílu regulárních jazyků, neboť je ho možné nahradit vnořenými exkluzivními výběry, ale počet operací výběru roste exponenciálně s počtem elementů, a tak operátor permutace výrazně zjednodušuje zapsaný výraz. Navíc je libovolné pořadí elementů zcela běžné v XML dokumentech používaných v praxi (viz. [1]).

Problematika odvozování regulárních výrazů už byla rozebírána v mnoha pracech. Zde je uveden algoritmus, který čerpá z poznatků předchozích řešení,



19: Editační vzdálenost stromů

```
<xs:element name="rodic">
<xs:complexType>
<xs:choice>
<xs:sequence>
<xs:element name="A" type="TA" />
<xs:element name="B" type="TB1" />
</xs:sequence>
<xs:sequence>
<xs:element name="C" type="TC" />
<xs:element name="B" type="TB2" />
</xs:sequence>
</xs:choice>
</xs:complexType>
</xs:element>
```

20: Schéma s dvěma elementy stejného jména

snaží se výhody některých těchto řešení spojit dohromady a přidat nové vlastnosti. Uvedený algoritmus je heuristický, ve smyslu jak je uvedeno v kapitole

```
procedure rozdelElement(seznam)
  vzdalenost = new matice
  foreach dvojice (a,b) ze seznam
    vzdalenost[a,b] = vzdalenost[b,a] = vzdalenostStromu(a,b)
  end for
  sousede = new matice
  foreach dvojice (a,b) ze seznam
    sousede[a,b] = poradi hodnoty na radce ve vzdalenost
  end for
  foreach dvojice (a,b) ze seznam
    if vzdalenost > MAX then continue;
    if vzdalenost < MIN then slouči skupiny
    else if sousede[a,b] + sousede[b,a] < K then slouči skupiny
  end for
  return skupiny
end procedure
```

21: Rozdělení elementů do skupin

`Rozdělení`. Navíc je tento algoritmus dokonce nedeterministický. Podle dělení v uvedené kapitole se jedná o algoritmus konstruktivní, i když již na rozhraní s výběrovými, protože konstruuje několik řešení a z nich vybírá. Uvedený algoritmus by se dal charakterizovat jako iterativní heuristický algoritmus s pozitivní (a částečně i negativní) zpětnou vazbou. Jako základ je použita modifikovaná optimalizace nazývaná ACO (Ant Colony Optimisation) prezentovaná v `[1]`.

5.5.1 Princip MDL

Aby bylo možné procházet prostorem řešení a hledat nejlepší zobecnění vstupních sekvencí, je třeba nadefinovat metriku hodnocení kvality schématu. Toto hodnocení musí zohledňovat dva protichůdné požadavky - obecnost a zachování detailů.

Pro reprezentaci řešení jsou použita gramatická pravidla tří druhů. První druh je jednoduché pravidlo tvaru $A \to tB$, kde A a B jsou neterminály a t je terminál. Toto pravidlo popisuje přijetí jednoho terminálu vstupu. Druhé pravidlo popisuje tzv. lambda přechod, tedy změnu stavu (neterminálu) bez posunu v přijímání vstupu. Pravidlo má tvar $A \to B$, kde A a B jsou neterminály. Poslední pravidlo zavádí permutační operátor. Má tvar $A \to B_1\&...\&B_nC$. Symboly A, B_i a C jsou neterminály. Neterminál A můžeme přepsat na neterminály B_i v libovolném pořadí, přičemž na konci se vždy vyskytuje neterminál C. Abychom zůstali v hranicích regulárních výrazů, je třeba, aby platilo

$$\forall i : not B_i \to^* A$$

nesmí tedy být jakýmkoliv způsobem možné odvodit z permutujících podčástí cyklicky zdrojový neterminál. Taková gramatika už by neodpovídala regulární gramatice, ale bezkontextové gramatice.

Pro hodnocení schématu je použit princip MDL (Minimum Description Length). Dobré schéma je dostatečně jednoduché (s čímž souvisí vhodné zobecnění), ale zároveň postihuje dostatečné množství detailů. Jednoduché schéma je malé, ve smyslu počtu stavů (neterminálů). Naopak schéma, které poskytuje velké množství detailů je takové, které umožňuje zakódovat instance schématu pomocí krátkých kódů (protože mnoho informací je již ve schématu a není potřeba je kódovat). Z tohoto principu vychází zvolené hodnocení, které se skládá z velikosti schématu a velikosti kódů pro vstupní řetězce. Velikost schématu spočteme jeko součet velikostí pravých částí pravidel gramatiky.

Pro velikost kódování instance `i` mějme posloupnost p_i pravidel schématu, jak jsou použita k přepsání počátečního neterminálu na instanci `i`. Velikost kódování instancí je součet počtu bitů potřebných k vybrání každého pravidla z této posloupnosti. Výběr pravidla k přepsání neterminálu je jednoduše jeho pořadové číslo. Počet bitů potřebných k zakódování pořadového čísla (použijeme binární kódování) je $\lceil \log_2(N) \rceil$, kde N je počet pravidel, kde je shodný neterminál na levé straně.

5.5.2 Ant Colony Optimisation

Hlavní modifikace algoritmu spočívá ve změně hodnocení nalezených schémat (a tím pádem i ve změně hodnocení kroků mravenců). Menší změnou je pak zavedení dočasné negativní zpětné vazby. Princip algoritmu by se dal popsat takto: Několik mravenců prochází prostorem a na základě jednoduché heuristiky hledá řešení. Každý takový mravenec najde některé lokální optimum funkce. V dalších iteracích pak mravenci využívají zkušeností mravenců z minulých iterací v podobě pozitivní zpětné vazby. Tedy informace o tom jak dobré řešení nalezli předchozí mravenci s pomocí daného kroku.

Dočasná negativní zpětná vazba funguje v rámci každé jedné iterace (na konci iterace je vynulována) a přiřazuje se ve chvíli, kdy mravenec provede krok. Tato negativní zpětná vazba je zavedena, aby mravenci v rámci iterace prošli co nejširší prostor řešení. Kdykoliv mravenec provede krok, přiřadí mu vhodné menší negativní hodnocení, takže má tento krok pro ostatní mravence nižší pravděpodobnost, čímž jsou nuceni použít jinou cestu. Ve většině případů nezávisí výsledné řešení na pořadí kroků mravence, a tak omezení kroku, který už byl proveden jiným mravencem, není újmou k nalezení správného řešení. Kdyby omezený krok měl vést k řešení, má možnost toto řešení najít mravenec, který provedl krok první.

Kostra heuristiky je uvedena v diagramu `22: Odvozování regulárních výrazů - hlavní program`. Nejdříve je z podelementů vstupních elementů vytvořena abeceda a vstupní slova. Z těchto vstupních slov je vytvořen prefixový strom (respektive gramatika odpovídající prefixovému stromu). Následně jsou prováděny hlavní iterace algoritmu. V každé iteraci je vytvořeno pole mravenců, tedy objektů procházejících prostor řešení. Každý mravenec prochází prostorem řešení dokud je živý. Jakmile mravenec zemře (zastaví se), prochází prostorem další. Mravenci v jedné iteraci se ovlivňují pouze negativně - vždy, když mravenec provede krok (posune se v prostoru řešení), přiřadí tomuto kroku postih. Pozitivní zpětná vazba je aktualizována hromadně, vždy na konci iterace. Jakmile nastala v iteraci změna k lepšímu, je vynulován čítač cyklů, jinak je čítač zvětšen. Algoritmus skončí, jakmile v daném počet po sobě následujících cyklů nenastane změna.

Stěžejním bodem výše uvedeného algoritmu je krok mravence. Příslušný podprogram je uveden v diagramu `23: Krok mravence`. Každý krok se skládá z generování možných přesunů, jejich ohodnocení a provedení. Pro generování možných přesunů může být, a je použito několika různých metod. Díky možnosti ohodnotit všechny kroky jednotným způsobem a následně vybrat nejvhodnější přesun, můžeme v každém stavu zvolit nejvhodnější metodu zobecnění. Tímto kombinujeme několik různých metod odvozování regulárních výrazů a jejich výhody.

Generování možných kroků popíšeme v samostatné kapitole. V kapitole `Princip MDL` byla popsána metoda MDL pro hodnocení nalezeného řešení. Stejného principu využijeme při hodnocení kroku mravence. Hodnocení kroku mravence je tedy rozdíl hodnocení schématu před provedením kroku a po jeho provedení. K hodnocení kroku ještě přičteme bonus z předchozích iterací a odečteme postih, pokud některý mravenec již podobný krok provedl.

Uvažujeme 2 různé druhy kroků mravence. První je jednoduché sloučení neterminálů, druhý je zavedení permutačního pravidla. Zavedení permutačního operátoru také znamená sloučení odpovídajících si částí v různých alternativách. A tak je možné unifikovat bonusy kroku do jediné matice, kde evidujeme bonus pro sloučení dvou neterminálů. U permutačního pravidla je pak výsledný bonus průměr z bonusů na sloučení jednotlivých částí (jejich počátků).

5.5.3 Generování kroků mravence

Metoda k,h-context

Pro generování přesunů je použito tří metod. Metoda `k,h-context` je převzatá z `[3]`. V této práci je definována identifikovatelná podtřída regulárních jazyků, která předpokládá omezený kontext elementů.

Definice Ekvivalence k,h-context: *Mějme slovo* $\mathtt{s}s_1,\ldots,s_k$ *délky* `k`. *Mějme stavy* p_i *a* q_i *takové, že* $d(p_i, s_{i+1}) = p_{i+1}$, $d(q_i, s_{i+1}) = q_{i+1}$ *a* $p_k = q_k$. *Pak* $\forall_{i \geq k-h}$: $p_i \equiv q_i$.

```
procedure vytvorSchema(skupina)
  priklady = new Pole[skupina.pocet]
  i = 0
  foreach element ve skupina
    priklady[i] = vytvorJazykoveEntityZPodelementu(element)
    i = i + 1
  end for
  schema = vytvorPrefixovyStrom(priklady)
  cyklus = 0
  bonusy = new MaticeBonusu
  nejlepsiReseni = neexistuje
  while cyklus < MAXIMUMCYKLU
    zmena = false
    postihy = new MaticePostihu
    mravenci = vytvorPoleMravencu(schema, bonusy, postihy)
    soucetHodnoceni = 0
    foreach mravenec v mravenci
      while mravenec.jeZivy
        mravenec.udelejKrok
      end while
      if mravenec.reseni.hodnoceni < nejlepsiReseni.hodnoceni then
        nejlepsiReseni = mravenec.reseni
        zmena = true
      end if
      soucetHodnoceni = soucetHodnoceni + mravenec.reseni.hodnoceni
    end for
    foreach mravenec v mravenci
    pomernyVysledek = mravenec.reseni.hodnoceni / soucetHodnoceni
      bonusy.aktualizujBonusy(mravenec.cesta, pomernyVysledek)
    end for
    if zmena then
      cyklus = 0
    else
      cyklu = cyklus + 1
    end if
  end while
  return nejlepsiReseni
end procedure
```

22: Odvozování regulárních výrazů - hlavní program

Dva elementy jsou prohlášeny za identické, pokud existují stejné cesty délky `k`, které končí v těchto elementech. Navíc pokud existují takové dvě cesty, tak prohlásíme za identické nejen poslední elementy, ale i `h` předchozích. V programu procházíme do hloubky celou gramatiku od jejího kořene a postupně zaznamenáváme všechny cesty dané délky. Poté vyhodnotíme, které

```
procedure Mravenec::udelejKrok()
  moznosti = new Seznam
  pridejKHKontextoveMoznosti(moznosti, this.aktualniReseni)
  pridejSKStringMoznosti(moznosti, this.aktualniReseni)
  pridejPermutacniMoznosti(moznosti, this.aktualniReseni)
  ohodnotMoznosti(moznosti)
  krok = vyberNahodne(moznosti)
  MaticePostihu.aktualizujPostih(krok, krok.hodnoceni*KOEFICIENTPOSTIHU)
  krok.proved(this.aktualniReseni)
end procedure
```

23: Krok mravence

cesty se vyskytují vícekrát a sloučení příslušných neterminálů přidáme mezi možnosti dalšího pohybu mravence.

Metoda s,k-string

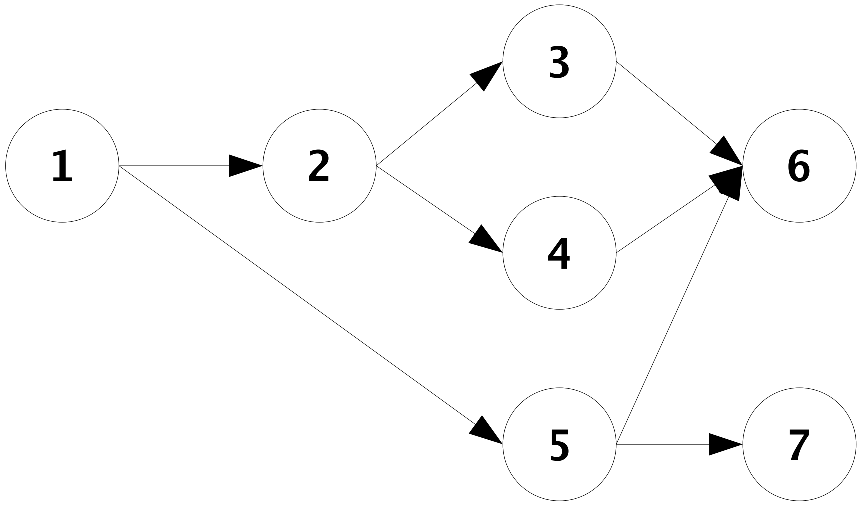
Druhou metodou je metoda s,k-string prezentovaná v [1]. Tato metoda vychází z Nerodovy ekvivalence stavů, že stavy automatu jsou ekvivalentní, pokud všechny cesty vedoucí z těchto dvou stavů do koncových stavů jsou identické. Taková podmínka je příliš striktní a těžko vyhodnotitelná. Omezíme se tedy místo na cesty, na k-stringy. K-string je cesta, která má délku `k` nebo končí v koncovém stavu. Stavy jsou potom ekvivalentní, pokud k-stringy z nich vedoucí jsou identické. Zde existuje několik forem této podmínky. Mějme uzly `A` a `B`. Pro každý uzel mějme seznam `k-stringů` S_A(resp. S_B) vedoucích z uzlu `A` (resp. `B`). Pro první variantu platí:

$$S_A \subset S_B \text{ a zároveň } S_B \subset S_A$$

Ostatní formy jsou analogií té první, například:

$$S_A \subset S_B \text{ nebo } S_B \subset S_A$$

V této práci je použita varianta "a zároveň". Tuto podmínku ještě dále uvolníme tak, že budeme pro ekvivalenci uvažovat jen `s` procent nejpravděpodobnějších k-stringů. Při konstrukci automatu si zaznamenáváme, kolikrát je která hrana použita. Poměr použití hran z jednoho vrcholu nám pak dává jejich pravděpodobnost. Pravděpodobnost cesty je součin pravděpodobností všech hran v cestě obsažených. Uvolnění uvažující pouze nejpravděpodobnější cesty reflektuje ignorování minoritních singularit. V programu tedy pro každou dvojici uzlů vygenerujeme k-stringy, ty seřadíme podle pravděpodobnosti a porovnáme.



24: Ukázka Společného následníka

Permutační operátory

Jak již bylo řečeno, zcela novou vlastností uvedeného algoritmu je zavedení permutačních operátorů. Příslušný algoritmus je uveden v diagramu `25: Generování permutačních kroků`. Pro hledání možností zavedení permutačních operátorů definujeme tzv. Společné následníky uzlu.

Definice Společný následník 1: *Společný následník uzlu je takový uzel, kterým procházejí všechny cesty vedoucí ze zdrojového uzlu.*

Na obrázku `24: Ukázka Společného následníka` je jednoduchý automat. Uzel 2 má v tomto automatu společného následníka 6. Naopak uzel 1 nemá žádného společného následníka, protože cesty končí ve dvou různých vrcholech 6 a 7. Alternativně je možné definovat společného následníka takto:

Definice Společný následník 2: *Množina společných následníků uzlu je průnik společných následníků jeho přímých následníků.*

Této definice využijeme při konstrukci dané množiny.

Dále definujeme Společné předchůdce. Toto je analogie Společných následníků, avšak cesty orientujeme proti směru šipek. Jinými slovy:

Definice Společný předchůdce 1: *Společný předchůdce je takový uzel, který je obsažen ve všech cestách přicházejících do daného uzlu.*

I alternativní rekurzivní definici můžeme analogicky zformulovat pro Společné předchůdce:

Definice Společný předchůdce 2: *Množina Společných předchůdců uzlu je průnik Společných předchůdců jeho přímých předchůdců.*

Avšak pro naše účely musíme tuto definici trochu upravit. Cílem této definice je ověření, zda neexistuje cesta, která neprochází uzlem `A`, ale prochází

```
procedure generujPermutacniKroky(reseni)
  foreach uzel v reseni
    if not uzel.vystupniStupen > 1 then continue
    spolecniNasledovnici = uzel.spolecniNasledovnici
    if spolecniNasledovnici.pocet = 0 then continue
    konecBloku = nejblizsi(uzel, spolecniNasledovnici)
    spolecniPredchudci = konecBloku.spolecniPredchudci(uzel)
    if not spolecniPredchudci.obsahuje(uzel) then continue
    generujPermutaceProBlok(uzel, konecBloku)
  end for
end procedure

procedure generujPermutaceProBlok(zacatekBloku, konecBloku)
  vetve = rozdelBlokNaVetve(zacatekBloku, konecBloku)
  horni = neprirazeno
  celkoveParovani = prazdne
  foreach vetev in vetve
    if horni je neprirazeno then
      horni = vetev
      continue
    end if
    spodni = vetev
    graf = prazdny
    foreach horniCast in horni.casti
      foreach dolniCast in dolni.casti
        podobnost = podobnost(horniCast, dolniCast)
        nova podobnost = podobnost
        graf.pridejHranu(horniCast, dolniCast, podobnost)
        do
          podobnost = novapodobnost
          dolniCast = dolniCast.rozsir
          novapodobnost = podobnost(horniCast, dolniCast)
          if novapodobnost > podobnost then
            graf.pridejHranu(horniCast, dolniCast, novapodobnost)
          end if
        while novapodobnost > podobnost
      end for
    end for
    parovani = najdiParovani(graf)
    celkoveParovani.pridej(parovani)
    horni = spodni
  end for
  pridej krok podle celkoveParovani
end procedure
```

25: Generování permutačních kroků

některým(i) uzly mezi `A` a `B`. Avšak cesty, které zasahují jen a pouze v bodě B nám nevadí. Modifikovaná definice bude tedy znít:

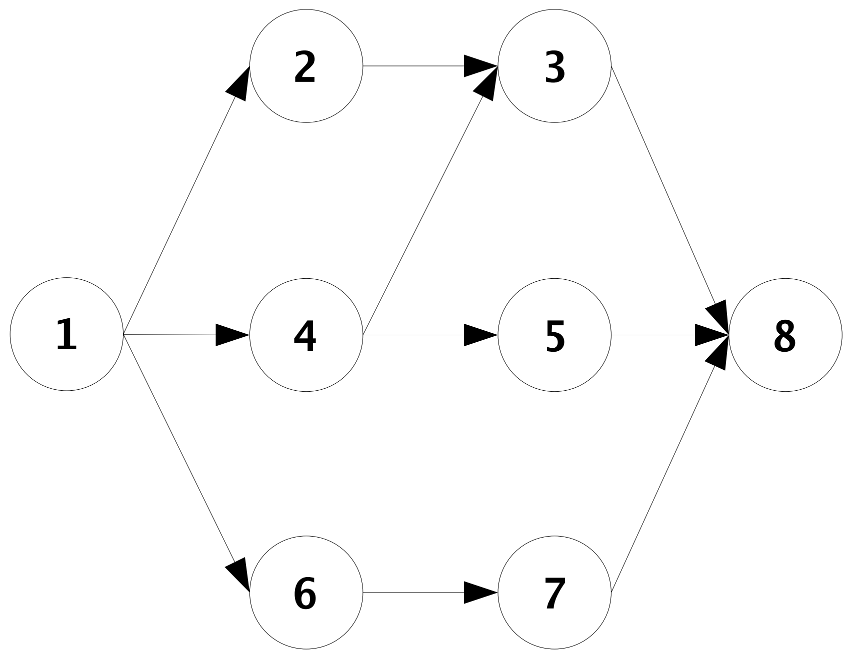
Definice Společný předchůdce 3: *Mějme blok mezi uzly* `A` *a* `B`. *Množinu Společných předchůdců* `B` *vzhledem k* `A` *definujeme jako průnik Společných předchůdců (nyní již úplných - podle původní definice "Společný předchůdce 2") všech přímých předchůdců, kteří mají (nepřímého) předka* `A`.

Na obrázku `24: Ukázka Společného následníka` má uzel 6 úplného společného předka uzel 1. Avšak společný předek uzlu 6, vzhledem k uzlu 2 je navíc sám uzel 2

Nyní již máme dostatečný aparát pro nalezení bloků, v nichž je možné zkoušet nalézt permutaci podvýrazů. Uzel automatu je začátkem takového bloku, pokud splňuje následující podmínky:

- Uzel má výstupní stupeň větší než 1. Jinými slovy, cesty se v tomto uzlu rozdělují.
- Množina společných následovníků je neprázdná. První (podle vzdálenosti) z těchto uzlů je pak konec bloku.
- Množina společných předků konce bloku vzhledem k začátku bloku obsahuje začátek bloku.

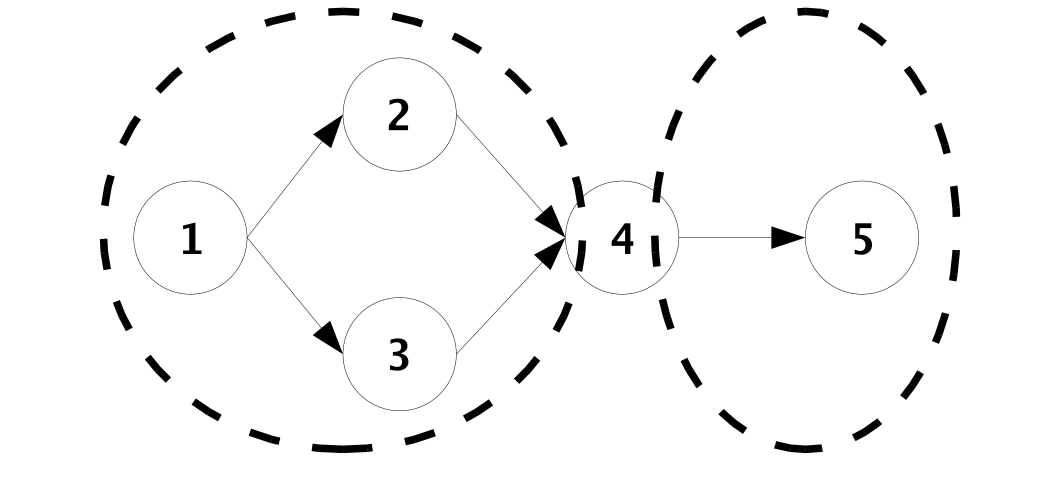
Tyto podmínky zajišťují, že se automat v daném uzlu větví, tedy existuje více alternativ, a že alternativy jsou úplné, v tom smyslu že nejsou cesty, které by tento blok opouštěly jinak než koncovým uzlem a analogicky neexistují cesty, které by do tohoto bloku přicházely jinudy než počátečním uzlem.



26: Rozdělení bloku na větve

Větve permutace

Každý blok, s potenciálem pro permutační operátory, je podroben podrobnější analýze. Dalším krokem je rozdělení bloku na jednotlivé alternativy - větve. Větev je množina cest, které začínají v počátečním uzlu bloku a uvnitř bloku se protínají v jiném bodě než je koncový bod bloku. Na obrázku `26: Rozdělení bloku na větve` je vidět schéma bloku, kde jsou 2 větve. Horní větev zahrnuje uzly 1,2,3,4,5,8 a druhá, spodní, větev zahrnuje uzly 1,6,7,8. Uzly 2,3 a 4,5 nemohou být rozděleny do různých větví, protože existuje hrana mezi uzly 4 a 3. Pokud v této fázi zjistíme, že existuje pouze jediná větev ve zkoumaném bloku, je hledání permutací irelevantní a je možné analýzu tohoto bloku ukončit.



27: Ukázka rozdělení větve

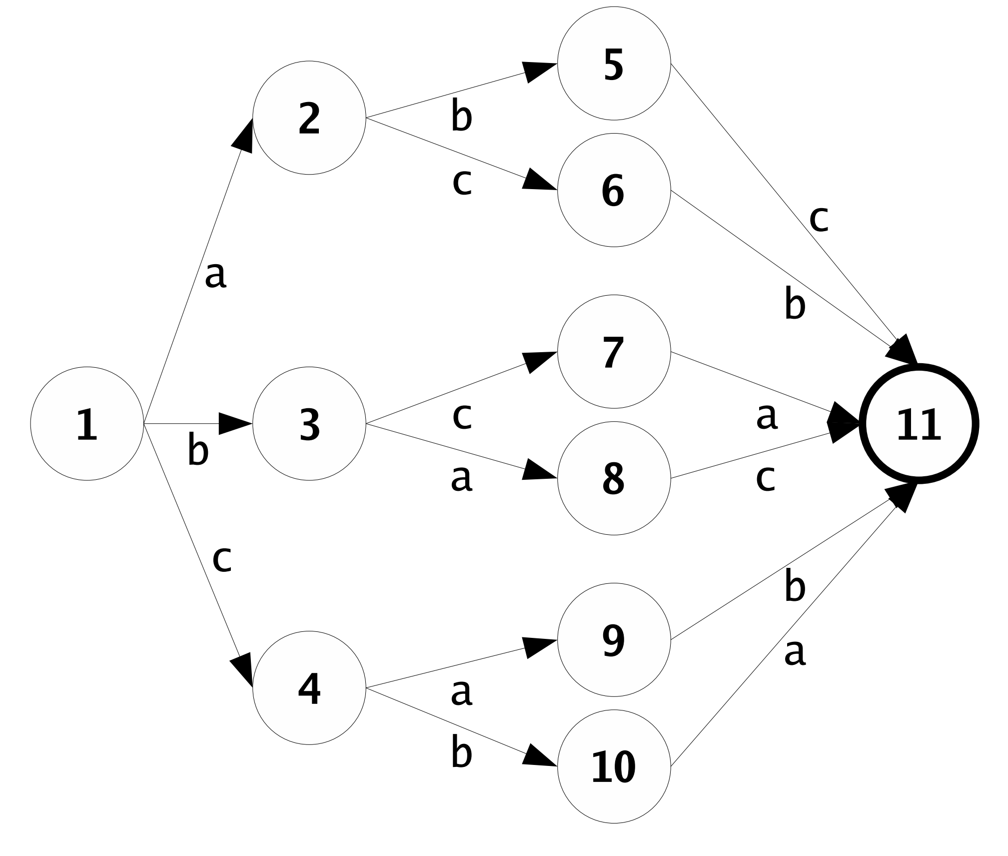
Každou identifikovanou větev je možné rozdělit na jednotlivé části potenciální permutace. Protože jednotlivé části jsou nezávislé na ostatních, v automatu musí existovat právě jeden dělící bod. Tedy uzel, kde je možné části rozdělit a neexistuje jiná cesta mimo tento dělící bod. Přesněji vyjádřeno:

Definice Dělící bod: *Dělící bod je uzel, kterým procházejí všechny cesty dané větve.*

K identifikaci takového uzlu použijeme již dříve definovanou množinu Společných následníků, takže další dělící bod je nejbližší společný následník aktuálního dělícího bodu. Prvním dělícím bodem je samozřejmě začátek bloku, v tomto bodě je však potřeba Společné následníky omezit na cesty vedoucí danou větví. Příklad rozdělení je vidět na obrázku `27: Ukázka rozdělení větve`. Rozdělení na části je vyznačeno tlustými šedými elipsami. Dělícím bodem je uzel 4, který obě elipsy sdílejí.

Definice Nedělitelná větev: *Nedělitelná větev je taková větev bloku, kde existují pouze 2 dělící body, a to začátek a konec bloku.*

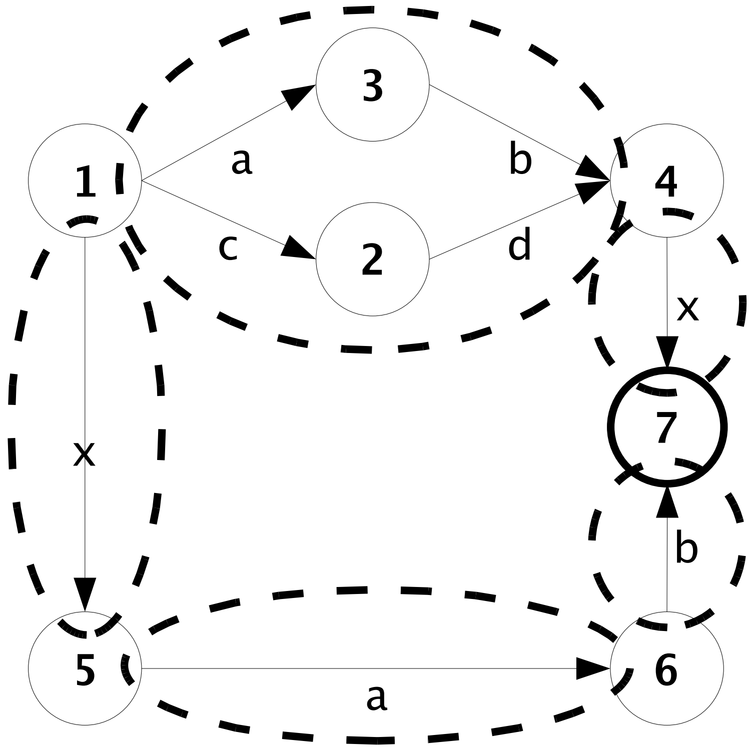
Nedělitelná větev by tak obsahovala pouze jedinou část a v takovém případě nemá smysl uvažovat o permutacích. Pokud tedy v bloku nalezneme nedělitelnou větev, můžeme analýzu bloku ukončit. Příklad nedělitelné větve je vidět na obrázku `26: Rozdělení bloku na větve`. Horní větev obsahující uzly 1,2,3,4,5,8 není možné rozdělit na části.



28: Postupné oddělování větví

Tím, že vytváříme ze vstupních vzorků prefixový strom, nebudou jednotlivé větve úplně izolované, ale větve se stejnou počáteční sekvencí budou mít i společnou část v prefixovém stromě. Na obrázku `28: Postupné oddělování větví` je prefixový strom pro řetězce `abc`, `acb`, `bac`, `bca`, `cab`, `cba`, což jsou všechny permutace písmen `a`, `b` a `c`. Respektive se jedná o prefixový strom se sloučenými koncovými stavy (do stavu 11). Protože na vstupu bylo 6 variant permutujících podvýrazů, mělo by být ve schématu 6 různých větví. Ale z počátečního stavu 1 vychází pouze 3 hrany a až v následujícím stavu se rozvětvují na 6 různých větví. Tuto skutečnost je třeba zohlednit při rozdělování větví na části.

Pokud nalezneme část, jejíž konec je totožný s koncovým uzlem bloku a tato podčást je rozdělitelná na víc jak jednu větev, je třeba prověřit i rozdělení na více samostatných větví. Avšak ne vždy je toto rozdělení správné. Proto zde vznikají 2 varianty výpočtu. Jedna, která použije část s více větvemi jako celek a druhá, která použije každou větev části zvlášť.



29: Ukázka spojování částí

Párování částí

Protože podvýrazy permutace mohou být složitější výrazy než sekvence terminálů, nemusí se části větví úplně shodovat. Vzhledem k metodě rozdělování větve na části se navíc nemusí shodovat ani počet částí v jednotlivých větvích i když se skutečně jedná o permutace částí. Taková situace nastane, pokud podvýraz povoluje alternativní sekvence a v některé větvi je zastoupena pouze jedna alternativa. Na obrázku `29: Ukázka spojování částí` je uveden příklad takové situace (šedivé elipsy ukazují rozdělení větví na části). Automat vznikl z instancí `abx`, `cdx` a `xab` regulárního výrazu $(ab|cd)\&x$. Algoritmus správně nerozpoznal, že v třetí větvi patří stavy 5,6 a 7 do jedné skupiny a rozdělil je ve stavu 6. Proto při párování nepárujeme přímo rozdělené části, ale skupiny po sobě následujících částí. Generované skupiny se překrývají. Při párování hledáme takové párování skupin, které pokrývají všechny nalezené části a nepřekrývají se.

Při párování procházíme po dvojicích po sobě následující větve a vždy pro tuto dvojici hledáme vhodné párování částí (skupin) tak, aby si co nejvíce odpovídaly. V ideálním případě by bylo třeba nalézt spojení obdobné párování, ale pro n-tice, tedy vždy jednu komponentu z každé větve spojit dohromady. Zde je problém zjednodušen na hledání párování a následné spojení párů dohromady. Navíc nehledáme párování mezi každou dvojicí větví, ale vždy mezi po sobě následujícími větvemi. Máme tedy 2 větve - horní a dolní.

Procházíme každou dvojici částí a spočteme jejich podobnost. Hranu s příslušným hodnocením pak přidáme do bipartitního grafu částí (skupin). Nyní zkusíme zvětšit část spodní větve o část následující (bezprostředně vpravo) a znovu spočítat jejich podobnost. Dokud se podobnost zlepšuje opakujeme rozšiřování částí a přidáváme do grafu nové hrany. Horní větev, až na vyjímku, kdy je to nejhornější větev celého bloku, už je rozdělena na skupiny z párování s předchozí větví. Skupiny horní větve je třeba zachovat, aby bylo možné spojit párování mezi větvemi dohromady. A tak je část horní větve rozšiřována pouze v jediném uvedeném případě.

Když je spočítána podobnost mezi všemi dvojicemi částí (a skupinami) dané dvojice větví, nalezneme jejich nejlepší párování. Je třeba nalézt takové párování skupin, které pokrývá všechny části, skupiny se nepřekrývají a součet hodnocení na hranách je největší (skupiny si jsou nejvíce podobné). Takový výběr hran (a skupin) pak jednoznačně určuje hledaný přesun mravence, a tak jej přidáme do seznamu možností.

5.5.4 Determinismus modelu

Jazyk XML Schema dovoluje svými konstrukcemi definovat nedeterministický regulární výraz. Omezením typu `<xs:all` ... je však vždy možné definovat deterministický konečný automat. Omezení permutačního typu je nutné, protože takový operátor regulárního výrazu nemá přímou obdobu v konečných automatech. A tak, pokud by bylo možné vytvořit nedeterministické schéma s permutačním operátorem, nešlo by vytvořit deterministický konečný automat, který by byl schopen validovat dokumenty podle tohoto schématu. Díky omezením kladeným na permutační typ(smí obsahovat pouze elmenty a nejvýše jednou) je vždy hrana rozšířeného konečného automatu reprezentující tento typ deterministická, a tak je možné pro celý výraz zkonstruovat deterministický automat.

Zde prezentovaný algoritmus může vygenerovat schéma, které z tohoto hlediska neodpovídá specifikaci jazyka XML Schema. Může tedy vygenerovat nedeterministický model. Z hlediska kvality schématu se jedná o lepší řešení, avšak podle takového schématu by bylo složité ověřovat validitu dokumentů.

Úprava (spíše by se jednalo o zjednodušení), která by zajistila konstrukci správného schématu podle specifikace jazyka XML Schema není složitá. Při rozdělování větví na části by byly části jasně definované na pouze jediný element (jediné pravidlo v řešení).

6 Implementace

Součástí práce je ukázková implementace navržených algoritmů v podobě aplikace nazvané Schema Miner. Tato kapitola obsahuje detaily implementace algoritmů, použité nástroje a knihovny a popis použití aplikace.

6.1 Použité nástroje

Pro ukázkovou implementaci navržených algoritmů byl zvolen jazyk Java (konkrétně byla aplikace testována s verzí SDK 1.4.2). Jazyk Java byl zvolen kvůli přenositelnosti výsledné aplikace a existenci knihoven pro práci s XML dokumenty. Pro zpracování XML dokumentů je použit SAX parser standardně dodávaný s Javou.

6.2 Architektura

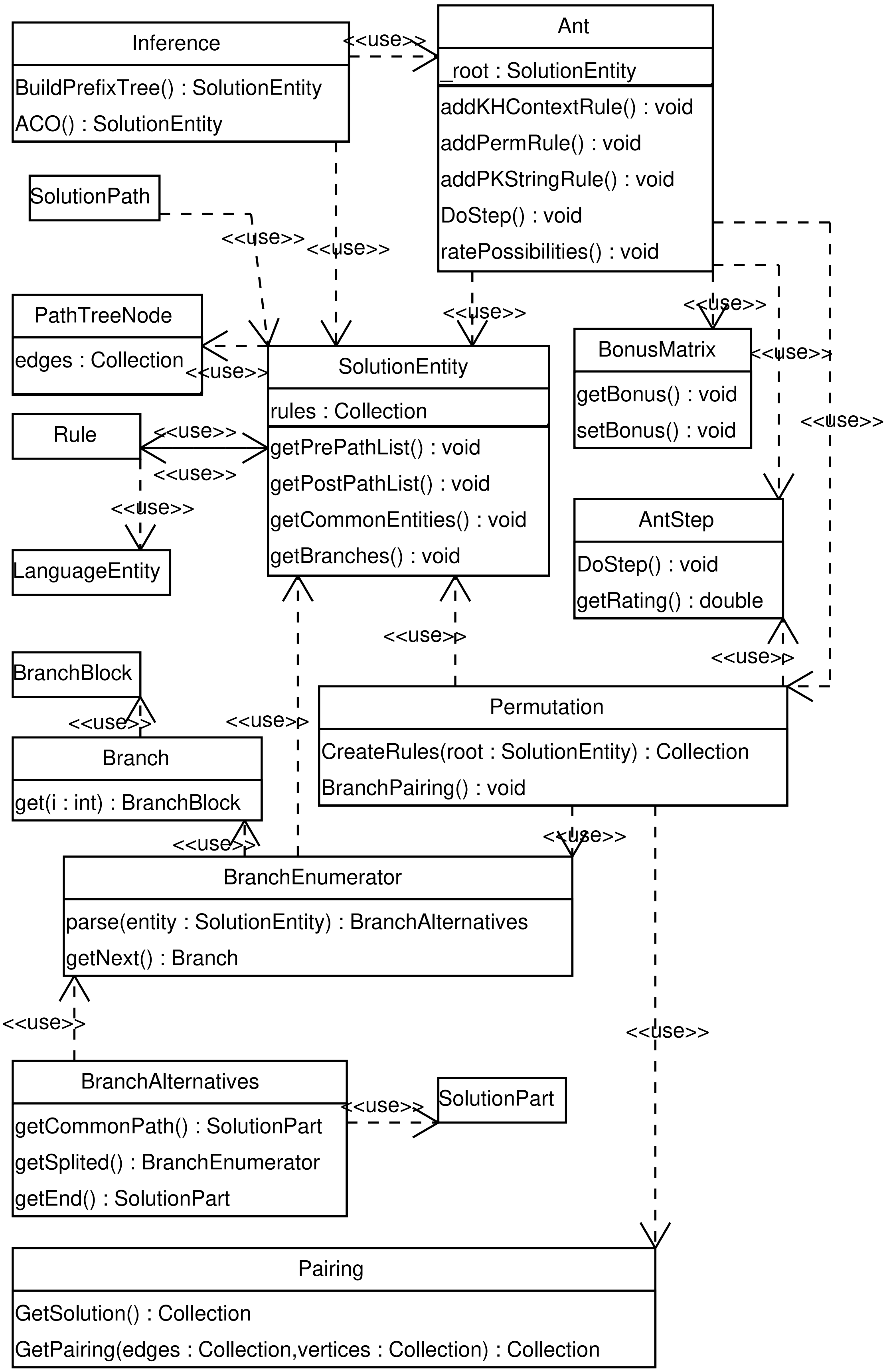
Architektura aplikace sestává ze dvou základních a několika pomocných balíků. Tyto balíky odpovídají dvěma stěžejním vlastnostem navrženého algoritmu.

- **Default:** Implicitní balík obsahuje třídy pro zpracování vstupu a zápis výstupu
- **Clustering:** Tento balík obsahuje třídy implementující algoritmus pro rozdělení elementů do skupin podle podobnosti.
- **GI:** Tento balík obsahuje třídy pro odvozování regulárních výrazů ze vstupních řetězců.
- **Xml:** Tento balík obsahuje pomocné algoritmy pro práci s XML dokumenty a XML Schématy

6.2.1 Balík Clustering

Tento balík implementuje algoritmy pro tvorbu grafu závislostí a rozdělení elementů do skupin podle podobnosti. Balík se skládá z těchto tříd:

- **DependencyGraphNode:** Reprezentuje uzel grafu závislosti
- **NodeCluster:** Reprezentuje skupinu podobných uzlů
- **DependencyBuilder:** Obsahuje metody pro tvorbu grafu závislosti
- **ClusterBuilder:** Obsahuje metody pro vytvoření skupin



30: UML diagram tříd balíku GI

6.2.2 Balík GI

Tento balík implementuje algoritmy pro odvozování regulárních výrazů z pozitivních příkladů, které byly navrženy v kapitole `Odvozování regulárních výrazů`. Balík se skládá ze tří hlavních tříd a několika pomocných, ve většině případů statických (obsahují pouze statické metody implementující konkrétní část algoritmu), tříd. UML diagram tříd většiny tříd v balíku GI je uveden na obrázku `30: UML diagram tříd balíku GI`. Nejdůležitější jsou následující třídy:

- **Infernece:** Tato třída, respektive její metoda ACO, je vstupním bodem celého balíku. Nejdříve volá metodu BuildPrefixTree a poté vytváří třídy Ant a provádí kostru heuristiky ACO.
- **Ant:** Tato třída reprezentuje mravence, tedy entitu procházející prostorem řešení a vyhledávající optimální řešení. Zde jsou implementovány algoritmy pro generování kroků na základě analýzy současného řešení. Dále obsahuje metody pro hodnocení nalezených kroků a výběr kroku.
- **SolutionEntity:** Tato třída reprezentuje stav pseudoautomatu použitého k popisu řešení. Tato třída zároveň implementuje algoritmy pro analýzu struktury automatu, jako je hledání společných následníků a předchůdců, hledání společných cest a k-stringů. Také implementuje hodnocení schématu podle principu MDL. Dále implementuje operace nad schématem jako je slučování a rozdělování uzlů.

Důležité statické třídy jsou zejména tyto:

- **Permutation:** Tato třída obsahuje algoritmus pro zavádění permutačních operátorů.
- **Pairing:** Tato třída implementuje algoritmus na hledání párování mezi větvemi automatu.

Zajímavější pomocné třídy jsou pak tyto:

- **AntStep:** Tato třída reprezentuje možné kroky mravence.
- **Rule:** Pravidla přechodů mezi stavy (neterminály, objekty SolutionEntity)
- **BonusMatrix:** Třída pro objekty udržující bonusy pro jednotlivé kroky mravenců. Pro bonusy i postihy je použita stejná třída.
- **BranchEnumerator:** Objekty pro procházení jednotlivými větvemi při hledání permutačních operátorů.

6.3 Ovládání

Ukázková implementace se spouští z příkazového řádku. Jako parametry programu se na příkazové řádce zadávají XML dokumenty, které mají být zpracovány. Pokud je parametrem adresář, program prohledá tento adresář včetně všech podadresářů a všechny soubory s příponou .xml zařadí do zpracování.

Výsledné schéma program vypisuje na standardní výstup. Pokud program uspěje, pak je jeho návratový kód 0, jinak je jeho návratový kód menší než nula a specifikuje kód chyby. Při výskytu libovolné chyby je na chybový výstup vypsána zpráva obsažená ve výjimce, která pád ohlásila. Přehled chybových kódů:

- **-1:** Obecná chyba, blíže nespecifikovatelná.
- **-2:** Chyba při načítání vstupů. Na chybový výstup pak je vypsán název chybného souboru.
- **-3:** Chyba při rozdělování elementů do skupin. Blíže nespecifikovatelná.
- **-4:** Chyba při odvozování regulárních výrazů. Blíže nespecifikovatelná.
- **-5:** Chyba při generování schématu. Blíže nespecifikovatelná.

6.4 Omezení implementace

Přiložená implementace je pouze ukázkovou implementací a pro její praktické nasazení by bylo potřeba dořešit některé technické detaily. Asi nejdůležitějším omezením je skutečnost, že program načítá všechny vstupy (celé dokumenty) do paměti, a všechny operace provádí v paměti. Tudíž při rozsáhlejších vstupech může program předčasně ukončit svoji činnost kvůli nedostatku paměti. Zároveň jsou některé podružné algoritmy, které nejsou stěžejní pro tuto práci, implementovány s ohledem na přehlednost a ne s ohledem na efektivitu (jak časovou tak paměťovou).

7 Experimenty

V této kapitole jsou představeny a ukázány praktické výsledky práce navrženého algoritmu. Z dostupných reálných dokumentů byly vybrány ty množiny, které již obsahovaly popis struktury dokumentů. Tyto množiny pak byly rozděleny podle charakteru struktury do několika typů. Každá množina pak byla předložena implementovánemu algoritmu jako vstup a výsledné XML Schema bylo porovnáváno s původním popisem struktury. Odchylky algoritmu pak byly analyzovány v rámci typu i mezi typy. Protože reálně používané dokumenty nepokrývaly možnosti algoritmu, byly dále vytvořeny umělé množiny dokumentů, které demonstrují zbývající vlastnosti algoritmu.

V prvních odstavcích každé z podkapitol jsou stručně charakterizovány použité typy dokumentů. Potom následuje popis odchylek výsledků algoritmu od očekávaných výstupů (případně originálních popisů struktur). První podkapitola se zabývá reálnými dokumenty a druhá podkapitola pak dokumenty uměle vytvořenými. Všechny použité dokumenty, originální popisy struktury a vygenerovaná schémata jsou přiložena na disku CD-ROM v adresáři `/data`.

7.1 Reálné dokumenty

Reálně používané dokumenty, respektive jejich struktura je ve většině případů velmi jednoduchá. Dokonce i pokud je popisovaná struktura (DTD nebo XML Schema) složitější, uživatelé typicky možnosti struktury nevyužívají a dokumenty odpovídají mnohem jednodušší podmnožině definované struktury. Mezi reálnými dokumenty byly nalezeny tyto typy struktury (s uvedením zdrojů):

- **Typ 1:** Dokumenty podléhají velmi jednoduché a obecné struktuře typu $(A|...|B)^*$. Nevyužívají volitelné elementy, hlubší strukturu exkluzivních výběrů ani sekvence elementů.
  - *Arthurs Classic Novels (http://arthursclassicnovels.com/)*
- **Typ 2:** Tyto dokumenty naopak vůbec nepoužívají operátor exkluzivního výběru, ale pouze volitelné elementy, sekvence a opakování.
  - *Niagara (http://www.cs.wisc.edu/niagara/data.html)*
  - *RNAdb (http://research.imb.uq.edu.au/rnadb/xmldownloads/default.aspx)*
- **Typ 3:** Tyto dokumenty využívají všech možností jazyka DTD. Tedy využívají opakování elementů, exkluzivní výběry i volitelné podvýrazy.
  - *Bible v XML (http://www.softcorporation.com/products/xmllight/largeDoc.zip)*
  - *Části Shakespearových her (http://www.ibiblio.org/bosak/)*

  - *Niagara (http://www.cs.wisc.edu/niagara/data.html)*
  - *Oval XML files (http://oval.mitre.org/oval/download/datafiles.html)*
- **Typ 4:** Tyto dokumenty využívají velmi pravidelnou strukturu odpovídající spíš tabulce. Typicky je v kořenovém elementu libovolněkrát opakován element se strukturou n-tice údajů v přesně daném pořadí. Viz. `31: DTD dokumentů s pravidelnou strukturou`
  - *Niagara (http://www.cs.wisc.edu/niagara/data.html)*
- **Typ 5:** Tyto dokumenty využívají operátor <`xs:all` ... jazyka XML Schema. Takových dokumentů je však velmi málo a typicky tato vlastnost struktury není využita.
  - *pdb-mmcif (http://www.rcsb.org/pdb/uniformity/)*
- **XSD:** Jak již bylo řečeno v úvodu práce, dokument popisující strukturu v jazyce XML Schema je sám XML dokumentem. A existuje XML Schema dokument popisující svou vlastní strukturu. Proto byly jako pokus zařazeny do experimentů i dokumenty popisující schéma jiných dokumentů.
  - *Michigan (http://www.eecs.umich.edu/db/mbench/)*
  - *NASArrays (http://arabidopsis.info/bioinformatics/narraysxml/)*
  - *XBench (http://db.uwaterloo.ca/ ddbms/projects/xbench/index.html)*

```
<!ELEMENT W4FDOC (Movie)*>
<!ELEMENT Movie (Title,Year,DirectedBy,Genres,Cast)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT DirectedBy (Director)*>
<!ELEMENT Director (#PCDATA)>
<!ELEMENT Genres (Genre)*>
<!ELEMENT Genre (#PCDATA)>
<!ELEMENT Cast (Actor)*>
<!ELEMENT Actor (FirstName,LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

31: DTD dokumentů s pravidelnou strukturou

Výsledky

Odchylky vygenerované struktury od očekávané jsou u všech typů v zásadě podobné, proto je budeme popisovat pro všechny typy dohromady. Nejmarkantnější odchylkou je téměř vždy vygenerování podmnožiny očekávaného schématu. Tato odchylka souvisí s typickým nevyužitím detailů struktury v dokumentech. Samozřejmě, pokud některý detail struktury není v žádném vstupním dokumentu obsažen, nemůže být automatickým generátorem vygenerován. Dokonce je možné říci, že toto je vhodná vlastnost algoritmu, protože vygenerování schématu pro podmnožinu dokumentů může být právě záměr uživatele.

Horší odchylkou od očekávaných výsledků je přílišné zobecnění schématu. Konkrétně se jedná o nastavení opakování místo od jedné do nekonečna na od nuly do nekonečna. Algoritmus tedy dává možnost vynechat element tam, kde to nebylo záměrem. Tato odchylka se vyskytuje pouze vyjímečně, ale na typově podobných místech. Ve většině případů se jedná o opakování prosté sekvence elementů.

Poslední zjištěnou odchylkou je neefektivní zápis regulárních výrazů. Toto není závažná odchylka, protože z hlediska jazyka popisovaného výrazem není ve výrazech rozdíl. Jedná se zejména o výrazy typu `(prefix tělo suffix| tělo)`, které by bylo možné zjednodušit na `(prefix? tělo suffix?)`. A o výrazy typu (*výraz výraz**), které by bylo možné zjednodušit na (*výraz*$^+$). Tento zápis je dán implementací algoritmu, který převádí konečný automat na regulární výraz a neoptimalizuje zápis tohoto výrazu.

XSD

Dokumenty XSD obsahující XML Schema popisy struktury jiných dokumentů jsou rozděleny do dvou množin. První množina jsou náhodně nalezené dokumenty z různých zdrojů. Druhá množina jsou popisy vygenerované navrženým algoritmem.

Výsledek práce algoritmu na první množině je výrazně ovlivněn různorodostí použitých dokumentů. XML Schema je velmi komplexní a složitý jazyk na popis struktury. Má mnoho nepovinných výrazových prostředků sloužících k dodatečnému popisu struktury a mnohdy je možné tutéž věc zapsat několika způsoby. Ač se jedná stále o dokumenty v jazyce XML Schema, různí tvůrci využívají různé možnosti a styly zápisu. V zásadě je však z výsledku patrné že se jedná o XML Schema.

Druhá množina (část výsledku je zkázána v `32: Jeden typ z generování schématu pro XML Schema`), protože je z jediného zdroje, podléhá mnohem méně různorodosti dokumentů. Zde je patrná tendence algoritmu zachovávat pro podobné dokumenty větší podrobnosti struktury dokumentů (respektive vytvářet méně obecné schéma). Přizpůsobením parametrů implementace by bylo možné docílit výsledků bližších očekávání, tedy obecnějšího schématu.

7.2 Umělé dokumenty

Umělé dokumenty ukazují vlastnosti algoritmu, které nebylo možné pozorovat při zpracovávání reálných dokumentů, protože nebyly nalezeny dokumenty s vhodnou charakteristikou. Zejména se jedná o tyto vlastnosti:

```
<xs:complexType name="Txs:choice1" >
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="xs:element" type="Txs:element1"
                minOccurs="1" maxOccurs="1" />
      <xs:element name="xs:element" type="Txs:element1"
                minOccurs="0" maxOccurs="unbound" />
    </xs:sequence>
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="xs:sequence" type="Txs:sequence1"
                minOccurs="1" maxOccurs="1" />
      <xs:element name="xs:sequence" type="Txs:sequence1"
                minOccurs="0" maxOccurs="unbound" />
    </xs:sequence>
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="xs:element" type="Txs:element1"
                minOccurs="1" maxOccurs="1" />
      <xs:element name="xs:element" type="Txs:element1"
                minOccurs="0" maxOccurs="unbound" />
      <xs:element name="xs:sequence" type="Txs:sequence1"
                minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

32: Jeden typ z generování schématu pro XML Schema

- **Zavedení permutačního operátoru:** Byly vytvořeny dokumenty, kde se v různém pořadí vyskytují tytéž elementy.
- **Různá struktura elementů stejného jména:** Byly vytvořeny dokumenty, kde se element stejného jména vykytuje v různých kontextech.

Permutační operátor

Pro účely testování této vlastnosti navrženého algoritmu bylo vytvořeno několik množin dokumentů lišících se ve dvou parametrech. První parametr je velikost permutující množiny, tedy kolik elementů se v různém pořadí vyskytuje. Druhý parametr pak je kolik procent permutací je v dokumentech obsaženo. Pokud uvažujeme o permutacích N elementů, pak máme N! různých řetězců, které tomuto typu odpovídají. Druhý parametr tedy určuje, kolik permutací z počtu N! se reálně v dokumentech vyskytuje.

Velikost permutující množiny nemá příliš velký vliv na zavádění permutačního operátoru. Má vliv pouze na dobu výpočtu. Naproti tomu procento použitých permutací má podle očekávání zásadní vliv na zavádění tohoto operátoru. V tabulce `33: Vliv počtu použitých permutací na zavedení permutačního operá-`

`toru` je tento vliv přehledně znázorněn. V řádcích je velikost permutující množiny a ve sloupcích jsou procenta použitých permutací. Hodnota "Ne" značí že nebyl použit permutační operátor. Hodnota "Částečně" značí že byl použit, ale ne na celou strukturu jak bylo očekáváno. A nakonec hodnota "Ano" značí že byl použit permutační operátor tak jak bylo očekáváno. Tabulka ukazuje, že permutační operátor se začíná vyskytovat okolo hodnoty třiceti procent použitých permutací. Od hodnoty osmdesát procent je už vždy plně použit.

	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
3	ne	částečně	ne	částečně	ne	ne	částečně	ano	ano	ano
4	ne	ano	částečně	ano	částečně	částečně	částečně	ano	ano	ano
5	ne	ne	částečně	částečně	částečně	částečně	částečně	ano	ano	ano

33: Vliv počtu použitých permutací na zavedení permutačního operátoru

Různá struktura elementů stejného jména

Pro účely testování této vlastnosti navrženého algoritmu bylo vytvořeno několik množin dokumentů. Parametrem množiny je kolik procent elementů v klíčových řetězcích je stejných. V zásadě je možné říci, že se algoritmus choval na testovacích případech zcela podle očekávání. V závislosti na konkrétní struktuře vztahu elementů se jednotlivé skupiny spojují pokud se překrývají přibližně z padesáti procent.

8 Závěr

Cílem práce bylo prozkoumat možnosti a omezení automatického generování schémat pro sadu XML dokumentů a na základě této analýzy navrhnout a implementovat vlastní řešení.

Nejprve byla provedena analýza existujících řešení tohoto problému. Výsledkem této analýzy bylo formulování kritérií pro hodnocení algoritmů z hlediska uživatele ([3]), kategorizace existujících algoritmů na základě přístupu k řešení problému a detailní analýza vybraných metod.

Na základě této analýzy, a zejména na základě zjištěných vlastností, výhod a nevýhod existujících řešení, byly navrženy vlastní algoritmy. Heuristická metoda ACO byla zvolena pro svůj potenciál v rozšiřitelnosti vlastní metody o nové vlastnosti a schopnost spojit tyto různorodé vlastnosti ve spolupracující celek. Jako jazyk pro popis výsledného schématu byl zvolen jazyk XML Schema, kvůli jeho vyjadřovacím možnostem a zároveň kvůli tomu, že je standardem konzorcia w3.

Přínos této práce spočívá ve volbě cílového jazyka pro zápis výsledného schématu a využití jemu odpovídajících možností a vylepšení existujících algoritmů. Nejdůležitějším rozšířením je zavádění permutačních operátorů v popisu struktury elementu (tedy zavedení typu `<xs:all` ... jazyka XML Schema). Zavedením tohoto operátoru může dojít k výraznému zlepšení a zjednodušení výsledných schémat. Dalším přínosem je možnost specifikovat různé typy (různou strukturu) elementům stejného jména vyskytujících se v jiném kontextu struktury dokumentu. Posledním původním rozšířením je možnost určit několik různých kořenových elementů vygenerovaného schématu.

Ačkoliv byly všechny stanovené cíle v zásadě splněny, je v navrženém řešení prostor pro další rozšiřování. Jednou z oblastí by mohlo být využití dalších vlastností jazyka XML Schema pro další vylepšení schématu. Zejména využití substitucí a skupin elementů a atributů. Další oblastí, kde by bylo možné navržené řešení rozšířit je volba jiného jazyka pro popis schématu a využití jeho předností. Poslední oblastí, kde by mohlo být navržené řešení rozšířeno je interakce s uživatelem. V poslední oblasti je asi největší potenciál, protože plně automatické algoritmy pouze reverzním inženýrstvím přibližně hádají záměr uživatele, kdežto pokud by algoritmus umožnil vhodnou formou interakci s uživatelem, výsledek by přesně odpovídal požadavkům.

Reference

[1] Extensible Markup Language (XML). W3C Recommendation, 2004. **http://www.w3.org/TR/REC-xml/**.

[2] XML Schema. W3C Recommenation, 2004. **http://www.w3.org/TR/xmlschema-1/**.

[3] Helena Ahonen: Generating Grammars for Structured Documents Using Grammatical Inference Methods. Report A-1996-4, Department of Computer Science, University of Finland, 1996.

[4] Henning Fernau: Learning XML Grammars. 2001. *In P. Perner, editor, Machine Learning and Data Mining in Pattern Recognition MLDM'01*, **2123**: 73-87

[5] Raymond K. Wong, Jason Sankey: On Structural Inference for XML Data. 2003. **citeseer.ist.psu.edu/wong03structural.html**.

[6] E. Mark Gold: Language identification in the limit. 1967. *Information and Control*, **10(5)**: 447-474

[7] Chuang-Hue Moh, Ee-Peng Lim, Wee-Keong Ng: Re-engineering Structures from Web Documents. 2000. *In ACM Digital Libraries 2000*: 67-76

[8] Keith E. Shafer: Creating DTDs via the GB-Engine and Fred. 1996. *SGML: The World Tour*

[9] A. W. Biermann, J. A. Feklman: On the synthesis of finite-state machines from samples of their behaviour. 1972. *IEEE Transactions on Computers*, **21**: 591-597

[10] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, Kyuseok Shim: XTRACT: A System for Extracting Document Type Descriptors from XML Documents. Bell Labs Tech. Memorandum, 1999.

[11] M. Charikar and S. Guha: Improved combinatorial algorithms for the facility location and k-median problems. 1999. *Proc. of the Ann. Symp. on Fundations of Computer Science (FOCS)*

[12] Jean Berstel, Luc Boasson: XML Grammars. 2000. *Mathematical Foundations of Computer Science (MFCS'2000)*, **1893**: 182-191

[13] Anil K. Jain, Richard C. Dubes : Algorithms for Clustering Data. Prentice Hall, 1988.

[14] Rafael C. Carrasco, Jose Oncina: Learning Stochastic Regular Grammars by Means of a State Merging Method. 1994. *The 2nd Intl. Collo. on Grammatical Inference and Applications*: 139-152

[15] Laurent Mignet, Denilson Barbosa, Pierangelo Veltri: The XML Web: a First Study. 2003. *In Proc. of International World Wide Web Conf*: 500-510

A Obsah CD-ROM

Součástí diplomové práce je přiložený CD-ROM obsahující text práce a především zdrojové soubory včetně dokumentace a přeložené soubory ukázkové implementace SchemaMiner. CD-ROM obsahuje následující soubory a adresáře:

- obsah.txt - soubor s popisem obsahu CD-ROM
- /text - adresář s textem diplomové práce
- /src - adresář se zdrojovými soubory ukázkové implementace
- /doc - adresář s dokumentací k ukázkové implementaci (vygenerováno programem javadoc)
- /release - adresář s pželoženou aplikací SchemaMiner
- /data - adresář s daty použitými k ověření ukázkové implementace v sekci .