Dokumentace k pilotní verzi ročníkového projektu

Vedoucí projektu: **Barbora Vidová-Hladká**
Autor: **Vladimír Rovenský**
Téma: **Rozpoznání smysluplnosti české věty**
Platformy: **Unix, Windows**
Programovací jazyk: **C++ (IDE MS Visual Studio)**

Uživatelská dokumentace

Hardwarové požadavky: žádné speciální
Softwarové požadavky: OS Unix/Linux/Windows, tool_chain

Stručný popis programu
Jedná se o nástroj ke zjišťování smysluplnosti české věty na základě tvaroslovných (morfologických) informací. Vstupem programu je řetězec českých slov zpracovaný automatickou morfologickou analýzou (prostřednictvím nástroje *tool_chain*). Výstupem programu je informace o tom, je-li či není vstupní řetězec smysluplnou českou větou.
Větou může být i otázka.

Instalace programu pod OS Windows
Stačí do libovolného adresáře rozbalit verzi aplikace pro OS Windows, tj. adresář *release\windows* z instalačního balíčku.

Instalace programu pod OS Unix
Nejprve do libovolného adresáře rozbalte soubory v adresáři *source/sense* z instalačního balíčku. Dále je třeba standardním způsobem tyto zdrojové kódy zkompilovat (například spuštěním příkazu *„g++ `ls ./*.cpp` -w –o sense"* v adresáři, kam byly rozbaleny) .

Spuštění programu
Samotné spuštění je stejné pod Windows i Unixem. V adresáři, kam byl program nainstalován, najdete dva soubory: *conditions.txt a sense.exe.* V prvním jsou uložené vztahy, podle kterých program větu analyzuje (jejich formát bude popsán dále). Při dodržení tohoto formátu je možné tento soubor libovolně rozšiřovat a upravovat. Ke spouštění slouží soubor *sense.exe*, který přijímá tyto volby:

-i soubor	specifikace souboru se vstupem, jímž je věta zpracovaná programem *tool_chain* ve formátu CSTS. Program očekává na vstupu větu prošlou tokenizací, morfologickou analýzou a tagováním nástroje *tool_chain.*
-h	Zobrazí krátkou nápovědu..
-v	Zapnutí podrobného výstupu, program vypíše nalezené vztahy (viz dále)

Použití nástroje *tool_chain*
Jelikož program k použití vyžaduje na vstupu text analyzovaný nástrojem *tool_chain*, zde je krátký popis jeho použití. Podrobnější informace o nástroji *tool_chain* lze nalézt na adrese **http://ufal.mff.cuni.cz/rest/CAC/doc-cac20/cac-guide/cz/html/ch3.html#nastroje-zprac.**

Jedná se o nástroj zajišťující tvaroslovnou analýzu českých textů, pro potřeby dokumentovaného programu jsou důležité pouze služby tokenizace, morfologická analýza a tagování, ze kterých analýza smysluplnosti vychází. Pro analýzu vstupní věty nástrojem *tool_chain* je třeba spustit jej následovně:

tool_chain –tAT –i soubor_s_větou –o soubor_s_výstupem

Volba –tAT zajistí, že se provedou všechny tři výše uvedené služby, *soubor_s_větou* je textový soubor (prostý text) obsahující vstupní větu, *soubor_s_výstupem* je jméno souboru, kam má být uložen výsledek zpracování.

Základní popis algoritmu

Na vstupu máme nástrojem *tool_chain* analyzovanou větu ve formátu CSTS (popsán na **http://ufal.mff.cuni.cz/pdt2.0/doc/data-formats/csts/html/DTD-HOME.html**). Takto může vypadat příklad vstupu programu ve formátu CSTS po analýze nástrojem *tool_chain* (tokenizace, morfologická analýza, tagování, tedy volba -tAT).
Vstupní větou byl řetězec „Před domem stojí auto a v domě štěká pes.“

```
<csts lang=cs>
<doc file="in" id=1>
<a>
<mod>?
<txtype>?
<genre>?
<med>?
<temp>?
<authname>?
<opus>in
<id>001
</a>
<c>
<p n=1>
<s id="in:001-p1s1">
<f cap>před<MDl src="m">před<MDt src="m">R--------------
<f>domem<MDl src="m">dům<MDt src="m">NNIS7-----A----
<f>stojí<MDl src="m">stát-4<MDt src="m">VB-S---3P-AA---
<f>auto<MDl src="m">auto<MDt src="m">NNNS4-----A----
<f>a<MDl src="m">a-1<MDt src="m">J^-------------
<f>v<MDl src="m">v-1<MDt src="m">RR--6----------
<f>domě<MDl src="m">dům<MDt src="m">NNIS6-----A----
<f>štěká<MDl src="m">štěkat<MDt src="m">VB-S---3P-AA---
<f>pes<MDl src="m">pes<MDt src="m">NNMS1-----A----
<D>
<d>.<MDl src="m">.<MDt src="m">Z:-------------
</c>
</doc>
</csts>
```

Takovýto vstup je rozdělen na jednotlivé slovní jednotky (neboli tokeny – nejčastěji jedno slovo nebo interpunkční znaménko) a ty jsou uloženy do příslušných struktur spolu se svou morfologickou informací, která je reprezentována jako patnáctipoziční značka (tag) pro každou slovní jednotku. Každá pozice této značky jednoznačně určuje jednu informaci, například slovní druh, rod, osobu, číslo nebo pád slovní jednotky. Jejich přesný popis je k nahlédnutí na
http://ufal.mff.cuni.cz/~hladka/rp200809/cz-appendix-D.pdf. V CSTS výstupu nástroje *tool_chain* jsou tyto informace uloženy ve značkách <f>, <d>, <MDl> a <MDt>.

Pokud je na vstupu souvětí, je nejprve rozděleno na věty jednoduché. Celé souvětí je smysluplné, právě když jsou smysluplné všechny jeho věty jednoduché. Větou jednoduchou je myšlena věta, která obsahuje nejvýše jedno sloveso. Souvětí je věta obsahující alespoň dvě slovesa, skládající se z vět jednoduchých.

Algoritmus dělení souvětí na věty jednoduché

Program nejprve najde pozici prvního slovesa ve větě, tuto si zapamatuje. Následně postupuje od posledního slova směrem k prvnímu a hledá sloveso (ne infinitiv). Když na nějaké narazí, pokračuje v načítání slov dokud nenarazí na slovní jednotku, která může oddělovat věty (např. spojka, čárka).

V tom okamžiku byla nalezena věta jednoduchá. Program načte všechny další případné oddělovače (může jich být za sebou víc, např. „, ale i“) , uloží větu jednoduchou a opakuje. V okamžiku, kdy narazí na první sloveso věty, automaticky přidá coby větu jednoduchou zbytek vstupu a končí.

Příklad:
„Umění sahá hodně daleko zpátky , a čím dál se díváme, tím kontroverznější jsou důkazy.“
Prvním slovesem je „sahá“. Algoritmus postupuje od konce věty. Najde sloveso sahá, dojde k čárce a přidá větu „, tím kontroverznější jsou důkazy.“ Dále najde sloveso díváme, po přečtení „,a čím“ přidá další větu („a čím dál se díváme“). Dále narazí na sloveso „sahá“, to je prvním slovesem věty a proto celý zbytek vstupu přidá coby poslední větu jednoduchou („Umění sahá hodně daleko zpátky“).

Algoritmus zjišťování smysluplnosti jednoduché věty

1) Podmínky pro smysluplné dvojice slov

Analýza jednoduché věty probíhá tak, že se pro každou dvojici slovních jednotek pokusíme zjistit, zda je smysluplná. K tomu jsou pro každou dvojici slovních druhů definovány podmínky, za kterých je tato smysluplná. Těchto podmínek může být pro jednu dvojici slovních druhů několik, jedna, nebo vůbec žádná, pokud neexistuje smysluplné spojení těchto slovních druhů.

Program prochází všechny dvojice slovních jednotek ve větě a na základě jejich slovních druhů vybere příslušné podmínky, jejichž platnost následně ověřuje. Pokud testovaná dvojice slovních jednotek splňuje alespoň jednu z nich, potom tvoří smysluplnou dvojici slovních jednotek.

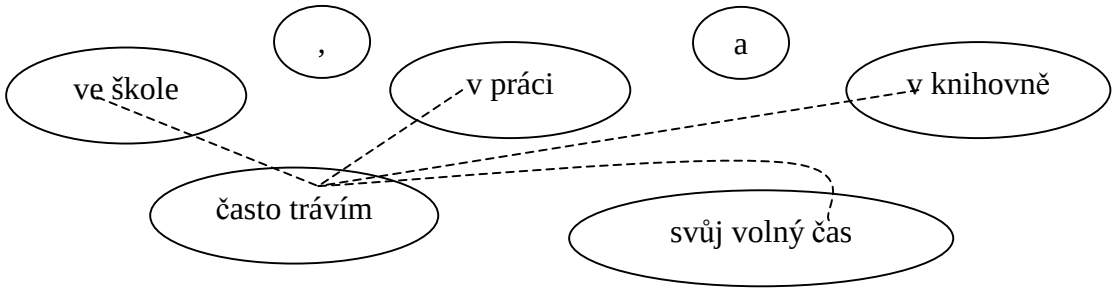
Příklad:
Dvojice podstatné a přídavné jméno je smysluplná, pokud mají stejný rod, pád i číslo.
Dvojice podstatné jméno a předložka je smysluplná, pokud předložka ve větě předchází podstatnému jménu, a to je ve správném pádě vzhledem k základnímu tvaru předložky.
Pro ilustraci bude dále sloužit (jednoduchá) věta: „Ve škole, v práci a v knihovně často trávím svůj volný čas.“

Program při zkoumání této věty nalezne následující smysluplné dvojice slovních jednotek dle podmínek typu (1):

ve škole + v práci + v knihovně	- podmínky pro spojení předložky a podstatného jména lze definovat například tak, že předložka musí být ve větě před podstatným jménem, a to musí mít správný pád vzhledem k základnímu tvaru předložky.
často trávím	- dvojice příslovce a sloveso je většinou smysluplná bez dalších omezení, resp. omezení by ubrala na obecnosti.
svůj čas + volný čas	- dvojice podstatné jméno + slovní druh rozvíjející podstatné jméno (například přídavné jméno, zájmeno) lze rozpoznat podle shody v rodu, pádu, čísle atp.
trávím v + trávím čas	- pro pospojování logických částí věty (příslovečné určení místa, podmětná část, přísudková část atp.) jsou definovány (velmi obecné) podmínky spojující například sloveso s předložkou, nebo sloveso s podstatným jménem.

Jakmile máme takto definovány potřebné podmínky, můžeme větu přepracovat na graf G= (V,E), kde V = slovní jednotky věty a $\forall t_1,t_2$ slovní jednotky: $t_1,t_2 \in E \Leftrightarrow \exists$ podmínka pro smysluplnou dvojici, které t_1,t_2 vyhovují. Smysluplnost věty je potom definována jako souvislost takto vzniklého „grafu smysluplnosti."

Takto vypadají komponenty konstruovaného grafu po aplikaci podmínek (1):



Čárkovaně jsou vyznačeny hrany odpovídající spojením, která jsou natolik obecná, že budou omezena dalšími typy podmínek (výše zmíněná spojení předložky a slovesa nebo slovesa a podstatného jména).

Základem jsou tedy podmínky a jejich definice. K tomuto účelu slouží soubor *conditions.txt* v adresáři s programem. V něm jsou veškeré tyto podmínky definovány a do něj je možné další doplňovat, jeho formát bude popsán dále.

Takto definované podmínky ale neposkytují dostatečně silný prostředek k rozpoznání nesmysluplnosti věty, neboť například spojení slovesa a podstatného jména je natolik obecné, že je podmínka prakticky prázdná. Potom stačí, aby v každé komponentě grafu bylo jedno podstatné jméno a v celé větě jediné sloveso a graf bude souvislý, bez ohledu na smysluplnost věty. Proto lze definovat další typy podmínek, které smysluplná věta musí splňovat.

2) Podmínky pro spojování částí věty

Výše definované podmínky dokáží vcelku rozumně rozložit (implicitně) větu na několik úseků, například rozvité podstatné jméno, určení místa atp. Problémem je rozumné spojení těchto úseků, viz výše zmíněný problém se slovesem a podstatným jménem a rozbor ilustrační věty.

K tomuto účelu je slouží druhý druh podmínek – definice ternárního vztahu slovní druh – spojka (čárka) – slovní druh, který říká, za jakých podmínek může spojka spojovat dvojici slovních jednotek s danými slovními druhy. Je možné přesně definovat morfologické vlastnosti slovních jednotek na každé straně spojovacího výrazu a tím zpřísnit spojování úseků věty.

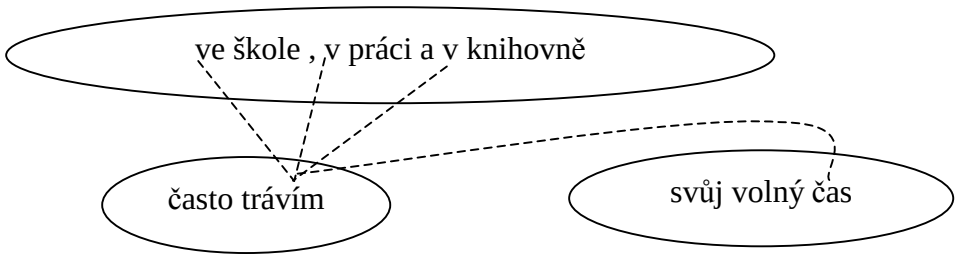
Příklad:
Vraťme se k rozpracované větě „Ve škole, v práci a v knihovně často trávím svůj volný čas.“ Při hledání trojic splňujících některou podmínku typu (2) budou nalezena spojení:
„škole , práci“ + „práci a knihovně“ – v obou případech se jedná o spojení podstatných jmen ve stejném pádě, jednou čárkou a jednou spojkou.

Obecně jsou pro každou dvojici shodných slovních druhů definovány podmínky (2), například dvě přídavná jména mohou být spojena spojkou (čárkou), pokud mají stejný pád a slovní poddruh, slovesa mohou být spojena spojkou, pokud jsou obě v infinitivním tvaru atp.

V této větě byly tedy podmínkami (2) spojeny tři určení místa do jednoho souvislého úseku a rovněž byly do věty zapojeny slovní jednotky „,“ a „a“ – i ty jsou brány v potaz při určování smysluplnosti.

Pro tento příklad by se zřejmě zdálo logičtější, kdyby byly podmínkami (2) spojeny předložky místo podstatných jmen, nicméně algoritmus se zastaví u první nalezené dvojice, jež může zkoumaná spojka / čárka spojovat – pro potřeby analýzy smysluplnosti stačí, že existuje nějaká taková dvojice, kdyby hledání pokračovalo dál, byla by nalezena i možnost spojení předložek.

Takto budou vypadat komponenty konstruovaného grafu po aplikaci podmínek (2):



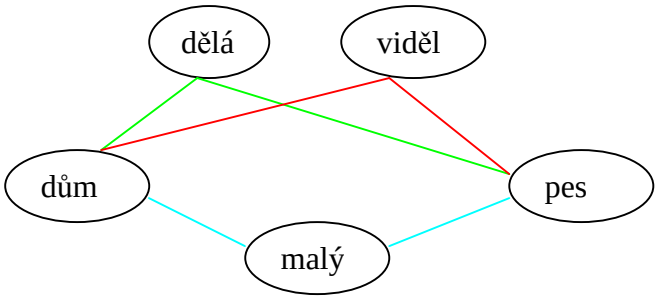
3) Podmínky pro maximální počet vztahů

Dalším prostředkem pro zpřísnění analýzy jsou podmínky pro omezení počtu vztahů. Pokud není počet vztahů nijak omezen, nastává problém s rozpoznáváním nesmyslných vět, například podmínka spojující částici / citoslovce se slovesem opět nemá prakticky žádné morfologické informace, které by se daly kontrolovat, tedy jakékoli citoslovce se naváže na všechna slovesa věty bez omezení. Tato podmínka poskytuje prostředek k vyjádření podmínek typu „jedna částice se může navázat nejvýše na jedno sloveso“ atp. V tomto případě se jedná o nesymetrickou relaci.

Vladimír Rovenský
V.Rovensky@lit.cz

Příklad:
Do třetice všeho dobrého se podívejme na větu „Ve škole, v práci a v knihovně často trávím svůj volný čas.“ Tato věta je smysluplná, proto zde podmínky typu (3) nehrají zásadní roli. Zde by se projevila pouze jediná podmínka (3) říkající, že podstatné jméno se naváže nejvýše na jednu předložku. Tím pádem by například nedošlo ke vzniku dvojic „ve práci“ nebo „v škole“. Na výsledku to však nic nezmění, věta by byla vyhodnocena jako smysluplná i bez podmínek typu (3).

Nyní uvažme zjevně nesmyslnou větu „Dům dělá malý viděl pes.“ Tato „věta“ neobsahuje spojku ani čárku, takže dělením na věty jednoduché projde beze změny a bude analyzována coby věta jednoduchá. Takto by vypadal zkonstruovaný graf bez podmínek (3):



Všechny vyznačené hrany (všech barev) odpovídají podmínkám (1). Graf je zjevně souvislý, navzdory naprosté nesmyslnosti věty. Zde poskytuje zavedení podmínek (3) větší přesnost. Máme například definovány podmínky (3) určující, že přídavné jméno se smí navázat nejvýše na jedno podstatné jméno a podstatné jméno nejvýše na jedno sloveso. Obarvené hrany nyní odpovídají těmto omezením: z každé množiny hran stejné barvy můžeme použít pouze (v tomto případě) jednu.

Nyní už graf díky menšímu počtu hran souvislý nebude (bez ohledu na to, jaké hrany vybereme - nejvýše lze vybrat tři, ale kostra grafu na pěti vrcholech má čtyři hrany).

Popis formátu souboru podmínek

Soubor s podmínkami má speciální formát, který je třeba dodržet při jeho případném rozšiřování. Ve finální verzi bude nejspíše s programem dodávána i k tomu určená utilita. V souboru je možné psát jednořádkové komentáře uvozené //. Nezáleží na pořadí definovaných podmínek.

Definice podmínek pro smysluplné dvojice (1):

rel	*slovní druh 1*	*slovní druh 2*
[!][1]	*[hodnota]*	*[hodnota]*
[!] [2]	*[hodnota]*	*[hodnota]*
[!] [3]	*[hodnota]*	*[hodnota]*
[!] [4]	*[hodnota]*	*[hodnota]*
…		
[!] [15]	*[hodnota]*	*[hodnota]*
[!][lemma]	*[*hodnota*]*	*[hodnota]*
[!][ord]	*1 / 2*	*1 / 2*
end		

Deklarace podmínky je uvozena klauzulí *rel*, za ní následují *identifikátory Slovní druh 1 a Slovní druh 2* určující dvojici slovních druhů, pro které je podmínka smysluplnosti definována.

Identifikátory slovních druhů:

Podstatné jméno	**N**	Příslovce	**D**
Přídavné jméno	**A**	Předložka	**R**
Zájmeno	**P**	Spojka	**J**
Číslovka	**C**	Částice	**T**
Sloveso	**V**	Citoslovce	**I**

Ačkoli sémanticky na pořadí nezáleží, v souboru musí být uveden jako první slovní druh s nižším číslem (podstatné jméno = 1, citoslovce = 10). Oddělovačem je mezera nebo tabulátor.

Následují podmínky pro hodnoty tagů obou slovních jednotek, na jednom řádku jedna podmínka, uvozena je číslem pozice dané morfologické kategorie (1-15), poté pro každou slovní jednotku požadovaná *hodnota*.

Hodnota může být buď jedna konkrétní hodnota dané kategorie, více možných hodnot ve tvaru (hod1 hod2 hod3) – tedy v závorkách oddělené mezerami, nebo znak + který zastupuje libovolnou hodnotu, nebo znak -, který znamená rovnost hodnotě stejné morfologické kategorie druhé slovní jednotky (pokud obě slovní jednotky mají coby hodnotu jedné kategorie -, mohou být hodnoty libovolné, ale musejí se rovnat). *Hodnota* také nemusí být vyplněna vůbec, v tom případě je význam stejný jako +.

Dále se lze odkázat na základní tvary (lemmata) obou slovních jednotek, v tomto případě je *hodnota* příslušný základní tvar (seznam základních tvarů / + / - jako v prvním případě)

Na slovosled se lze odkázat pomocí klauzule *ord* a pořadového čísla pro každou slovní jednotku (1 – slovní jednotka je první z dvojice, 2 – slovní jednotka je druhá z dvojice).

Před řádkem je možné uvést nepovinně symbol ! označující negaci. Pokud je řádek negovaný, znamená to, že pokud je podmínka určená tímto řádkem splněna, celá podmínka splněna není. Tedy například „!lemma matka +" říká, že základní tvar první slovní jednotky nesmí být „matka".

Prakticky všechny tyto podmínky jsou nepovinné, lze vytvořit i úplně prázdnou podmínku, která bude splněna pro každou dvojici slovních jednotek s odpovídajícími slovními druhy.

Definice vztahu končí řádkem uvozeným klauzulí *end*.

Definice podmínek pro spojování částí věty (2):

con	*slovní druh 1*	*slovní druh 2*
[!][1]	*[hodnota]*	*[hodnota]*
[!] [2]	*[hodnota]*	*[hodnota]*
[!] [3]	*[hodnota]*	*[hodnota]*
[!] [4]	*[hodnota]*	*[hodnota]*
…		
[!] [15]	*[hodnota]*	*[hodnota]*
[tok]	*lemma*	
end		

Podmínka říká, kdy můžou být slovní jednotky se slovními druhy *slovní druh 1* a *slovní druh 2* spojeny spojkou nebo čárkou, tedy vztah je ternární, ale syntax má téměř stejnou jako předchozí binární. Hodnoty tagů jednotlivých slovních jednotek se definují stejně. Za klauzulí *tok* následuje základní tvar spojky, pokud musí být nějaký konkrétní.

Podmínka je aplikována tak, že pokud je ve větě nalezena spojka nebo čárka, najde se k ní nejbližší pár slovních jednotek (jeden zleva, jeden zprava), který vyhovuje některé z takto definovaných podmínek. Další slovní jednotky již tato konkrétní spojka (čárka) neváže.

Definice podmínek pro omezení počtu relací (3)
max slovní druh 1 slovní druh 2 maximum

Tato podmínka je na jediný řádek. Uvozena klauzulí *max*, u následující dvojice identifikátorů slovních druhů záleží na pořadí (sémanticky i syntakticky), *maximum* je číslo větší nebo rovné 1. Takováto podmínka má význam „Jedna konkrétní slovní jednotka se slovním druhem *slovní druh 1* se může navázat nejvýše na *maximum* různých slovních jednotek se slovním druhem *slovní druh 2.“* Navázat znamená vytvořit hranu v konstruovaném grafu.

Vynucování této podmínky je realizováno tak, že pokud ze všech smysluplných vztahů lze vybrat alespoň jednu podmnožinu splňující podmínku (3), stačí to pro smysluplnost věty. Proto je její ověřování náročnější než u ostatních podmínek, v současné verzi program s jistou optimalizací možnosti generuje.

Časová složitost algoritmu

Označme n = počet slovních jednotek ve větě. Načtení věty proběhne zjevně v $O(n)$, načtení souboru s podmínkami $O(s)$, kde s je délka souboru s podmínkami.

Rozdělení souvětí na věty jednoduché proběhne v čase $O(n)$, následné hledání dvojic podle podmínek (2) rovněž $O(n)$, neboť v obou částech je nejhorším případem průchod celé věty.

Při hledání podmínek (1) jsou zkoumány všechny dvojice slovních jednotek věty a pro každou by měly být v konstantním čase (asociativní pole) vyhledány a ověřeny odpovídající podmínky. Pokud budeme předpokládat, že počet definovaných podmínek pro jednu dvojici slovních jednotek bude konstantní, vychází časová složitost této fáze $O(n^2)$.

Dále je třeba projít větu a zanést do konstruovaného grafu informaci o podmínkách (3), pokud opět budeme uvažovat konstantní počet těchto podmínek, vychází čas $O(n)$.

Poslední fází je kontrola souvislosti grafu, ta sestává jednak z DFS průchodu grafu. Zde je třeba uvažovat složitost $O(V+E)$, kde V je počet vrcholů grafu a E je počet hran. Vzhledem k tomu, že vrcholy grafu tvoří slovní jednotky, vychází časová složitost $O(n^2)$.

Konečně nejnáročnější fází je hledání řešení, které splňuje všechny definované podmínky (3). Označíme-li p počet hran omezených podmínkami (3), potom všech podmnožin této množiny je až 2^p a kontrola jedné z nich trvá až $O(n)$, takže tato fáze trvá až $O(n*2^p)$. Jak n tak p jsou ale relativně velmi malá čísla a součástí algoritmu je rovněž optimalizační procedura. Tato část algoritmu bude ve finální verzi nejspíše upravena pro lepší asymptotickou časovou složitost.

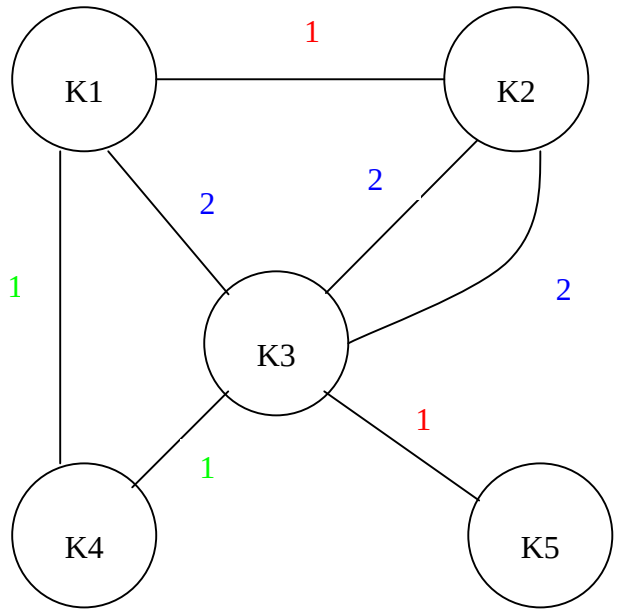
Celková složitost pilotní verze tedy vychází v nejhorším případě $O(s) + O(n^2) + O(n*2^p)$, kde s je délka souboru s podmínkami, n je počet slovních jednotek věty a p je počet dvojic slovních jednotek omezených podmínkami (3).

Vladimír Rovenský
V.Rovensky@lit.cz

Shrnutí algoritmu

Následuje stručné shrnutí celého použitého algoritmu

1) Načíst vstupní větu a soubor s podmínkami.
2) Pokud je na vstupu souvětí, je rozděleno na věty jednoduché. Celá věta je smysluplná právě když jsou smysluplné všechny její věty jednoduché.
3) Pro každou větu jednoduchou zkonstruovat „graf smysluplnosti“, vrcholy = slovní jednotky věty a hrany přidávány na základě definovaných podmínek, tedy:
    a. Najít všechny spojky a čárky ve větě a aplikovat podmínky (2)
    b. Projít všechny dvojice slovních jednotek ve větě a aplikovat pro ně podmínky (1)
4) Zkontroluje se souvislost vzniklého grafu, věta je smysluplná, právě když je graf souvislý. Zároveň je třeba kontrolovat platnost podmínek (3).
    a. Provést DFS[1] na sestavený graf, bez použití jakékoli hrany, která je omezená některou z podmínek (3). Pokud jsou takto dosažitelné všechny vrcholy, věta je smysluplná .
    b. Pokud vznikne několik komponent, mezi nimi budou hrany omezené podmínkami (3). To znamená, že nemusí být možné je použít všechny, proto program hledá způsob, jakým vybrat podmnožinu těchto hran, která všechny komponenty spojí do jedné a zároveň neporuší žádnou z podmínek (3). Pokud takovou najde, věta (jednoduchá, ne celé souvětí) je prohlášena za smysluplnou. Situaci popisuje obrázek.

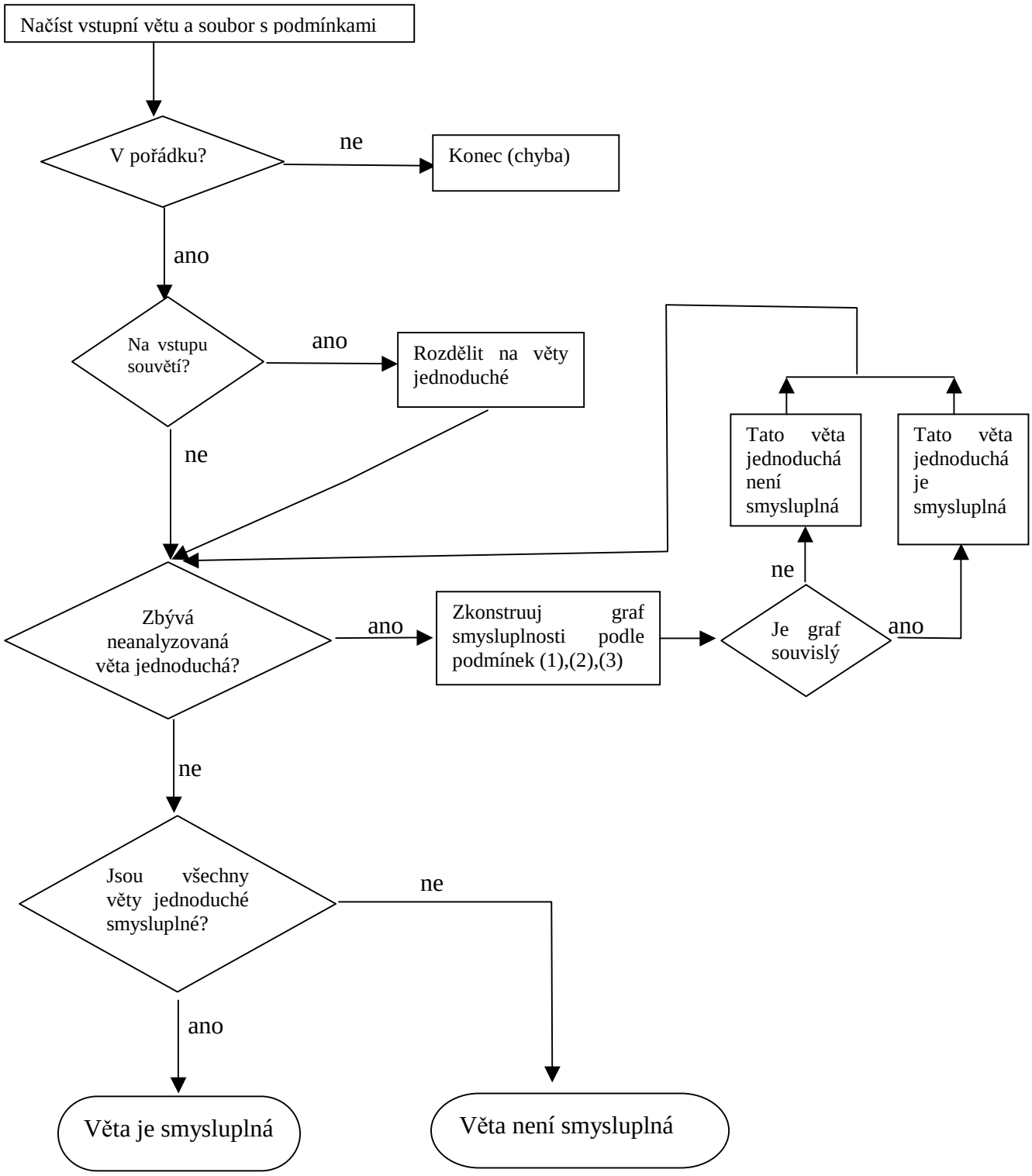


K1 – K4 jsou komponenty vzniklé DFS průchodem po hranách neomezených podmínkami (3), mezi nimi vedou hrany omezené podmínkami (3). Z hran označených jednou barvou lze vybrat pouze k hran, kde k je číslo uvedené u každé hrany této barvy. Situaci na obrázku by např. vyhovovala množina hran K3K5, K1K4,K1K3 a K2K3.

[1] Depth First Search, tedy průchod grafem do hloubky – algoritmus, který dokáže projít postupně všechny vrcholy (souvislého) grafu. Podrobný popis lze nalézt na **http://en.wikipedia.org/wiki/Depth-first_search** nebo v češtině na **http://cs.wikipedia.org/wiki/DFS**

Vladimír Rovenský
V.Rovensky@lit.cz

Vývojový diagram



Vladimír Rovenský
V.Rovensky@lit.cz

Programátorská dokumentace

Popis programu

Následuje popis struktury celého programu, tříd, metod, atd. Tato část dokumentace je z velké části obsažena i v komentářích přímo v programu, zde bude jen základní popis.

Struktura programu

Základ programu tvoří třída *Sense* deklarovaná v souboru *sense.h*, která pokrývá veškerou funkčnost. Stará se o načtení a rozdělení věty, její uložení do připravených struktur a uchovávání globální konfigurace.

Třída *Tester* obstarává kontrolu platnosti podmínek a jejich načítání i uchovávání ve vhodných strukturách. Instance třídy *Tester* je obsažena v třídě *Sense*, která jí posílá dvojice slovních jednotek ke kontrole platnosti podmínek. Je deklarovaná v souboru *tester.h*.

Grafové operace provádí třída *Graph (*soubor *graph.h*), jedná se o graf realizovaný seznamem následníků, poskytuje metody pro kontrolu spojitosti.

Poslední z podstatných tříd je *Token* (soubor *token.h*), která reprezentuje jednu slovní jednotku věty spolu s jejími morfologickými údaji.

V souboru *constants.h* jsou uloženy globální konstanty programu, v *exceptions.h* jsou deklarovány použité typy výjimek a ve *functions.h* jsou deklarace různých globálních funkcí použitých v programu.

Popis důležitých funkcí, typů a rozhraní

Třída Sense

Privátní data

- `std::vector<Sentence*> bareSentences_`
  Seznam jednoduchých vět (naplněn po rozdělení souvětí).
- `Typ Sentence`
  Definován jako std::vector<Token*> - jedna věta.
- `Tester tester_`
  Instance třídy *Tester*, poskytující veškerou kontrolu podmínek.
- `bool verbose_`
  Uchovává informaci o tom, zda má být poskytnut podrobný výstup.

Privátní metody

- `void freeData()`
  Uvolní naalokovanou paměť, zatím pouze volá *freeSentences()*.
- `void freeSentences()`
  Uvolní naalokované věty (tokeny).
- `void readSentence(std::ifstream& from,std::vector<Token*>& to)`
  Načte větu z *from* do *to*.
- `bool endOfSentence(const std::string& line)const`
  Vrací true, pokud *line* je řádek CSTS výstupu nástroje *tool_chain* indikující konec věty.
- `Token* getToken(const std::string& line,int tokenNumber)const`
  Z řádky *line* CSTS výstupu nástroje *tool_chain* vyrobí jeden token.
- `int checkMark(unsigned int from,const std::string& line,const std::string& mark)const`
  V řádce CSTS výstupu *line* hledá od pozice *from* značku *mark*, vrací index prvního nemezerového znaku po této značce, případně –1, pokud značku nenajde.

- **`void splitSentence(std::vector<Token*>& tokens)`**
  Rozdělí větu na věty jednoduché.
- **`bool checkSentence(int i)const`**
  Otestuje, zda je jednoduchá věta číslo *num* smysluplná. Zkonstruuje graf, přidá do něj hrany pomocí *checkConjunctions()* a *objektu* tester_, aplikuje podmínky (3) pomocí *setMaxConditions()* a zkontroluje souvislost grafu.
- **`void checkConjunctions(Sentence* sentence,Graph& g)const`**
  Projde větu a pokud narazí na spojku nebo čárku, pokusí se najít dvě slovní jednotky (podle definovaných podmínek), které by mohla spojovat. Hledá tak, že od spojky postupuje nalevo a pro každou slovní jednotku se pokusí napravo od spojky najít další, která by vyhovovala některé z definovaných podmínek(2). První takto nalezený pár je zanesen do grafu a hledá se další spojka (čárka).
- **`void setMaxConditions(Graph& g,const int num)const`**
  Zanese do grafu věty číslo *num* informaci o podmínkách typu (3) – omezení počtu hran pro DFS. V podstatě označí hrany grafu podobným způsobem jako u ilustračního obrázku výše, akorát místo barev používá k odlišení skupinek hran čísla.
- **`void transformSentence(Sentence* sentence)`**
  Projde větu a udělá různé drobné úpravy zjednodušující analýzu. Momentálně téměř prázdná, ve finální verzi by měla sloužit například k zachycení závorek, uvozovek atp.

Veřejné metody

- **`void readInput(const std::string& filename)`**
  Načte ze souboru *filename* větu a rozparsuje ji.
- **`bool check()const`**
  Hlavní metoda, zkontroluje smysluplnost načtené věty.

Třída Tester

Privátní data

- **`posCouple`**
  Typ definován jako *std::pair<char,char>* - dvojice slovních druhů.
- **`CondList`**
  Typ pro uložení podmínek, definován jako *std::map<posCouple,std::vector<Condition> >*, jedná se o asociativní pole dvojice slovních druhů -> seznam podmínek, za kterých je tato dvojice smysluplná. Stačí, aby platila jediná z podmínek.
- **`ConjCondList`**
  Typ pro ukládání podmínek pro spojky a čárky, definován jako *std::map<posCouple,std::vector<ConjunctionCondition> >*, opět asociativní pole dvojice slovních druhů -> seznam podmínek.
- **`MaxCondList`**
  Typ pro uchovávání podmínek (3), definován jako *std::map<posCouple,int>*, asociativní pole dvojice slovních druhů -> maximální počet relací. V tomto případě je dvojice slovních druhů nesymetrická.
- **`CheckingFunction`**
  Typ pro funkci ověřující nějakou podmínku mezi dvěma slovními jednotkami. Definován jako *`bool (Tester::*checkingFunction)(const Condition*,const Token&,const Token&)const.`* Tedy ukazatel na funkci přijímající podmínku a dva kontrolované tokeny, vracející true pokud tokeny podmínce vyhovují, false pokud ne.
- **`std::vector<checkingFunction>checkingFunctions_`**
  Seznam kontrolních funkcí, které budou použity při ověřování platnosti podmínek.

Vladimír Rovenský
V.Rovensky@lit.cz

Privátní metody

- `bool checkCondition(const Token& t1,const Token& t2)const`
  Projde všechny dostupné podmínky (1) pro tokeny *t1 a t2* a pokud je alespoň jedna z nich splněna, vrátí true, jinak false.
- `bool checkValues(string val1,string val2,std::vector<std::string> cond1,vector<string> cond2,int tagNum=-1)const`
  Porovná hodnoty *val1,val2* (jedná se o hodnoty tagů / základní tvar atp) s hodnotami v podmínce (viz popis formátu souboru s podmínkami).
- `void readTags(std::istringstream& from,std::vector<char>& to)`
  Načte pomocí *getTags()* hodnoty tagů z výstupu nástroje *tool_chain*.
- `void getTags(const std::string& from,std::vector<char>& to)`
  Parametr *from* je string obsahující hodnoty tagů z výstupu nástroje *tool_chain*, tato funkce je načte do *to*.
- `void readLemmas(std::istringstream& from, std::vector<std::string>& to)`
  Stejná funkce jako *readTags()* pro základní tvar.
- `void getLemmas(const std::string& from,std::vector<std::string>& to)`
  Stejná funkce jako *getTags()* pro základní tvary.
- `bool checkTag(const std::string& c1,const std::string& c2,const std::string& val1,const std::string& val2)const`
  Zkontroluje hodnoty jedné pozice tagu (obou kontrolovaných slovních jednotek) proti hodnotám v podmínce. V *c1,c2* jsou hodnoty dané pozice načtené z podmínek, ve *val1,val2* jsou hodnoty této pozice načtené ze slovních jednotek. Funkce tyto hodnoty porovná a vrátí true, pokud souhlasí, false pokud ne. Hodnoty z podmínek mohou obsahovat i znaky -,+,*(,)* (viz formát souboru podmínek).
- `bool checkConjunctionCondition(const ConjunctionCondition* cond,const Token& t1,const Token& t2,const Token& conj)const`
  Zkontroluje zda tokeny *t1,t2* mohou být spojeny spojkou (čárkou) *conj*. V *cond* je jedna podmínka typu (2), kterou bude funkce kontrolovat.
- `void initCheckingFunctions()`
  Inicializace kontrolních funkcí - naplnění vektoru *checkingFunctions_*. Při přidávání nových funkcí je třeba je ve této funkci do vektoru rovněž přiřadit.
- ***`bool useCheckingFunctions(const Condition* cond,const Token& t1,const Token& t2)const`***
  Použije dostupné kontrolní funkce k ověření, zda tokeny *t1,t2* vyhovují podmínce *cond*. Vrací true, pokud všechny kontrolní funkce vrátí true, jinak false.
- `bool ccCheckTags(const Condition* cond,const Token& t1,const Token& t2)const`
  Kontrolní funkce, ověří, že hodnoty tagů v tokenech odpovídají hodnotám v podmínce.
- `bool ccCheckWordOrder(const Condition* cond,const Token& t1,const Token& t2)const`
  Kontrolní funkce, ověří, že slovosled tokenů odpovídá slovosledu vyžádanému v podmínce.
- `bool ccCheckLemmas(const Condition* cond,const Token& t1,const Token& t2)const`
  Kontrolní funkce, ověří, že hodnoty základních tvarů v tokenech odpovídají základním tvarům v podmínce.
- `bool checkEq(const std::string& c,const std::string& val)const`
  Vrací true, pokud se parametry (znaky v pozicích tagů používaných nástrojem *tool_chain*) buď rovnají, nebo pokud jeden je symbolem pro množinu obsahující druhý. Například Y,M atp. (viz dokumentace nástroje *tool_chain)*

Veřejné metody

- `bool check(const Token& t1,const Token& t2)const`
  Podívá se na slovní druhy tokenů *t1*,*t2* a najde všechny odpovídající podmínky typu (1) pro tuto dvojici slovních druhů. Dále zkontroluje, zda alespoň jedna z nich pro tyto tokeny platí, pokud ano, vrátí true, pokud žádná neplatí (nebo pokud žádná podmínka pro tuto dvojici slovních druhů není definována), vrátí false. Implementováno pomocí privátní funkce *checkCondition()*
- `bool checkConjunction(const Token& t1,const Token& t2,const Token& conj)const`
  Podívá se na slovní druhy tokenů *t1*,*t2* a najde všechny odpovídající podmínky typu (2) pro tuto dvojici slovních druhů. Dále zkontroluje, zda alespoň jedna z nich pro tyto tokeny platí, pokud ano, vrátí true, pokud žádná neplatí (nebo pokud žádná podmínka pro tuto dvojici slovních druhů není definována), vrátí false.
- `bool getMaxCondition(posCouple c,int& max)const`
  Dostane dvojici slovních druhů a pokud je pro ně definována podmínka (3), vrátí true a do *max* uloží hodnotu z podmínky. Jinak vrací false.

Struktura TagVal

Zastupuje jednu pozici jednoho tagu v podmínce – de facto jeden řádek v definici podmínky typu (1) určující přípustné hodnoty na dané pozici tagu.

- `int tagNumber`
  Číslo pozice, které se podmínka týká.
- `std::vector<char> firstValue,secondValue`
  Seznamy přípustných hodnot na této pozici tagu pro první, respektive druhou kontrolovanou slovní jednotku.
- `bool neg`
  True pokud je tento řádek negován (symbol ! v definici podmínky, viz formát souboru podmínek).
- `void clear()`
  Vyprázdní strukturu (smaže oba vektory, *neg* nastaví na false)-

Struktura BasicCondition

Základní třída reprezentující podmínku. V této podmínce lze definovat pouze požadavky na hodnoty tagů.

- `std::vector<TagVal> tagValues_`
  Seznam požadovaných hodnot na všech pozicích tagu.

Struktura Condition

Odvozena od *BasicCondition*, rozšířená o možnost odkazovat se na základní tvary, slovosled a podmínky typu (3). Určena k definici podmínek typu (1).

- `std::vector<std::string> firstLemma_,secondLemma_`
  Možné základní tvary pro první, respektive druhou slovní jednotku.
- `std::pair<bool,int> order_`
  Zachycení slovosledu. První hodnota = true, pokud na slovosledu záleží. V takovém případě je druhá hodnota = 1 pokud slovní jednotka musí být ve větě jako první z dvojice, nebo 2 pokud má být druhá. Pokud na slovosledu nezáleží (podmínka *ord* nedefinována), je první hodnota = false, druhá nedefinována.
- `bool negLemma`
  Příznak negace podmínky pro základní tvar (!*lemma* v definici podmínky).
- `bool negOrder`
  Příznak negace podmínky pro slovosled (!*ord* v definici podmínky).
- `int firstMax_`
  Podmínka typu (3). Říká, v kolika nejvýše vztazích definovaných touto podmínkou (celou strukturou) může být první slovní jednotka. Pokud není podmínka definována, hodnota je -1.

- `int secondMax_`
  Podmínka typu (3). Říká, v kolika nejvýše vztazích definovaných touto podmínkou (celou strukturou) může být druhá slovní jednotka. Pokud není podmínka definována, hodnota je -1.

Struktura ConjunctionCondition

Odvozena od *BasicCondition*, určená k definici podmínky typu (2), rozšířená o možnost odkázat se na základní tvar spojovací slovní jednotky.

- `std::string conjunction_`
  Základní tvar spojovací slovní jednotky (řádek *tok* v definici podmínky (2)). Prázdné pokud na něm nezáleží.

Soubor constants.h

Tento soubor obsahuje deklarace globálních konstant.

- `const int PARTOFSPEECH_NUMBER = 10`
  Počet slovních druhů.
- `const std::string CONDITIONS_FILENAME = "conditions.txt"`
  Jméno souboru s podmínkami.
- `const int TAGS_NUMBER = 15`
  Počet pozic tagu na výstupu nástroje *tool_chain*.
- `const int MAX_ERASED_SPLITTERS = 3`
  Program pro zjednodušení odstraňuje ze začátků vět slovní jednotky oddělující věty, např. čárku, spojky atp. Tato konstanta udává maximální počet takových slovní jednotek, které mohou být odstraněny. Délka se vztahuje na souvislý úsek slovních jednotek, mazání končí první neoddělující slovní jednotkou.

Soubor exceptions.h

Obsahuje definice tříd určených pro výjimky programu.

- `WrongConditionFileE`
  Výjimka vyvolaná když program narazí na chybu v souboru s podmínkami při jeho načítání. Typicky nedodržení formátu souboru.
- `WrongInputFileE`
  Výjimka vyvolaná při neúspěšném parsování vstupu, nejspíše neplatný soubor CSTS.

Soubor_functions.h

Obsahuje různé globální funkce potřebné v programu, sémanticky nenáležející žádné z tříd.

- `bool isSpace(char c)`
  Definice symbolů, jež jsou chápány jako bílé znaky (oddělovače řetězců).
- `std::vector<std::string> stringify(const std::vector<char>& vect)`
  Převede vektor znaků na vektor řetězců délky jedna. Používá se pro zjednodušení porovnávání hodnot atributů slovních jednotek s podmínkami.
- `int posCharToNum(char pos)`
  převede značku pro slovní druh vyprodukovanou nástrojem *tool_chain (*viz příslušná dokumentace) na číslo tohoto slovního druhu (podstatné jméno – 1, citoslovce – 10).
- `bool isSplitter(const Token& t)`
  definice řetězců, které mohou dělit souvětí na jednoduché věty, například spojky nebo některé typy zájmen (který, jež…)

Třída Graph

Zastupuje graf ve formátu seznamu vrcholů a následníků. Má možnost kontroly souvislosti, při které zohledňuje i podmínky typu (3). Ty jsou do grafu zaneseny (při jeho konstrukci) označením příslušných hran. Označení hrany má tvar dvojice čísel – číslo skupiny, do které patří, a maximálního počtu hran, které z této skupiny lze vybrat. Kontrola souvislosti potom probíhá v několika krocích:

1) DFS průchod grafu zcela bez použití označených hran. Tím se graf rozpadne na několik komponent. Pokud je jediná, graf je souvislý. Jinak vzniklo více komponent, mezi kterými mohou být pouze označené hrany.
2) Je třeba vybrat podmnožinu těchto hran tak, aby pospojovala všechny komponenty a zároveň byly uspokojeny všechny podmínky (3) – označení hran.
   a. Optimalizace – nejprve se vyberou ty hrany, které spojují komponenty, které žádná jiná hrana nespojuje. Ty zřejmě musíme vybrat, jinak graf souvislý být nemůže.
   b. Hledání podmnožiny. Probíhá generováním řešení – postupně jsou vybírány možné hrany, spojovány komponenty až do chvíle, kdy jsou buď všechny sloučeny do jedné a řešení existuje – graf je souvislý, nebo není další možnost výběru hrany – došlo by k porušení některé z podmínek. Graf tedy souvislý není.

Privátní data

- **`std::vector<int> vertices_`**
  Seznam vrcholů.
- **`std::map<int,std::vector<int> > neighbours_`**
  Seznam následníků, klíčem je číslo vrcholu, hodnotou seznam následníků.
- **`VerboseStruct`**
  Typ pro předávání informací z grafu za účelem poskytnutí podrobného výstupu (volba –v). Definován jako `std::pair<std::vector<std::pair<int,int> >,std::vector<int> >.` První hodnotou je seznam hran použitých při kontrole souvislosti grafu, druhou je mapa komponent (indexem je vrchol, hodnotou na tomto indexu číslo komponenty vrcholu), po DFS průchodu bez použití hran označených podmínkou (3).
- **`EdgeGroups`**
  Struktura udržující označení hran (podmínky (3)). Typ je definován jako `std::map<std::pair<int,int>,std::vector<std::pair<int,int> > >` tedy asociativní pole, jehož klíčem je hrana a hodnotou seznam skupin, do nichž hrana náleží (typicky jednoprvkový). Skupina = číslo skupiny + počet hran, které z této skupiny mohu použít.
- **`EdgeLine`**
  Seznam skupin pro jednu hranu. Definován jako `std::pair<std::pair<int,int>,std::vector<std::pair<int,int> > >` tedy dvojice hrana a seznam skupin, skupina = číslo skupiny a počet hran, které z ní lze vybrat.

Privátní metody

- **`bool isMarked(int v1,int v2)const`**
  Vrátí true, pokud je hrana *v1v2* označena nějakou podmínkou (3).
- **`std::vector<std::vector<int> > dfsUnmarked()const`**
  Provede DFS na neoznačených hranách grafu, vrátí mapu komponent po tomto DFS.

- `void useEdge(int nextEdge,const std::vector<edgeLine>& list,const std::vector<int>& cMap, std::vector<edgeLine>& newList,std::vector<int>& newcMap)const`
  Použije hranu při generování vhodné podmnožiny označených hran. Použít znamená sloučit komponenty spojené touto hranou do jedné (nová mapa komponent) a odpovídajícím způsobem upravit označení hran (některé spojují jiné komponenty než předtím). Parametry: nextEdge – hrana k použití, index do *list*, *list* – seznam hran a skupin, v nichž jsou, *cMap* – mapa komponent, *newList*,*newCmap* – nový seznam hran a mapa komponent, po použití hrany.
- `void optimizeEdgeList(std::vector<edgeLine>& list,std::vector<int>& cMap)const`
  Provádí optimalizaci, viz krok 2a) v popisu třídy *Graph*.
- `bool recurseEdgeList(std::vector<edgeLine> list,std::vector<int> cMap)const`
  Funkce hledající podmnožinu označených hran spojující všechny komponenty a vyhovující všem podmínkám (3). Dostane seznam hran a jejich skupin a mapu komponent, použije některou hranu, upraví si tyto dvě struktury a rekurzivně se zavolá znovu, dokud není nalezeno řešení, nebo vyčerpány možnosti.

Veřejné metody

- `void addVertex(int v)`
  Přidá vrchol v.
- `void addEdge(int v1,int v2)`
  Přidá hranu mezi v1 a v2.
- `bool checkConnectivity()const`
  Vrátí true, pokud je graf souvislý.
- `void markEdge(int v1,int v2,std::pair<int,int> group)`
  Označí hranu *v1v2* – přiřadí jí další skupinu. Skupinou se rozumí číslo skupiny a počet hran, které z ní můžu použít. Volá se při aplikování podmínky typu (3).
  Příklad. Mějme slovní jednotku T a podmínku typu (3) říkající, že T se může navázat nejvýše na K sloves. Program najde všechna slovesa $S_1 \ldots S_n$ ve stejné větě s T a každou z n hran TS_i označí stejným číslem skupiny a číslem K.

Třída Token

Reprezentuje jednu slovní jednotku věty.

Privátní data

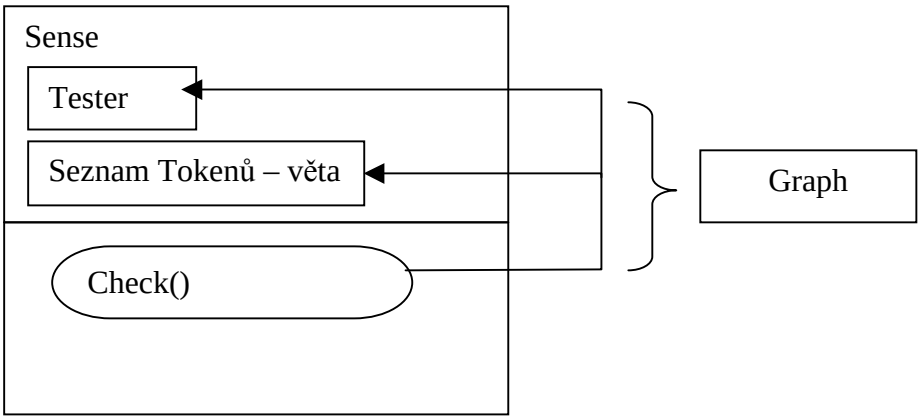
- `std::string token_`
  Vlastní slovní jednotka.
- `std::vector<char> tags_`
  Seznam hodnot všech pozic tagů z výstupu nástroje *tool_chain*.
- `int number_`
  (Pořadové) číslo slovní jednotky ve větě.
- `std::string lemma_`
  Základní slovní tvar (lemma) slovní jednotky.

Veřejné metody

- `int getPartOfSpeechNum()const`
  Vrací číslo slovního druhu slovní jednotky (1 – podstatné jméno, 10 – citoslovce).
- `char getPartOfSpeech()const`
  Vrací značku (jako tool_chain) slovního druhu slovní jednotky.

Graf struktury programu

Hrubý graf použití tříd definovaných v programu. Hlavní třídou je *Sense*, ten ke kontrole smysluplnosti *Tokenu* používá třídu *Tester*. Při kontrole je vytvořena instance třídy *Graph*.



Vladimír Rovenský
V.Rovensky@lit.cz

Příklad spuštění programu

Mějme vstupní větu „Před domem stojí auto a v domě štěká pes.“ Tuto necháme zpracovat nástrojem *tool_chain* a výstup uložíme do souboru *in.txt*. Dále spustíme program příkazem “*sense.exe –i in.txt*.“ Výstupem bude řetězec „OK“ signalizující smysluplnost věty. Pokud přidáme volbu *–v*, výstup bude následující (vysvětlivky modrou barvou):

```
Checking sentence:                      Kontrolovaná věta.
v domě štěká pes
Setting conjunction conditions:         Aplikace podmínek (2) – zde nejsou
Setting relation conditions:            Aplikace podmínek (1) – seznam smysluplných
                                        dvojic.
Relation: v---domě
Relation: v---štěká
Relation: domě---štěká
Relation: štěká---pes
Setting max number of relations conditions:         Aplikace podmínek (3) – seznam
                                                    označených hran.
Marking edge: v---štěká group: 1 max: 1
Marking edge: domě---v group: 2 max: 1
Marking edge: domě---štěká group: 3 max: 1
Marking edge: pes---štěká group: 4 max: 1
Checking connectivity:                              Kontrola souvislosti:
Optimization procedure found 4 edges.               Kolik hran nalezla
                                                    optimalizační procedura
                                                    (Graph::optimizeEdgeList()).
Components after dfs not using marked edges:        Komponenty po provedení DFS bez
                                                    označených hran:
v---0
domě---1
štěká---2
pes---3
Marked edges used:              Vygenerované řešení (vybrané označené hrany):
Used edge: domě---v
Used edge: v---štěká
Used edge: domě---štěká
Used edge: pes---štěká
Sentence OK                     Vyhodnocení jednoduché věty.

Checking sentence:              Další kontrolovaná věta.
před domem stojí auto
Setting conjunction conditions:
Setting relation conditions:
Relation: před---stojí
Relation: domem---stojí
Relation: stojí---auto
Setting max number of relations conditions:
Marking edge: před---stojí group: 1 max: 1
Marking edge: domem---stojí group: 2 max: 1
Marking edge: auto---stojí group: 3 max: 1
Checking connectivity:
Optimization procedure found 3 edges.
Components after dfs not using marked edges:
před---0
domem---1
stojí---2
auto---3
Marked edges used:
Used edge: před---stojí
Used edge: domem---stojí
Used edge: auto---stojí
Sentence OK
OK                              Celkový výsledek (všechny jednoduché věty)
```

Literatura

- Popis nástroje *tool_chain*:
  Barbora Vidová Hladká a kol., Český akademický korpus 2.0, v tisku
  http://ufal.mff.cuni.cz/rest/CAC/cac_20.html
- Popis morfologických značek:
  http://ufal.mff.cuni.cz/rest/CAC/doc-cac20/cac-guide/cz/html/ch13.html