PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Implementace expertního systému s dopředným řetězením

Forward-chaining Expert System Implementation

2010

Jakub Kaláb

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně.

5.8.2010 Jakub Kaláb

Anotace

Expertní systémy mají v praxi bohaté využití. Jejich smyslem je asistovat expertovi na danou problematiku, či jej plně nahradit. V příloze bakalářské práce implementuji prázdný expertní systém s dopředným řetězením inspirovaný systémem CLIPS jako knihovnu v programovacím jazyku Common Lisp tak, aby jej bylo možno plně integrovat do dalších programů.

Děkuji Mgr. Martinu Dostálovi, Ph.D. za vedení této bakalářské práce.

Obsah

Seznam obrázků

1. Experní systém

Expertní systém je počítačový program, který se pokouší nalézt řešení problému v situaci, kde by jinak bylo zapotřebí jednoho či několika odborníků. Idea expertního systému byla představena vývojáři ze Standfordského projektu *Heuristic programming* v čele s Edwardem A. Feigenbaumem na systémech Dendral a MYCIN. Hlavními přispěvateli tohoto projektu byli Bruce Buchanan, Edward Shortliffe, Randall Davis, William vanMelle a Carli Scott ([3]).

Expertní systém patří do skupin KBS (Knowledge-Based System, „systém založený na znalostech“) a RBS (Rule-Based System, „systém založený na pravidlech“), spadá tedy do odvětví informatiky označované jako umělá inteligence.

Vývojář expertního systému nejprve definuje vhodnou reprezentaci znalostí nějaké domény a zákonitostí mezi těmito znalostmi. Poté probíhá proces učení systému – odborníci na danou problematiku buď sami, nebo (častěji) skrze znalostního inženýra (*knowledge engineer*) zadají systému základní informace o problematice, čímž sestaví tzv. vědomostní bázi (*knowledge base*) a bázi pravidel (*rule base*). V provozu pak systém ze zadaných znalostí pomocí sestavených pravidel odvozuje nová fakta. Díky tomu je jej možno využít v praxi jako asistenta odborníka v daném oboru, jehož práce je tímto značně zefektivněna, nebo odborníka, v ideálním případě, zcela nahradit.

Pro lepší představu uvedu příklad – program na bázi expertního systému v ordinaci praktického lékaře. Na začátku zadá lékař expertnímu systému informace o příznacích známých chorob, možnostech jejich léčby, medikaci, konfliktních lécích, atd. Expertní systém potom při zadání příznaků pacienta s jistou pravděpodobností (odvislou od pravděpodobností zadaných v jednotlivých pravidlech příznak → choroba) určí možné příčiny, alternativy léčby, apod. Takto nějak právě funguje zmíněný systém MYCIN.

1.1. Charakteristické vlastnosti expertních systémů

Expertní systémy typicky

- simulují lidské usuzování na základě dedukce – systém neprovádí nad množinou znalostí algebraické výpočty
- usuzují nad libovolnou množinou správně reprezentovaných znalostí – tato znalostní báze je zcela oddělena od odvozovacího (*inferenčního*) mechanismu a není na něm nijak závislá
- problémy řeší heuristickým přístupem, často také jen na základě pravděpodobností – úspěch tedy není vždy zaručen. Díky tomu tyto systémy na druhou stranu nevyžadují perfektní data, pokud se uživatel spokojí s jistou pravděpodobností správného výsledku

— Citováno z [2].

1.2. Dopředné a zpětné řetězení

- systém s **dopředným řetězením** vyvozuje z poskytnutých dat možné závěry – jako v systém asistující lékaři ve výše uvedeném příkladu – lékař při zadávání dat systému ještě pochopitelně neví, co bude výsledkem – jaké příčiny problému systém vyhodnotí a jaké možnosti léčby pacientu doporučí

- u systému se **zpětným řetězením** je tomu naopak – systému zadáme cíle, jichž bychom rádi dosáhli, např. „rád bych založil softwarovou firmu“. Vědomostní báze systému musí disponovat informacemi o tom, co je pro založení takové firmy potřeba, tyto informace budou pravděpodobně větvené – pro firmu potřebuji prostory, zaměstnance, potřebné úřední formality. Vyřízení formalit obnáší podnikatelský záměr, návrh rozvahy rozpočtu, ... Systém z těchto dat vyhodnotí, zda je zadaný cíl možno rovnou provést, či co je k jeho splnění ještě potřeba. V příkladu u lékaře by byl takovýto systém nepoužitelný.

2. Uživatelská příručka

Tato práce se zabývá implementací již známých algoritmů, je tedy spíše praktického zaměření. Úmyslně jsem tedy posunul kapitolu s hlubším popisem implementovaných algoritmů až za uživatelskou příručku, aby byl čternář při její četbě již seznámen s používanými datovými strukturami, strukturou pravidel, apod.

Tato kapitola vyžaduje alespoň zběžnou znalost programovacího jazyka Common Lisp, jeho základních datových typů (symbol, seznam) a metod pro práci s nimi.

V kapitole budu občas uvádět v závorkách a kurzívou názvy některých použitých tříd či metod. Je to kvůli snadnější orientaci při následné četbě programátorské dokumentace. Čtenář, který tuto číst nehodlá je může směle přeskakovat.

2.1. Použité datové struktury

Uživatel knihovny Exil (dále jen knihovny) se nejčasteji setká se dvěma datovými strukturami, jsou to fakta (třída *fact*) a pravidla (třída *rule*). Fakta znalostní báze mohou být buď jednoduchá (třída *simple-fact*), specifikovaná prostým seznamem atomů (např. `(on book table)`), či strukturovaná (třída *template-fact*. Pro použití strukturovaných fakt je třeba nejdříve definovat šablonu makrem `deftemplate` následovně:

```
(deftemplate in (object location (ammount :default 1)))
```

Volání makra je velice podobné volání vestavěného makra Common lispu `defclass`, jen neobsahuje seznam tříd, neboť neposkytuje dědičnost. Prvním parametrem je jméno šablony, dalším pak seznam slotů (pojmenované části faktu) s případnými implicitními hodnotami. Specifikace takovéhoto strukturovaného faktu pak vypadá takto: `(in :object fridge :location kitchen)`.

Pozn.: Při opakovaném volání `deftemplate` se stejným jménem šablony dochází k přepsání původní definice. Má-li nově definovaná šablona jiné sloty než šablona původní a existují-li ve znalostní bázi fakta vytvořená pomocí původní šablony, vzniká nekonzistence. Makro je tedy třeba volat s rozmyslem.

Kromě fakt se uživatel setká ještě s tzv. *patterny* (pro neznalost výstižného českého ekvivalentu se budu držet dobře zažitého výrazu anglického). Tyto jsou faktům velmi podobné, jen se v nich mohou vyskytovat proměnné. Název proměnné v mé implementaci vždy začíná otazníkem (stejně jako v systému CLIPS) a je tedy od běžného atomu jasně patrný. Patterny mohou být, stejně jako fakta, jednoduché či strukturované. Při snaze použít proměnnou v popisu faktu skončí vyhodnocení výrazu chybou.

Odvozovací pravidlo je druhým stavebním kamenem expertního systému. Umožňuje inferenčnímu mechanismu vyvozovat ze zadaných fakt fakta nová. Pravidlo sestává ze dvou částí – podmínkek a aktivací. Podmínky určují, za jakých

okolností je pravidlo splněno fakty znalostní báze a je jej možno zařadit do seznamu pravidel k aktivaci (tento seznam budu dále označovat jako *agenda*). Seznam aktivací je tvořen libovolným počtem lispových výrazů (jež pochopitelně mohou, a velmi často budou, zahrnovat i knihovnou definované metody a makra), které se při aktivaci pravidla vyhodnotí. Pravidlo tedy typicky při platnosti nějakých fakt či neplatnosti jiných přidá do znalostní báze nějaká další fakta, či některerá z báze odebere.

2.2. Manipulace s fakty znalostní báze

K přidání faktu do znalostní báze souží makro `assert`, přidání tedy vypadá následovně:

```
(assert (on book table))
(assert (in :object fridge :location kitchen))
```

Při pokusu o opětované přidání již existujícího faktu se nestane nic.

Odebrání faktu z báze se provádí makrem `retract` se stejnou syntaxí. Při pokusu o odebrání neexistujícího faktu se opět nic nestane.

Posledním makrem z této skupiny je `modify` to slouží k modifikaci faktu (jeho nahrazení jiným) a používá se takto:

```
(modify (in :object snack :location fridge)
        (in :object snack :location schoolbag))
```

Toto makro je ve skutečnosti jen zkratkou pro postupné volání `assert` a `retract`. Neslouží ani tak k ušetření zdrojového kódu, jako spíš ke zčitelnění aktivací pravidla – z jeho použití je evidentní vztah dvou faktů.

2.2.1. Definice skupiny faktů

Pro pohodlnější práci s větším množstvím faktů jsem (stejně jako systém CLIPS) implementoval makro `deffacts`. Volání tohoto makra obsahuje název skupiny faktů náledovaný jejich seznamem:

```
(deffacts stuff-i-ve-got-in-me-fridge
  (in :object coke :location fridge)
  (in :object cheese :location fridge)
  (is-expired cheese)
  (in :object butter :location fridge)
        ...
  )
```

Takto definovaná fakta se ve znalostní bázi neobjeví hned, nýbrž až po zavolání makra `reset` (bez parametrů). Toto makro odstraní ze znalostní báze všechna fakta a poté ji naplní fakty definovanými ve skupinách makrem `deffacts`. K pouhému vyčištění znalostní báze slouží makro `clear` (opět bez parametrů).

2.3. Manipulace s odvozovacími pravidly

Definice odvozovacího pravidla se provádí makrem `defrule` a zahrnuje jméno pravidla a seznamy podmínek a aktivací. Podmínky jsou od aktivací odděleny symbolem `=>`. Vypadá to tedy nějak takto:

```
(defrule find-and-take-green-fruit-to-the-left-of-orange-vegetable
  (green ?x)
  (fruit ?x)
  (orange ?y)
  (vegetable ?y)
  (next-to :left-one ?x :right-one ?y)
=>
  (take ?x)
  (take ?y))
```

Jak je z příkladu zřejmé, podmínky pravidel mohou obsahovat proměnné, jde tedy o patterny. Podmínky mohou být také negované, pak se ověřuje neexistence odpovídajícího faktu ve znalostní bázi. Negované podmínky jsou specifikovány symbolem '-' (minus) ještě před prvním atomem (či před názvem šablony u složených patternů).

Při opakovaném volání `defrule` se stejným jménem pravidla dochází k přepsání jeho definice.

K zneplatnění definice pravidla slouží makro `undefrule`, jehož jediným parametrem je název pravidla.

2.4. Spouštění inferenčního mechanismu

Jakmile jsou systému zadána inicializační fakta a pravidla, je třeba spustit výpočet. Ten pak probíhá v kolech, kde se střídají dvě akce. Nejdříve inferenční machanismus zjistí, která pravidla mají splněné podmínky a jakým faktem je ta která podmínka splněna. Následně je ze seznamu splněných pravidel (*agendy* vybráno pravidlo a toto je aktivováno. Po vyhodnocení aktivací pravidla se opět aktualizuje seznam splněných pravidel atd.

Výpočet inferenčního mechanismu se spouští funkcí `run` (bez parametrů). Chceme-li výpočet sledovat po krocích, můžeme použít funkci `step` (též bez parametrů), která vždy spustí jen jedno kolo výpočtu.

Dále je k dispozici funkce `halt`, která výpočet přeruší. Tuto funkci nemá smysl používat samostatně, její volání je ale někdy vhodné zařadit do seznamu aktivací některých pravidel. Tím signalizujeme, že jsou-li podmínky pravidla splněny, dosáhli jsme, čeho jsme chtěli a výpočet může skončit.

2.5. Strategie výběru pravidla k aktivaci

Výběr následujícího pravidla k aktivaci je ovlivněn zvolenou strategií. K výběru strategie slouží makro `set-strategy`, jehož jediným parametrem je název kýžené strategie. Kromě vestavěných strategií je možné definovat také strategie vlastní. To se provádí makrem `defstrategy`, jemuž předáme název nové strategie a funkci, která z agendy, již dostane parametrem vybere jednu z aktivací a tu vrátí. Definice strategií tedy mohou vypadat třeba takto:

```
(defstrategy fast-strategy #'first)
(defstrategy random
  (lambda (agenda)
    (nth (random (length agenda))
         (agenda))))
```

O vestavěných strategiích budu hovořit v následující části textu.

2.6. Sledování změn prostředí

Poslední věcí sloužící k zefektivnění práce s knihovnou je možnost sledovat změny ve znalostní bázi či agendě. Sledovat je možno přidávání či ubíraní faktů do a ze znalostní báze, definice a zneplatnění pravidel a změny v agendě.

Sledování se zapíná makrem `watch` a vypíná makrem `unwatch`. Obě makra berou očekávají jako parametr předmět sledování, tedy jeden ze symbolů `facts`, `rules` či `activations`.

2.7. Příklad práce s knihovnou

Na závěr této sekce uvedu krátký příklad práce s knihovnou. Řádky začínající "`EXIL>`" označují vstup uživatele, zbytek je výstup programu.

```
EXIL> (deftemplate goal (action object from to))
#<TEMPLATE GOAL ((ACTION :DEFAULT NIL) (OBJECT :DEFAULT NIL)
                 (FROM :DEFAULT NIL) (TO :DEFAULT NIL))>

EXIL> (deftemplate in (object location))
#<TEMPLATE IN ((OBJECT :DEFAULT NIL) (LOCATION :DEFAULT NIL))>

EXIL> (deffacts world
        (in :object robot :location A)
        (in :object box :location B)
        (goal :action push :object box :from B :to A))
T
```

```
EXIL> (defrule stop
        (goal :object ?x :to ?y)
        (in :object ?x :location ?y)
      =>
        (halt))
#<RULE STOP>

EXIL> (defrule move
        (goal :object ?x :from ?y)
        (in :object ?x :location ?y)
        (- in :object robot :location ?y)
        (in :object robot :location ?z)
      =>
        (modify (in :object robot :location ?z)
                (in :object robot :location ?y)))
#<RULE MOVE>

EXIL> (defrule push
        (goal :object ?x :from ?y :to ?z)
        (in :object ?x :location ?y)
        (in :object robot :location ?y)
      =>
        (modify (in :object robot :location ?y)
                (in :object robot :location ?z))
        (modify (in :object ?x :location ?y)
                (in :object ?x :location ?z)))
#<RULE PUSH>

EXIL> (watch facts)
T

EXIL> (watch activations)
T

EXIL> (reset)
==> (IN (OBJECT . ROBOT) (LOCATION . A))
==> (IN (OBJECT . BOX) (LOCATION . B))
==> (GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
==> Activation MOVE:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . B))
 (IN (OBJECT . ROBOT) (LOCATION . A)))
NIL
```

```
EXIL> (run)
Firing Activation MOVE:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . B))
 (IN (OBJECT . ROBOT) (LOCATION . A)))
<== (IN (OBJECT . ROBOT) (LOCATION . A))
==> (IN (OBJECT . ROBOT) (LOCATION . B))
==> Activation PUSH:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . B))
 (IN (OBJECT . ROBOT) (LOCATION . B)))

Firing Activation PUSH:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . B))
 (IN (OBJECT . ROBOT) (LOCATION . B)))
<== (IN (OBJECT . ROBOT) (LOCATION . B))
==> (IN (OBJECT . ROBOT) (LOCATION . A))
<== (IN (OBJECT . BOX) (LOCATION . B))
==> (IN (OBJECT . BOX) (LOCATION . A))
==> Activation STOP:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . A)))

Firing Activation STOP:
((GOAL (ACTION . PUSH) (OBJECT . BOX) (FROM . B) (TO . A))
 (IN (OBJECT . BOX) (LOCATION . A)))
Halting
NIL
```

Příklad je převzat z [2] a je původně určen pro systém CLIPS.

3. Použité algoritmy

Přiložená implementace expertního systému zahrnuje několik z literatury čerpaných algoritmů. V prvé řadě je to algoritmus Rete, který se stará o pattern matching podmínek odvozovacích pravidel proti faktům znalostní báze, tedy o zjišťování, která odvozovací pravidla je možno aplikovat v dalším kroku výpočtu. Poté je použito několik velice jednoduchých algoritmů pro výběr pravidla, jenž bude jako další aktivováno.

3.1. Algoritmus Rete

Algoritmus Rete (materiály nabízí několik alternativ čtení názvu tohoto algoritmu, mezi nimi „reet“ [rýt] či „ree-tee“ [rýtý] [1]) kontroluje splnění podmínek inferenčních pravidel fakty znalostní báze. Použití naivního přístupu, tedy zkoušení platnosti všech podmínek proti všem faktům při každé změně znalostní báze není možné, neboť tato kontrola se již při několika stovkách faktů či desítkách pravidel (závisí samozřejmě na výkonu výpočetního hardware, jeho zatížení, atd.) stává příliš časově náročnou.

Algoritmus pracuje na základě dataflow sítě (či chcete-li sítě toku dat), kterou průběžne konstruuje a upravuje při každém přidání či odebrání inferenčního pravidla. Přepočet splněných pravidel algoritmem Rete je o tolik rychlejší díky tomu, že tato síť uchovává v každém uzlu výsledky z minulých výpočtů a nově přidaný (či odebraný) fakt touto sítí jen velice rychle „proteče“ a aktualizuje výsledky jen ve velmi malé části uzlů.

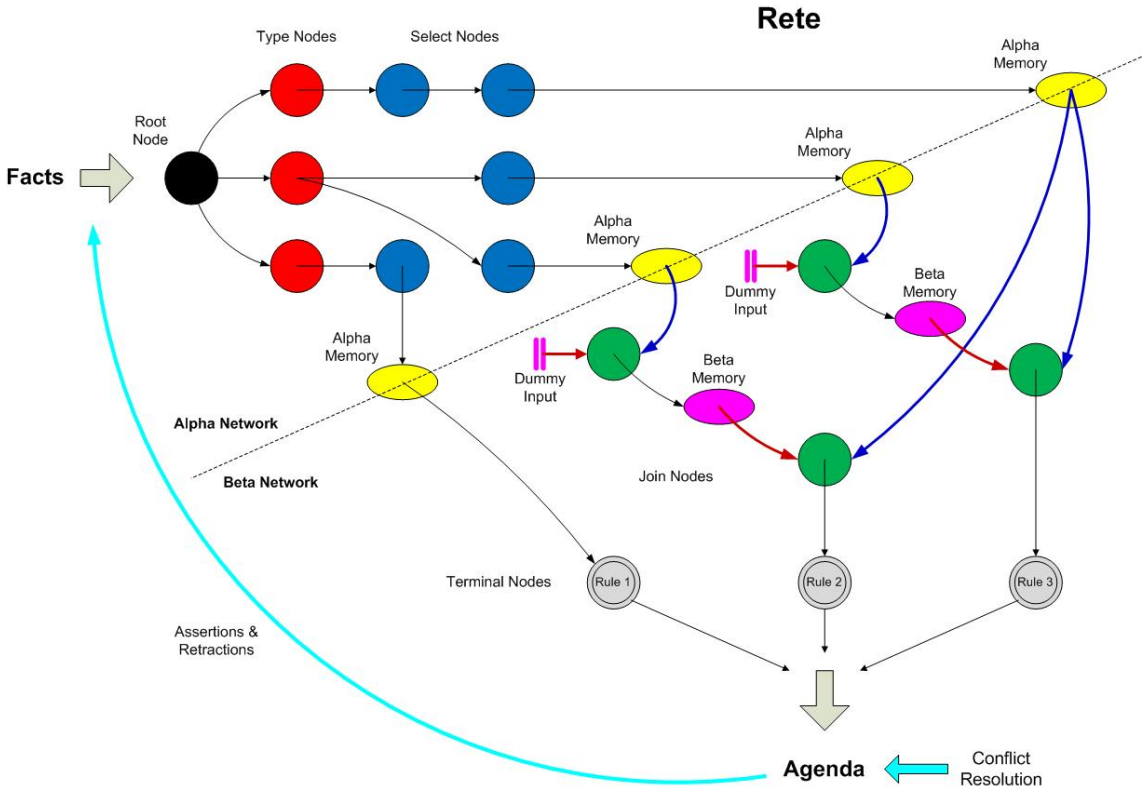
Dataflow síť algoritmu je z pohledu teorie grafů spojitý orientovaný graf. Máme-li dva uzly a orientovanou hranu, která je spojuje, budu uzel, z nějž hrana vede, označovat jako rodiče a uzel, do nějž vede, jako potomka. Síť je tvořena dvěma částmi označovanými jako alfa a beta. Navštívení uzlu sítě Rete algoritmu nazýváme aktivací. Některé uzly mohou být aktivovány i více způsoby.

Pozn.: V této práci budu slovo potomek používat v neživotném tvaru. Není to pravděpodobně zcela gramaticky korektní, ovšem označní „ti potomci“, mluvíme-li o uzlech dataflow sítě, mi přijde poněkud směšné.

Na obrázku 1. je znázorněna dataflow síť Rete algoritmu. Pokud by ovšem mělo znázornění odpovídat používané terminologii, měla by být alfa část sítě na pravé straně. To si ale čtenář jistě snadno představí.

3.1.1. Alfa síť

Alfa část sítě řeší konstantní porovnávání přidávaných faktů s patterny v podmínkách pravidel, ignoruje tedy vazby proměnných a jejich konzistenci. Jde v podstatě o strom, jehož každý uzel testuje právě jeden atom faktu. Kořen alfa sítě je aktivován faktem, faktem aktivuje všechny své potomky. Tyto už provádějí testy jednotlivých atomů, má-li atom požadovanou hodnotu, předá uzel fakt svým



Obrázek 1. Dataflow síť Rete algoritmu[4]

potomkům a ty pokračují v testování. Projde-li fakt všemi konstatními testy, aktivuje poslední testovací uzel fakterm k němu přidružený paměťový alfa-uzel. Ten fakt uloží do své paměti a aktivuje jím své potomky v beta částí sítě.

Kořen alfa části sítě může ještě rozlišovat typ faktu (jde-li o fakt jednoduchý či složený, ve druhém případě pak typ šablony). A podle toho aktivovat faktem jen část potomků.

Pro lepší představu uvedu příklad. Mějme následující pravidlo:

```
(defrule move
  (goal :object ?x :from ?y)
  (in :object ?x :location ?y)
  (- in :object robot :location ?y)
  (in :object robot :location ?z)
  =>
  (modify (in :object robot :location ?z)
  (in :object robot :location ?y)))
```

Kořen alfa sítě tedy bude mít dva potomky – pro šablony `goal` a `in` (ignorujme prozatím, že třetí podmínka pravidla je negovaná, to řeší až beta část dataflow sítě). Z uzlu pro šablonu `goal` povede hrana rovnou do alfa-paměti, všechny zbylé

atomy podmínky jsou totiž proměnné. Z uzlu pro šablonu povede také hrana do další alfa paměti, neboť totéž platí pro druhou podmínku. Bude mít ale i dalšího potomka a to testovací uzel, zda je slot `object` roven hodnotě `robot`. Ten pak už aktivuje poslední paměťový alfa-uzel.

Máme tedy dva typové uzly (pro šablony `goal` a `in`, jeden testovací uzel a tři uzly paměťové. To zatím není tak působivé, uvědomme si ale, že definujeme-li další pravidlo, které bude mít některé podmínky stejné jako výše uvedené, žádné uzly nám pro tyto podmínky nepřibudou. A budou-li podmínky alespoň podobné, přibudou jen testovací uzly pro atomy patternu, které se budou lišit.

Zde je možno pozorovat první z výhod oproti naivnímu přístupu. Totiž máme-li pravidla se stejnými či podobnými podmínkami, velká část testů se provádí jen jednou, což nám markantně snižuje časovou náročnost výpočtu.

3.1.2. Beta síť

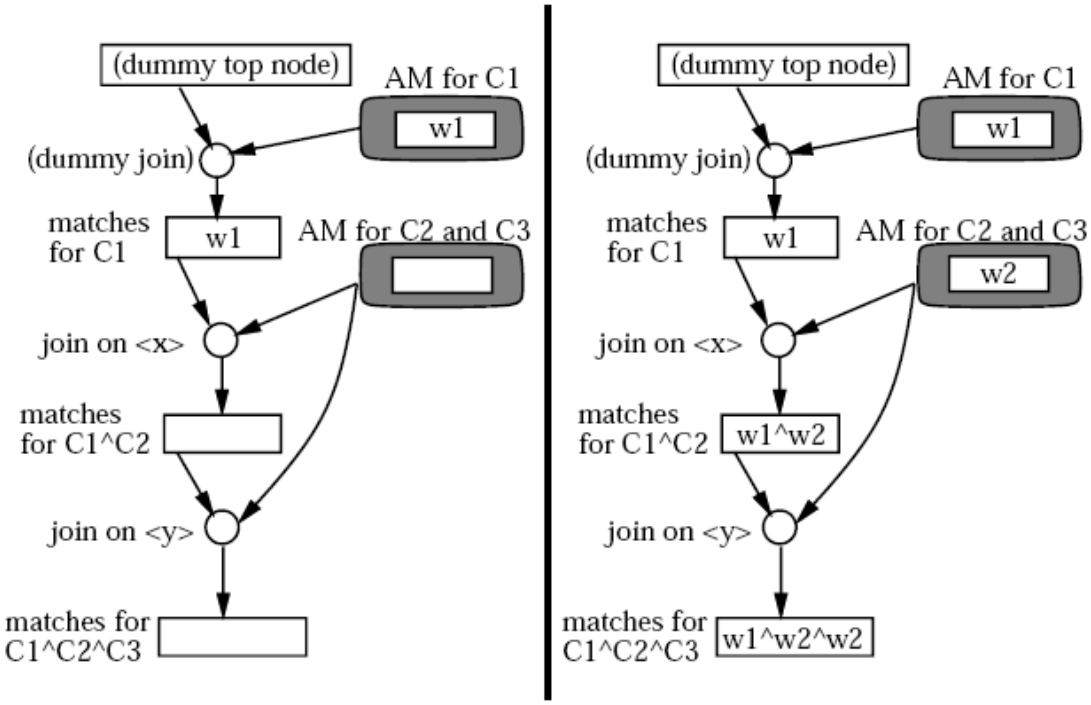
Beta část sítě je opět tvořena několika typy uzlů. Kromě uzlů paměťových jsou zde uzly, které testují konzistenci vazeb proměnných mezi jednotlivými podmínkami inferenčního pravidla, uzly negativních testů a produkční uzly.

Paměťové beta-uzly už neuchovávají jednotlivá fakta, jako je tomu v alfa části dataflow sítě, nýbrž tzv. *tokeny*, tedy skupiny faktů. Každý paměťový uzel udržuje seznam tokenů, jimiž byly splněny dosavadní podmínky inferenčního pravidla (tzv. *partial match*). Máme-li tedy pravidlo o čtyřech podmínkách, budeme pro něj potřebovat čtyři paměťové beta-uzly. První uzel uchovává jednomístné tokeny s fakty, které prošly první podmínkou. Neliší se tedy příliš od paměťového alfa-uzlu. Druhý už ale uchovává tokeny s dvojicí faktů – faktem, jenž prošel první podmínkou a faktem, který prošel druhou. Takto se fakta v tokenech nabalují, až máme konečně v posledním uzlu token stejné délky, jako je počet podmínek inferenčního pravidla. Tento uzel označíme jako produkční – je-li aktivován tokenem, našli jsme kompletní shodu faktů s podmínkami pravidal (*complete match*), v tuto chvíli algoritmus Rete ohlásí nalezení shody (dvojice pravidlo-token) prostředí a ta je přidána do seznamu aktivací.

Hybnou silou beta síťě jsou spojovací uzly (tzv. *join node*). Ty testují konzistenci vazeb proměnných mezi fakty splňujícími jednotlivé podmínky inferenčního pravidla. Jsou-li vazby konzistentní, uzel vytvoří nový token a aktivuje jím přidružený paměťový beta-uzel. Spojovací uzel může být aktivován dvěma způsoby – buď nově přidaným či ubraným (a konstantními testy prošlým) faktem z paměťového alfa-uzlu (tzv. *aktivace zprava*; toto je místo, kde beta část sítě navazuje na alfa část), nebo tokenem z nadřazeného paměťového beta-uzlu (*aktivace zleva*). V obou případech ale spojovací uzel prohledá druhou paměť toho druhého rodičovského uzlu pro získání konzistentních tokenů (resp. faktů) a při nalezení shody aktivuje potomky. nnn Speciálním případem spojovacích uzlů jsou uzly s negativními testy. Tyto mohou být, stejně jako běžné spojovací uzly, aktivovány z obou stran. Podstatný rozdíl je ale v tom, že negativní spojovací uzel aktivuje své po-

tomky jedině tehdy, nenalezl-li pro daný token žádný konzistentní fakt ve paměti svého rodičovského alfa-uzlu.

Opět uvedu příklad. Zůstaňme u pravidla z příkladu předchozího. Alfa část dataflow sítě pro uvedené pravidlo má tři paměťové uzly. V beta části budeme potřebovat tři spojovací uzly (z nichž jeden bude negativní). První bude testovat konzistenci vazeb mezi prvními dvěma podmínkami pravidla, druhý mezi třetí podmínkou a prvními dvěma, atd. Druhý i třetí spojovací uzel bude mít zprava připojen týž paměťový alfa-uzel (opět recyklace uzlu, tentokrát dokonce v rámci jednoho pravidla). Ke každému spojovacímu uzlu je jako potomek připojen paměťový beta-uzel, který uchovává tokeny s částečnými (v prvních třech případech) či úplnými (ve čtvrtém případě) shodami.



Obrázek 2. Spojovací uzly mezi jednotlivými podmínkami inferenčního pravidla[1]

3.1.3. Přidání a odebrání inferenčních pravidel

Definuje-li uživatel nové inferenční pravidlo je třeba pro něj vytvořit nové či pospojovat již existující uzly dataflow sítě. Algoritmus postupně prochází podmínky pravidla, vytváří (či recykluje) pro každou paměťový beta-uzel a spojovací beta-uzel, který napojí na následující paměťový beta-uzel. Ke každému spojovacímu uzlu také vytvoří (či plně nebo částečně recykluje) cestu v alfa části sítě. Poslední paměťový uzel označí jako produkční a uloží v něm pravidlo aby mohlo být při splnění všech podmínek předáno spolu s tokenem zpět prostředí a zařazeno

do agendy.

Při odstranění pravidla ze systému je pravidlo odstraněno také z odpovídajícího produkčního uzlu a všechny nyní zbytečné uzly jsou ze sítě odstraněny.

3.2. Strategie pro výběr pravidla k aktivaci

Expertní systém je velice obecný výpočetní mechanismus a proto není možné určit jednu odvozovací strategii, která by pokrývala všechny domény problémů, v nichž ho lze využít. Implementace proto poskytuje několik vestavěných strategií pro výběr pravidla z agendy k následné aktivaci a umožňuje i definici uživatelských strategií.

- **Strategie nejnovější aktivace (*depth strategy*)** vybírá z agendy pravidlo, které sem bylo přidáno jako poslední. Inferenční mechanismus tedy provádí jakýsi výpočet do hloubky, spouští nově aktivovaná pravidla dokud to lze a teprve při jejich vyčerpání se ve výpočtu vrací k pravidlům aktivovaným dříve.

- **Strategie nejstarší aktivace (*breadth strategy*)** vybírá z agendy pravidlo, které sem bylo přidáno jako první. Je tedy přesným opakem strategie předchozí, provádí výpočet „do šířky“.

- **Strategie nejobecnějšího pravidla (*simplicity strategy*)** vybírá pravidlo s nejnižším počtem podmínek, tedy obecnější (statisticky čím méně je podmínek, tím více najdeme shod, které pravidlo splní).

- **Strategie nejkonkrétnějšího pravidla (*complexity strategy*)** vybírá pravidlo s nejvyšším počtem podmínek, tedy specifičtější.

– Volně převzato z [2].

4. Popis implementace

Knihovna je napsána v programovacím jazyku Common Lisp. K jejímu vývoji jsem použil textový editor GNU Emacs s rozšířením SLIME (The Superior Lisp Interaction Mode for Emacs) a kompilátor SBCL (Steel Bank Common Lisp). Vývoj jsem prováděl na architektuře GNU/Linux x86. Funkčnost knihovny jsem testoval také ve vývojovém prostředí LispWorks®.

Zdrojový kód knihovny je rozdělen celkem do 15 souborů:

```
packages.lisp
utils.lisp
templates.lisp
facts.lisp
patterns.lisp
rules.lisp
rete-generic-node.lisp
rete-alpha-part.lisp
rete-beta-part.lisp
rete-net-creation.lisp
matches.lisp
activations.lisp
strategies.lisp
environment.lisp
export.lisp
```

- V souboru `packages.lisp` definuji package („balíček“) `exil`, který bude definované funkce, metody a makra obsahovat.
- Soubor `utils.lisp` obsahuje jednoduché obecné funkce a makra, které jsem během vývoje pro zjednodušení další práce napsal.
- V souboru `export.lisp` jsou všechny metody a makra, které knihovna poskytuje uživateli. Tyto jsou z package `exil` exportovány, lze je tedy volat bez nutnosti vstupu do package (před název je třeba připojit “`exil:`”).
- Ve zbytku souborů definuji jednotlivé třídy a metdody. Kód v každém ze souborů staví na konstruktech definovaných v předchozích souborech. Prostředí (třída `environment`), definovaná ve stejnojmenném souboru nakonec všechny dříve definované třídy zakomponuje do jednoho celku a vytvoří celek, na kterém může soubor `export` vystavět makra, která uživatel bude nakonec používat.

4.1. Templates

Soubor vytváří základní mechanismus pro práci se strukturovanými daty. Třída `template` obsahuje sloty pro uchování informací o uživatelsky definovaných šablonách. Všechna strukturovaná data nebo patterny jsou pak instancemi tříd odvozených od `template-object`. Při vytváření těchto instancí mechanismus prochází kromě inicializačních parametrů také specifikaci slotů uloženou v konkrétní instanci třídy `template` a tak umožňuje nastavení implicitnch hodnot. Krom toho soubor definuje predikát `tmpl-object-specification-p`, který podle specifikace faktu (resp. patternu) určí, jde-li o fakt jednoduchý či strukturovaný.

4.2. Facts a Patterns

Definice základních datových struktur `fact` a `pattern` a od nich odvozených `simple-fact`, `template-fact`, `simple-pattern` a `template-pattern`. Nevyžaduje hlubší popis.

4.3. Rules

Definice třídy `rule` pro uchování inferenčních pravidel. Třída nese informace o jméně pravidla, jeho podmínkách (typu `pattern`) a aktivacích (prostý seznam lispových výrazů, které se při aktivaci pravidla postupně vyhodnotí).

4.4. Rete

Algoritmus Rete je nejsložitější částí implementace knihovny, jeho kód je tedy pochopitelně nejdelší. Je rozdělen do čtyř souborů.

První z nich, `rete-generic-node` definuje třídu `node`, která je základem všech uzlů dataflow sítě algoritmu. Tím je zajištěno, že každý uzel sítě uchovává seznam svých potomků a implementuje metody `activate` a `inactivate`, skrze něž je svým rodičem upozorňován na změny ve znalostní bázi. Každý typ uzlu na tyto aktivace reaguje jinak.

Implementace aktivací konkrétních souborů zabírá většinu kódu následujících dvou souborů – `rete-alpha-part` a `rete-beta-part`. Kořenem celé dataflow sítě algoritmu Rete je `alpha-top-node`. Ten uchovává hashovací tabulku jednotlivých podsítí alfa-sítě podle typu šablony faktů, které ta která síť testuje. Je zde navíc jedna podsíť pro fakta jednoduchá (ta nemají šablonu).

Jakmile je určena podsíť, jíž bude nově přidaný (resp. odebraný) fakt procházet, je aktivovaný příslušný „podkořen“ (`alpha-subtop-node`), ten už má potomky typu `alpha-test-node`. Při průchodu každým testovacím uzlem je otestován jeden atom faktu. Poslední testovací uzel předá při kladném výsledku testu fakt paměťovému uzlu (`alpha-memory-node`), ten si jej uloží a pošle ho beta části sítě.

V souboru `rete-beta-part` je kromě jednotlivých typů uzlů definována třída `token`, uchovávající skupinu faktů prošlých podmínkami inferenčního pravidla. Relativně zajímavým prvkej je zde také přístup k testování kozistence vazeb proměnných mezi jednotlivými podmínkami pravidla. Algoritmus zde při aktivaci spojovacího uzlu (`beta-join-node` a `beta-negative-node`) vůbec nehledí na konkrétní proměnné, jak by čtenář mohl očekávat. Při tvorbě dataflow sítě je totiž pro každý spojovací uzel vytvořen seznam objektů typu `test`. Každý test slouží k otestování konzistence jednoho pole aktuální podmínky s předchozími podmínkami. Test obsahuje tři hodnoty – název (v případě strukturovaného patternu podmínky) či index (v případě nestrukturovaného) pole aktuálně testované podmínky, číslo značící, o kolik podmínek zpět má hledat konzistentní pole a název či index tohoto pole.

Pro lepší porozumnění uvedu příklad. Mějme pravidlo

```
(defrule find-green-block-on-a-red-one
  (green ?x)
  (red ?y)
  (on ?x ?y)
=>
  ; zde budou aktivace)
```

a sestavme testy pro `beta-join-node` testující konzistenci mezi vazbami proměnných ve třetí podmínce proti podmínkám předchozím. Podmínky jsou typu nestrukturovaných patternů, pole číslujme od 0. Test konzistentní vazby proměnné `?x` bude mít hodnoty `(1 2 1)`, testujeme totiž pole aktuální podmínky s indexem 1 proti poli s indexem 1 podmínky o 2 zpět. Pokud je prostřední hodnota testu 0, znamená to, že testujeme konzistenci vazby jen v rámci aktuální podmínky. To se může stát, objevuje-li se daná proměnná v podmínce vícekrát. Test vazby proměnné `?y` bude mít, jen pro úplnost, hodnoty `(2 1 1)`.

Jak tedy vidíme, hotový `beta-join-node` na proměnné už vůbec nehledí, jen mechanicky vykoná testy a skončí-li všechny úspěšně, přidá k tokenu fakt prošlý aktuální podmínkou a aktivuje jím své potomky (ty jsou typu `beta-memory-node`.

Testování se poněkud komplikuje u negativních spojovacích uzlů (`beta-negative-node`). Zde musíme při každé aktivaci (přidání nového faktu) zkontrolovat, zde není konzistetní s některými z dosud prošlých faktů. Pokud takovou shodu nalezneme, je třeba informovat potomky, aby všechny tokeny s těmito fakty ze svých proměnných odstranily. Při inaktivaci (odstranění faktu ze znalostní báze) je naopak třeba zkontrolovat, zda toho nebyl poslední fakt, který danému tokenu v nesplnění testu bránil (nezapomínejme že je řeč o negativních uzlech). Pokud tomu tak je, je třeba tokenem aktivovat potomky uzlu.

Kdyby nám nezáleželo na efektivitě, bylo by implementačně nejjednodušší po každé aktivaci (resp. inaktivaci) přepočítat možné shody veškerých tokenů v rodičovské beta-paměti proti všem faktům z alfa-paměti. Toto je ale neúnosně časově

náročné. Přidal jsem proto k tomuto účelu do třídy `token` ještě slot, v němž si token v paměti negativního spojovacího uzlu uchovává seznam konzistentních faktů, jež mu brání v úspěšném nesplnění testů. Když je negativní uzel inaktivován odebráním faktu ze znalostní báze, stačí tedy jen odstranit fakt z těchto seznamů v tokenech a je-li po tomto odstranění seznam některého tokenu prázdný, může tento pokračovat dataflow sítí níže.

`Beta-memory-node` má kromě uchovávání tokenů ještě jednu funkci – slouží jako produkční uzel – lze v něm uložit seznam pravidel, která jsou uloženými tokeny splněna. Uzel tedy v případě aktivace novým tokenem zkontroluje, zda nějaká pravidla v seznamu má a pokud ano, zavolá metodu `add-match` a předá jí splněné pravidlo a token, jenž jeho podmínkami prošel. Tento pár (označovaný jako `match`) je poté přidán k agendě.

4.5. Matches

Soubor `matches` definuje třídu `match`. Dokud jsem nezačal implementovat různé strategie pro výběr pravidla k aktivaci, ukládal jsem v agendě jen (pravidlo . token) a této třídy nebylo třeba. Některé strategie ale při výběru pravidla zkoumají, kdy bylo pravidlo na agendu přidáno. To si vyžádalo definici vlastní třídy `match`, která tuto informaci kromě pravidla a tokenu uchovává.

4.6. Activations

Metody definované v souboru `activations` řeší samotnou aktivaci pravidla. Výrazy v seznamu aktivací totiž není možné bez předchozí úpravy vyhodnotit, neboť velice často obsahují proměnné navázané v podmínkách. Je tedy třeba z kombinace seznamu podmínek pravidla a tokenu, který pravidlo splnil dostat vazby těchto proměnných. Proměnné v aktivacích jsou poté nahrazeny takto zjištěnými atomy a teprve tehdy je možno aktivace vyhodnotit.

4.7. Strategies

Soubor `strategies` obsahuje definice jednotlivých strategií pro výběr pravidla z agendy k následující aktivaci. Jde tedy pouze o seznam funkcí, kterým je agenda jako parametr předána a ony jeden z objektů typu `match` vrátí. Ten je poté z agendy odstraněn.

4.8. Environment

V souboru `environment` definuji třídu stejného jména, která pouze zastřešuje všechny doposud definované konstrukty. Obsahuje seznam definovaných šablon, znalostní bázi faktů, definované skupiny faktů, seznam pravidel a agendu. Kromě toho obsahuje instanci třídy `rete`, jejíž jediným zajímavým prvkem je odkaz

na kořen stromu dataflow sítě, seznam definovaných strategií a název aktuálně zvolené strategie. Posledním prvkem prostředí je seznam aktuálně aktivních *watcherů*, tedy sledovaných objektů (viz 2.6.).

5. Závěr

Má implementace prázdného expertního systému podává (při použití stejné inferenční strategie) stejné výsledky jako systém CLIPS, jímž byla inspirována. Neposkytuje ale všechny jeho možonsti a nedosahuje pochopitelně zdaleka takovéhu výpočetního výkonu. V dalším studiu bych se rád zabýval právě zvyšováním efektivity výpočtů a tedy snižováním reakční doby systému. Věnovat bych se chtěl také rozšíření systému o další inferenční strategie (např. LEX a MEA), které podobné systémy často poskytují a rozšiřováním syntaxe pravidel. Systém CLIPS např. poskytuje možnost „uložení“ faktu, jež vyhovuje dané podmínce pravidla, do proměnné a následné práce s ním. Při rozhodování o použité syntaxi, se kterou se knihovní funkce volají jsem se snažil co nejvíce přiblížit lispovým idiomům, (volání makra `deftemplate` je např. velice podobné volání `defclass` standardního objektového systému CLOS) rád bych v dalším studiu poskytl rozšíření syntaxe tak, aby bylo možno programy vytvrořené pro systém CLIPS přímo použít v mé implementaci.

Reference

[1] Doorenbos, R. B.: *Production Matching for Large Learning Systems*. 1995.

[2] Jackson, P.: *Introduction to Expert Systems*. Addison Wesley, 1998, ISBN 0-201-87686-8.

[3] Wikipedia: Expert system — Wikipedia, The Free Encyklopedia. 2010, [Online].
URL `<http://en.wikipedia.org/wiki/Expert_system>`

[4] Wikipedia: Rete algorithm — Wikipedia, The Free Encyklopedia. 2010, [Online].
URL `<http://en.wikipedia.org/wiki/Rete_algorithm>`