VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYHLEDÁVÁNÍ SPOJŮ V JÍZDNÍCH ŘÁDECH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Ondřej Žižka
AUTHOR

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYHLEDÁVÁNÍ SPOJŮ V JÍZDNÍCH ŘÁDECH

FINDING CONNECTIONS IN SCHEDULE-BASED TRANSIT NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Ondřej Žižka
AUTHOR

VEDOUCÍ PRÁCE Ing. Jiří Jaroš
SUPERVISOR

BRNO 2008

Abstrakt

Nedílnou součástí moderní společnosti je pravidelná přeprava osob. Za tím účelem existují přepravní systémy, jejichž spoje se řídí předem daným jízdním řádem.

Tato práce si klade za cíl prozkoumat možnosti automatického vyhledávání spojů z jednoho místa na jiné, takové vyhledávání implementovat, a prozkoumat možnosti použití metod soft-computingu pro urychlení hledání v rozsáhlých jízdních řádech.

V průběhu implementace se také pokusíme prozkoumat možnosti hromadných pseudo-paralelních výpočtů v procedurálních rozšířeních jazyka SQL.

Klíčová slova

Jízdní řády, spoje, spojení, linka, vyhledávání, hledání, MHD, doprava, Integrovaný dopravní systém, IDS, Celostátní informační systém jízdních řádů, CIS JŘ, IDOS

Abstract

Everyday need of modern society is a mass personal transit on a regular basis. For this purpose, mass transit systems exist which obey aforethought schedule.

This thesis' goal is to examine the means of automatic search of connections from one place to another, implement such search, and to advance the search algorithm using the soft-computing paradigms. Minor goal would be a research of SQL language procedural capabilities, which could support mass pseudo-parallel computations.

Keywords

Timetable, time table, schedule, transport service, mass transit system, traffic, urban mass transportation, search, national transport schedule information system

Citace

Žižka Ondřej: Vyhledávání v jízdních řádech. Brno, 2008, diplomová práce, FIT VUT v Brně.

Vyhledávání spojů v jízdních řádech

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Jiřího Jaroše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Další informace mi poskytli pracovníci firem Chaps, s.r.o., Dopravní podnik města Brna, a.s., a autor programu timetab Pavel Machek.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Poděkování

Děkuji vedoucímu diplomové práce, Ing. Jiřímu Jarošovi, za ochotnou spolupráci při její tvorbě; Davidu Švingrovi, Tomáši Chlebničanovi a Peterovi Chlebničanovi z firmy Chaps, s.r.o., za vysvětlení praktických problémů při řešení vyhledávání v jízdních řádech; panu Saidlovi z oddělení jízdních řádů Dopravního podniku města Brna za předvedení vnitřní informační struktury dopravní sítě MHD a vysvětlení problémů při tvorbě jízdních řádů; Dopravnímu podniku města Českých Budějovic za poskytnutí reálných testovacích dat; Ing. Danielovi Rozsnyóovi za poskytnutí výpočetního výkonu pro učení a testování neuronových sítí.

Poděkování patří také mým přátelům, kteří se ujali role korektorů, a všem dalším lidem, kteří mi pomáhali při opravách textu a testování praktické části.

© Ondřej Žižka, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1
1 Úvod....3
1.1 Cíle práce....3
1.1.1 Prozkoumání obvyklých řešení....3
1.1.2 Implementace prostého hledání do šířky (BFS)....3
1.1.3 Prozkoumání možností heuristiky a optimalizace....4
1.1.4 Průzkum možností aplikovat principy soft-computingu....4
1.2 Předpokládané znalosti čtenáře....4
1.3 Terminologie....4
2 Stávající systémy....5
2.1 Existující uživatelská rozhraní....5
2.1.1 Další možná rozšíření....6
2.2 Podrobnosti o systému IDOS....6
3 Analýza úlohy....8
3.1 Úloha z pohledu uživatele (motivace)....8
3.2 Úloha z pohledu programátora....8
3.3 Dopravní systém z pohledu cestujícího....9
3.4 Dopravní systém z pohledu programátora....9
3.4.1 Pojmy a terminologie....9
3.4.2 Vymezení pojmu „trasa“....10
3.5 Diagram entit jízdních řádů a jejich vztahů....11
3.6 Import dat....11
4 Tradiční řešení - hledání hrubou silou....15
4.1.1 Dijkstrův algoritmus....16
4.1.2 Floydův-Warshallův algoritmus....16
4.1.3 Optimalizace....17
4.2 Implementace....19
4.2.1 Relační databázový systém MySQL....19
4.2.2 Implementace vyhledávání hrubou silou – procedura mhd_VyhledejSpoje()....21
4.2.3 Princip práce procedury mhd_VyhledejSpoje()....21
4.3 Použití procedury mhd_VyhledejSpoje()....25
5 Optimalizace s využitím neuronových sítí pro heuristiku....27
5.1 Analýza úlohy neuronové sítě....27

5.1.1 Způsob a druh použité sítě....28
5.1.2 Architektura sítě....28
5.1.3 Způsob zakódování vstupů sítě....31
5.1.4 Způsob zakódování výstupů sítě....35
6 Implementace neuronové sítě....37
6.1 Relační model neuronové sítě....37
6.2 Vytvoření neuronové sítě....38
6.3 Výpočet neuronové sítě....40
6.4 Backpropagation....42
6.5 Učení neuronové sítě....44
6.5.1 Procedura mhd_nn_TeachStationNet()....45
6.6 Efektivita implementace....48
6.7 Úspěšnost učení....50
6.8 Integrace výsledků neuronové sítě do algoritmu vyhledávání spojů....52
7 Další možnosti optimalizace....53
7.1 Genetické algoritmy....53
7.1.1 Přímé využití pro hledání....53
7.1.2 Využití GA při trénování sítě pro heuristiku....55
7.2 Hopfieldova síť....56
7.3 Teorie mraveniště....56
8 Závěr....58
8.1 Výsledky práce....58
8.2 Další vývoj....59
Literatura....59
Seznam příloh....60

1 Úvod

Nedílnou součástí moderní společnosti je pravidelná přeprava osob. Za tím účelem existují přepravní systémy, jejichž spoje se řídí předem daným jízdním řádem.

Tato práce si klade za cíl prozkoumat možnosti automatického vyhledávání spojů z jednoho místa na jiné, takovéto vyhledávání implementovat, a dále prozkoumat možnosti urychlení vyhledávání v příliš rozsáhlých jízdních řádech pomocí metod umělé inteligence.

Práce také zkoumá možnosti hromadných výpočtů v procedurálních rozšířeních jazyka SQL a případné možnosti jejich paralelizace.

1.1 Cíle práce

Tato práce se zabývá úlohou vyhledávání spojů v databázi jízdních řádů a její časovou optimalizací. Nejprve prozkoumává současné teoretické poznatky o problému a následně ověří jejich praktickou aplikovatelnost v prostředí relační databáze a aplikačního serveru. Konkrétně bude práce postupovat následujícími kroky.

1.1.1 Prozkoumání obvyklých řešení

Obvyklým řešením je myšleno dnes nejspíše nejčastěji používané řešení, tedy vyhledávání do šířky bez použití metod soft-computingu. Za tímto účelem se také uskutečnila návštěva u výrobce největšího systému tohoto druhu v České republice, ve firmě Chaps, s.r.o. Schůzka s pracovníky oddělení jízdních řádů Dopravního podniku města Brna nás obohatila o znalosti praktických problémů, které se musí odrazit v návrhu informačního systému, zejména struktury entit. Díky návštěvě v Dopravním podniku města Českých Budějovic máme k dispozici data jízdních řádů, na kterých budeme naši implementaci testovat.

1.1.2 Implementace prostého hledání do šířky (BFS)

Dalším cílem práce je implementovat obvyklý způsob vyhledávání spojů v jízdních řádech. Zamýšlené prostředí pro implementaci je relační databázový systém, který podporuje ukládané procedury (např. MySQL, PostgreSQL nebo Oracle).

1.1.2.1 Výhody back-endu v SQL

Přesun algoritmické části úlohy do databázového systému umožňuje její efektivní programování v jazyce SQL, kde se kód úlohy redukuje z mnoha zdrojových souborů s implementací různých tříd

tvořících algoritmus na několik stránek kódu SQL příkazů. SQL jazyk totiž v podstatě popisuje operace nad množinami, na rozdíl od programovacích jazyků, ve kterých je tyto operace nutné většinou naprogramovat krok po kroku.

Výhoda ovšem spočívá i v lepší škálovatelnosti. U rozsáhlých jízdních řádů (např. spojený Londýnský jízdní řád veřejné dopravy a soukromých dopravců) by v případě implementace na úrovni aplikačního kódu mohlo dojít k dosažení limitu operační paměti a systém by musel neustále „zahazovat“ objekty načtené z perzistenční vrstvy (databáze) a načítat jiné. Při implementaci v SQL se o „kešování“ stará databázový systém.

1.1.3 Prozkoumání možností heuristiky a optimalizace

Výše zmiňovaná obvyklá cesta řešení úlohy umožňuje zapojit určité heuristické a obecně optimalizační postupy. Jedním z cílů práce je tyto postupy prozkoumat a pokusit se o jejich implementaci.

1.1.4 Průzkum možností aplikovat principy soft-computingu

Ačkoliv ve spojitosti s jízdními řády se principy soft-computingu využívají spíše pro jejich tvorbu, na úlohu vyhledávání se dají aplikovat také. S jakým výsledkem, to bude další oblast zkoumání této diplomové práce.

1.2 Předpokládané znalosti čtenáře

Pro plné porozumění praktické části práce se u čtenáře předpokládá zejména znalost databázových technologií založených na SQL, výhodou je potom znalost konkrétně MySQL 5.0 a „ukládaných procedur“ (stored procedures). Dále je vhodné znát základy teorie neuronových sítí.

1.3 Terminologie

V průběhu seznamování s problematikou jsem často narazil na termíny, které nemají vhodný český ekvivalent a používají se počeštěné anglické termíny. V takových případech nebudu termín překládat a používám jej v zažité podobě.

Některé pojmy jsou definované zákonem, konkrétně zákonem č. 111/1994 Sb. o silniční dopravě a jeho pozdějšími úpravami, § 2 odst. 3.

Další pojmy jsem převzal tak, jak se jejich používání ustálilo v organizacích uvedených v bodě 1.1.1. Jejich význam vysvětlujeme po krátkém teoretickém úvodu v bodě 3.4.1.

2 Stávající systémy

2.1 Existující uživatelská rozhraní

V České republice je stav jednoduchý – existuje jediný celostátní systém a tím je IDOS od firmy Chaps (někteří dopravci sice mají vlastní vyhledávací formuláře, které jsou ovšem jen front-endem pro IDOS). IDOS má velmi jednoduché uživatelské rozhraní:

V podstatě se jedná jen o nalezení cílové a výchozí zastávky podle názvu a vyhledání trasy pro uvedené datum a čas. Jedinou zajímavou vlastností je, že je možné hledat podle příjezdu do cílové zastávky, což se v praxi v algoritmu hledání nejspíše projeví „obrácením“ vyhledávacích kritérií – hledání bude probíhat od cílové zastávky k výchozí a hledat se bude nikoliv první následující průjezd trasy zastávkou, ale poslední předchozí. Omezení přestupů vypadá nevinně, ale v podstatě určuje nejvyšší počet průchodů vyhledávacím algoritmem (viz naše implementace v části 4.2.2). Systém dále umožňuje omezené prohledávání databáze v dalších záložkách a základní personalizaci – nastavení zkratek k předvyplněným údajům.

V rozhovorech s tvůrci systému vyplynulo, že v minulosti zkoušeli systém rozšířit o různé doplňkové služby či vlastnosti, ale odezva ze strany uživatelů byla minimální, tudíž z obchodního hlediska nevýhodná, a vrátili se k původnímu jednoduchému formuláři.

Proto se podívejme do zahraničí. Velmi propracované uživatelské rozhraní mají německé Deutsche Bahn na adrese http://reiseauskunft.bahn.de/bin/query.exe/en. Umožňují vyhledání zpáteční jízdy, počet cestujících a další podrobnosti. Po vyhledání umožňují rezervaci a zaplacení lístku, případně místenky. Z URL je vidět, že jádrem systému je aplikace pro operační systém Windows.

Uživatelské rozhraní pro samotné vyhledávání spojů příliš rozšiřovat nejde – konec konců, vždy jde jen o vyhledání vhodného spoje odněkud někam v určitý čas. Správnou cestou je integrace

s jinými službami a aktivitami – tak, jak se to daří například rakouskému systému Scotty, který nabízí hledání spojení nejen podle zastávky, ale i podle adresy. To evidentně zapojuje geografický systém, pravděpodobně ve spolupráci s jinou společností.

Úplně nejdále v nápaditém využití vyhledávacího systému jde londýnský dopravní server Transport for London (www.tfl.gov.uk). V pokročilém vyhledávacím formuláři nabízí tyto možnosti:

- Umožňuje hledat spojení nejen ze zastávky, ale také podle adresy, PSČ, nebo podle významných míst.
- Nabízí vyhledání nejen nejrychlejších, ale také nejlevnějších spojení a také takových, u kterých bude potřeba co nejméně chodit pěšky.
- Umožňuje filtrovat spojení podle dostupnosti zdravotně postiženým (nutnost použít schody, eskalátory, výtahy, bezbariérový přístup).
- Můžete také specifikovat, jak moc dobří jste chodci – rychlost, maximální doba chůze, a jestli chcete pěší přesun upřednostnit, pokud to urychlí celkovou cestu.
- A asi nejzajímavější: umí naplánovat kompletní cyklistický výlet: vyhledá spojení na místo, cyklistickou trasu podle doby, jak dlouho se chcete projíždět, a spojení zpět; navíc jde zvolit, jestli chcete kolo nechat ve stanici, naložit do vozidla, nebo cestovat jen na kole.

2.1.1 Další možná rozšíření

Londýnský systém částečně rozvíjí směr, kterým se dá využit vyhledávání spojů v dopravních systémech: Od prostého vyhledávání k plánování programu. Systému zadáte, co chcete dělat, jakou dobu se chcete této činnosti věnovat, koho chcete vzít s sebou, kolik chcete utratit, a systém vám sestaví program se spojením. Nejednalo by se tedy už jen o vyhledávání v dopravním systému, ale o jakýsi všeobecný integrovaný vyhledávací systém.

Vyhledávání by mohlo být napojené na sociální síť, databázi firem, databázi aktuálních kulturních a sportovních akcí, a další podobné služby. Sociální síť by vzala v potaz program vašich přátel, případně jim zaslala návrh na program. Databáze kulturních akcí by poskytla informace o akcích konajících se v danou dobu jako cíl cesty, a podle databáze firem by systém doporučil vhodné restaurační zařízení na cestě zpět. Návrh vlastností takového systému již ale jde daleko mimo téma této práce.

2.2 Podrobnosti o systému IDOS

IDOS (zkratka „Informační dopravní systém“, nebo také španělsky „odejděte“) je informační systém vyvíjený Brněnskou firmou Chaps, s.r.o. Jeho hlavním účelem je vyhledávat spojení v jízdních řádech. Nad jedním jádrem a stejnou databází běží mnoho uživatelských rozhraní – dnes je

nejznámější asi webové rozhraní na adrese jizdnirady.idnes.cz; v době psaní práce byl zahájen testovací provoz nového rozhraní na adrese beta.jizdnirady.idnes.cz. Dále existuje WAP rozhraní, SMS služba, webová služba, programy pro Windows, programy pro SmartPhone. Se systémem IDOS jsou dále spjaté další produkty – viz web firmy Chaps. Na základě návštěv ve firmě IDOS víme o systému IDOS trochu více.

Jeho jádrem je obecná knihovna `tt.dll`, vyvíjená již od roku 1994. Jejíž schopnosti a aplikační rozhraní jsou prý natolik obecné, že dokáže zvládnout prakticky jakoukoliv úlohu týkající se vyhledávání spojů. Počítá s tím, že jsou data jízdních řádů v paměti počítače. Na současných počítačích je natolik výkonná, že všechna on-line rozhraní systému IDOS obsluhují tři servery. Knihovna je vnitřně vysoce optimalizovaná, ale jen na programové úrovni a nepoužívá žádnou formu umělé inteligence.

Nad knihovnou `tt.dll` bývalo rozhraní TTCOM napsané v jazyce Visual Basic, využívající technologii Microsoft COM [26]. To se časem ukázalo jako nevyhovující řešení a nad vrstvou TTCOM vznikla vrstva TTWS, která rozhraní vystavuje v podobě webové služby. Všechna rozhraní systému IDOS postupně přecházejí na používání této služby, a až mezivrstvu TTCOM nebude potřebovat žádné rozhraní, bude odstraněna.

Data systém IDOS čerpá především z *Celostátního informačního systému o jízdních řádech* (CIS JŘ), kam ze zákona musí povinně posílat svoje jízdní řády v elektronické podobě ve formátu předepsaném firmou Chaps všichni dopravci provozující pravidelné spoje pro osobní přepravu (postupování jízdního řádu do CIS JŘ ukládá § 17 odst. 2 zákona č. 111/1994 Sb). K datům má však s posvěcením Ministerstva dopravy monopolní přístup firma Chaps. V zákoně je sice stanovena povinnost data zveřejňovat, ale s tím se firma Chaps vypořádala tak, že zveřejnila jízdní řády ve formátu PDF, tedy tak, aby data nebylo možné zpracovat strojově a vytvořit si kopii databáze, a firma si tak snáze mohla udržet své postavení na trhu.

Data jízdních řádů dodávaná jednotlivými dopravci se mění překvapivě často. Například pražské jízdní řády se mění každý den. Pokud například onemocní řidič okrajové nedůležité linky, dopravní podnik změní na dva dny jízdní řád. V systému IDOS ale každá změna jízdních řádů vyžaduje provedení různých operací (optimalizace, indexace), proto je aktualizace CIS JŘ omezená, nejčastěji ji dopravci mohou provádět třikrát týdně.

Další informace získané ve firmě Chaps nelze stručně obsáhnout v několika odstavcích. Jedná se hlavně o praktické zkušenosti s modelováním jízdních řádů, které se promítly v implementaci vyhledávání.

3 Analýza úlohy

3.1 Úloha z pohledu uživatele (motivace)

Z pohledu uživatele vypadá úloha celkem jednoduše:

Hledáme dopravní spoj ze zastávky *A* do zastávky *B* v okamžiku *T*. Očekáváme, že dostaneme několik nejrychlejších spojů vyjíždějících ze zastávky *A* od okamžiku *T* dále. Alternativně spoje, které dorazí do zastávky *B* nejpozději do okamžiku *T*. Okamžik *T* se dá stanovit na jakýkoliv bod v čase, tedy včetně data.

Nejraději bychom pochopitelně byli, kdybychom se do cílové zastávky dostali bez nepohodlného přestupování, ale pokud to ušetří hodně času, chceme raději nalézt spoj s přestupem. Nechce se nám ale přestupovat mnohokrát, chceme mít možnost počet přestupů omezit.

Rádi se během dlouhé cesty protáhneme a projdeme, chceme tedy možnost vybrat si typ dopravních prostředků.

Ve výpisu spojů musí být samozřejmě všechny časy nástupů a výstupů a názvy / čísla spojů. V ideálním případě bychom se měli dozvědět i cenu a uvítáme i další detaily typu „nízkopodlažní vůz“, „přeprava kol“ a podobně.

3.2 Úloha z pohledu programátora

Při úvahách o úloze jsem si vzpomněl na svého kamaráda Karla Kyriana, který svoji odpověď skoro na jakoukoliv otázku ohledně programování začíná slovy „*No tak to je jasný, ne?*“. I tato úloha se na první pohled jeví jako jednoduchá a připomíná příklady z hodin diskrétní matematiky; po hlubší analýze ale zjistíme, že to zas až tak „jasný“ není.

Vágní zadání z předchozího bodu se vejde na několik odstavců, ale implementace rozhodně není triviální. První, co programátora napadne, je, že se jedná o grafovou úlohu, kde zastávky jsou uzly grafu a spoje jsou jeho hrany, jejichž váhy jsou doby jízdy spojů. Hrany jsou orientované (spoj jede z místa *A* do místa *B*, ale nemusí jet zároveň i další spoj naproti). Jedná se o multigraf, protože mezi zastávkami určitě jede více spojů. Hrany i vrcholy mají další atributy, které mají vliv na požadovaný výstup (a to bude kamenem úrazu pro *hurá-řešení*). Další vlastnosti grafu záleží na konkrétní síti dopravních spojů.

Jakmile si tedy programátor uvědomí „grafovitost“ úlohy, začne uvažovat o použití některého z algoritmů hledání nejkratší cesty v grafu – například Dijkstrův, Floydův-Warshallův či A*. Ve svých úvahách ale zjistí, že při aplikaci základní podoby těchto algoritmů naráží na jejich přílišnou jednoduchost v tom smyslu, že mají málo parametrů. Bude si je tedy muset poupravit.

3.3 Dopravní systém z pohledu cestujícího

Cestující je osoba relativně nevědomá. Vezměme si za příklad třeba městskou hromadnou dopravu v Brně. Cestující ví, že chce jet ze zastávky Tylova na Fakultu informačních technologií na obhajobu této diplomové práce, a ze zkušenosti ví, že má jet linkou číslo 1, protože žádná jiná linka touto trasou nejezdí. Jenže najednou přijede linka číslo 8. Jedná se o výjimku, protože ačkoliv linka číslo 8 ve své „oficiální“ trase zastávku Tylova nemá, tato konkrétní tramvaj na ní zastavuje, protože jede z vozovny a když už tudy jednou jede, proč nevzít cestující.

3.4 Dopravní systém z pohledu programátora

Předchozí situace ovšem pro programátora není výjimkou. Systém vyhledávání spojů, který programuje, musí podchytit i takové situace a s daným spojem počítat. V algoritmických úlohách však nelze dělat výjimky – a proto je třeba algoritmus vyhledávání a datový model upravit tak, aby se výjimka stala jen jednou z více běžných možností.

A zde se dostáváme k faktu, že pro cestujícího je dopravní systém zjednodušen na minimum. Linka 8 je zkrátka linka 8, i když zrovna projíždí zastávkou Tylova, kde nemá co dělat. Programátor si však uvědomí, že „linka 8“ je jen mlhavé označení a v datovém modelu si musí zavést jiné pojmy.

3.4.1 Pojmy a terminologie

V následující tabulce definic jsou dopravní termíny kvůli jednoznačnosti podtržené.

Tabulka 3.1: Pojmy používané v textu

Linka	Mlhavé označení pro potřeby zjednodušení dopravního systému pro cestující. V podstatě říká, že dopravní prostředek (jízda) označený číslem (případně názvem) linky jede víceméně po trase obvyklé pro danou linku, a to v jednom ze dvou směrů.
Směr	Mlhavé rozlišení pro potřeby zjednodušení dopravního systému pro cestující. U linek, které jezdí z konečné zastávky *A* do konečné zastávky *B* a také z *B* do *A*, představuje směr rozlišení, odkud kam daný dopravní prostředek (jízda) jede.

Zastávka	Význam termínu se prakticky shoduje s intuitivním chápáním pojmu zastávka. Jedná se o označené fyzické místo, kde se nacházejí jednotlivé uzly tras a kde jednotlivé jízdy provádějí stání.
Výchozí / konečná z.	Opět se shoduje s běžným chápáním. Je to první / poslední zastávka na trase.
Trasa	Uspořádaný sled uzlů tras. Je velmi důležité, že se jedná o multimnožinu zastávek, protože jedna zastávka může být na trase vícekrát – bude to mít dopad na složitost implementace. Termín má také spojitost s pojmem linka – linka je totiž výběr tras.
Uzel trasy	Výskyt zastávky na trase. Dva různé uzly stejné trasy mohou představovat zastavení ve stejné zastávce.
Jízda	Jízda je „instance“ trasy. Dá se chápat jako jízda dopravního prostředku po určité trase od výchozí zastávky do konečné.
Stání	Stání je „instance“ zastávky. Pokud jede určitá jízda po trase, tak na jejích zastávkách „provádí“ stání. V reálném světě to jednoduše znamená, že dopravní prostředek v určitém čase zastaví v nějaké zastávce, cestující nastoupil vystoupil a prostředek jede po trase dál.
Spoj	Běžně se pojem „spoj“ používá ve stejném smyslu, jaký má v této práci pojem jízda. My budeme pod pojmem spoj myslet možnost, jak se pomocí dopravního systému dostat z místa *A* na místo *B*. To může zahrnovat použití více dopravních prostředků s přestupy.

Jak bylo zmíněno výše, některé pojmy jsou definované zákonem, konkrétně zákonem č. 111/1994 Sb. o silniční dopravě a jeho pozdějšími úpravami, § 2 odst. 3.

3.4.2 Vymezení pojmu „trasa“

Když jsme si definovali pojmy, můžeme se na dopravní systém podívat s trochu odbornějším komentářem a pojmy si více vysvětlit.

Už tedy víme, že pojem linka neznamená skoro nic. Když tedy jede například linka 1 z Řečkovic do Bystrce (to jsou okrajové části Brna), projede sledem zastávek, tedy všemi zastávkami trasy. V Bystrci se na smyčce otočí a vyjede na další jízdu – tentokrát ale po *jiné* trase. Na noc řidič nenechá čekat tramvaj na smyčce, ale zaveze ji do vozovny. Proto na zastávce Semilasso cestujícím automat oznámí, že „Vůz – jede do vozovny… Medlánky.“ Pro cestujícího je to stále ta samá linka 1, ale z hlediska informačního systému je to pokaždé jiná trasa.

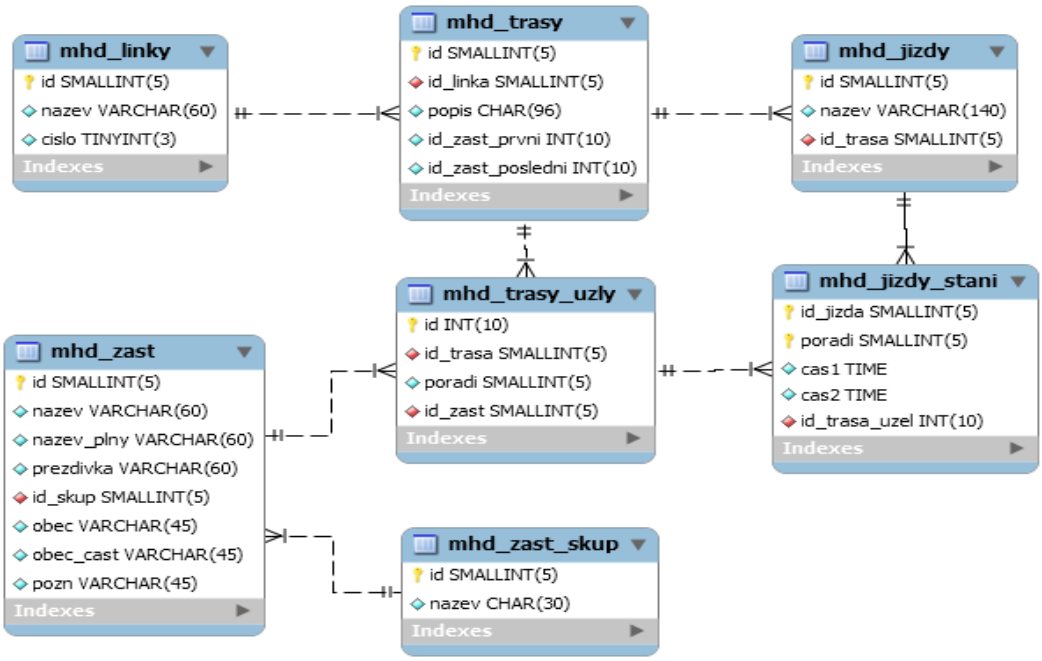
Jakákoliv sebemenší odchylka ve sledu zastávek dává vzniknout nové trase. Pokud tedy například autobus 64 někdy zajíždí k Nákupnímu centru Královo pole, je to jiná trasa, než když tuto

zastávku vynechává. Ale dokonce i když trasa fyzicky zastávkou projíždí, jako v případě linky č. 1 zastávkou Janáčkovo divadlo, ale obsluhuje ji jen v určité době, jedná se o různé trasy.

Jízda je jednotlivá „instance“ trasy. Pokud tedy na trasu Řečkovice – Bystrc vyjíždí tramvaj v 14:10, a ve 14:20 další, jedná se o dvě různé jízdy po stejné trase. Obě tramvaje zastaví na stejné zastávce Semilasso, ovšem jedná se o jiná stání.

3.5 Diagram entit jízdních řádů a jejich vztahů

Když jsme tedy vyjasnili pojmy, můžeme si vše přehledně shrnout na ER-diagramu na obrázku 3.1.



Obrázek 3.1 – ER-diagram základních entit systému.

Jak je vidět, entity jsou poměrně složitě provázané a vznikají cyklické vazby. A to jsou teprve základní entity bez entit pro rozlišení různých dnů, různých typů dopravních prostředků a dalších detailů.

Tato struktura vznikla podle vstupních dat získaných z databáze reálných jízdních řádů. Postup importu dat popisuje následující část.

3.6 Import dat

Pro potřeby testování dosud vyvinutých postupů máme k dispozici starší data jízdních řádů Dopravního podniku města České Budějovice. Data jízdního řádu tvoří zejména tyto soubory:

```
CASKODY.TXT
LINKY.TXT
PEVNYKOD.TXT
SPOJE.TXT
UDAJE.TXT
ZASLINKY.TXT
ZASSPOJE.TXT
ZASTAVKY.TXT
```

Jsou ve formě obyčejných textových souborů s tabulkovými daty oddělenými čárkami – formát CSV (*comma separated values*, čárkami oddělené hodnoty)[10]. Z tohoto formátu se dále vyvinul formát JDF, formálně standardizovaný Ministerstvem dopravy České republiky podle požadavků firmy Chaps, spol. s r. o. [4]:

> „Dopravce je podle /.../ vyhlášky Ministerstva dopravy a spojů č. 388/2000 Sb., o jízdních řádech veřejné linkové osobní dopravy /.../ povinen schválený jízdní řád /.../ postoupit v elektronické podobě ve formátu a struktuře dat stanovenými Ministerstvem dopravy do celostátního informačního systému o jízdních řádech.“

Nejvíce ze všech nás zajímá soubor `Zasspoje.txt`. Ten je zhruba v následujícím formátu:

```
"000009","1","1","530","2","","","0","","0411";
"000009","1","2","115","2","","","0","","0412";
"000009","1","3","72","2","","","1","","0414";
"000009","1","4","488","2","","","2","","0415";
"000009","1","5","74","2","","","3","","0417";
"000009","1","6","453","2","","","3","","0418";
"000009","1","7","37","2","","","3","","0419";
...
```

Soubor `Zasspoje.txt` obsahuje pro každý spoj a každou zastávku linky jeden záznam. Význam sloupců je následující:

Tabulka 3.2.Význam sloupců v souboru `Zasspoje.txt`

Sloupec 1	Číslo linky. Zde nezáleží na trase, trasa je daná sledem zastávek.
Sloupec 2	Číslo jízdy. Lichá čísla jsou jízdy v jednom směru, sudá ve směru opačném. V Českých Budějovicích okružní linky nejsou, takže toto pravidlo platí u všech linek.
Sloupec 3	Číslo uzlu trasy. Rostoucí sekvence – proto můžeme říci, že uzly s vyšším číslem z tohoto sloupce jsou na trase dále.
Sloupec 4	Číslo zastávky daného uzlu.
Sloupec 5	Číslo stanoviště („sloupek“).
Sloupce 6 a 7	Pevné kódy; cizí klíč do číselníku `pevnykod.txt`
Sloupec 8	Počet kilometrů od začátku trasy.
Sloupec 9	Čas příjezdu do zastávky ve formátu `HH:MM`, nebo znak < nebo `|`. Povinný v konečné zastávce.

Sloupec 10	Čas odjezdu ze zastávky ve formátu `HH:MM`, nebo znak < nebo `|`.. Nepovinný v konečné zastávce.

Jednoznačnost záznamu je určena číslem linky, číslem jízdy a tarifním číslem zastávky. To znamená, že zastávky spojů jsou seřazeny vždy podle zastávek linky, a proto časové a km údaje jsou pro směr zpět (jsou-li záznamy pro spoj setříděny podle tarifních čísel zastávky) uvedeny v opačném pořadí, t.j od cílové stanice do výchozí.

Pro každou zastávku spoje jsou uvedeny časy odjezdu vyjma případu, kdy spoj zastávkou projíždí (čas odjezdu obsahuje znak "|") nebo jede jiným směrem (čas odjezdu obsahuje znak "<"). Je-li uvedeno, že spoj jede jiným směrem, jsou km údaje prázdné. V případě, že spoj stojí v zastávce déle než 5 minut, musí být uveden i čas příjezdu [4].

Sloupce 9 a 10 mohou obsahovat místo času hodnotu „<". Ta znamená, že spoj touto zastávkou neprojíždí. Nutnost tohoto značení je důsledkem použití koncepce linek i pro definici dat. Pokud tedy trasa může vést z konečných zastávek *A1* nebo *A2*, bude v tomto souboru uvedena vždy stejná sekvence zastávek, ale u těch, kterou daná trasa neprojíždí, bude znak „<".

Dále nás zajímá soubor `zastavky.txt`, který obsahuje data o zastávkách dané linky. Logičtější by sice bylo tento soubor vytvořit jediný pro celý dopravní systém, naštěstí se ale čísla totožných zastávek u všech tras shodují.

```
"10","České Budějovice","","Žižkova - PVT","CB","CZ","","","",...;
"11","České Budějovice","","Nádraží","CB","CZ","00002","","",...;
"37","České Budějovice","","Družba","CB","CZ","00001","","",...;
...
```

Význam sloupců je po řadě: číslo zastávky, oblast, obec, jméno zastávky, ID dopravního systému, ID státu dopravního systému, příznaky ze souboru `pevnykod.txt`. Vazba je realizována přes číslo zastávky (ze souborů `Zasspoje.txt` a `Zaslinky.txt`).

V souboru `caskody.txt` jsou informace o tom, kdy která část jízdního řádu platí. Jeho obsahem se zatím nebudeme zabývat.

Soubor `spoje.txt` obsahuje spoje a jejich příznaky s vazbou na soubor `pevnykod.txt`. V následující ukázce spoje 5 a 7 jezdí jen v pracovní den, a spoj 9 jen o víkendu a ve svátky.

```
"000009","5","00003","","","","","","","","","";
"000009","7","00003","","","","","","","","","";
"000009","9","00004","00005","","","","","","","","";
```

Samotný import dat je poměrně složitá procedura. Ve stručnosti vypadají kroky převodu takto:

1. Do SQL databáze převedeme jednotlivé soubory – každému podstatnému souboru vytvoříme tabulku a naplníme ji jeho daty.
2. Data vyčistíme od nepotřebných záznamů a doplníme chybějící zastávky.
3. Pro každou linku začneme vyhledávat stejné trasy.

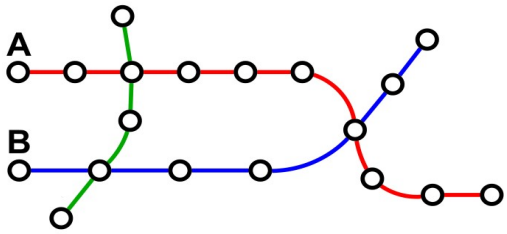
4. Pro každou nalezenou trasu vytvoříme záznam v nové tabulce tras.
5. Následně všem jízdám přiřadíme ID příslušné trasy, vzniklé v předchozím kroku.

Podrobné vysvětlení importu by zabralo několik stran. Proto zájemce odkazuji na okomentovaný zdrojový kód v příloze.

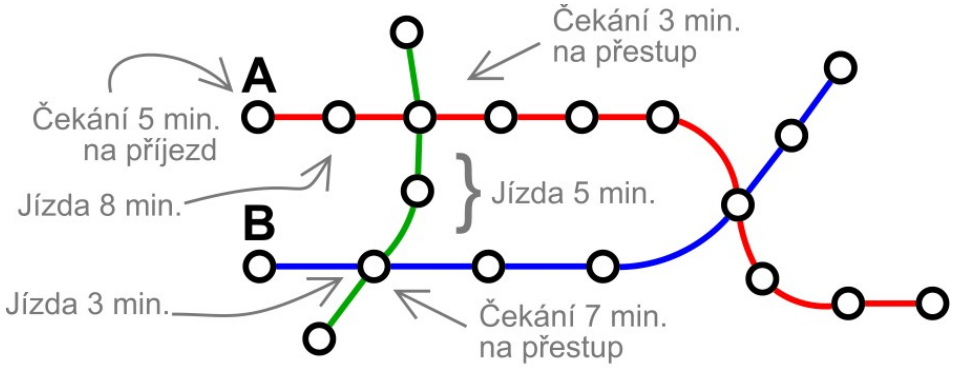
4 Tradiční řešení - hledání hrubou silou

Když tedy máme připravená testovací data, ve kterých můžeme vyhledávat, přejděme k implementaci samotného hledání. Nejprve se zaměříme na tradiční prosté hledání hrubou silou – tedy procházení grafu do šířky.

Jak jsme již zmínili, úloha má do značné míry charakter vyhledávání optimální cesty grafem. Síť dopravního systému si můžeme zobrazit jako graf:



Představme si tedy situaci, kdy se chceme dostat z vrcholu *A* do bodu *B*. V případě jednoduchého hledání nejkratší cesty grafem bychom potřebovali mít ohodnocené hrany. Ovšem my takové hodnocení nemáme. Místo toho atributy hran a vrcholů vypadají takto:



V tomto případě nejprve čekáme 5 minut v uzlu *A* na příjezd, pak jedeme 8 minut o dva uzly dále, atd. Ale pokud bychom například jeli až do uzlu, kde se střetá cesta s uzlem *A* s cestou s uzlem *B*, hodnoty by mohly vypadat jinak: Například do posledního úseku před uzlem *B* bychom se dostali pravděpodobně pozdější jízdou, která může mít v jízdním řádu pro tento úsek vyhrazené 4 minuty (třeba kvůli dopravní špičce). Ceny hran se tedy liší v závislosti na předchozím průběhu prohledávání grafu.

Tato vlastnost grafu přináší komplikace při aplikaci obyčejných algoritmů hledání nejkratší cesty grafem a vyžaduje jejich úpravy. Nejprve se na tyto algoritmy podívejme.

4.1.1 Dijkstrův algoritmus

Takto je Dijkstrův algoritmus popsán v [6]:

Mějme graf G, v němž hledáme nejkratší cestu. Řekněme, že V je množina všech vrcholů grafu G a množina E obsahuje všechny hrany grafu G. Algoritmus pracuje tak, že si pro každý vrchol v z V pamatuje délku nejkratší cesty, kterou se k němu dá dostat. Označme tuto hodnotu jako $d[v]$. Na začátku mají všechny vrcholy v hodnotu $d[v] = \infty$, kromě počátečního vrcholu s, který má $d[s] = 0$. Nekonečno symbolizuje, že neznáme cestu k vrcholu.

Dále si algoritmus udržuje množiny Z a N, kde Z obsahuje už navštívené vrcholy a N dosud nenavštívené. Algoritmus pracuje v cyklu tak dlouho, dokud N není prázdná. V každém průchodu cyklu se přidá jeden vrchol $vmin$ z N do Z, a to takový, který má nejmenší hodnotu $d[v]$ ze všech vrcholů v z N.

Pro každý vrchol u, do kterého vede hrana (označme její délku jako $l(vmin, u)$) z $vmin$, se provede následující operace: pokud $d[vmin] + l(vmin, u) < d[u]$, pak do $d[u]$ přiřaď hodnotu $d[vmin] + l(vmin, u)$, jinak neprováděj nic.

Až algoritmus skončí, potom pro každý vrchol v z V je délka jeho nejkratší cesty od počátečního vrcholu s uložena v $d[v]$.

Jak je vidět, algoritmus počítá s pevně danou délkou hrany. V našem případě ale délku hrany předem neznáme, protože je závislá na okamžiku, pro který z daného uzlu (zastávky) hledáme spoj. Kromě toho navíc nemá obdobu čekání na zastávce na daný spoj. Algoritmus je tedy vhodný například pro silniční síť, kde lze cestovat libovolně a kdykoliv a na uzlech (křižovatkách) nečekáme. Princip algoritmu se nám ale bude hodit – trochu si jej budeme muset upravit.

4.1.2 Floydův-Warshallův algoritmus

Dalším inspirujícím algoritmem pro nás bude Floydův-Warshallův algoritmus. Pokud bychom opět řešili např. GPS navigaci a hledali spoje pro silnice, byl by tento algoritmus dokonce ještě lepší než Dijkstrův – snadněji se programuje a nalezne nejkratší cestu nejen z bodu A do bodu B, ale dokonce nejkratší cesty mezi všemi vrcholy.

Vrcholy grafu očíslujeme čísly od jedničky do n. Všechny dvojice vzdáleností si nejsnáze uložíme do matice $n \times n$. Celý trik Floyd-Warshallova algoritmu spočívá v tom, že vzdálenosti nepočítáme přímo, ale v n iteracích. V i-té iteraci spočítáme matici D_i.

Hodnota $D_i[u, v]$ je délka nejkratší cesty z u do v, která smí procházet pouze přes vrcholy $\{1, 2, . . . , i\}$. Jinými slovy, $D_i[u, v]$ je délka nejkratší cesty v podgrafu indukovaném vrcholy $\{1, 2, . . . , i\}$. V nulté iteraci začneme s maticí D_0. Hodnota $D_0[u, v]$ je délka hrany uv, pokud z u vede hrana do v, a nekonečno jinak. Matice D_0 v podstatě odpovídá upravené matici sousednosti, která místo jedniček obsahuje délky hran a místo nul nekonečna. V poslední iteraci skončíme s maticí D_n, která už bude obsahovat hledané vzdálenosti, protože cesty mezi u a v smí procházet přes všechny vrcholy.

Jenže ani tento algoritmus bez úprav použít nemůžeme. Hrany jsou sice již orientované, ale opět se nepočítá se závislostí délky hrany na okamžiku, kdy ji zjišťujeme. Podcesty v grafu nalezené při předchozím procházení do šířky si totiž ukládá do svého druhu „cache“ (tedy do matice D_i) a pro cestu o půl třetí z Českých Budějovic do Brna by klidně použil spoj v sedm večer z Jihlavy do Náměšti nad Oslavou. Kromě toho se opět nepočítá s prodlevou při čekání na spoj.

4.1.3 Optimalizace

Metody optimalizace algoritmického hledání nejkratší cesty spočívá zejména v ovlivňování pořadí, v jakých jsou při prohledávání grafu vybírány uzly. Běžně se používají tyto postupy:

4.1.3.1 Obousměrné prohledávání

Graf se prohledává paralelně ve dvou směrech na dvou grafech se stejnou množinou vrcholů – jednak od výchozího uzlu A na grafu G, a jednak od cílového uzlu B na grafu G', jehož hrany jsou „obrácené“ hrany grafu G – tedy $E' = \{(u, v) \mid (v, u) \in E\}$. Ve chvíli, kdy algoritmus nějaký uzel $v \in V$ označí za stabilní v obou paralelních procesech, lze najít nejkratší cestu přes uzel $u \in V$ mezi A a B jako spojení nejkratších cest z A do u a z B do u.

Tento postup je ovšem pro potřeby vyhledávání spojů v jízdních řádech nepoužitelný, jelikož v cílovém uzlu bychom museli nejprve určit čas, od kterého vyhledáváme (v čase směrem zpět).

4.1.3.2 Hledání vstříc cíli neboli algoritmus A*

Tato technika, prvně popsaná v [1], upravuje prioritu výběru uzlů k procházení Dijkstrovým algoritmem tak, že k prioritě uzlu přidávají tzv. potenciál, který závisí na jeho (geografické) vzdálenosti od cílového uzlu – čím blíže, tím vyšší potenciál a tedy pravděpodobnost výběru. Tím se sníží doba běhu algoritmu, a algoritmus stále vrací nejkratší cestu.

Tento postup je vhodný pro vyhledávání např. v silniční síti; pro vyhledávání spojů v jízdních řádech jeho použití nepřináší dobré výsledky, a to opět z toho důvodu, že předem neznáme

ohodnocení hran – cesta spojem, který jede „oklikou“ a s přestupem, může vyjít časově lépe než přímý spoj, který ovšem jede za dlouho.

Navíc, jelikož v naší implementaci pomocí SQL se uzly přidávají po celých množinách (všechny zastávky na určité trase od nynější dále), zjišťování vzdálenosti všech uzlů od cílového a jejich následné řazení a podmínečný výběr do dalšího cyklu prohledávání by mohlo zvýšit režii SQL dotazů natolik, že by se již optimalizace nemusela vyplatit. Princip algoritmu si ovšem upravíme tak, že zastávky tras, po nichž pravděpodobně optimální cesta nevede, ponecháme v „meziskladu zastávek“ a prohledávání z nich dále do grafu pozdržíme o jedno či více kol.

4.1.3.3 Metoda založené na hierarchii

Některé metody optimalizace vycházejí z výsledků předem provedené analýzy dat. Hierarchická metoda spočívá v tom, že hranám grafu přiřadíme atribut úrovně v pomyslné hierarchii – například rozdělení silnic na dálnice a rychlostní komunikace, okresní silnice a vedlejší silnice. Dále se graf rozdělí na oblasti tak, aby co nejméně hran spojovalo vrcholy v různých oblastech. Předvýpočet je provedený tak, že nejvýhodnější cesta mezi uzly, které jsou propojené hranami určité úrovně hierarchie, vede pouze po hranách stejné nebo vyšší úrovně – při hledání cesty grafem tedy můžeme hrany nižší úrovně (ideálně tedy většinu hran) zanedbat. Představme si například státy a hraniční přechody propojené dálnicemi. Při cestě ze španělské Valencie do Brna do Králova Pole nás tak algoritmus navede do Barcelony a odtud po dálnici až do Brna, aniž by se zabýval každou polní cestou v Rakousku. Při dosažení prvního vrcholu v oblasti Brno se algoritmus přepne na nižší úroveň a pro dohledání cesty do Králova pole již použije všechny cesty v grafu.

Další metoda využívající hierarchii funguje podobně, s tím rozdílem, že hrany vyšší úrovně si do grafu dotváříme. Předem vybereme klíčové uzly (nejlépe s největší konektivitou – např. dopravní uzly), vyhledáme nejvýhodnější cesty mezi nimi a vložíme je do grafu s atributem, který odliší takto uměle vytvořené hrany. Při algoritmickém hledání s nimi ovšem zacházíme transparentně - stejně jako s obyčejnými hranami. Metodu lze kombinovat s předchozí.

Opět je třeba mít na paměti, že zacházíme s „vrstvovým“ grafem. Jedna nalezená nejkratší cesta mezi vybranými uzly tedy platí pouze pro určitý čas, a pro čas o minutu později již opět optimalizaci nemáme. Nemusíme ovšem vyhledávat cesty pro všechny časy – ostatně, mohlo by to být i kontraproduktivní kvůli zvětšení objemu dat a indexů v databázi. Stačí tedy optimalizovat například pro časy, pro které systém vyhledává spoje nejčastěji.

4.2 Implementace

Jako všechny algoritmické části práce je i vyhledávání spojů implementováno jako sada procedur v relačním databázovém systému MySQL 5.0. Nejprve tedy něco o tomto systému.

4.2.1 Relační databázový systém MySQL

MySQL je opensource RDMBS vzniklý kolem roku 1993 s motivací vytvořit rychle pracující databázový systém bez zátěže plynoucí z historického vývoje. Je založen víceméně na standardech SQL [11]. V říjnu 2005 oficiálně vyšla verze 5.0, která podporuje tzv. ukládané procedury.

K efektivní implementaci v SQL bylo zapotřebí využít i další koncepty, které MySQL nabízí, ale nejsou ve standardech SQL nebo jsou vlastním rozšířením MySQL. Pro lepší porozumění si je stručně popíšeme.

4.2.1.1 Ukládané procedury

Termínem ukládané procedury (angl. stored procedures) se označuje mechanismus ukládání sekvence SQL příkazů vložených do základní programové logiky (větvení programu, cykly, volání procedur a funkcí navzájem, atd.). Jazyk SQL sám o sobě se dělí na příkazy pro manipulaci s daty (DML – Data Manipulation Language) a příkazy pro definici struktury dat (DDL – Data Definition Language). V ukládaných procedurách lze použít všechny příkazy obou skupin, které MySQL podporuje i pro samostatné zpracování [12].

Používání ukládaných procedur s sebou nese mnoho výhod. Někdy klient potřebuje určitou informaci, k jejímuž získání ovšem potřebuje zpracovat mnoho dat z databáze. Data je tedy nutné přenést ze serveru, na straně klienta zpracovat, zažádat o nová data, a tak proces pokračuje.

Pokud algoritmus zpracování přeneseme na server a klient volá pouze aplikační rozhraní, kterému předává parametry, získáme následující výhody:

- Menší zatížení přenosového pásma pro komunikaci mezi klientem a serverem
- Lepší využití možností hardwaru a softwaru
- Lepší zabezpečení
- Nižší náklady na vývoj a distribuci, vyšší spolehlivost (známe cílovou platformu)
- Centralizované zabezpečení, správa a údržba společných rutin

Některé databázové systémy (např. DB2 od IBM) navíc umožňují volání procedur napsaných v jiných jazycích (např. Java); MySQL se mezi ně zatím neřadí, i když prototypy pro verzi 6.0 již fungují.

4.2.1.2 Self-JOIN

Termín „self-JOIN“ označuje napojení tabulky na sebe sama. Vhodnou volbou spojovacích pomínek a hodnot vypočítaných do výsledné množiny záznamů lze velmi efektně (a často i efektivně) řešit velmi složité úlohy typu procházení stavovým prostorem pomocí jediného dotazu `SELECT`. O síle self-joinu vypovídá například možnost v jediném `SELECT` dotazu vyřešit sudoku [13].

4.2.1.3 Proměnné

Nejen v procedurách, ale i v dotazech `SELECT` lze použít proměnné, se kterými MySQL v průběhu dotazu pracuje. Lze tak spočítat hodnotu v jedné řádce a použít ji v některé z dalších. Například následující dotaz očísluje vrácené řádky od 1 výše:

```
SET @pos = 0;
SELECT @pos := @pos + 1 AS pozice, t.* FROM table1 AS t;
```

S využitím této vlastnosti se dají v jediném dotazu řešit úlohy, které by jinak vyžadovaly procedurální zpracování.

4.2.1.4 Logování, funkce pro práci s řetězci

V kódu se hojně vyskytují volání procedury `LoggP()` a odvozených:

```
CALL LoggP('mhd_VyhledejSpoje',F4('mhd_VyhledejSpoje( From #{1} to #{2}
  at {3}, max {4} rounds )', idZastFrom, idZastTo, whn,  iMaxRounds));
```

Tato volání slouží k zaznamenávání procesu pro účely ladění aplikace.

Jejich parametr tvoří obvykle funkce `F1()` až `F10()`. Ty jsou jednoduchou obdobou funkce `sprintf()` a zkracují zápis.

4.2.1.5 Názvové konvence

Jelikož MySQL nenabízí žádný mechanismus přímého předávání výsledkových sad mezi procedurami, je nutné vymyslet a dodržovat nějaký nepřímý. Jediným rozumným způsobem je předávat data v dočasných (`TEMPORARY`) tabulkách. Abychom se vyhnuli kolizím ve jmenném prostoru (MySQL 5.0 má jediný společný), zavedeme konvenci, že každá procedura pro komunikaci s okolím používá dočasné tabulky, jejichž jméno začíná jménem procedury. Sice to povede k dlouhým názvům typu `nn_ts_CorrectWeights_InternalValues`, ale účel to splní.

4.2.1.6 Nedostatky v konceptu `GROUP BY`

Pro agregaci používá jazyk SQL klauzuli `GROUP BY(`*`výraz1, výraz2, ...`*`)`. Ta způsobí seskupení výsledků do skupin se stejnými hodnotami vyjmenovaných výrazů. Sloupce, které se v klauzuli `GROUP BY` nevyskytují, můžeme ve výrazech tvořících výsledná data používat pouze

v takzvaných agregačních funkcích [14]. To stačí, pokud chceme informaci souhrnného typu. Pokud ovšem chceme provést nějaký výpočet uvnitř této skupiny, musíme se uchýlit k řešení pomocí poddotazů nebo triku. Poddotazy jsou (aspoň v MySQL) zdrojem značných poklesů výkonu dotazů, proto se obvykle využívá tzv. *CONCAT-trik*. Konkrétní řešení popisujeme dále.

4.2.2 Implementace vyhledávání hrubou silou – procedura `mhd_VyhledejSpoje()`

Vyhledávání spojů hrubou silou provádí procedura `mhd_VyhledejSpoje()`:

```
CREATE PROCEDURE `mhd_VyhledejSpoje`(
  idZastFrom INT UNSIGNED,
  idZastTo INT UNSIGNED,
  whn DATETIME,
  iMaxRounds TINYINT UNSIGNED
)
```

Jejími parametry jsou:

- `idZastFrom` – ID zastávky, ze které má spoj vyjíždět (výchozí zastávka). Odkazuje do tabulky `mhd_zast`.
- `idZastTo` – ID zastávky, na kterou má spoj dorazit (cílová zastávka). Odkazuje do tabulky `mhd_zast`.
- `whn` – určuje okamžik, ve kterém začínáme spoj hledat.
- `iMaxRounds` – maximální počet kol, ve kterých se bude hledat. Určuje také maximální počet přestupů.

4.2.3 Princip práce procedury `mhd_VyhledejSpoje()`

Princip, na kterém procedura `mhd_VyhledejSpoje()` pracuje, je hybrid mezi výše uvedenými algoritmy používanými k hledání nejkratší cesty. Vyhledávání pracuje na základě procházení grafem podobně jako Dijkstrův algoritmus v tom smyslu, že si udržuje množinu navštívených a nenavštívených vrcholů, a u každého vrcholu si pamatuje „nejkratší vzdálenost“, tedy v našem případě nejkratší čas, a způsob, jakým se dá do vrcholu v tomto čase dostat.

Za tímto účelem si v metodě zřizujeme dočasnou tabulku `tmp_pokryte_zast`. Na začátku do tabulky vkládáme výchozí zastávku s časem dosažení rovným času, od kterého máme vyhledávat.

```
CREATE  TEMPORARY TABLE tmp_pokryte_zast (
    id_zast INT UNSIGNED NOT NULL PRIMARY KEY,
    fastest TIME NOT NULL, INDEX(fastest),
    id_jizda_fastest INT UNSIGNED,
    id_zast_from INT UNSIGNED
  ) ENGINE Memory;
INSERT INTO tmp_aktualni_zast
  SELECT idZastFrom AS id_zast, whn AS fastest,
```

```
      NULL AS id_jizda_fastest, NULL AS id_zast_from;
INSERT INTO tmp_pokryte_zast
    SELECT * FROM tmp_aktualni_zast;
```

Stejnou strukturu má také tabulka `tmp_aktualni_zast`, která uchovává množinu aktuálně zpravovávaných zastávek. Na začátku do ní vložíme obsah tabulky `tmp_pokryte_zast`.

Z důvodu optimalizace si také uchováváme seznam použitých tras. Pokud bychom chtěli napodobit funkci systému IDOS, který vyhledává více spojů v různých časech, mělo by smysl použít stejnou trasu vícekrát. Jelikož ale hledáme pouze nejlepší spoj, nemá smysl jet po stejné trase znovu – to by nás dovedlo v pozdějším čase do zastávek, kterých již jsme dosáhli v kratším čase.

```
CREATE  TEMPORARY TABLE tmp_pokryte_trasy (
    id_trasa INT UNSIGNED NOT NULL PRIMARY KEY
) ENGINE Memory;
```

Tolik příprava; následně zavedeme proces do smyčky. V každé její iteraci provádíme následující kroky:

- Zjistíme, zda v dosažených zastávkách je ta, na kterou se chceme dostat. Pokud ano a nechceme najít potenciálně lepší spojení s více přestupy, které možná existuje, ukončíme hledání.

```
SELECT COUNT(*) INTO @iCnt FROM tmp_aktualni_zast
  WHERE id_zast = idZastTo;
SET @iKola = @iKola + 1;
IF @iCnt != 0 OR @iKola = iMaxRounds
  THEN LEAVE loopHledani; END IF;
```

- Získáme trasy projíždějící aktuálními zastávkami.

```
CREATE TEMPORARY TABLE tmp_uzel_tras ENGINE Memory
SELECT za.id_zast, za.fastest, tu.id, tu.id_trasa, tu.poradi
  FROM tmp_aktualni_zast AS za
  LEFT JOIN mhd_trasy_uzly AS tu ON tu.id_zast = za.id_zast;
```

- Pro každou trasu zjistíme její první stání v dané zastávce. Získáme ID jízdy, dále čas, kdy zastávkou projíždí, a pořadí zkoumané zastávky na trase té které jízdy. Zde se objevuje CONCAT-trik zmiňovaný v bodu Nedostatky v konceptu GROUP BY(4.2.1.6).

```
CREATE TEMPORARY TABLE tmp_stani_prvni ENGINE Memory
SELECT tu.id_zast, tu.id_trasa, tu.poradi,
   -- Chceme vědět, za jak dlouho od momentu, kdy jsme schopni
   -- se do zastávky dostat, kdy jede první jízda, a její ID.
   -- CONCAT(): spojení kvůli přenesení z GROUPování
   MIN(CONCAT( lib_GetDistanceToTime( fastest,
     IFNULL(js.cas1,js.cas2) ),',',js.id_jizda )) AS zakdy_jizda
FROM tmp_uzel_tras AS tu
LEFT JOIN mhd_jizdy_stani AS js ON js.id_trasa_uzel = tu.id
GROUP BY tu.id_zast, tu.id_trasa;
```

- Z implementačních důvodů (nedostatky v konceptu `GROUP BY`) musíme provést další dotaz, kterým rozdělíme řetězec na čas a ID příslušné jízdy.

```
    CREATE TEMPORARY TABLE tmp_jizdy_zastavkou (
      id_zast INT UNSIGNED NOT NULL,
      id_trasa INT UNSIGNED NOT NULL,
      zakdy TIME NOT NULL,
      id_jizda INT UNSIGNED NOT NULL,
      poradi SMALLINT NOT NULL
    ) ENGINE Memory
      SELECT id_zast, id_trasa, LEFT(zakdy_jizda,8) AS zakdy,
  SUBSTR(zakdy_jizda,10) AS id_jizda, poradi FROM tmp_stani_prvni;
```

- Do seznamu pokrytých tras přidáme aktuálně nalezené.

```
    INSERT IGNORE INTO tmp_pokryte_trasy (id_trasa)
      SELECT id_trasa FROM tmp_jizdy_zastavkou;
```

- Následuje poměrně složitý SQL příkaz, kterým sestavíme seznam zastávek, do kterých se dostaneme v tomto kole jízdami vybranými výše.

```
INSERT tmp_aktualni_zast(id_zast,fastest,id_jizda_fastest,id_zast_from)
SELECT
  tu.id_zast AS id_zast,
  lib_GetNearerTime( whn,zp.fastest,IFNULL(js.cas1,js.cas2)) AS fastest,
  jz.id_jizda AS id_jizda_fastest,
  jz.id_zast AS id_zast_from
  -- Nejprve všechny první jízdy zastávkou.
  FROM tmp_jizdy_zastavkou   AS jz
  -- Na ně napojíme všechna následující stání.
  LEFT JOIN mhd_jizdy_stani  AS js
     ON js.id_jizda = jz.id_jizda AND js.poradi > jz.poradi
  -- K nim připojíme příslušné uzly tras.
  LEFT JOIN mhd_trasy_uzly   AS tu ON js.id_trasa_uzel = tu.id
  -- A k tomu ještě připojíme již pokryté zastávky,
  -- a použijeme jejich čas, pokud je menší.
  LEFT JOIN tmp_pokryte_zast AS zp ON zp.id_zast = tu.id_zast
  -- Vyloučení konečných zastávek - u jejich jízd
  -- nejsou žádná další stání, tedy napoji se NULL row.
  WHERE js.id_jizda IS NOT NULL
  -- Pokud do jedné zastávky můžeme v tomto kole
  -- dojet více způsoby, vybereme ten rychlejší.
ON DUPLICATE KEY UPDATE
      fastest = lib_GetNearerTime(whn, tmp_aktualni_zast.fastest,
  VALUES(fastest)),
      id_jizda_fastest = IF( lib_T1SoonerThanT2(whn, VALUES(fastest),
  tmp_aktualni_zast.fastest), VALUES(id_jizda_fastest),
  tmp_aktualni_zast.id_jizda_fastest),
      id_zast_from     = IF( lib_T1SoonerThanT2(whn, VALUES(fastest),
  tmp_aktualni_zast.fastest), VALUES(id_zast_from),
  tmp_aktualni_zast.id_zast_from);
```

- Nakonec do seznamu dosažených zastávek přidáme ty, na které jsme se dostali v tomto kole.

```
INSERT INTO tmp_pokryte_zast
  SELECT * FROM tmp_aktualni_zast
  ON DUPLICATE KEY UPDATE
      fastest = lib_GetNearerTime(whn, tmp_pokryte_zast.fastest,
  VALUES(fastest)),
```

```
        id_jizda_fastest = IF( lib_T1SoonerThanT2(whn,
    VALUES(fastest), tmp_pokryte_zast.fastest),
    VALUES(id_jizda_fastest), tmp_pokryte_zast.id_jizda_fastest),
        id_zast_from      = IF( lib_T1SoonerThanT2(whn,
    VALUES(fastest), tmp_pokryte_zast.fastest),
    VALUES(id_zast_from),      tmp_pokryte_zast.id_zast_from);
```

Tím končí první smyčka. Po ní následuje další smyčka, ve které sestavujeme tabulku `tmp_nalezene_trasy` s nalezenou nejkratší trasou. Tato smyčka pracuje takto:

Máme k dispozici všechny zastávky, kterých jsme dosáhli s daným počtem přestupů, a u nich uložený údaj, jakou jízdou a ze které výchozí zastávky jsme se do zastávky dostali.

Stejné údaje máme k dispozici i u výchozí zastávky spoje; můžeme tedy zpětně sestavit cestu, jak se do jakékoliv zastávky dostat z výchozí.

Pokud je v množině dosažených zastávek cílová zastávka (předaná v parametru `idZastTo`), budeme zpětně sestavovat cestu od ní k výchozí zastávce celého spojení (její ID máme v parametru `idZastFrom`).

Nejprve tedy vytvoříme tabulku, do které uložíme výslednou nalezenou trasu:

```
    CREATE  TEMPORARY TABLE tmp_nalezene_trasy LIKE tmp_pokryte_zast;
    ALTER TABLE tmp_nalezene_trasy ADD COLUMN pos TINYINT UNSIGNED FIRST;
```

Aby bylo zpracování výsledků pro klienta snazší, druhým příkazem `ALTER TABLE` přidáváme sloupec, do kterého uložíme pořadí jízd, v jakém byly použité ve spoji.

Dále si připravíme pracovní tabulku `tmp_aktualni_zast2`.

```
    CREATE  TEMPORARY TABLE tmp_aktualni_zast2 LIKE tmp_pokryte_zast;
    INSERT INTO tmp_aktualni_zast2
      SELECT * FROM tmp_pokryte_zast WHERE id_zast = idZastTo;
    SET @iFound = ROW_COUNT();
```

Tabulce dáváme stejnou podobu, jakou má tabulka `tmp_pokryte_zast`.

Příkazem `INSERT` do ní vložíme řádek z tabulky `tmp_pokryte_zast` – neboli prvek množiny pokrytých zastávek – který odpovídá cílové zastávce (`idZastTo`).

V posledním příkazu vytváříme proměnnou `@iFound` a do ní ukládáme počet nalezených zastávek; v tomto případě buď `0` nebo `1`, což indikuje, jestli jsme v předchozí smyčce dosáhli cílové zastávky. Pokud ne (počet je 0), v první iteraci smyčky se hned na jejím začátku ukončí sestavování trasy a vrátí se prázdná tabulka:

```
loopSestaveni: LOOP
  IF @iFound = 0 OR @iKola <= 0 THEN LEAVE loopSestaveni; END IF;
  SET @iKola = @iKola - 1;    -- Dekrementace počtu přestupů.
```

Pokud ale nějaké zastávky nalezeny byly, vložíme jejich záznam společně s pořadovým číslem spoje do tabulky nalezených tras. Jelikož hledáme pouze nejrychlejší spoj, bude se vždy jednat pouze o jednu zastávku.

```
INSERT INTO tmp_nalezene_trasy
  SELECT @iKola, zast.* FROM tmp_aktualni_zast2 AS zast;
```

Následuje převod zastávek z minulého kroku. Budeme se z nich vracet blíže k výchozí zastávce. Nejprve posuneme tabulky zastávek tohoto a předchozího kola (bráno z pohledu smyčky tvorby spojů):

```
ALTER TABLE tmp_aktualni_zast2 RENAME tmp_aktualni_zast;
CREATE TEMPORARY TABLE tmp_aktualni_zast2 LIKE tmp_aktualni_zast;
```

Potom do tabulky s množinou zkoumaných zastávek vložíme zastávku, ze které jsme se dostali do té z aktuálního kola, respektive spoj, kterým jsme se do ní dostali.

```
INSERT INTO tmp_aktualni_zast2
SELECT zp.* FROM tmp_aktualni_zast AS za
  LEFT JOIN tmp_pokryte_zast AS zp ON zp.id_zast = za.id_zast_from
  WHERE zp.id_zast_from IS NOT NULL;
```

Posledním příkazem opět zjistíme počet odpovídajících zastávek. Poté je konec smyčky, hodnota proměnné `@iFound` se tedy použije opět na začátku smyčky pro rozhodnutí o ukončení cyklu. Pokud by byla nulová, znamená to, že do zastávky již nemáme jak se dostat, případně na místě identifikátoru předchozí zastávky je `NULL`, což odpovídá výchozí zastávce, pro kterou jsme `NULL` uložili při přípravě na začátku procedury. V obou případech ukončujeme sestavování a vracíme aktuální obsah tabulky.

```
  SET @iFound = ROW_COUNT();
END LOOP loopSestaveni;
```

Nakonec již jen přejmenujeme pracovní tabulku tak, aby odpovídala názvovým konvencím, které jsme si zvolili (viz 4.2.1.5).

```
ALTER TABLE tmp_nalezene_trasy RENAME TO mhd_VyhledejSpoje;
```

4.3 Použití procedury `mhd_VyhledejSpoje()`

Jak je uvedeno v části 4.2.2, proceduře `mhd_VyhledejSpoje()` předáváme v parametrech tyto údaje:

- Identifikátor zastávky, ze které hledáme spojeni.
- Identifikátor zastávky, do které hledáme spojeni.
- Čas, od kterého má procedura hledat spojení.
- Maximální počet přestupů.

Volání procedury potom vypadá následovně:

```
CALL mhd_VyhledejSpoje( 5, 141, '2008-05-14 21:53:00', 3 );
SELECT sp.*, zast.nazev_plny FROM mhd_VyhledejSpoje AS sp
  LEFT JOIN mhd_zast AS zast ON sp.id_zast = zast.id
  ORDER BY pos ASC;
```

Druhý příkaz zobrazí výsledek a k zastávkám připojí jejich názvy.

Tabulka 4.1: Obsah tabulky mhd_VyhledejSpoje po volání stejnojmenné procedury.

pos	id_zast	fastest	id_jizda_fastest	id_zast_from	nazev_plny
1	11	22:06:00	13505	5	'Nádraží'
2	141	04:57:00	13176	11	'Křižíkova'

5 Optimalizace s využitím neuronových sítí pro heuristiku

V kapitole 4 jsme se seznámili s běžnými způsoby optimalizace prohledávání grafu. Také jsme si vysvětlili, proč nejdou použít pro vyhledávání spojení v jízdních řádech. Optimalizace však principiálně možná je.

Inspirací nám budiž pozorování člověka, který cestuje dopravní sítí: Přijde na zastávku a chce se dostat na jinou. Pokud počítáme s člověkem znalým dopravní sítě, pravděpodobně nepůjde ke stojanu s vývěskou a nebude analyzovat trasy linek a počítat jejich časy. Místo toho je intuitivně naučený, na kterou linku má nastoupit, v závislosti na poloze cílové zastávky. Jak mezi linkami vybírá?

Zaprvé pravděpodobně využije místopisných znalostí, případně mapu, a bude se snažit vyhledávat spoje, které pravděpodobně povedou do cíle – tedy ty, jejichž trasa fyzicky míří zhruba směrem na cíl. To je chování podobné algoritmu A*.

To ovšem nestačí. Co kdyby cestující chtěl např. z Českých Budějovic na Sicílii? Nejlepší spoj (z pohledu algoritmu A* překvapivě) nepovede od Českých Budějovic na jih, ale naopak – nejprve povede do Prahy na letiště. Algoritmus A* s předpokladem vzdušné čáry jako nejkratší cesty by tedy vyhledání spoje neuspíšil, ale naopak zdržel.

Cestující tedy navíc použije svoje zkušenosti, dalo by se říct empirické znalosti, které mu napovídají, že na Sicílii se nejlépe dostane letadlem z Prahy. Proto si nejprve vyhledá spoj do Prahy a následně odtud na Sicílii.

Ke svému úsudku tedy podvědomě použil svoje dosavadní zkušenosti s cestováním. Při dostatečném počtu pokusů ze všech zastávek ve všech časových úsecích dne může empirickými znalostmi pokrýt všechny situace na celé dopravní síti.

My se to pokusíme udělat stejně. Empirické znalosti se pokusíme uložit do neuronových sítí, jejíchž úkolem bude rozhodovat na příslušné zastávce v daném čase, kterou trasou se vydat (a případně i kde vystoupit), pokud mají tyto akce vést k dosažení cílové zastávky v co nejkratší době.

5.1 Analýza úlohy neuronové sítě

Při použití neuronových sítí je důležité zvážit zejména tyto faktory:

- Způsob použití sítě

- Druh použité sítě
- Architektura sítě
- Způsob zakódování vstupů sítě
- Způsob zakódování výstupů sítě

Postupně provedeme diskusi ke všem čtyřem faktorům.

5.1.1 Způsob a druh použité sítě

Úloha "nápovědy", kterou trasu zvolit v daném čase pro dosažení dané cílové zastávky z dané výchozí zastávky, je v podstatě fuzzy mapováním z *N*-rozměrného prostoru vstupů do *M*-rozměrného prostoru výstupů, kde *M* je počet tras, které v dané zastávce můžeme zvolit, a *N* je počet vstupů, který odvodíme dále. Výstupy označují pravděpodobnost, s jakou daná trasa povede k nejrychlejšímu spoji. Jedná se tedy o selektivní model [15].

Z toho důvodu je zřejmé, že nejlepší způsob využití neuronových sítí je přiřadit každé zastávce jednu neuronovou síť, kterou použijeme, pokud budeme vyhledávat spoj z dané zastávky, a na základě vstupů doporučí použití některé z tras, které zastávkou procházejí.

Ze stejného důvodu se také jako nejlepší volba druhu sítě jeví perceptron učený algoritmem backpropagation - perceptrony jsou pro selektivní modely empirických znalostí ideální.

Jednotlivé případy úlohy vyhledávání v jízdních řádech jsou nezávislé na ostatních úlohách. Nezáleží na tom, kolik a jakých vyhledávání jsme provedli dříve; vždy budeme chtít stejný výsledek.

Z toho plynnou dva fakty:

a) nepotřebujeme rekurentní neuronové sítě, tedy takové, které jako vstup používají svůj předchozí výstup,

b) nepotřebujeme na vstupu jiné informace než ty, které se týkají aktuálního případu úlohy.

Použijeme tedy dopředný perceptron.

5.1.2 Architektura sítě

Úloha je s největší pravděpodobností lineárně neseparabilní. (Záleží na konkrétním jízdním řádu a kódování vstupů, ale možnost, že by všechny vektory vstupů byly lineárně separabilní, je mizivá.) Z toho plyne nutnost použít vícevrstvý perceptron [17], [16]. Kolik vrstev konkrétně použít a jaký počet neuronů zvolit již není tak jednoduchá otázka.

Nejužitečnějším průvodcem byl dokument "comp.ai.neural-nets FAQ" [21], což je dokument o rozsahu knihy, ve kterém jsou shrnuté znalosti mnoha expertů na neuronové sítě a jejich odpovědi na často kladené otázky na stejnojmenné diskusní skupině.

5.1.2.1 Počet neuronů první skryté vrstvy

Existuje několik zaběhnutých, ale zcela zcestných "vodítek", které se vyskytují i v odborné literatuře [18] jako tzv. "rules of thumb", například:

- Počet neuronů střední vrstvy by měl být někde mezi počtem vstupů a počtem výstupů.
- Počet neuronů střední vrstvy by měl být 2/3 počtu vstupů plus počet výstupů.
- Počet skrytých neuronů by měl být menší než dvojnásobek počtu vstupů.

Existují také různé experimentální přístupy – vyzkoušet naučit síť s různými počty neuronů a vrstev [19]. Bohužel zde trvá učení tak dlouho, že vyzkoušení tohoto přístupu nezbyl čas. Odhadem počtu neuronů se zabývají i některé patenty, např. [20]. My jsme zvolili odhad počtu neuronů na základě výpočtu.

Víme, že první vrstva skrytých neuronů při použití rostoucí aktivační funkce v podstatě dělí hyperprostor na poloprostory. My chceme prostor rozdělit na takové oblasti, které budou reprezentovat jednotlivé cílové zastávky v různých časových úsecích dne. Pokud si tedy požadovanou granularitu výsledků stanovíme na šestiminutové intervaly (240) pro každou jednotlivou zastávku, kterých je Z, počet potřebných podprostorů je tedy $240 \cdot Z$. V jízdním řádu, který máme k dispozici, je 159 zastávek, potřebujeme tedy 38160 podprostorů. Zbývá otázka, kolik nadrovin rozdělí N-rozměrný nadprostor na p podprostorů.

V jednorozměrném prostoru dělí každá nadrovina (bod) prostor přímky tak, že přibude další jeden podprostor. Ve dvourozměrném prostoru vzniká s každou přidanou přímkou vždy o jeden prostor více než s předchozí. Ve trojrozměrném prostoru potom vzniká s n-tou přidanou rovinou o tolik více podprostorů, kolik podprostorů vzniklo ve dvourozměrném prostoru při přidání n-té přímky. Obecně potom pro N-rozměrný prostor platí, že přidáním n-té nerovnoběžné nadroviny vznikne přibude tolik podprostorů, kolik celkem jich vzniklo rozdělením N-1-rozměrného prostoru n nerovnoběžnými nadrovinami. Rekurentní výpočet by potom mohl vypadat takto:

$$\begin{aligned} p(N,n) &= p(N,n-1) + p(N-1,n-1) \\ p(1,n) &= n+1 \end{aligned}$$

Případně přímý výpočet vypadá takto: $p = \binom{n}{0}\binom{n}{1}\cdots\binom{n}{N-1}\binom{n}{N}$

Potřebujeme ovšem znát opačný výpočet – známe dimenzionalitu prostoru, víme, kolik chceme výsledných podprostorů, a chceme vědět, kolik nadrovin jej takto rozdělí, resp. kolik máme použít neuronů na první vrstvě. Výpočtem se zde nebudeme podrobně zabývat, stačí nám jeho odhad. Výše uvedený výpočet se dá odhadnout jako $p(N,n) \approx n^N / N!$

Vyjádření n potom vypadá takto: $n \approx \sqrt[N]{N! \cdot p} \approx N \cdot \sqrt[N]{p}/\mathrm{e}$

kde (pro připomenutí) n je počet nadrovin, N je dimenzionalita prostoru, p je počet podprostorů a e je Eulerova konstanta.

Ve výpočtech zacházíme s dimenzionalitou prostoru. Ta je rovna počtu vstupů sítě, který určíme v části 5.1.3. Pro úplnost dodejme, že v našem případě se bude jednat o pětirozměrný prostor, a pro $240 \cdot 159$ požadovaných podprostorů budeme potřebovat minimálně 16 neuronů skryté vrstvy. Jelikož mnohé nadroviny mají tendenci se "zatoulat" a dělit prostor celkem neúčelně, experimentálně jsme vyzkoušeli, že rozumný počet je zhruba 25 až 30 neuronů vnitřní vrstvy.

5.1.2.2 Počet vrstev sítě

Heaton ve své knize [18] uvádí následující vodítko:

Skryté vrstvy	Výsledek
žádné	Dokáže reprezentovat jen lineárně separabilní funkce nebo rozhodování.
1	Zvládá libovolnou aproximaci skoro jakékoliv funkce, která obsahuje spojité mapování jednoho prostoru do druhého.
2	Se vhodnými aktivačními funkcemi zachytí libovolné rozhodovací hranice s libovolnou přesností a dokáže aproximovat libovolné hladké mapování s jakoukoliv přesností.

Ovšem důvěryhodnost tohoto zdroje jsme poznali v odstavci 5.1.2.1. Ostatní zdroje se této otázce buď vyhýbají, nebo ji probírají formou matematického modelu, ovšem na zjednodušených modelech sítí s booleovskými hradly [25].

Co tedy jednotlivé vrstvy dělají? Jak uvádíme v odstavci 5.1.2.1, každý neuron se sigmoidou jako aktivační funkcí funkcí představuje rozdělení nadprostoru jednou (fuzzy) nadrovinou. Vícevrstvý perceptron s jednou skrytou vrstvou je schopný rozdělit trénovací vzorky do několika skupin. V případě úlohy výběru vhodné trasy se vstupy času a prostorového umístění cílové zastávky skrytá vrstva rozčleňuje nadprostor na časoprostorové „kapsy“, například „oblast na severu města kolem sedmé večer“.

Jelikož u dvouvrstvého perceptronu s dvěma skrytými vrstvami se dá považovat výstup první skryté vrstvy za vstupní vrstvu perceptronu s jednou skrytou vrstvou, druhá vrstva přebírá rozdělení trénovacího vzorku jednotlivými nadrovinami do těchto kapes a na základě tohoto rozdělení provádí další dělení nadprostoru nadrovinami. Pro určení vhodné trasy by ale mělo zcela postačovat již určení oné kapsy – při známém úseku času a známé části prostoru, ve kterém se nachází cílová zastávka, by již měla neuronová síť mít dostatek informací k určení vhodné trasy. Proto volíme jednu skrytou vrstvu.

5.1.3 Způsob zakódování vstupů sítě

5.1.3.1 Zakódování cílové zastávky

Cílová zastávka je v databázi identifikována svým číselným identifikátorem. Identifikátor sám o sobě nevypovídá vůbec nic – přiřazení identifikátorů zastávkám může být libovolné.

Pokud chceme neuronové síti předat informaci o výběru některého prvku z mnoha, které spolu nijak nesouvisejí, jedinou rozumnou možností zakódování je vytvořit jeden vstupní neuron pro každý takový prvek (případně o jeden méně, pokud používáme na první skryté vrstvě biasy – to abychom předešli vzniku lineárních závislostí). Pro dopravní síť malého města se 159 zastávkami bychom tedy potřebovali 159 vstupních neuronů, a pro dopravní síť Prahy už by se jednalo o neúnosný počet vstupů (řádově tisíce). Nejenže by výpočet takové sítě byl pomalý (doba výpočtu první vrstvy roste lineárně s počtem vstupů), ale navíc by se taková síť pravděpodobně špatně učila a podávala špatné výsledky – viz [21], část 2, *What is the curse of dimensionality?*:

> The curse of dimensionality causes networks with lots of irrelevant inputs to be behave relatively badly: the dimension of the input space is high, and the network uses almost all its resources to represent irrelevant portions of the space.

Proto raději zastávky zakódujeme jinak. Jediná možnost, jak snížit počet vstupů pro určení prvku, je najít nějakou jejich společnou vlastnost, která se dá vyjádřit malým počtem hodnot ze spojitých intervalů. V případě zastávek se přímo nabízí jejich geografická poloha vyjádřená souřadnicemi. Nese to jednu velkou výhodu: Generalizaci.

Generalizace je v oblasti neuronových sítí velmi žádoucí jev. Jedná se o tu schopnost neuronové sítě, díky které dokáže poskytnout rozumný výstup i pro vstupní hodnoty, které nejsou blízké žádnému ze vstupních vektorů. V případě geografické pozice potom můžeme zadat pozici mezi dvěma zastávkami, a neuronová síť by měla doporučit vydat se trasami, které povedou na jednu nebo na druhou z nich. To je výhodné také proto, že učení pro jednu cílovou zastávku zároveň příliš nezruší nastavení vah pro blízkou zastávku na stejné trase.

Všem zastávkám tedy potřebujeme přiřadit geografické souřadnice a ty následně normalizovat do rozmezí <-1; 1>. Horní a dolní hranice intervalů souřadnic X, resp. Y určí nejzápadnější a nejvýchodnější, respektive nejsevernější a nejjižnější zastávka dopravního systému. Hypotetická zastávka uprostřed tedy ponese souřadnice [0,0].

Pro určení GPS souřadnic všech zastávek jsme použili servery www.mapy.cz a www.amapy.cz. Bohužel ani u jednoho není k dispozici API rozhraní, které by umožňovalo extrahovat umístění zastávek automaticky.

Následné převedení GPS souřadnic ve formátu `Loc: S°MM'SS.sss"N S MM'SS.sss"E` na dvojici reálných čísel provádí funkce `geo_GetDoublesFromGPS()`.

```
CREATE PROCEDURE `geo_GetDoublesFromGPS`(IN sLoc VARCHAR(60), OUT dLat
   DOUBLE, OUT dLon DOUBLE) BEGIN

  -- Zpracováváme formát: 'Loc: 48°58\'40.53"N, 14°29\'26.277"E';
  SET @s = REPLACE(sLoc, 'Loc: ','');
  SET @s = REPLACE(@s, '"N','');
  SET @s = REPLACE(@s, '"E','');
  SET @s = REPLACE(@s, "'",':');
  SET @s = REPLACE(@s, "°",':');
  SET @tLat = CAST( SUBSTRING_INDEX(@s,',',1) AS TIME);
  SET @tLon = CAST( LTRIM(SUBSTRING_INDEX(@s,',',-1)) AS TIME);

  SET @dLat = HOUR(@tLat) + MINUTE(@tLat)/60 + (SECOND(@tLat) + 0.000001
   * MICROSECOND(@tLat)) /3600;
  SET @dLon = HOUR(@tLon) + MINUTE(@tLon)/60 + (SECOND(@tLon) + 0.000001
   * MICROSECOND(@tLon)) /3600;
END $$
```

Nakonec ještě potřebujeme hodnoty převést do intervalu <-1; 1>.

```
CREATE PROCEDURE `mhd_geo_NormalizeLatLon`() BEGIN
  --  Získáme šířku / délku z GPS řetězce "Loc: ..."
  UPDATE jizdnirady_cb.mhd_zast SET lat = geo_GetLatFromGPS(gps_loc),
                                    lon = geo_GetLonFromGPS(gps_loc);
  --  Rozdíl mezi největší a nejmenší šířkou / délkou.
  SELECT MIN(lat),  MAX(lat)-MIN(lat) AS lat_diff,
         MIN(lon),  MAX(lon)-MIN(lon) AS lon_diff
  INTO @dLatMin, @dLatDiff, @dLonMin, @dLonDiff
  FROM mhd_zast;

  --  Normalizujeme pozici do intervalu <-1;1>.
  UPDATE jizdnirady_cb.mhd_zast SET
    lat_norm = (lat - @dLatMin) / @dLatDiff * 2 - 1
   ,lon_norm = (lon - @dLonMin) / @dLonDiff * 2 - 1 ;
END $$
```

Takto převedené souřadnice již můžeme použít přímo jako vstup pro neuronovou síť.

5.1.3.2 Zakódování času

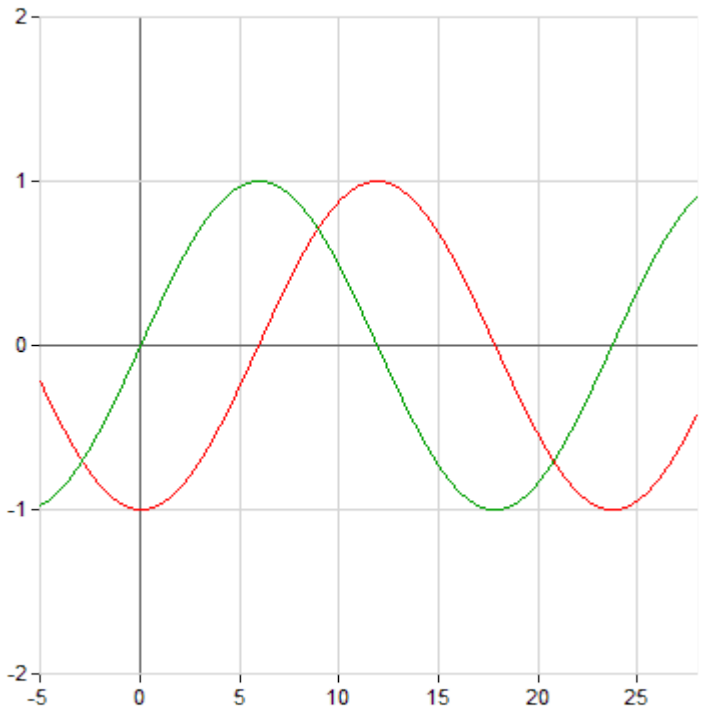
Zakódování času se zdá být jednoduché. Konvenční značení času od 00:00 do 23.59 svádí jednoduše čas převést do intervalu <-1; 1> stylem *(hodiny · 60 + minuty) / 24 · 60 · 2 – 1*; takové řešení ovšem vede k situaci, kdy stejný okamžik, tedy půlnoc, reprezentuje jak hodnota -1.0, tak hodnota 1.0, a časy blízké půlnoci z opačné strany časové osy jsou z pohledu neuronové sítě diametrálně odlišné. To je pochopitelně nežádoucí a vedlo by to ke špatným, ne-li žádným výsledkům v časech kolem půlnoci.

Řešením je čas zakódovat pomocí nějaké periodické funkce, či spíše tak, aby hodnota funkce opět klesla a na pomyslné přímce času tvořila spojitou křivku. Ovšem zakódováním např. pomocí funkce sin(t) s periodou 24 (hodin) by vedlo ke dvěma nepříznivým vlivům:

a) Poblíž maxima a minima by se rychlost změny hodnoty funkce snížila oproti inflexním bodům. Při kódování času ale není taková charakteristika nijak opodstatněná; ideálně by funkce měla mít konstantní hodnotu derivaci.

b) Horší důsledek je ten, že dvěma různým časovým okamžikům by odpovídala stejná hodnota; z pohledu neuronové sítě by tedy byly tyto dva okamžiky nerozpoznatelné.

Řešením je zakódovat čas více vstupy, minimálně dvěma tak, aby každý okamžik byl jednoznačně určený kombinací jejich hodnot, např. již zmíněný sin(t) a k němu cos(t), kde t je čas dne lineárně mapovaný na interval <0; 2π).



$$\cos\left(\frac{t-6}{\pi\cdot 1{,}2}\right) \qquad \sin\left(\frac{t-6}{\pi\cdot 1{,}2}\right)$$

Tento převod provádí procedury `mhd_nn_TimeInputCos()` a `mhd_nn_TimeInputSin()`:

```
CREATE FUNCTION `mhd_nn_TimeInputCos`( t TIME ) RETURNS double BEGIN
  RETURN COS( TIME_TO_SEC(t) * 0.0000727221 );
END $$
CREATE FUNCTION `mhd_nn_TimeInputSin`( t TIME ) RETURNS double BEGIN
  RETURN SIN( TIME_TO_SEC(t) * 0.0000727221 );
END $$
```

Ještě lepší je ovšem do kódování zahrnout rozumovou úvahu a neuronové síti trochu napovědět souvislosti, které o dopravním systému známe. V našem případě se jedná o dopravní systém městské hromadné dopravy, ve kterém první ranní vozidla vyjíždějí kolem 4:10 a poslední dojíždějí do konečných zastávek kolem 23:30. Noční spoje potom vyjíždějí kolem 23:10 a končí jízdy

kolem 4:30. Bylo by tedy vhodné sítí dát explicitně najevo, zda je denní či noční režim a v jaké fázi daného režimu se nacházíme. Proto je ideální řešení vstupu takové:

Jeden vstup pro určení, zda je denní či noční režim. Jeho hodnota nabývá přes den hodnoty 1.0, přes noc -1.0. V mezních časech (4:00 až 4:30 a 23:00 až 23:30) lineárně stoupá a klesá mezi těmito hodnotami.

```
CREATE FUNCTION `mhd_nn_TimeInputDayNight`(t TIME) RETURNS double BEGIN

  SET @t = TIME_TO_SEC(t);
  --                        4:00    4:30   23:00   24:00
  SET @i = INTERVAL( @t, 14400,  16200,  82800,  86400 );

  RETURN ELT( @i + 1 ,
    -1.0,
    (@t - 14400) / 1800.0 * 2 - 1,
    1.0,
    (@t - 82800) / 3600.0 * -2 + 1,
    -1.0
  );
END $$
```

Druhý vstup udává, v jaké fázi denního režimu se nacházíme. V průběhu noci nabývá hodnoty 0.0, v průběhu dne mezi 4:10 a 23:30 hodnota lineárně stoupá od -1.0 do 1.0, v mezních časech lineárně přechází k 0.0.

```
CREATE FUNCTION `mhd_nn_TimeInput1`( t TIME ) RETURNS double BEGIN
  /**
   *   Vrací hodnotu od -1.0 do 1.0, lineárně mapovanou od 4:00 do 24:00.
   *   Mimo tento interval vrací 0.0.
   */
  IF t < CAST('4:00:00' AS TIME) OR t > CAST('24:00:00' AS TIME) THEN
    RETURN 0.0; END IF;
  --   50400 == (10 + 4) * 3600.
  RETURN ((TIME_TO_SEC(t) - 50400) / 36000.0);
END $$
```

Třetí vstup naopak udává fázi nočního režimu od 23:00 do 4:30, a podobně, avšak opačně než vstup pro denní režim nabývá přes den hodnoty 0.0, přes noc lineárně roste od -1.0 do 1.0 a v mezních časech se lineárně blíží k 0.0.

```
CREATE FUNCTION `mhd_nn_TimeInput2`( t TIME ) RETURNS double BEGIN
  /**
   *  Vrací hodnotu od -1.0 do 1.0, lineárně mapovanou od  23:30 do 4:10.
   *  Mimo tento interval vrací 0.0.
   */
  IF t > CAST('4:10:00' AS TIME) AND t < CAST('23:30:00' AS TIME) THEN
   RETURN 0.0; END IF;
  SET @iTime = TIME_TO_SEC(t) + 1800;       --  23:30 -> "24:00"
  -- Později než 10:00 --> převedeme na "předchozí den".
  IF  @iTime > 36000 THEN SET @iTime = @iTime - 86400; END IF;
  RETURN  (@iTime / 16800.0) * 2 - 1;
END $$
```

Tímto kódováním by mělo dojít k lepší generalizaci, protože pro očekávaně příbuzné (související) výsledky nabízíme síťi podobné vstupy.

5.1.3.3 Zakódování dalších parametrů

Další možné vstupní parametry pro vyhledávání jsou převážně binární a určují fakta typu zda jde o pracovní den nebo o víkend či svátek; dále zda jsou prázdniny, a podobně. Pravděpodobně lepším přístupem je však ponechat množinu vstupů co nejmenší a pro takto časově odlehlé a nepříliš související jízdní řády je lepší raději natrénovat další neuronové sítě.

```
CREATE FUNCTION `mhd_nn_TimeInput_WeekDay`(d DATE) RETURNS double BEGIN

  /**
   *   Vrací -1.0 pro pracovní dny a 1.0 pro víkendové.
   */

  RETURN IF( WEEKDAY(d) < 5, -1.0, 1.0 );

END $$
```

5.1.4 Způsob zakódování výstupů sítě

Jak jsme uvedli v analýze úlohy optimalizace, úkolem neuronové sítě je napovědět, kterou trasou se vydat v dané situaci. Zřejmě nejlepším a asi jediným rozumným kódováním tohoto výstupu je vytvořit jeden neuron pro každou trasu. Při učení sítě bude mít výstup pro trasu vedoucí k nejlepšímu spoji jako požadovanou hodnotu 1.0, ostatní výstupy 0.0. Při používání sítě potom budou jednotlivé výstupy určovat pravděpodobnost, s jakou daná trasa povede k nejlepšímu spoji.

```
CREATE FUNCTION `mhd_nn_FormatDesiredOutput_TracesOnly`(
  iTracesCnt INT UNSIGNED,
  iSelectedTracePos INT UNSIGNED
) RETURNS text CHARSET utf8 BEGIN

  /**
    Vytváří řetězec čárkami oddělených požadovaných
    výstupních hodnot neuronové sítě.

    Očekává:
      Počet výstupních neuronů, počet tras, pozici vybrané trasy,
      vzdálenost vybrané stanice.

    Výstup:
      Řetězec typu "1,0,0,0,0,1,0,0,0,0,0,0".
      1 pro vybrané a 0 pro ostatní.

    Použití:
      SELECT mhd_nn_FormatDesiredOutput_TracesOnly( 4, 1 );
      -- Vrátí '1,0,0,0'
      SELECT mhd_nn_FormatDesiredOutput_TracesOnly( 4, 3 );
      -- Vrátí '0,0,1,0'
  */

  CALL LoggP('mhd_nn_FormatDesiredOutput',
   F2('mhd_nn_FormatDesiredOutput( iTracesCnt: {1}, iSelectedTracePos:
   {2})', iTracesCnt, iSelectedTracePos));

  -- Pokud, je vybraná trasa mimo povolený rozsah, zapíšeme varování.
```

```
  IF iSelectedTracePos < 1 OR iSelectedTracePos > iTracesCnt THEN
    CALL LoggP_warn('mhd_nn_FormatDesiredOutput', F2('iSelectedTracePos
  ({2}) out of range <1, {1}>.', iTracesCnt, iSelectedTracePos));
    SET iSelectedTracePos = -1;
  END IF;

  SET @saOutput = REPEAT('0,', iTracesCnt);
  SET @saOutput = INSERT( @saOutput, 2*iSelectedTracePos-1,2, '1,' );
  RETURN TRIM( TRAILING ',' FROM @saOutput);

END $$
```

Mezi přílohami jsou ještě další verze funkcí pro kódování vstupů a výstupů, například kódování času pomocí sinus a cosinus a kódování požadovaného vstupu i pro model sítě, který do výstupu zahrnuje i pozici výstupní zastávky.

5.1.4.1 Převod výstupů funkcí Softmax

Pro výstup by bylo vhodné použít funkci Softmax, která jednak zjistí, že součet hodnot výstupů je roven 1, a zadruhé zvýrazňuje rozdíly mezi jednotlivými výstupy, což je pro úlohu určení vhodnosti volby jednotlivých možností žádoucí. Kvůli zachování jednoduchosti implementace však funkci Softmax případně aplikujeme až na výsledné výstupy, nikoliv na hodnoty samotných neuronů. Rozdíl je v tom, že takto nebude mít Softmax vliv na učení. Vzorec funkce Softmax je:

$$p_i = \frac{e^{q_i}}{\sum_{j=1}^{c} e^{q_j}}$$

kde q_i je hodnota i-tého výstupu z množiny výstupů q, a c je počet všech výstupů.

```
CREATE PROCEDURE `neural_network`.`nn_Softmax`() BEGIN
  SELECT SUM(val) INTO @dSum FROM nn_Softmax;
  CREATE TABLE nn_Softmax_output
   SELECT id_neuron, val / @dSum AS val FROM nn_Softmax;
END $$
```

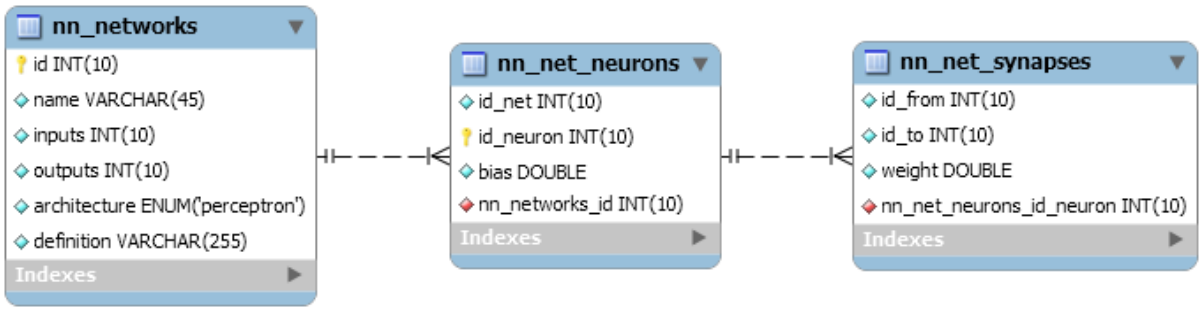
6 Implementace neuronové sítě

Jedním z cílů práce bylo vyzkoumat možnosti implementace neuronových sítí v jazyce SQL a ukládaných procedurách. To se jevilo jako obzvlášť výhodné pro situace, kdy velikost a počet sítí přesahují paměťové možnosti počítače. Výpočet potom probíhá v paměti počítače prostřednictvím nástrojů systému řízení relační databáze (DML, DDL).

Další výhody plynou z toho, že datová reprezentace neuronových sítí je okamžitě k dispozici pomocí standardního rozhraní SQL, které může využívat prakticky jakýkoliv klient. Data jsou navíc strukturovaná podle známého relačního modelu, proto je velmi snadné data využít k dalším účelům – např. pro genetické algoritmy, kdy křížení dvou sítí můžeme provést v jediném SQL dotazu. Kromě toho jsou některé databázové servery schopné distribuovat výpočet jednoho SQL dotazu na více procesorů a disků, někdy i na více serverů, čímž by se u velkých sítí dal využít koncept distribuce výpočtu, což je jinak u kompletně provázaných vrstev složité [22]. A nakonec, pokud využijeme některé knihovny pro objektově-relační mapování (ORM) jako jsou Hibernate nebo Oracle Toplink, dostáváme velmi efektivní ukládací prostor pro neuronové sítě, získáváme navíc možnost rychlého výpočtu v překládaném jazyce pomocí knihoven typu Joone, a zůstává nám možnost provádět hromadné operace s entitami pomocí HQL - jazyka podobného SQL. Další výhody implementace v SQL jsou popsané na [23]. Nyní k samotné implementaci.

6.1 Relační model neuronové sítě

Samotné jádro relačního modelu neuronové sítě je poměrně jednoduché. Pokud uvažujeme dopřednou síť bez zpožděných vstupů, vypadá model jako na obrázku 6.1. Tabulka `nn_networks` uchovává informace o jednotlivých sítích. V tabulce `nn_net_neurons` jsou jednotlivé neurony sítí a řádky v tabulce `nn_net_synapses` představují spojení (synapse) mezi neurony. Pomocí tohoto modelu je možné popsat perceptrony, po lehkých úpravách i složitější architektury, např. rekurentní sítě nebo sítě propojené mezi více vrstvami a nehomogenně propojené sítě. Z tabulky `nn_net_synapses` vedou do tabulky `nn_net_neurons` dva vztahy – jeden přes cizí klíč `id_from` pro výchozí neuron a jeden přes cizí klíč `id_to` pro cílový neuron. Na tuto dvojici sloupců je navíc uplatněno omezení unikátnosti – z jednoho neuronu do druhého vede jen jediná synapse.



Obrázek 6.1: ER diagram neuronových sítí

Na vstupní a výstupní neurony jsou napojené pseudosynapse, jejichž druhý konec má místo ID neuronu uloženou hodnotu `NULL`. Ty slouží k vyhledání vstupních a výstupních neuronů. Pokud bychom je hledali pomocí `LEFT JOIN`u, daly by se tyto záznamy ušetřit; časově by to ovšem bylo náročnější, protože k dohledání záznamů, na které nejsou napojené jiné záznamy přes klíč, vyžaduje zpracování všech záznamů levé i pravé tabulky.

6.2 Vytvoření neuronové sítě

Síť lze vytvořit prostým vkládáním záznamů do příslušných tabulek. Pro zjednodušení máme ale připravenou proceduru `nn_CreatePerceptron()`, která vytváří perceptron s daným počtem vstupů, výstupů a plně propojených skrytých vrstev s daným počtem neuronů.

```
CREATE PROCEDURE `nn_GeneratePerceptron`(
  sLayers VARCHAR(255), dWeightsRange DOUBLE, dBiasRange DOUBLE
)
```

Procedura `nn_CreatePerceptron()` přebírá čtyři vstupní a jeden výstupní parametr:

- `sLayers VARCHAR(255)` – čárkami oddělené počty neuronů na jednotlivých vrstvách. První číslo určuje počet vstupů, poslední číslo počet výstupů, a čísla mezi nimi určují počty neuronů ve skrytých vrstvách.
- `sName VARCHAR(255)` – název sítě; slouží jen pro orientaci vývojáře, nemá žádný sémantický význam.
- `dWeightsRange DOUBLE` – rozsah, ve kterém se mají generovat váhy; například hodnota 0,05 říká, že generované váhy mají být v rozsahu <-0,05; 0,05>.
- `dBiasRange DOUBLE` – obdobně jako pro váhy, určuje tento parametr rozsah hodnot, které mají být generované pro biasy.
- `OUT out_iNetId INT UNSIGNED` – výstupní parametr, do kterého se uloží vygenerovaný identifikátor nově vzniklé sítě.

Volání procedury vypadá následovně:

```
CALL nn_CreatePerceptron('2,2,1', 'XOR test', 0.05, 0.5, @out_NetID);
```

Tímto voláním vznikne síť, což si můžeme ověřit v tabulce `nn_networks`:

```
SELECT * FROM nn_networks;
```

Tabulka 6.1: Záznam v tabulce nn_networks, který reprezentuje nově vzniklou síť.

id	name	inputs	outputs	architecture	definition
2	'XOR'	2	1	'perceptron'	'2,2,1'

Procedura je ovšem jen obalem pro volání procedury `nn_GeneratePerceptron()`. Od té převezme vygenerovanou síť v dočasných tabulkách a zajistí jejich vložení do globální databáze entit.

Procedura `nn_GeneratePerceptron()` pracuje takto (zkráceno):

```
body: BEGIN

  CREATE  TEMPORARY TABLE nn_GeneratePerceptron_neurons(
    `id_neuron` int unsigned NOT NULL auto_increment,
    `bias` double NOT NULL,
    PRIMARY KEY  (`id_neuron`)
  ) ENGINE=Memory COMMENT='Neural net neurons';

  CREATE  TEMPORARY TABLE nn_GeneratePerceptron_synapses(
    `id_from` int unsigned,
    `id_to` int unsigned,
    `weight` double NOT NULL,
    UNIQUE KEY  (`id_from`,`id_to`)
  ) ENGINE=Memory COMMENT='Connections between neurons';

  CREATE  TEMPORARY TABLE nn_GeneratePerceptron_tmp(
    `id_neuron` int unsigned NOT NULL auto_increment,
    `bias` double NOT NULL,
    PRIMARY KEY  (`id_neuron`)
  ) ENGINE=Memory COMMENT='New layer neurons';

  CREATE  TEMPORARY TABLE nn_GeneratePerceptron_prev_layer(
    `id_neuron` int unsigned,  UNIQUE KEY (`id_neuron`)
  ) ENGINE=Memory COMMENT='IDs of previous layer neurons.';

  --  Vytvoříme pseudo-synapse do vstupní vrstvy.
  SET @iNeurons = CAST( @sLayers AS UNSIGNED );
  CALL lib_GenerateSequence(1, @iNeurons, 1);
  --  Vytvoříme virtuální vrstvu, do které první krok smyčky WHILE
  --  vytvoří synapse.
  INSERT INTO nn_GeneratePerceptron_prev_layer
   VALUES (NULL); -- Pouze jedna pseudosynapse na do každého neuronu.

  SET @iLayer = 0;
  create_layers:  WHILE @sLayers != '' DO

    --  Zjistíme počet jednotek v další vrsvě.
    SET @sHead = SUBSTRING_INDEX(@sLayers, ',', 1);
    SET @sLayers = SUBSTRING( @sLayers, LENGTH(@sHead)+2 );

    SET @iNeurons = CAST( @sHead AS UNSIGNED );
    IF NOT @iNeurons THEN ITERATE create_layers; END IF;
    SET @iLayer = @iLayer + 1;
```

```
    --- Vytvoření vrstvy ---

    -- Poslední vložené ID.
    SELECT IFNULL(MAX(id_neuron),0) INTO @iLastId
      FROM nn_GeneratePerceptron_neurons;
    -- Vytvoříme ID nových neuronů.
    CALL lib_GenerateSequence(@iLastId+1, @iLastId + @iNeurons, 1);
    -- Vložíme neurony s příslušnými ID.
    INSERT INTO nn_GeneratePerceptron_neurons
      SELECT NULL AS id_neuron, (RAND()-0.5)*2*dBiasRange AS bias
        FROM lib_GenerateSequence;

    -- Spojíme tuto novou vrstvu s předchozí vrstvou.
    INSERT INTO nn_GeneratePerceptron_synapses
      SELECT
        nn_GeneratePerceptron_prev_layer.id_neuron AS id_from,
        lib_GenerateSequence.i AS id_to,
        (RAND()-0.5)*2*dWeightsRange AS weight
      FROM nn_GeneratePerceptron_prev_layer
      CROSS JOIN lib_GenerateSequence;

    -- Nová vrstva se pro další kolo stává "předchozí" vrstvou.
    TRUNCATE nn_GeneratePerceptron_prev_layer;
    INSERT INTO nn_GeneratePerceptron_prev_layer
      SELECT i AS id_neuron FROM lib_GenerateSequence;

  END WHILE;

  -- Vytvoříme pseudo-synapse do výstupní vrstvy.
  INSERT INTO nn_GeneratePerceptron_synapses
    SELECT id_neuron AS id_from, NULL AS id_to, 1.0 AS weight
      FROM nn_GeneratePerceptron_prev_layer;
  -- Váhy pseudosynapsí nastavíme na 1.0.
  UPDATE nn_GeneratePerceptron_synapses
    SET weight = 1.0 WHERE id_from IS NULL;
  -- Biasy vstupních neuronů nastavíme na 0.0.
  UPDATE nn_GeneratePerceptron_neurons AS neu
    LEFT JOIN nn_GeneratePerceptron_synapses AS syn
      ON neu.id_neuron = syn.id_to
    SET bias   = 0.0 WHERE syn.id_from IS NULL;

END $$
```

6.3 Výpočet neuronové sítě

Výpočet sítě zajišťuje procedura `nn_ComputeNet()`.

```
CREATE PROCEDURE `nn_ComputeNet`(
  iNetId INTEGER, sadInput VARCHAR(255), bStoreInternalValues BOOLEAN
)
```

Procedura přebírá tři parametry:

- `iNetId INTEGER` – Identifikátor sítě, která se má vypočítat.
- `sadInput VARCHAR(255)` – serializované čárkami oddělené vstupy sítě.
- `bStoreInternalValues BOOLEAN` – zda se mají uchovat vypočítané výstupní hodnoty skrytých neuronů. Užitečné pro učení pomocí algoritmu backpropagation.

Volání procedury vypadá následovně:

```
CALL nn_ComputeNet(2, '-1, 1', TRUE);
```

Výsledkem volání je tabulka `nn_ComputeNet` s hodnotami na jednotlivých výstupech.

```
SELECT * FROM nn_ComputeNet;
```

Tabulka 6.2: Obsah tabulky `nn_ComputeNet` po volání procedury `nn_ComputeNet()`.

id_neuron	val
663	0.44019863481252

Pokud měl parametr `bStoreInternalValues` hodnotu `TRUE`, potom také vznikla tabulka `nn_ComputeNet_InternalValues` s hodnotami všech neuronů.

Tabulka 6.3: Obsah tabulky `nn_ComputeNet_InternalValues` po volání procedury `nn_ComputeNet()`.

id_neuron	val
659	-1
660	1
661	0.440817674829587
662	0.610166742015456
663	0.44019863481252

Procedura `nn_ComputeNet()` pracuje takto (zkráceno):

```
body: BEGIN

  --  Vytvoříme tabulku se vstupními hodnotami v řádcích.
  CALL lib_Explode(',', sadInput);
  CALL nn_GetInputNeuronIDs( iNetId );

  SET @iCntInputNeurons = (SELECT COUNT(*) FROM nn_GetInputNeuronIDs);
  SET @iCntInputValues  = (SELECT COUNT(*) FROM lib_Explode);
  IF @iCntInputNeurons != @iCntInputValues THEN  LEAVE body;  END IF;

  --  Vytvoříme pracovní tabulku a naplníme ji ID vstupních neuronů.
  CREATE  TEMPORARY TABLE nn_ComputeNet_prev_layer  (
    id_neuron INT UNSIGNED NOT NULL
    ,PRIMARY KEY USING HASH (id_neuron)
    ,val DOUBLE NOT NULL
  ) ENGINE = Memory  ROW_FORMAT = FIXED
  SELECT id_neuron, val  FROM lib_Explode AS ex
    LEFT JOIN nn_GetInputNeuronIDs AS inp USING(pos);

  --  Pokud máme ukládát hodnoty neuronů skrytých vrstev, učiníme tak.
  IF @bStoreInternalValues THEN
    DROP TEMPORARY TABLE IF EXISTS nn_ComputeNet_InternalValues;
    CREATE  TEMPORARY TABLE nn_ComputeNet_InternalValues (
      id_neuron INT UNSIGNED NOT NULL
      ,PRIMARY KEY USING HASH (id_neuron)
      ,val DOUBLE NOT NULL
```

```
    ) ENGINE = Memory  ROW_FORMAT = FIXED
      SELECT * FROM nn_ComputeNet_prev_layer;
  END IF;

  SET @iInputNeurons = (SELECT COUNT(*) FROM nn_ComputeNet_prev_layer);
  SET @iRound = 1;

  compute_layers: LOOP
    SET @iRound = @iRound +1;

    --  Vypočteme hodnoty následující vrstvy.
    CREATE  TEMPORARY TABLE nn_ComputeNet_new_layer  (
      id_neuron INT UNSIGNED NOT NULL
      ,PRIMARY KEY USING HASH (id_neuron)
      ,val DOUBLE NOT NULL
    ) ENGINE = Memory  ROW_FORMAT = FIXED
    SELECT neu.id_neuron,
      1 / (1 + EXP(- (SUM(inp.val * syn.weight)+neu.bias) ) ) AS val
      FROM nn_ComputeNet_prev_layer AS inp
      LEFT JOIN nn_net_synapses AS syn ON inp.id_neuron = syn.id_from
      LEFT JOIN nn_net_neurons  AS neu ON syn.id_to = neu.id_neuron
      WHERE neu.id_neuron IS NOT NULL
      GROUP BY(neu.id_neuron);

    --  Nastavíme počet "vstupních neuronů" následující vrstvy.
    SELECT COUNT(*) INTO @iInputNeurons FROM nn_ComputeNet_new_layer;
    --  Pokud žádné nejsou, ukončíme smyčku.
    IF 0 = @iInputNeurons THEN LEAVE compute_layers; END IF;

    --  Posun vrstev  --
    DROP TEMPORARY TABLE IF EXISTS nn_ComputeNet_prev_layer;
    ALTER TABLE nn_ComputeNet_new_layer RENAME TO
   nn_ComputeNet_prev_layer;

    --  Pokud máme ukládát hodnoty neuronů skrytých vrstev, učiníme tak.
    IF @bStoreInternalValues THEN
      INSERT INTO nn_ComputeNet_InternalValues
        SELECT * FROM nn_ComputeNet_prev_layer;
    END IF;

  END LOOP;
  ALTER TABLE nn_ComputeNet_prev_layer RENAME TO nn_ComputeNet;
END $$
```

Procedura má také modifikaci pro výpočet více sítí najednou s použitím stejného vstupního vektoru – viz příloha.

6.4 Backpropagation

Pro učení neuronové sítě jsme implementovali algoritmus backpropagation. Opět je experimentálně napsaný v jazyce SQL - ze stejných důvodů, z jakých jsme jej použili pro implementaci výpočtu neuronových sítí. Úpravu vah provádí procedura `nn_CorrectWeights()`:

```
CREATE PROCEDURE `nn_CorrectWeights`(
  id_net INTEGER, sadValuesWanted VARCHAR(255),
  dLearn DOUBLE, OUT out_dErrorSum DOUBLE )
```

Procedura pracuje takto: Předpokládá, že existují dočasné tabulky se skutečnými výsledky výpočtu sítě a s výsledky neuronů skrytých vrstev (`nn_CorrectWeights` a `nn_CorrectWeights_InternalValues`). Další postup je okomentovaný v následujícím zdrojovém kódu procedury (zkráceno oproti skutečnému kódu).

```
BEGIN body: BEGIN

  --  Vytvoříme tabulku s požadovanými hodnotami v řádcích. Výsledek:
  --  TEMPORARY TABLE nn_CorrectWeights_Wanted( id_neuron, val ).
  CALL lib_Explode(',', sadValuesWanted);
  ALTER TABLE lib_Explode RENAME TO nn_CorrectWeights_Wanted;

    --  Získáme ID výstupních neuronů.
    CALL nn_GetOutputNeuronIDs( id_net );

    --  Proměníme pozice požadovaných neuronů na ID neuronů této sítě.
    ALTER TABLE nn_CorrectWeights_Wanted
      RENAME TO nn_CorrectWeights_Wanted_pos;
    CREATE  TEMPORARY TABLE nn_CorrectWeights_Wanted (
      id_neuron INT UNSIGNED NOT NULL PRIMARY KEY,  val  DOUBLE NOT NULL
    ) SELECT out_neu.id_neuron, want.val
      FROM nn_CorrectWeights_Wanted_pos AS want
      LEFT JOIN nn_GetOutputNeuronIDs AS out_neu USING(pos);
    DROP TEMPORARY TABLE nn_CorrectWeights_Wanted_pos;

  --  Zjistíme počty požadovaných a vypočtených hodnot
  --  a počet výstupních neuronů.
  SELECT COUNT(*) INTO @iCntWanted FROM nn_CorrectWeights_Wanted;
  SELECT COUNT(*) INTO @iCntComputed FROM nn_CorrectWeights;
  SELECT COUNT(*) INTO @iCntOutputNeurons FROM nn_GetOutputNeuronIDs;

  --  Přejmenujeme vstupní tabulku na nn_CorrectWeights_Computed.
  ALTER TABLE nn_CorrectWeights RENAME TO nn_CorrectWeights_Computed;

  /*  Nyní máme k dispozici tabulky:
      nn_GetOutputNeuronIDs        (pos, id_neuron)
      nn_CorrectWeights_Computed   (id_neuron, val)
      nn_CorrectWeights_Wanted     (id_neuron, val)
      nn_CorrectWeights_InternalValues (id_neuron, val)
  */

  --  Vytvoříme tabulku pro data "aktuální vrstvy" -
  --  ID neuronu and rozdíl příslušné požadované a vypočtené hodnoty.
  CREATE  TEMPORARY TABLE `nn_CorrectWeights_cur_layer` (
    id_neuron INTEGER UNSIGNED NOT NULL PRIMARY KEY
    ,error DOUBLE NOT NULL DEFAULT 0.0
  ) ENGINE Memory   ROW_FORMAT = FIXED
  SELECT comp.id_neuron
    ,(want.val - comp.val)   *   comp.val * (1-comp.val) AS error
    FROM `nn_CorrectWeights_Computed` AS comp
    LEFT JOIN `nn_CorrectWeights_Wanted` AS want USING(id_neuron);

  --  Zjistíme celkovou chybu výstupní vrstvy.
  --  Použijeme ji níže pro řízení učebního procesu.
  SELECT SUM(ABS(error))/@iCntOutputNeurons INTO out_dErrorSum
    FROM `nn_CorrectWeights_cur_layer`;
```

```
  SET @iLayers = 0;
  layers: LOOP
    SET @iLayers = @iLayers + 1;

    --  Vytvoříme tabulku pro neurony předchozí vrstvy.
    --  Vypočítáme jejich chybu a vypočteme hodnotu
    --  derivace aktivační funkce neuronu.
    CREATE  TEMPORARY TABLE nn_CorrectWeights_prev_layer (
      id_neuron INTEGER UNSIGNED NOT NULL
       ,PRIMARY KEY USING HASH (id_neuron)
      ,error     DOUBLE NOT NULL
    ) ENGINE = Memory  ROW_FORMAT = FIXED
    SELECT upstream.id_neuron,  SUM( cur.error * syn.weight )  *
                      upstream.val * (1-upstream.val)  AS error
      FROM nn_CorrectWeights_cur_layer AS cur
        LEFT JOIN nn_net_synapses  AS syn ON syn.id_to = cur.id_neuron
        LEFT JOIN nn_CorrectWeights_InternalValues AS upstream
               ON syn.id_from = upstream.id_neuron
      WHERE upstream.id_neuron != 0
      GROUP BY upstream.id_neuron;

    --  Pokud před touto vrstvou nejsou žádné neurony, ukončíme smyčku.
    IF 0 = (SELECT COUNT(*) FROM nn_CorrectWeights_prev_layer)
      THEN LEAVE layers; END IF;

    --  Upravíme bias neuronů aktuální vrstvy.
    UPDATE nn_CorrectWeights_cur_layer AS cur
      LEFT JOIN nn_net_neurons AS neu USING(id_neuron)
      SET neu.bias = neu.bias + dLearn * cur.error;

    --  Upravíme váhy synapsí do neuronů předchozí vrstvy.
    UPDATE nn_CorrectWeights_cur_layer   AS cur
      LEFT JOIN nn_net_synapses      AS syn ON syn.id_to = cur.id_neuron
      LEFT JOIN nn_CorrectWeights_InternalValues AS iv
        ON syn.id_from = iv.id_neuron
    SET syn.weight = syn.weight + dLearn * cur.error * iv.val;

    ---  Posun vrstev  ---
    ALTER TABLE nn_CorrectWeights_prev_layer
      RENAME TO nn_CorrectWeights_cur_layer;

  END LOOP layers;

END body;
END $$
```

Po provedení procedury jsou váhy sítě upravené algoritmem backpropagation podle zadaných požadovaných výstupů a podle předaných tabulek se skutečnými výsledky.

6.5 Učení neuronové sítě

Proces učení neuronové sítě spočívá v opakovaném aplikování učícího algoritmu na vstupní vektory trénovací množiny – viz např. [24]. V našem případě tedy proces vypadá takto:

- Připravíme si trénovací množinu – vektory vstupů – jako SQL tabulku.
- Pro každý záznam této tabulky:

- Spočteme výstup neuronové sítě pomocí procedury `nn_ComputeNet()`, přičemž necháme ukládat hodnoty vnitřních neuronů.
- Zjistíme požadovaný výstup neuronové sítě a zakódujeme jej jako výstup sítě.
- Zjistíme úroveň chyby skutečného výstupu.
- Za určitých okolností (např. každé třicáté kolo) zjistíme střednědobý průměr úrovně chyby. Pokud dosáhne cílové úrovně (viz níže), učení ukončíme. V aktuální verzi implementace nevytváříme a nepoužíváme testovací množinu a raději všechny vektory používáme k učení.
- Voláním procedury `nn_CorrectWeights()` upravíme váhy sítě podle dat získaných v předchozích třech krocích cyklu.

- V případě, že zpracujeme všechny vektory vstupních hodnot, použijeme znovu stejnou množinu, ovšem v jiném pořadí.

Tento proces provádí procedura `mhd_nn_TeachStationNet()`.

6.5.1 Procedura `mhd_nn_TeachStationNet()`

```
CREATE PROCEDURE `mhd_nn_TeachStationNet`(
  iStationID INT UNSIGNED, iNetID INT UNSIGNED,
  dTargetError DOUBLE, iMaxCases INT UNSIGNED
)
```

Procedura přebírá čtyři argumenty:

1. `iStationID INT UNSIGNED` – ID stanice, která patří k neuronové síti, kterou chceme učit. Tato informace není v databázi explicitně obsažena, protože bychom museli buď vytvořit novou výpočty zdržující tabulku 1:N, či ještě hůř, přizpůsobit obecnou strukturu modelu neuronových sítí konkrétnímu problému.
2. `iNetID INT UNSIGNED` – ID neuronové sítě, kterou chceme učit. Opět uvádíme explicitně v parametru, protože na stejnou zastávku lze použít různé neuronové sítě (pochopitelně natrénované pro ni).
3. `dTargetError DOUBLE` – cílová úroveň chyby. Dosáhne-li během procesu učení střednědobý průměr úrovně chyby pod tuto úroveň, učení se ukončí.
4. `iMaxCases INT UNSIGNED` – maximální počet případů učení

Detaily principu fungování procedury jsou okomentované v následujícím zdrojovém kódu (zkráceno oproti skutečnému kódu):

```
body: BEGIN

  /**
   *   Trénuje neuronovou síť zastávky.
   *
   *   Dokud není dosaženo ukončujících podmínek,
   *   procedura pracuje v následující smyčce:
   *
   *   1) Vybere náhodné stání a zjistí jeho čas a pozici.
   *         Tímto způsobem získáme distribuci vzorků, která koreluje
   *         se skutečným rozložením spojů v dopravním systému.
   *   2) Zjistí nejlepší spojení hledáním do šířky.
   *   3) Vypočte výstup neuronové sítě zastávky.
   *   4) Naučí síť algoritmem backpropagation; požadované hodnoty
```

```
 *      vytvoří podle zjištěného nejlepšího spojení.
 */

DECLARE iJizdaID, iZastDestID INT UNSIGNED;
DECLARE iStaniPos INT;
DECLARE tTime TIME;
DECLARE dTime1, dTime2, dTimeDN, dLatNorm, dLonNorm DOUBLE;
DECLARE sZastNazev VARCHAR(255);

DECLARE bDone INT DEFAULT 0;
DECLARE curStani CURSOR FOR

  SELECT st.id_jizda   AS id_jizda, st.poradi AS pos_stani,
    st.cas2,
    mhd_nn_TimeInput1(st.cas2),
    mhd_nn_TimeInput2(st.cas2),
    mhd_nn_TimeInputDayNight(st.cas2),
    zast.lat_norm, zast.lon_norm,  zast.id AS id_zast, zast.nazev_plny
  FROM mhd_jizdy_stani AS st
    LEFT JOIN mhd_trasy_uzly AS tu  ON tu.id = st.id_trasa_uzel
    LEFT JOIN mhd_zast AS zast  ON tu.id_zast = zast.id
  WHERE tu.id_zast != iStationID  AND  st.cas2 IS NOT NULL
  ORDER BY RAND();

DECLARE CONTINUE HANDLER FOR NOT FOUND  SET bDone = 1;


--  Vybere k zastávce všechny procházející trasy, očísluje podle pořadí
--  pos INT,      id_trace INT,     onward_stops INT
CALL mhd_nn_GetStationTracesInfo(iStationID);

--  Vytvoří sekvenci pro proceduru mhd_nn_FormatDesiredOutput().
SELECT COUNT(*) INTO @iTracesCnt FROM mhd_nn_GetStationTracesInfo;
SELECT MAX(onward_stops), SUM(onward_stops)
  INTO @iMaxOnward, @iOnwardSum FROM mhd_nn_GetStationTracesInfo;
CALL lib_GenerateSequence( 1, GREATEST(@iTracesCnt, @iMaxOnward), 1 );
ALTER TABLE lib_GenerateSequence
  RENAME TO mhd_nn_FormatDesiredOutput_sequence;

--  Počet výstupů.
SET @iOutputCnt = @iTracesCnt + @iOnwardSum;

SET @dErrorSum = 0.0;
SET @iErrorNum = 0;
SET @dLearn = 0.05; -- Prvotní hodnoty nízké, pro určení znamének vah.

SET @iCases = 0;
rounds: BEGIN

  OPEN curStani;
  FETCH curStani INTO iJizdaID, iStaniPos, tTime, dTime1, dTime2,
             dTimeDN, dLatNorm, dLonNorm, iZastDestID, sZastNazev;
  WHILE NOT bDone DO

    SET @iCases = @iCases + 1;  IF @iCases > iMaxCases
      THEN LEAVE rounds; END IF;

    -- Hledáním do šířky najdeme nejrychlejší spoj do cílové zastávky.
    -- Ukázkový výsledek:
    -- id_zast   fastest     id_jizda_fastest        id_zast_from
    -- 465       04:35:00    13505                   115
```

```
      CALL mhd_VyhledejSpoje( iStationID, iZastDestID, ADDTIME(
        CAST( CURDATE() AS DATETIME ), CAST(tTime AS TIME)), 3);

      --  Uložíme údaje prvního spojení do proměnných.
      SELECT id_zast, id_zast_from, id_jizda_fastest
        INTO @idZastDest, @idZastFrom, @idJizda
        FROM mhd_VyhledejSpoje ORDER BY pos LIMIT 1;

      --  Zjistíme trasu vybrané jízdy.
      --  Momentálně jen pro první spoj, další ignorujeme.
      SET @iTrasaID = (SELECT id_trasa
                         FROM mhd_jizdy WHERE id = @idJizda);
      IF @iTrasaID IS NULL THEN CALL LoggP_error(
        'mhd_nn_TeachStationNet', F1('No trace for ride #%s',@idJizda));
        LEAVE rounds;
      END IF;

      --  Zjistíme, kolikátá trasa projíždějící touto zástavkou
      --  (řazeno podle ID) byla použita. Máme k dispozici tabulku
      --  mhd_nn_TSN_prochazejici_trasy / mhd_nn_GetStationTracesInfo.
      SET @iTrasaPos = (SELECT pos FROM mhd_nn_GetStationTracesInfo
                          WHERE id_trace = @iTrasaID);
      IF @iTrasaPos IS NULL THEN
        CALL lib_RaiseError(F2('mhd_nn_TeachStationNet(): Trace #{1}
         does not pass through station #{2}',NS(@iTrasaID), iStationID));
        LEAVE rounds;
      END IF;

  /*
      Nyní známe všechny informace potřebné k sestavení požadovaného
      výstupu sítě, který vypadá nějak takto:
      ( * == výstupní hodnota +1.0,  - == výstupní hodnota -1.0 )

          *   |  -  -  *  -  -  -
          -   |  -  -  -  -  -  -  -  -
          -   |  -  -  -  -

      Takový výstup by znamenal, že použijeme první trasu procházející
      zastávkou (řazeno podle ID trasy), a že bychom měli vystoupit
      na třetí zastávku od aktuální zastávky.

      Oddělená část - pořadí zastávky, kde vystoupit - není v konečné
      implementaci, protože určovat výstupní zastávku neuronovou sítí
      je příliš neefektivní. Zdrojový kód implementace, která výstupní
      zastávku určuje, je v příloze.
  */

    SET @saDesiredOutput =
      `mhd_nn_FormatDesiredOutput_TracesOnly`( @iTracesCnt, @iTrasaPos );

    ---  Vypočítáme výstup sítě pro tuto situaci.
    SET @saInput = CONCAT_WS(',',
                        dLatNorm, dLonNorm, dTime1, dTime2, dTimeDN );
    CALL nn_ComputeNet( iNetID, @saInput, TRUE );

    ---  Naučíme síť podle požadovaného výstupu.
    ALTER TABLE nn_ComputeNet RENAME TO nn_CorrectWeights;
    ALTER TABLE nn_ComputeNet_InternalValues
        RENAME TO nn_CorrectWeights_InternalValues;
    CALL nn_CorrectWeights(iNetID, @saDesiredOutput, @dLearn,
                              @out_dErrorSum);
```

```
    ---  Vypočítáme střednědobý průměr chyby.
      SET @dErrorSum = @dErrorSum + @out_dErrorSum, @iErrorNum =
            @iErrorNum + 1;

      ---  Pozorujeme průběh úrovně chyby.
      IF @iErrorNum >= 30 THEN
        SET @dErrorAvg = @dErrorSum / @iErrorNum;

        --  Pokud je střednědobý průměr chyby nižší než cílová úroveň,
        --  ukončíme smyčku.
        IF @dErrorAvg  <= dTargetError THEN LEAVE rounds; END IF;

        --  Nastavíme novou rychlost učení.
        SET @dLearn = INTERVAL(@dErrorAvg,
                 0.0005,     0.005,    0.01,      0.1,     1.0     );
        SET @dLearn = ELT(@dLearn+1,
            0.05,        0.1,     0.3,       0.5,     0.7,     1.0 );

        --  Vynulujeme proměnné pro výpočet střednědobého průměru.
        SET @dErrorSum = 0.0, @iErrorNum = 0;
      END IF;

      FETCH curStani INTO iJizdaID, iStaniPos, tTime, dTime1, dTime2,
                dTimeDN, dLatNorm, dLonNorm, iZastDestID, sZastNazev;
    END WHILE;
    CLOSE curStani;

  END rounds;

END $$
```

Pro kontrolu správné volby kódování vstupů sítě jsme napsali i proceduru, která počítá s kódováním času pomocí sin*(t)* a cos*(t)*. Zde jsou ty části její kódu, které se liší od té standardní:

```
CREATE `mhd_nn_TeachStationNet_CosSin`(
  iStationID INT, iNetID INT, dTargetError DOUBLE, iMaxCases INT )
body: BEGIN
...
DECLARE curStani CURSOR FOR
  SELECT st.id_jizda   AS id_jizda, st.poradi AS pos_stani, st.cas2,
      -mhd_nn_TimeInputCos(st.cas2),
      mhd_nn_TimeInputSin(st.cas2),
...
```

6.6 Efektivita implementace

Je zřejmé, že implementace pomocí databázových operací se nemůže rychlostí vyrovnat specializované implementací perceptronu v C++ nebo i s obecnou NN knihovnou v jazyce Java. Podle srovnávacích testů je SQL implementace zhruba padesátkrát až stokrát pomalejší než implementace v C++. Podrobnější srovnání viz následující tabulka:

Tabulka 6.4: Srovnání výkonu různých implementací perceptronu.

architektura	implementace			
	C++	Java (Joone)	SQL	SQL trénovací set
XOR 2,2,1	100	113	7810	8457
XOR 2,300,1	100	114	5031	5963
XOR 2,900,900,1	100	117	2407	2420

Tabulka obsahuje poměr hodnoty průměrného času tří měření, za jak dlouho daná implementace provedla 1000 kol učení na dané úloze, oproti referenční implementaci v C++. Testy proběhly na serverové sestavě (čtyřjádrový procesor Intel Xeon X3220 2.4 GHz 8 MB cache, 4× SAS disk 15 000 ot./min., softwarový RAID5) na 64-bitovém systému Linux (jádro v. 2.6.25). XOR jsme zvolili pro jednoduchost. Na vstupech a výstupech při testování výkonu implementace nezáleží – časová náročnost ani počet operací se nijak nemění.

V tabulce se dá vyčíst tendence ke snižování rozdílu mezi SQL implementací a srovnávacími implementacemi směrem k větším sítím. To můžeme přičíst na vrub tomu, že u větších sítí implementace v C++ i v Javě procházejí větším úsekem paměti, a při výpočtu hodnot neuronů další vrstvy musí velice často přecházet mezi paměťovými stránkami, čímž jde výkon výrazně dolů. Oproti tomu výkon SQL implementace na větších sítích výrazně neklesá, protože výpočet další vrstvy provádí jeden příkaz SELECT a databázové systémy jsou pro zpracovávání velkých objemů dat přímo určené; naopak u malých sítí vzniká poměrně velká režie (příprava dotazu, předávání volání mezi procedurami, přejmenovávání tabulek, kontrola práv a další náležitosti mohou v případě MySQL zabrat až 70 % času výpočtu [29].

Poslední sloupec v tabulce reprezentuje učení sítě provedené pomocí tabulek s připravenými trénovacími vektory. V něm se dá vyčíst tendence k výkonu srovnatelnému s trénováním bez připravených vektorů až u velkých sítí. Pravděpodobně to způsobuje dodatečná režie pro časté načítání trénovacích vektorů u menších sítí.

Efektivitu implementace posoudíme vzhledem k jejím cílům – viz úvod kapitoly 6. Měli jsme za cíl vytvořit robustní snadno škálovatelnou multiplatformní implementaci, která si poradí s mnoha velkými sítěmi s možností učení za běhu.

To se touto implementací zdařilo: Není problémem spočítat a učit síť prakticky libovolného rozsahu. Sítě jsou uložené v databázi, kde se s nimi také pracuje, proto není problém pracovat s různými neuronovými sítěmi, aniž by bylo nutné je neustále načítat z databáze a ukládat zpět. Tím se ušetří přenosové pásmo, sestavování sítě do datových struktur jiného jazyka a jejich zpětné rozkládání na SQL příkazy. Také je možné pracovat se všemi sítěmi paralelně – o rozdělení práce mezi procesory a případnou distribuci výpočtu mezi více počítačů se postará databázový server, a jeho clusterovací mechanismy se postarají o distribuci dat mezi úložiště.

I tak je ale výkon implementace relativně příliš slabý na to, aby jej bylo vhodné nasadit do časově kritické aplikace. Pomoci by mohla připravovaná podpora MySQL pro ukládané procedury psané v jazyce Java; záleží ještě na míře a způsobu podpory – tedy jestli bude možné v Java procedurách zpracovávat výsledkové sady (*result sets*). Dále jsou ve vývoji implementace úložišť (*storage engines*) pro MySQL specializované na rychlé zpracování neindexovaných dočasných tabulek – měly by být k dispozici pro MySQL 5.2 nebo 6.1. A nakonec, pro MySQL 6.1 bude pravděpodobně jednou z priorit zvýšení výkonu ukládaných procedur. Všechna tyto fakta dávají této implementaci šanci na výrazné zrychlení o desítky procent.

6.7 Úspěšnost učení

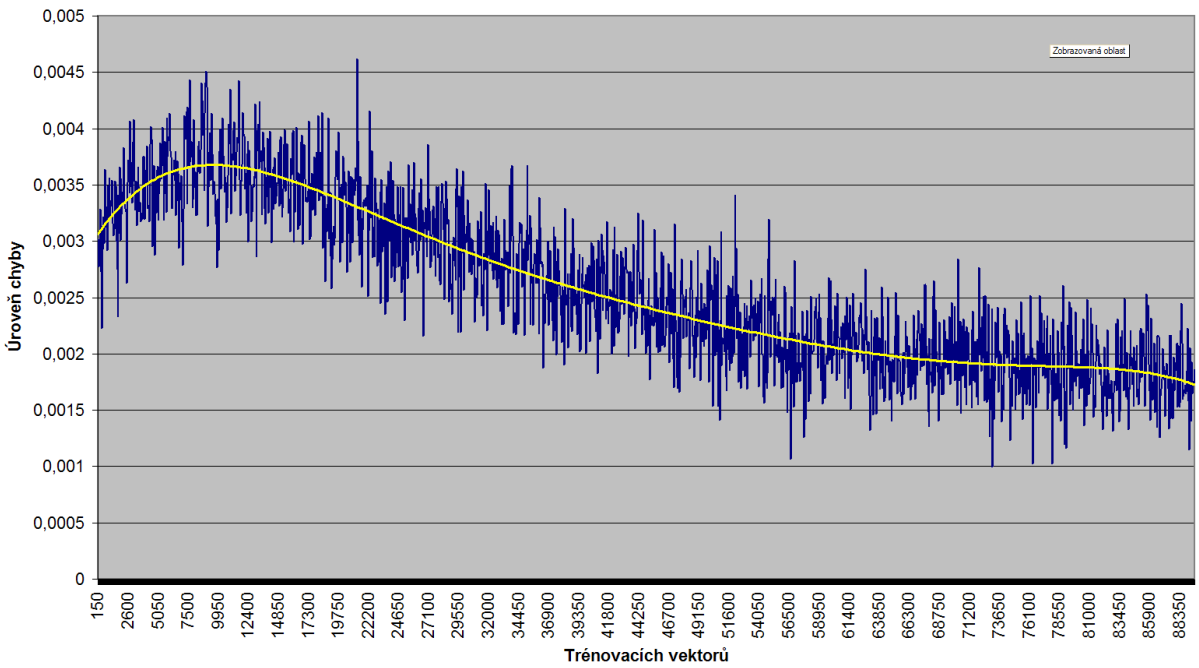
V době psaní této práce je hotová a otestovaná základní implementace algoritmu backpropagation. K učení se používá pouze inkrementální on-line stochastická metoda, případně se simulovaným žíháním. Není zatím implementovaná setrvačnost, a aktivační funkce všech neuronů je sigmoida se strmostí 1.0. Učení je tedy „zbaveno“ všech ověřených pomůcek, které učení urychlují. Jejich implementaci jsme záměrně odložili, protože v dosavadní sadě instrumentů, které dává k dispozici současná verze MySQL, chybějí prostředky pro jejich efektivní implementaci. To by se dalo obejít, nicméně zpomalení výkonu by pravděpodobně anulovalo kladný vliv rychlejšího naučení („rychlejšího“ ve smyslu „v méně kolech“).

Proto nelze očekávat rekordně rychlé naučení sítě. Konvergence úlohy XOR trvá kolem tří až pěti tisíc kol, v závislosti na zvolené strategii změn rychlosti učení a podle počátečního náhodného nastavení vah.

U složité úlohy, jakou je výběr vhodné trasy k dosažení nejlepšího spojení mezi danými zastávkami v daném čase, není zaručená konvergence ke globálnímu minimu. Je velmi pravděpodobné, že povrch účelové funkce (*objective function*) je plný hlubokých lokálních minim. Vzhledem k obrovské rozsáhlosti trénovacích dat (až 228 960 vzorků pro jednu zastávku, podle cílové přesnosti) a velké členitosti účelové funkce se dá očekávat velmi dlouhá doba učení.

Počet vzorků je roven počtu zastávek v systému bez jedné vynásobenému počtem časových úseků, na kolik si rozdělíme den. Systém rozlišuje čas s přesností na minuty, maximálně je tedy 1440 ($24 \cdot 60$) časových úseků. V databázi, na které provádíme testování, je 159 zastávek. Odtud maximální počet $24 \cdot 60 \cdot 158 = 228\,960$ vzorků.

Při pokusných učeních je tendence konvergovat evidentní. Křivka vývoje střednědobého průměru chybové funkce má typický průběh (viz obrázek 6.1): Velmi vysoká hodnota ihned po začátku, po několika málo vzorcích rychlý pokles, poté vzestup, a potom velice pozvolna klesající hodnoty, s jejichž poklesem zároveň snižujeme rychlost učení (*learn rate*).



Kresba 6.1: Pokles úrovně chyby během učení. První tři vzorky odebrány – dosahují přibližně desetinásobných hodnot oproti zbytku.

S tímto průběhem učení se síť zhruba po šedesáti tisících vektorů dostane do stavu, kdy dává vcelku rozumné výsledky a je možné uvažovat o jejím použití k optimalizaci vyhledávání v jízdních řádech. Bohužel těchto šedesát tisíc kroků učení trvá i na výše uvedené serverové sestavě desítky hodin. Paradoxně však učení nezdržuje implementace neuronových sítí, ale samotný předmět optimalizace. Abychom dostali trénovací vzorek, musíme vygenerovat cílovou zastávku a čas, a pro tuto kombinaci zjistit nejrychlejší spoj. Zjištění spoje zabírá z celého jednoho kroku učení zhruba 90 % času. Z toho důvodu postupoval výzkum učení poměrně dlouho.

Na druhou stranu se zde otevírá možnost využít fakt, že je síť v databázi neustále k dispozici k učení. Síť nemusíme nutně učit ve zvláštním odděleném prostředí. Místo toho ji můžeme zapojit do běžného procesu vyhledávání v reálném nasazení. Po nějaký čas ponecháme vyhledávání spojů čistě na algoritmu bez optimalizace a porovnáme jeho výstup s výstupem sítě – vypočítáme úroveň chyby. Dokud je úroveň chyby vysoká, síť učíme podle výsledků algoritmu. Jakmile bude dlouhodobě vykazovat dobré výsledky, využijeme je k optimalizaci hledání.

Tento přístup bude výhodný i pro případ drobných změn v jízdních řádech, které se dějí překvapivě často (viz podkapitola 2.2). Pokud dojde ke změně, opět můžeme začít kontrolovat správnost výběru trasy, kterou doporučuje neuronová síť, oproti algoritmu bez optimalizace. Nebude třeba vykonat algoritmus dvakrát – jednou s optimalizací, jednou bez. Místo toho stačí zvýšit počet kroků algoritmu tak, aby v pozdějších krocích obsáhl všechny trasy – tak zajistíme, že algoritmus opravdu najde nejrychlejší spojení. Princip optimalizace popisujeme v následující části.

6.8 Integrace výsledků neuronové sítě do algoritmu vyhledávání spojů

Začlenění výsledků neuronové sítě do algoritmu provedeme podobně, jako to dělá algoritmus A* – s tím rozdílem, že co A* dělá na úrovni grafů uzlu, my provedeme pro celé cesty, které představují trasy dopravního systému.

Obyčejný algoritmus hledání do šířky (viz 4.2.2) jednoduše z každé zastávky *Z* v aktuální množině zastávek prohledává všechny možnosti – do množiny prohledaných zastávek přidává všechny zastávky na všech trasách, které ze zastávky *Z* vedou. Optimalizovaná verze nejprve provede výpočet neuronové sítě zastávky *Z*, jehož výsledkem je pravděpodobnost, s jakou procházející trasy vedou k nejlepším spoji. Podle míry spolehlivosti neuronové sítě se v algoritmu rozhodne, které trasy nebude v tomto kole prohledávat. To lze provést například nastavením hranice v rozmezí v intervalu *<0,0; 1,0)* a z prohledávání vynecháme ty trasy, které neuronová síť ohodnotila pod tuto mez. Další možností je aplikovat na výsledky funkci Softmax (viz 5.1.4.1) a použít pouze trasy označené těmi výstupy, které tvoří část součtu nad touto hranicí, řazeno podle velikosti sestupně. Např. při hranici 0,5 a hodnotách 0,3, 0,1, 0,4 a 0,2 (součet 1,0) použijeme pro hledání do šířky pouze trasy, ke kterým náleží první a třetí výstup (součet 0,7).

Zajímavou vlastností algoritmu A*, která se do našeho algoritmu přenáší, je, že nemění výsledek výpočtu oproti neoptimalizované verzi, pokud jej necháme proběhnout až do splnění stejných ukončovacích podmínek (vyjma počtu kol). Pokud tedy z nějakého důvodu budeme považovat nalezený spoj za nedůvěryhodný, jednoduše můžeme pokračovat ve výpočtu bez ohledu na poradní hlas neuronové sítě. To lze v praktickém nasazení efektivně využít k neustálému učení sítě – viz konec předchozího bodu 6.7.

Zdrojový kód modifikované procedury `mhd_nn_VyhledejSpoje_NN()` je v příloze.

7 Další možnosti optimalizace

Heuristika využitím neuronových sítí je jen jednou z možností aplikace metod soft-computingu k urychlení vyhledávání spojů v jízdních řádech. V této kapitole navrhujeme některé další příležitosti, jak by paradigmata umělé inteligence mohla k této úloze přispět.

7.1 Genetické algoritmy

Teorie genetických algoritmů je inspirovaná principy, jakými „řeší úlohy“ příroda evoluční cestou – tedy prostřednictvím dědičnosti, mutace, křížení a přirozeného výběru.

7.1.1 Přímé využití pro hledání

Na úlohách typu vyhledání cesty grafem podle kritérií se obvykle používají tak, že jedinec představuje určitou cestu grafem (v našem případě spoj) a jeho geny jsou potom jednotlivé (navazující) hrany grafu.

7.1.1.1 Zápis vlastností jedince do chromozomu

V našem případě jsou životaschopní jedinci jen ti, kteří představují spoj, kterým se (obecně) dá dostat z dané výchozí zastávky do cílové zastávky počínaje v daném čase. Existence jedince, který by tuto vlastnost nesplňoval, prakticky nemá smysl (možnost, že by jeho další mutací vznikl životaschopný jedinec je mizivá). Jejich vzniku tedy předejdeme už při křížení.

Je tedy třeba explicitně volit místa, kde je vhodné dva chromozomy rozdělit a navzájem spojit. Je zřejmé, že tento předěl by měl být mezi geny, které představují hrany vedoucí ze stejného uzlu. V naší implementaci ovšem hrany fyzicky neexistují. Existuje jen sled uzlů a hrany existují pouze z definice. Například trasa je definována v databázové tabulce jako sled zastávek – v tabulce jsou tedy jen ID zastávek. Jedince tedy také zakódujeme jako sled uzlů:

v_1, v_2, v_3, v_3

Takový sled vrcholů reprezentuje trasu. Pokud budeme chtít pracovat rovnou s jednotlivými jízdami a tedy přímo vyhledávat spoje (nikoliv trasy), je třeba pracovat se sledem dvojic (vrchol, čas) :

$(v_1, t_1), (v_2, t_2), (v_3, t_3), \ldots$

Pokud bychom opět zamýšleli implementaci v SQL, v databázi by tyto dvojice představovaly jednotlivé záznamy v tabulce `mhd_jizdy_stani` (viz podkapitola 3.5). Genetický kód jedince by tedy byl v databázi uložen v tabulce s triviální strukturou:

```
id_jedinec INT UNSIDNED NOT NULL,
id_stani INT UNSIGNED NOT NULL
```

kde `id_stani` je cizí klíč do tabulky `mhd_jizdy_stani` a `id_jedinec` rozlišuje záznamy různých jedinců.

7.1.1.2 Operace křížení

Operace křížení bude probíhat nad dvojicemi jedinců, v jejichž chromozomech se vyskytuje aspoň jeden společný gen – tedy mezi spoji, které projíždějí stejnou zastávkou. V každé nalezené dvojici oba jedince za shodným genem rozpojíme a zkřížíme:

$(v_{a1}, t_{a1}), (v_{a2}, t_{a2}), \ldots, (v_{an}, t_{an}), (v_{an+1}, t_{an+1}), \ldots$

● $(v_{b1}, t_{b1}), (v_{b2}, t_{b2}), \ldots, (v_{bm}, t_{bm}), (v_{bm+1}, t_{bm+1}), \ldots$

$\rightarrow \{ ((v_{a1}, t_{a1}), (v_{a2}, t_{a2}), \ldots, (v_{an}, t_{an}), (v_{bm+1}, t_{bm+1}), \ldots),$

$((v_{b1}, t_{b1}), (v_{b2}, t_{b2}), \ldots, (v_{bm}, t_{bm}), (v_{an+1}, t_{an+1}), \ldots) \}$

Následně ověříme platnost obou nově vzniklých jedinců, tedy zda poslední čas v alele první části chromozomů je menší než v první alele druhé části chromozomů, tedy zda platí:

$$t_{an} < t_{bm+1} \quad \wedge \quad t_{bm} < t_{an+1}$$

Jedince, kteří tuto podmínku nesplňují, do další generace nezařadíme.

7.1.1.3 Fitness funkce, selekce, mutace

Pro základní požadavek na vyhledané spoje – co nejrychleji se dostat z výchozí do cílové zastávky – bude fitness funkce velmi jednoduchá a tedy i efektivní – vypočítá se jako rozdíl času poslední a první alely. Požadavek na co nejméně přestupů se dá zjistit jedním SQL dotazem podle počtu použitých unikátních jízd. Výpočtu fitness funkce, která by zahrnovala případné další požadavky, např. co nejnižší počet ujetých kilometrů nebo nejnižší cena jízdného, je nutné věnovat dodatečné implementační úsilí, obvykle by ale měly jít řešit jedním `SELECT` dotazem.

Pro selekci by bylo vhodné použít princip rulety a elitismu. Princip rulety vybíráme, protože chceme mít jistotu, že dáme šanci i řešením, která se od cíle nejprve vzdalují. Elitismus je vhodný proto, že v tomto typu úlohy rozhodně nechceme přijít o dosud nejlepší řešení.

Jediný typ mutace, který při výše uvedeném použití genetických algoritmů pro úlohu vyhledávání spojů v jízdních řádech dává smysl, je zkrácení chromozomu (od kraje), a připojení cíleně vygenerované sekvence genů na konec existujícího chromozomu. Jakákoliv jiná mutace (např. náhodná změna genu uvnitř chromozomu) by měla za následek vznik neplatného chromozomu.

7.1.1.4 Proč jsme GA nepoužili

Problémem (či možná spíše specifikem) genetických algoritmů, pokud bychom je použili výše navrženým způsobem, je, že by nastala silná tendence vytvářet spoje s mnoha přestupy. Pokud by totiž operace křížení rozpojovala a napojovala existující spoje, nutně by do další generace vznikali jedinci představující spoj s více přestupy. To by mohlo být výhodné u dopravní sítě, která s mnoha přestupy počítá – například pro dlouhé spoje přes několik malých států nebo pro cestování po státě s konkurenčním prostředím na trhu vlakové dopravy.

Bohužel však máme k dispozici pouze data jízdních řádů jednoho malého města, kde nejlepší spoj mezi libovolnými dvěma zastávkami má skoro bez výjimky vždy maximálně jeden přestup. Ani na dopravní síti MHD Brna či Prahy, které se úspěšně snaží o totéž, nebo na monopolní síti Českých drah, by genetické algoritmy nebyly příliš efektivní a neprojevil by se jejich optimalizační potenciál – na to bychom potřebovali např. jízdní řády Londýna, kde dopravu napříč městem z technických důvodů musí zajišťovat spoje s více přestupy.

7.1.2 Využití GA při trénování sítě pro heuristiku

Genetické algoritmy bychom také mohli použít jako alternativu k backpropagation při učení neuronové sítě, kterou využíváme pro heuristiku, jak je popsáno v kapitole 5. U některých úloh je učení křížením sítí efektivnější než backpropagation [27], s úspěchy se setkávají také hybridní algoritmy [28].

Genetické algoritmy však vyžadují ohodnocení jedinců. Jak jsme uvedli v podkapitole 6.7, ověření správnosti oproti nejlepšímu řešení je časově velmi náročné. Zatímco u backpropagation můžeme algoritmus nechat běžet do konce a výsledek použít pro učení s jistotou, že úroveň chyby sítě po učení přinejhorším nebude o moc horší, u geneticky vzniklé sítě si tím jisti být nemůžeme a bylo by ji třeba ohodnotit na více zadáních, nebo ji testovat při dalších hledáních při ostrém provozu. To by znamenalo např. vždy vzít více jedinců (sítí) a vyzkoušet, jak by se algoritmus hledání do šířky choval s jejich použitím. Při tak velké časové náročnosti algoritmu to ovšem nepřipadá v úvahu. Ani geneticky trénované sítě, ani hybridní trénování tedy pro tuto úlohu nejsou vhodné metody.

7.2 Hopfieldova síť

Druhým typem neuronové sítě, která by mohla posloužit účelu vyhledávání spojů, je Hopfieldova síť. Ta se často používá k hledání suboptimálních řešení Úlohy obchodního cestujícího. V podobném duchu by ji bylo možné využít i pro hledání spojů v jízdních řádech.

Obecně vzato Hopfieldovy sítě hledají energeticky nejstabilnější stav, tedy ve stavovém prostoru má tendenci ustálit se na místě s nízkou hodnotou energetické funkce ([25] str. 79). Úlohu obchodního cestujícího řeší spojitá Hopfieldova síť. Její strukturu a energetickou funkci bychom mohli upravit podobně jako pro problém obchodního cestujícího, avšak bez požadavku na to, aby cesta tvořila cyklus. Pokud bychom jako vrcholy Hopfieldovy sítě použili zastávky a jejich pozice v maticové struktuře sítě interpretovali jako pořadí, v jakém je nutné zastávkami projet k dosažení suboptimálního času, mohla by Hopfieldova síť řešit vyhledávání v jízdním řádu.

Tato možnost ale vyvolává více otázek než odpovědí. Jak konkrétně by bylo třeba energetickou funkci a strukturu sítě upravit, jak by se měly nastavovat váhy během adaptační fáze a jak se vyrovnat s nesymetričností spojů v jízdních řádech, to ponecháme jako předmět případnému dalšímu zkoumání. V této práci se tím hlouběji zabývat nebudeme.

7.3 Teorie mraveniště

Studenti ubytovaní na kolejích Pod Palackého vrchem vědí, že pokud nechají na zemi např. obal od Tatranek, po jisté době se v něm objeví první život. Nemyslíme tím ovšem cokoliv mikroskopického, ale mravence. Po pokojích se tu a tam projde mravenec, hledajíc, co by donesl do mraveniště. A když najde obal od Tatranek plný čokolády (mravenci milují sladké), vezme vzorek, jde po své stopě zpátky a zanechává na ní nové pachové značky, které znamenají „tudy k jídlu“. Po nějaké době se pokojem po linoleu klikatí neoptimální trasa od obalu k dutině v panelu. Čas od času se však nějaký mravenec jdoucí směrem k potravě od hlavního proudu oddělí a jde náhodnou cestou. Pokud má štěstí, najde kratší cestu. Tu potom vyzkoušejí další mravenci a zanechávají na ní stopu. Po kratší cestě se mravenci častěji vracejí s potravou (kratší cesta → více „otoček“ za stejnou dobu), proto jsou feromonové stopy na kratší cestě stále silnější a vydá se po ní více mravenců. Postupně od delší cesty všichni mravenci upustí. Cestu tak zoptimalizovali.

Teorie mravenišť představuje poměrně novou metodu optimalizace. Inspirace pochází z chování skutečných mravenčích kolonií. Aplikací jednoduchých pravidel, jimiž se řídí jednotliví jedinci, vzniká komplexní chování celku, schopné řešit složité optimalizační úlohy. Počet úspěšných aplikací těchto algoritmů exponenciálně roste zejména v oborech optimalizace, komunikačních sítí a robotiky [5].

Teorii mravenišť je možné aplikovat i na úlohu vyhledávání spojů v jízdních řádech. Opět by byly nutné úpravy algoritmů vzhledem k „jednosměrné“ povaze problému – návrat mravence od potravy (cílové zastávky) do mraveniště (výchozí zastávky) by hodnocení jeho úspěšnosti ovlivnil.

Řekněme tedy, že mravenci by se náhodně rozešli z výchozí zastávky hledat cílovou. Jeden by ji našel a v nulovém čase by se vrátil po své trase zpět, přičemž by ji označil jako cestu k cíli. Jiný mravenec by na ni narazil poté, co do stejné zastávky na trase nalezené prvním mravencem dorazil kratší cestou. Stejným principem, který jsme popsali výše, se cesta postupně optimalizuje na nejkratší.

Zádrhel tohoto přístupu je v tom, že takto řešení v podstatě redukujeme na algoritmus hledání do šířky s „náhodnou heuristikou“ – mravenec náhodně vybírá, kudy se vydat. Kromě toho, teorie mravenišť nepočítá s tím, že by se jednotliví mravenci učili, a odsimulovat cestování mravenců dopravním systémem při každém hledání spoje je přinejmenším neefektivní.

Z těchto důvodů jsme od aplikace této větve soft-computingu na naší úlohu upustili.

8 Závěr

8.1 Výsledky práce

Vyhledávání spojů v jízdních řádech je specifická modifikace úlohy hledání nejkratší cesty v grafu, která vyžaduje pozměnit algoritmy běžně k tomu využívané. V této práci jsme úlohu analyzovali, navrhli a implementovali algoritmus hledání a otestovali jej na reálných datech, která jsme získali z dopravního podniku menšího města. Testovací data bylo třeba importovat, opravit a převést do podoby vhodné k algoritmickému zpracování.

Práce se také zabývala možnostmi optimalizace vyhledávacího algoritmu pomocí metod soft-computingu. Nejperspektivnější metodu – heuristiku pomocí neuronové sítě – jsme vybrali k implementaci.

Obě implementace jsou naprogramované v jazyce SQL a jeho procedurálních rozšířeních. Podle očekávání sice výkon implementace v databázovém systému MySQL 5.0 nedosahuje rychlosti implementací v jazycích, které se pro takové úlohy běžně používají, ale získali jsme jiné výhody (popsané v práci). S vývojem MySQL, případně při přechodu na RDBMS více optimalizovaný pro práci s daty v paměti, je pravděpodobné značné zrychlení. Navíc jsme otevřeli slibnou cestu paralelizaci výpočtů zajištěné přímo databázovým systémem. Implementace neuronové sítě v SQL vzbudila zájem v komunitě vývojářů umělé inteligence a v komunitě testerů MySQL a možná se po úpravách stane jedním z tzv. *benchmarků* MySQL pro ukládané procedury a práci s paměťovými úložišti.

Spoluprací s dopravními podniky a se společností vyvíjející v České republice největší (a víceméně jediný kompletní) informační systém jízdních řádů IDOS a správcem Celostátního informačního systému o jízdních řádech, firmou Chaps s.r.o., se podařilo promítnout teoretické předpoklady oproti reálným potřebám uživatelů jízdních řádů – dopravců na jedné straně, cestujících na druhé straně, a správci databáze mezi nimi. Tato spolupráce umožnila udržet akademickou práci v podobě, kdy ji bude možné použít jako základ reálně nasazeného systému pro správu dat jízdních řádů a vyhledávání spojů v nich.

V předposlední kapitole jsme se zabývali možnostmi použití několika dalších metod soft-computingu pro optimalizaci vyhledávání v JŘ, a navrhli možné použití některých z nich.

8.2 Další vývoj

Další možný vývoj projektu je implementace rozhraní pro správu dat jízdních řádů, testování vývojových větví MySQL (zejména možnosti propojení s jazykem Java a částí optimalizované pro malé tabulky v paměti), experimentální přepsání kódu pro databázový systém Oracle či MS SQL, implementace dalších heuristických metod. a v případě, že rychlost vyhledávání bude vyhovující a podaří se získat pravidelný přísun aktuálních dat jízdních řádů, je v plánu i vytvoření veřejného webového uživatelského rozhraní pro vyhledávání spojů.

Literatura

[1] Vyhláška Ministerstva dopravy a spojů o jízdních řádech veřejné linkové osobní dopravy. Dokument dostupný na URL http://www.pravnipredpisy.cz/predpisy/ZAKONY/ 2000/388000/ Sb_388000_------_.php (leden 2008)

[2] Zákon č. 111/1994 Sb. o silniční dopravě. Dokument dostupný na URL http://www.lexdata.cz/lexdata/sb_free.nsf/c12571d20046a0b20000000000000000/c12571d20046a0b2c12566d40074a3e7?OpenDocument (leden 2008)

[3] Božek, L., Hakrová, J.: Celostátní informační systém o jízdních řádech. Dokument dostupný na URL: http://www.mvcr.cz/casopisy/s/2000/0008/8konz.html (leden 2008)

[4] Ministerstvo dopravy České republiky: Jednotný datový formát JŘ. 2000. Dokument dostupný na URL: http://www.mdcr.cz/NR/rdonlyres/80947B72-4490-402E-A7FD-4C074918CA7C/0/PopisJDFverze1_9.rtf (leden 2008), k dispozici v elektronické příloze.

[5] Němec, M.: Mravenčí kolonie - biologická inspirace. Dokument dostupný na URL: http://www.milosnemec.cz/clanek.php?69 (leden 2008).

[6] Dijstrův algoritmus. Dokument dostupný na URL: http://cs.wikipedia.org/wiki/Dijkstr%C5%AFv_algoritmus (leden 2008).

[7] Mařík, V., Štěpánková, O., Lažanský, J.: Umělá inteligence. Praha, Academia 2003, ISBN 80-200-1044-0

[8] Russell, S. J., Norvig, P.: Artificial Intelligence, A Modern Approach. Upper Saddle River, Prentice Hall, Inc. 1995. ISBN 0-13-360124-2.

[9] Winston, P. H.: Artificial Intelligence. Addison Wesley, Reading 1992. ISBN 0-201-53377-4.

[10] Shafranovich Y.: RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files. Boston, SolidMatrix Technologies, Inc. 2005. Dokument dostupný na URL: http://tools.ietf.org/html/rfc4180 (leden 2008)

[11] Languages used in information technology. Dokumenty dostupné na URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_ics_browse.htm?ICS1=35&ICS2=60 (květen 2008).

[12] MySQL 5.0 Reference Manual. Dokument dostupný na URL: http://dev.mysql.com/doc/refman/5.0/en/ (květen 2008)

[13] Vontobel, B.: The Lost Art of the Self Join. Dokument dostupný na URL: http://en.oreilly.com/mysql2008/public/schedule/detail/794 (květen 2008)

[14] Kruckenberg. M., Pipes, J: Essential SQL. Berkeley, Apress 2005. ISBN 978-1-59059-505-3

[15] Makhfi, P.: Introduction to Knowledge Modeling. MAKHFI.com 2006. Dokument dostupný na URL: http://www.makhfi.com/KCM_intro.htm (duben 2008).

[16] Pollack. J. B.: Connectionism: past, present, and future. Artificial Intelligence Review, 1989. ISSN 0269-2821

[17] Minsky, M., Papert, S.: Perceptrons, Perceptrons: An introduction to computational geometry, The MIT Press, Cambridge 1969. ISBN 0262-630222

[18] Heaton, J.: Introduction to Neural Networks with Java. Heaton Research Inc., 2008. ISBN 097732060X

[19] Sartori, M. A.: A Simple Method to Drive Bounds on the Size and to Trim Multilayer Neural Network. IEEE Trans on NN, 2, 1991.

[20] Tamura, S.: Method of determining an optimal number of neurons contained in hidden layers of a neural network [patent]. Nippondenso Co., Ltd., 1994. United States Patent 5596681, vydán 1997.

[21] Sarle, W. S.: comp.ai.neural-nets FAQ. 1997-2002. Dokument dostupný na URL: http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-10.html (květen 2008).

[22] Kontár, S.: Paralelní trénování neuronových sítí pro rozpoznávání řeči [diplomová práce]. Brno, Vysoké učení technické 2006.

[23] Žižka, O.: Why implement neural networks in SQL. Dokument dostupný na URL http://ondra.zizka.cz/stranky/programovani/neural_networks_in_sql_stored_procedures/index.texy (únor 2008)

[24] Mehrotra, K.: Artificial neural network. Cambridge, MIT Press 1997. ISBN 0-262-13328-8

[25] Šíma J., Neruda, R.: Teoretické otázky neuronových sítí. Praha, matfyzpress 1996.

[26] COM: Component Object Model Technologies. Dokument dostupný na URL: http://www.microsoft.com/com/default.mspx (květen 2008)

[27] Gupta, J., Sexton, R.: Comparing backpropagation with a genetic algorithm for neural network training. Chichester, Omega 1998.

[28] McInemey, M., Dhawan, A. P.: Use of Genetic Algorithms with Back Propagation in Training of Feed-Forward Neural Networks. In: IEEE International Conference on Neural Networks, 2, 1993, s. 1092-1095. ISBN 0-7803-0999-5

[29] Widenius, M., Pipes, J.: The Future of MySQL. In: MySQL User's Conference 2008. Dokument dostupný na URL: http://en.oreilly.com/mysql2008/public/asset/attachment/2250

[30] Bussieck, M.R., Winter, T., Zimmermann, U.T.: Discrete optimization in public rail transport. Mathematical Programming, 79, 1997, s. 415-444. Dokument dostupný na URL http://www.springerlink.de/content/hr315633l6566r23/fulltext.pdf.

Seznam příloh

Příloha 1. CD se zprávou v elektronické formě a s vybranými zdrojovými kódy implementace.