FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

KIV/TI - Teoretická informatika

Převod nedeterministického automatu na deterministický

Jaroslav Klaus (A13B0347P), Vladimír Láznička (A13B0371P)

18. ledna 2015, Plzeň

Obsah

1 Zadání

Implementujte algoritmus pro převod nedeterministického koneèného rozpoznávacího automatu (umožněte i existenci e-hran) na ekvivalentní deterministický automat. Navrhněte vhodný formát vstupních a výstupních dat.

Program odlaďte alespoň na 6 příkladech včetně příkladů prezentovaných na přednáškách a cvičeních.

Všechny testovací příklady uveďte v dokumentaci včetně ručního řešení.

1.1 Formát vstupních a výstupních dat

Jako formát vstupních a výstupních dat budeme volit textové soubory `*.TI` odpovídající definici nedeterministického konečného rozpoznávacího automatu (vstup) a deterministického konečného rozpoznávacího automatu (výstup) na stránce `http://home.zcu.cz/~vais/formaty.htm`. Cílem výstupu je pak možnost jej využít pro další zpracování (např. minimalizace) daného automatu dalším programem.

2 Analýza

Řešení úlohy lze rozdělit do následujících celků - načtení dat ze vstupu a jejich zpracování do příslušné struktury, samotný převod automatu podle dané reprezentace na deterministický typ, uložení dat vzniklých z převodu do příslušné struktury a nakonec výpis výsledku na výstup.

2.1 Zpracování vstupu

Pro zpracování vstupu bude zapotřebí připravit si funkci nebo metodu, která provede parsování vstupu na jednotlivé řetězce, ze kterých se poté získají hodnoty důležité pro reprezentaci automatu a jeho následný převod. Formát vstupního souboru bude naprosto zásadní dodržet, neboť v opačném případě může dojít k chybnému převodu nebo také k němu nemusí dojít vůbec. Vstupní soubor nám udává počet stavů automatu, velikost množiny vstupních symbolů, přechodovou tabulku automatu a nakonec výpis vstupních a výstupních stavů. Tyto informace bude třeba uložit do struktury nebo objektu reprezentující daný automat. Jako reprezentace automatu se pak využije zmíněná přechodová tabulka, která v sobě drží defacto všechny potřebné informace k jeho převodu na deterministický typ. Ostatní informace poslouží buď k vytvoření dalších celků (jako velikost polí na základě počtu stavů) nebo k závěrečné části převodu - určení vstupního stavu a výstupních stavů deterministického automatu. Samotný seznam stavů nebude ke způsobu zápisu vstupního souboru potřeba, neboť stavy vždy odpovídají velkým písmenům a jsou řazené podle abecedy (obdobně to platí pro množinu vstupních symbolů, nicméně tu nebudeme pro převodu automatu přímo potřebovat).

2.2 Převod nedeterministického automatu na deterministický

Převod bude probíhat pomocí přechodové tabulky, která se bude postupně upravovat, aby se z ní odstranily všechny nedeterminismy (více vstupních stavů, nejednoznačné přechody a přítomnost e-hran). To nám zajistí jistou inuitivnost a umožní snazší porovnání s ručním řešením, které bude rovněž prováděno na základě přechodové tabulky.

Při vytváření tabulky deterministického automatu se nejprve použije první řádek tabulky z původního automatu, který se bude procházet položku po položce, z nichž se vyberou ty stavy, které ještě nemáme zaznamenány (na počátku máme pouze jeden vstupní stav). Pokud bude položka obsahovat více stavů, tyto stavy se spojí v jeden nový stav a ten se zaznamená. Pro každý takto zaznamenaý stav se vytvoří další řádek, jeho položkky budou obsahovat stavy, do kterých bychom se z něj dostali v daném nedeterministickém automatu pomocí příslušného vstupního znaku. Pokud byl nově vzniklý stav z více původních stavů, v položkách pro tento stav budou zaznamenány stavy, do kterých se lze daným znakem dostat ze všech těchto původních stavů. Takto se bude pokračovat, dokud nebudou nalezeny všechny nové stavy a pro ně vytvořeny příslušné položky.

V případě více vstupních stavů se tyto stavy na počátku přechodu spojí v jeden a takto vzniklý stav se použije jako první zaznamenaný. Pokud bude nedeterministický automat obsahovat e-hrany, což je indikováno další položkou pro příslušný stav v původní přechodové tabulce (jejich počet pak o 1 přesahuje uvedenou velikost množiny vstupních znaků), budeme vytvářet navíc tzv. tabulku e-následníků, která bude pro každý stav z původního automatu uvádět, do jakých dalších stavů se lze dostat prostřednictvím e-hrany (včetně jeho samého).

Tyto stavy se pak sjednotí do jednoho nového, který bude mít vlastnosti všech v sobě obsažených stavů a bude reprezentovat původní stav, pro nějž se tito e-následníci zjišťovali.

Jakmile budeme mít vytvořenou přechodovu tabulku deterministického automatu, určíme pomocí seznamu výstupních stavů z nedeterministického automatu, jaké stavy budou výstupní v deterministickém automatu. Budou to ty, které v sobě obsahují nějaký výstupní stav z původního automatu. Jako stav vstupní se použije buď ten původní, pokud byl jeden, nebo stav vzniklý sjednocením několika původních stavů, pokud jich bylo více.

2.3 Uložení parametrů deterministického automatu a vypsání výstupu

Vzhledem k tomu, že z předchozí části již budeme mít všechny potřebné informace - počet stavů, přechodovou tabulku, vstupní stav a výstupní stavy, lze je jednoduše uložit do stejné struktury nebo objektu jako u nedeterministického automatu. Pak už jen stačí použít vhodnou funkci nebo metodu, která si tyto informace vezme a zapíše je do souboru v daném formátu (v zásadě bude fungovat přesně opačně než funkce pro načtení dat ze souboru).

3 Implementace programu

K implementaci programu jsme se rozhodli použít jazyk Java, který je jednak výhodný svým objektovým přístupem a také obsahuje několik knihovních tříd umožňující používání různých struktur jako seznamy apod., anož bychom je museli sami implementovat. Nevýhodou je pak samozřejmě pomalejší běh než např. při použití jazyka C, ale pro náš případ není vysoká rychlost zpracování až tak důležitá.

3.1 Objekt Automaton

Tento objekt slouží k uchování informací o automatu ve svých atributech. Těmito atributy jsou:

- `String automatonType` - řetězec obsahující zkratku typu automatu
- `int statusCnt` - počet stavů automatu
- `int inputCnt` - velikost množiny vstupních znaků
- `String[][] automatonTable` - pole uchovávající přechodovu tabulku automatu, každý řádek přísluší jednomu stavu (A... Z) a každý sloupec jednomu vstupnímu znaku (a... z).
- `ArrayList<String> inputStatuses` - seznam se vstupními stavy automatu
- `ArrayList<String> outputStatuses` - seznam s výstupními stavy automatu

Dále má samozřejmě metody pro ukládání a vracení těchto atributů, kde je to třeba. Konstruktor pak jako parametry přijímá zkratku automatu, počet stavů a počet vstupních znaků.

3.2 Načtení souboru a uložení dat

Načtení souboru je řešeno pomocí metody `static createAutomatonFromFile(String filePath)`, která přebírá jako parametr řetězec s cestou ke vstupnímu souboru a je obsažena ve třídě `Input_Output`. Metoda používá ke čtení souboru knihovní třídu `BufferedReader`, zejména její metodu `readLine()`, pomocí které získa řetězec představující obsah aktuálně načítaného řádku. Ten je pak rozdělen buď ručně nebo pomocí metody `split()`, přičemž jako dělící znak je použita mezera. Nejprve se načte typ automatu, počet stavů a počet vstupních znaků a tyto hodnoty se poté použijí k vytvoření objektu `Automaton`. Následně se načte přechodová tabulka a postupně uloží do dvourozměrného pole řetězců, které se pak objektu předá. Nakonec se načtou vstupní a výstupní stavy do příslušných seznamů a rovněž se předají objektu.

3.3 Algoritmus převodu

- doplnit -

3.4 Uložení výsledného automatu a výpis na výstup

Výsledný deterministický automat je vytvořen na konci metody pro převod. Nejprve se vytvoří objekt tohoto automatu, přičemž jako parametry jsou použity řetězec "DKAR"(deterministický konečný automat rozpoznávací - dle požadovaného formátu souboru), velikost seznamu s vytvořenými stavy a počet vstupních znaků z původního automatu (ten se převodem nijak nemění). Poté se uloží pomocí metody `static void setOuputStatusesToDka(Automaton nka, Automaton dka, LinkedList<String> statuses)` seznam výstupních stavů automatu postupem víceméně popsaným v analýze. V metodě také dojde k přejmenování stavů tak, aby vyhovovaly formátu souboru, který bude výstupem. Přejmenování probíhá procházením vytvořených výstupních stavů, které mohou být v tu chvíli označené jako složení původních stavů, přičemž se tyto stavy porovnávají se seznamem všech nově vytvořených stavů a jakmile dojde ke shodě, složený stav se přejmenuje podle formule `'A'+ index stavu v seznamu vytvořených stavů`. K podobnému přejmenování dojde i ve vytvořené přechodové tabulce v metodě `static void renameStatuses(String[][] dkaTable, LinkedList<String> statuses)`. Nakonec se tabulka předá objektu automatu a tytvoří se list s jednim stavem `"A"`, který je uložen jako vstupní (prezentuje množinu původní vstupních stavů).

Objekt se pak použije k výpisu informací do výstupního souboru pomocí metody `static void writeAutomatonToFile(Automaton a, String filepath)` ze třídy `Input_Output`. V zásadě funguje opačně než metoda pro získání dat ze souboru. Postupně skládá informace získané z objektu automatu do řetězců, představující jednotlivé řádky (podle požadovaného formátu) a ty pomocí metody `write(<retezec>)` z knihovní třídy `BufferedWriter` zapisuje do zadaného souboru. -doplnit-

4 Uživatelská dokumentace

-doplnit, podle domluvy-

5 Porovnání s ručním řešením převodu a ověření správnosti

-doplnit-

6 Závěr

-doplnit-

Řešení zadané úlohy lze rozdělit do několika funkčních celků – rozhodnutí, čím se bude **síť reprezentovat v programu**, resp. paměti počítače (soustava polí nebo řada na sebe navazujících spojových seznamů apod.), zjištění **parametrů výsledné neuronové sítě** (jako kolik bude mít vrstev a kolik neuronů bude obsahovat každá vrstva), **sestavení její struktury** na základě údajů získaných z předchozí části s využitím vhodných struktur, **uložení dat přečtených ze vstupních souborů do zmíněných struktur** a nakonec **výpočet aktivačních hodnot na základě přečtených dat** a poskytnutého výpočetního vzorce.

6.1 Reprezentace sítě v programu

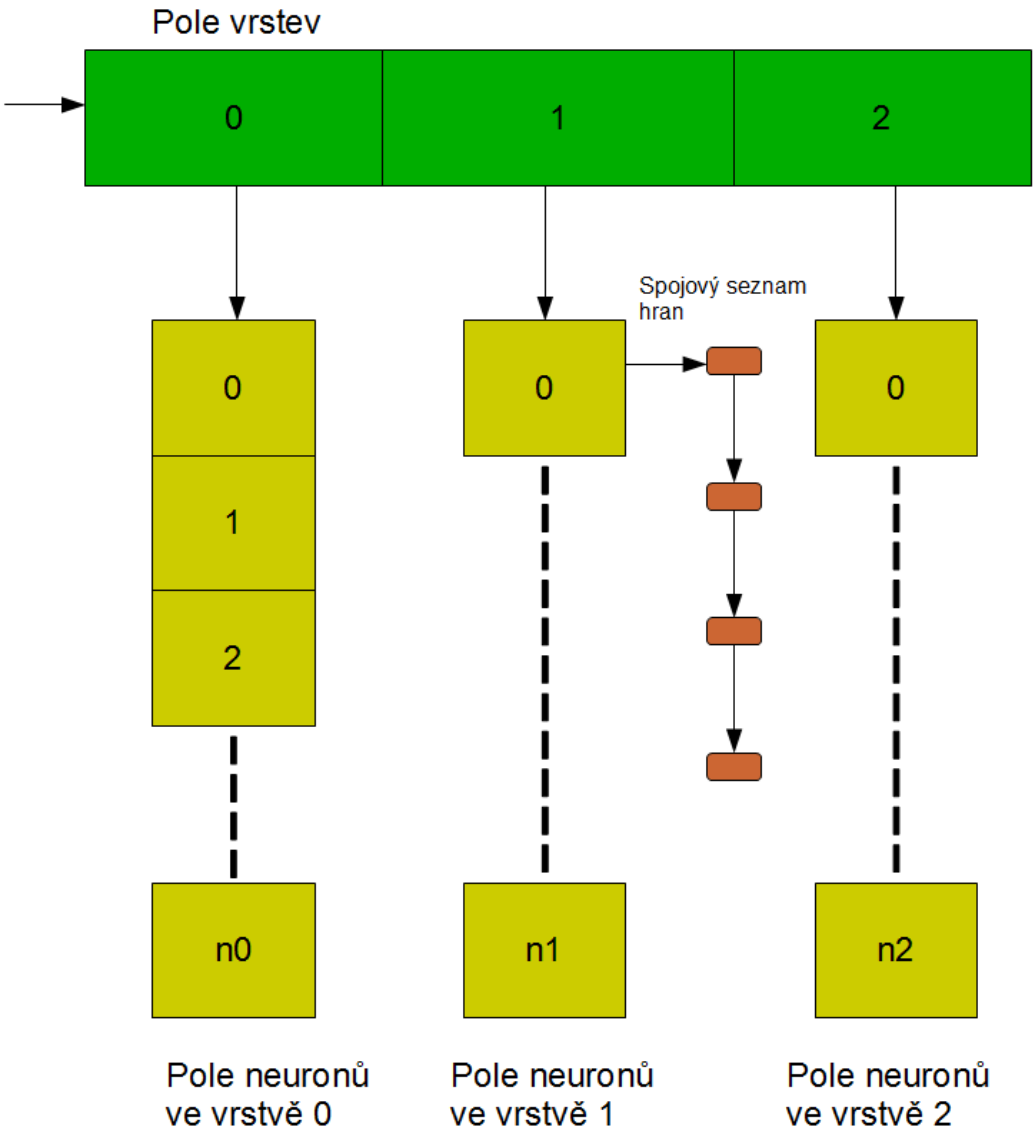
Reprezentaci sítě v programu je třeba navrhnout především s ohledem na jednoduchost jejího vytvoření a především potom na efektivitu závěrečného výpočtu. Je třeba sestavit jednotlivé vrstvy, jejich neurony a hrany mezi neurony do příslušné konstrukce tak, aby bylo umožněno nenáročné přecházení mezi těmito strukturami při následném ukládání přečtených dat a při výpočtu aktivačních hodnot.

V zásadě se nabízejí dvě hlavní možnosti – buď reprezentace sítě pomocí **soustavy spojových seznamů**, kdy do sítě se lze dostat přes seznam s vrstvami, přičemž z každé vrstvy povede ukazatel na první člen seznamu s neurony a od každého neuronu pak povede ukazatel na první člen seznamu s hranami, vedoucími do daného neuronu. Tato struktura pak nabízí jistou flexibilitu v tom smyslu, že v zásadě nemusíme dopředu řešit, jak velká síť bude, před jejím sestavením z příslušných struktur (mohli bychom tedy v podstatě vypustit část se zjišťováním parametrů sítě). Na druhou stranu nám zase nenabízí možnost se snadno dostat na nějaký specifický člen některé ze struktur bez toho, aniž bychom potřebovali vždy prohledávat příslušný seznam, což by způsobovalo pomalejší běh zejména v části provazování neuronů hranami (hrany pravděpodobně budou nejpočetnější množinou v síti).

Druhá možnost je pak reprezentovat síť pomocí **soustavy polí**. To nám poskytne výhodu v možnosti dostat se na nějakou specifickou strukturu pomocí indexu, což zefektivní některé části programu. Na druhou stranu potřebujeme pak vědět dopředu, jak bude neuronová síť velká, abychom mohli přiřadit polím se strukturami přesně velkou paměť. Hrany mezi neurony pak budou stále muset být reprezentovány spojovým seznamen, protože nelze snadno zjistit jejich počet u každého z neuronů. Na druhou stranu při jejich přidávání do seznamu nebo následném výpočtu nebude třeba znát nějaký index konkrétní hrany, neboť budeme vždy procházet jednu po druhé.

V případě využití konstrukce sítě pomocí polí u vrstev a jejich neuronů lze ještě uvažovat o **matici neuronů**, kdy první index bude představovat vrstvu a druhý index jejího neuronu. Výhoda je potom taková, že v zásadě není zapotřebí definovat strukturu pro samotné hrany a celkové zjednodušení schématu při implementaci, ale dost značná nevýhoda je ta, že velikost matice bude daná vrstvou s největším počtem neuronů a u ostatních vrstev nebude paměť zcela využita. Matice je pak speciálně nevýhodná v případě, kdy jsou mezi počtem neuronů v jednotlivých vrstvách propastné rozdíly.

Pro další části budeme počítat s reprezentací pomocí **polí u vrstev a neuronů a spojového seznamu u hran mezi neurony**.



Obrázek 1: Schéma neuronové sítě popsané v analýze

6.2 Parametry připravované sítě

Pro zjištění parametrů potřebných pro následné sestavení neuronové sítě bude zapotřebí provést **předběžné čtení souborů**. Ze **souboru s aktivačními hodnotami** pro vstupní vrstvu bude zapotřebí zjistit **počet hodnot**, které soubor obsahuje, tato hodnota nám pak bude indikovat, kolik neuronů se nachází ve vstupní vrstvě. Ze **souboru s topologií sítě** poté zjistíme, kolik vrstev bude následovat po vstupní vrstvě a kolik neuronů bude každá z nich obsahovat. Počet neuronů lze zjistit z části souboru, kde jsou vypsané **konstanty** pro každý neuron v dané vrstvě. Vzhledem k tomu, že každý neuron má práve jednu takovou konstantu, počet řádek s nimi v dané části/vrstvě se pak rovná počtu neuronů.

6.3 Sestavení neuronové sítě

Sestavení sítě proběhne v momentě, kdy budeme znát její parametry (počet vrstev a neuronů) a částečně při čtení hodnot ze **souboru s topologií sítě**. Pro vrstvy se vytvoří pole příslušných struktur, kdy každá taková struktura bude obsahovat **ukazatel na pole** se strukturami reprezentující neurony. Každy neuron pak bude obsahovat **ukazatel na první člen** spojového seznamu hran, které do něj vedou [1]. Do spojového seznamu s hranami se pak budou přidávat struktury jednotlivých hran podle toho, jak budou postupně čteny ze souboru i s jejich hodnotami (jaké neurony spojují a jakou mají váhu). Souběžně s přídáváním hran k jednotlivým neuronům k nim budeme přiřazovat v souboru uvedené konstanty, které se budou při výpočtu sčítat se sumou násobku váhy hrany a aktivační hodnoty neuronu z předchozí vrstvy, ze kterého hrana vede.

6.4 Výpočet aktivačních hodnot

Výpočet aktivačních hodnot pro všechny neurony ve vrstvách následující po vstupní vrstvě by vzhledem ke konstrukci sítě měl být poměrně přímočarý. Postupně budeme procházet jednotlivé vrstvy a pro každý jejich neuron provedeme **součet násobků váhy jednotlivých hran, k jejichž spojovému seznamu se lze dostat přes ukazatel uložený ve struktuře neuronu, a aktivačních hodnot neuronu z předchozí vrstvy, jejichž index v rámci pole neuronů bude uložen ve struktuře hrany**. K tomuto součtu se pak přičte na neuronu uložená konstanta a takto se bude cyklicky pokračovat pro všechny další neurony v dalších vrstvách. Nakonec se určí **nejvyšší aktivační hodnota u neuronů poslední vrstvy** a index neuronu s touto hodnotou se pak vrátí jako výsledek.

[1] Vzhledem k uvedenému vzorci výpočtu je výhodnější odkazovat se na seznam hran z neuronu, pro který chceme vypočítat jeho aktivační hodnotu.

7 Implementace programu

Program se mimo hlavního souboru `main.c` skládá z následujících modulů:

- `layer.h` a `layer.c`, které obsahují předpis struktury pro reprezentaci neuronových vrstev a funkcí pro jejich vytváření a pro vytváření polí pro ukazatele na jejich neurony,
- `neuron.h` a `neuron.c` obsahující předpis pro strukturu reprezentující jednotlivé neurony a funkce pro jejich vytváření,
- `edge.h` a `edge.c` s předpisem struktury reprezentující hrany mezi neurony a funkcí pro jejich přidávání do spojového seznamu,
- `my_functions.h` a `my_functions.c` obsahující funkce pro provedení jednotlivých kroků programu od načtení zdrojových souborů až po výpočet aktivačních hodnot a vrácení výsledku.

Struktura sítě použitá v programu odpovídá návrhu vybraném v analýze úlohy – **pole vrstev a neuronů a spojový seznam hran**.

7.1 Struktura pro vrstvy sítě

Struktura reprezentující neuronové vrstvy `layer` obsahuje jednak **pole ukazatelů** na její neurony a potom proměnnou typu `integer` uchovávající informaci o jejich počtu. Mezi funkce obsažené v modulu této kontrukce pak patří funkce `layer *createLayer(int neuron_count)`, která přijme informaci o počtu neuronů, alokuje paměť potřebnou pro strukturu `layer`, uloží informaci o počtu neuronů do připravené proměnné struktury a vrátí **ukazatel na danou strukturu**. Další funkcí pak je `void createNeuronArray(layer *l)` přijímající jako vstupní parametr **ukazatel na vrstvu**, přičemž alokuje paměť potřebnou pro vytvoření **pole ukazatelů** na struktury typu `neuron` a následně volá funkci na vytvoření jednotlivých neuronů a ukládání jejich ukazatelů na jednotlivé indexy v poli.

7.2 Struktura pro neurony ve vrstvách

Struktura reprezentující jednotlivé neurony `neuron` obsahuje dvě proměnné typu `double` pro uložení **aktivační hodnoty a konstanty ze souboru** a **dva ukazatele na spojový seznam s hranami** příslušející danému neuronu. Jeden ukazuje na první prvek v seznamu a druhý na poslední. Ukazatel na poslední prvek seznamu je ukládán pro rychlejší přidávání nových hran do seznamu, není pak zapotřebí procházet všechny již přidané hrany, abychom mohli přidat novou na jeho konec. Funkce, která má za úkol vytvářet neurony `neuron *createNeuron()` alokuje paměť potřebnou pro tuto strukturu, nastaví oba ukazatele na seznam neuronů na `NULL` a vrátí **ukazatel na daný neuron**.

7.3 Struktura pro hrany mezi neurony jednotlivých vrstev

Struktura reprezentující hrany `edge` obsahuje proměnnou typu `double` pro uložení **váhy hrany**, proměnnou typu `integer` pro uložení **indexu neuronu**, ze kterého hrana vychází [2] a

[2] Ukazatel na seznam s hranami vždy uchovává neuron, do kterého hrana směřuje, kvůli jednodušší situaci při výpočtu aktivačních hodnot.

ukazatel na následující hranu v seznamu. Funkce na přídání nové hrany `void addEdgeToList(edge **head, edge **tail, double weight, int neuron_index)` vždy přijme odkaz na **ukazatel hlavičky a konce seznamu, váhu hrany a index neuronu, ze kterého hrana vychází**. Poté alokuje paměť pro strukturu hrany a vložené hodnoty uloží do proměnných struktury. Následně ověří, jestli již nějaká hrana existuje na ukazateli na hlavičku seznamu, pokud ano, vloží ukazatel na novou hranu do proměnné pro ukazatel na další hranu u poslední vložené nacházející se na místě, kam ukazuje ukazatel na konec seznamu. V opačném případě se hlavička i konec seznamu nastaví na právě vkládanou hranu.

7.4 Postup činnosti programu

Nyní, když máme popsány jednotlivé struktury nacházející se v programu, rozebereme si postup funkce programu samotného tak, jak je specifikován ve funkci `main` (pozn.: všechny další funkce, s vyjímkou funkcí pro uvolňovaní paměti zabrané výše popsanými strukturami jsou obsaženy v modulu `my_functions`). Program si nejprve zkontroluje, zda byl zadán **dostatečný počet vstupních argumentů programu**, pokud ne, vypíše chybovou hlášku a svoji činnost ukončí. Následně si pomocí funkce `fopen` otevře proudy pro čtení ze **vstupních souborů**, jejichž umístění bylo specifikováno ve vstupních argumentech programu. Pokud jakýkoliv z nich nebude moci otevřít, oznámí chybu a ukončí svoji činnost.

Následuje volání funkce `int getInputLayerActivationsCount(FILE *f, int element_size)`, která má za úkol zjistit, **kolik aktivačních hodnot** obsahuje soubor s příponou `.dat` nebo-li jaký je **počet neuronů ve vstupní vrstvě**. Tuto hodnotu pak použije k alokaci paměti pro pole s hodnotami typu `float`, kam poté načte aktivační hodnoty z výše zmíněného souboru, který potom uzavře, neboť již dále není zapotřebí. Poté dojde k zavolání funkce `int getLayerCountFromFile(FILE *f, char *space)`, která přečte **první řádek souboru s topologií sítě** a zjistí z něj, kolik vrstev (mimo té vstupní) bude síť obsahovat. Na základě toho pak program alokuje paměť pro **pole ukazatelů** na struktury `layer` a u první z nich vytvoří pole ukazatelů na struktury `neuron` na základě hodnoty získané v předešlé části kódu. Nakonec pomocí funkce `copyActivationsToLayer(float *activations_buffer, layer **neural_layers, int layer_index)` přenese aktivační hodnoty z pole, které bylo vytvořeno pro jejich odložení, do **proměnných těchto struktur**.

Poté se volá funkce `int createNeuralLayersFromFile(FILE *f, char *space, layer **neural_layers)`, která má za úkol zjistit, **kolik neuronů bude každá vrstva obsahovat** a na základě této informace poté vytvořit pole ukazatelů na struktury `neuron` u dalších vrstev podobně jako u té vstupní. Funkce postupně prochází **soubor s topologií sítě** řádek po řádku, přičemž ověřuje, jestli se nachází v části, kde jsou uvedeny hrany mezi neurony dvou vrstev nebo konstanty pro neurony druhé z nich. Pokud se nachází v části s konstantami, **inkrementuje si proměnnou představující počet neuronů v aktuálně procházené vrstvě**. Jakmile se opět dostane do části s hranami, znamená to, že se zjišťují informace o další vrstvě a tedy je možné použít zmíněnou proměnnou pro alokování paměti pro dané pole ukazatelů na struktury `neuron`. Funkce takto pokračuje, dokud nepřečte celý soubor a nevytvoří pole pro všechny vrstvy.

Následně je volána funkce `int createEdgesFromFile(FILE *f, char *space, layer **neural_layers)`, která přečte daný soubor znovu obdobným způsobem jako v předchozím případě. Nicméně zde se snaží v části s konstantami **ukládat tyto do již vytvořených neuronů** a v části s hranami **přidává nové hrany do spojového seznamu k jednotlivým**

neuronům v poli. Obě popsané funkce vypisují na konzoli chybovou hlášku v případě, že narazí při čtení souboru na něco, co nejsou schopny zpracovat, a **jejich návratová hodnota je pak použita jako indikace ukončení programu**.

Nyní již máme v podstatě vytvořenou kompletní neuronovou síť, je zapotřebí jen vypočítat hodnoty aktivace u jednotlivých neuronů v jednotlivých vrstvách. To má na starost funkce `void setActivationValues(layer *l_curr, layer *l_prev)`, probíhající v cyklu nad polem ukazatelů na struktury `layer`. Jejími vstupními parametry je pak ukazatel na vrstvu, **pro jejíž neurony chceme vypočítat aktivační hodnoty**, a ukazatel na vrstvu předcházející, **z jejíchž neuronů potřebujeme tyto hodnoty přečíst a použít pro výpočet**. Funkce provádí výpočet postupně nad všemi neurony dané vrstvy tak, že na neuronu, kde chceme aktivaci vypočítat, prochází seznam hran a jejich váhy násobí aktivací specifického neuronu z předchozí vrstvy (k tomu je použit index neuronu uložený ve struktuře hrany). Tyto násobky jsou sčítany do výsledného součtu, ke kterému je poté přičtena konstanta daného neuronu a matematickou funkcí **tanh** je vypočítána výsledná aktivace.

Poslední část programu před uvolněním zabrané paměti a jeho ukončením je volání funkce `int getNeuronIndexWithMaxActivation(layer *l)`, která projde poslední vrstvu, porovná aktivační hodnoty jejích neuronů a vrátí **index toho neuronu, který měl hodnotu nejvyšší**. Index se poté vypíše na konzoli jako výsledek.

Nakonec se postupně uvolní paměť alokovaná jednotlivými strukturami. K tomu slouží specifické funkce obsažené v modulech těchto struktur. **Uvolňování paměti probíhá jakousi kaskádou**, kdy funkce zabývající se „vyšší“ strukturou (vrstva) volá funkci zabývající se „nižší“ strukturou (neuron nebo hrana v případě, že je vyšší struktura neuron) předtím, než pro tuto svou strukturu uvolní paměť.

7.5 Využití knihovních funkcí jazyka C

Na závěr popisu implementace ještě zmínka o použitých knihovních modulech. Pro práci se soubory jsou využité funkce z `stdio.h`, při načítání jednotlivých řádků souboru s topologií sítě jsou pak pro získávání hodnot použity funkce z `string.h`. Dále pro výpočet aktivační hodnoty přes matematickou funkci tanh je zahrnuta `math.h` a pro definování konstanty `DBL_MAX` (použitá při porovnávání aktivací) je zahrnuta `float.h`.

8 Uživatelská dokumentace

Sestavení programu ze zdrojových a hlavičkových souborů lze provést pomcí přiložených souborů `Makefile` (jeden pro systémy **Windows**, druhý pro **GNU/Linux**). V adresáři, kde se soubory nacházejí, je zapotřebí provést příkaz `make` pro příslušný `Makefile`, který pak zkompiluje funkční program spustitelný přes soubor `neural_net.exe`. Spuštení samotného programu lze pak realizovat opět z příkazové řádky kdy zadáme název spustitelného souboru a mezerami oddělené dva parametry, přičemž jejich počet a pořadí je naprosto zásadní. Jako první parametr se poskytne cesta k **souboru s topologií sítě**, který je poskytnutý ve formátu `.txt`. Jako druhý parametr je třeba zadat cestu **k souboru, který obsahuje aktivační hodnoty pro neurony vstupní vrstvy** v hexadecimální podobě a je poskytnutý s příponou `.dat`.

Podoba příkazu na spuštení programu je následující:
`neural_net.exe <soubor_topologie_site> <soubor_vstupnich_aktivaci>`,
tedy např. `neural_net.exe neuronova_sit.txt vstupy\0.dat`.

Pokud uživatel nezadá dostatečný počet perametrů nebo nastane nějaká neočekávaná situace při čtení souborů, **program skončí výpisem chybové hlášky**. Po správném průběhu programu je na konzoli vypsán index neuronu z výstupní vrstvy s nejvyšší hodnotou aktivace, což představuje výslednou třídu zkoumaného obrázku. [3]

```
C:\Users\Vlada47\Desktop\laznic-neural_net>mingw32-make
gcc -c -Wall -pedantic -ansi -O3 layer.c -o layer.o
gcc -c -Wall -pedantic -ansi -O3 neuron.c -o neuron.o
gcc -c -Wall -pedantic -ansi -O3 edge.c -o edge.o
gcc -c -Wall -pedantic -ansi -O3 my_functions.c -o my_functions.o
gcc -c -Wall -pedantic -ansi -O3 main.c -o main.o
gcc -lm layer.o neuron.o edge.o my_functions.o main.o -o neural_net.exe

C:\Users\Vlada47\Desktop\laznic-neural_net>
```

Obrázek 2: Příkaz pro kompilaci programu a jeho výsledek

```
C:\Users\Vlada47\Desktop\laznic-neural_net>neural_net.exe neuronova_sit.txt
You didn't specified all arguments!
 You must specify file with neural net and file with input data for the first la
yer of the net.
C:\Users\Vlada47\Desktop\laznic-neural_net>_
```

Obrázek 3: Chybová hláška po zadání nedostatečného počtu argumentů

[3] Na základě poskytnutých testovacích dat se jedná o číslici na obrázku odpovídajícím zadanému vstupu s aktivačními hodnotami neuronů vstupní vrstvy.

9 Závěr a zhodnocení práce

Implementace programu splnila předepsané zadání semestrální práce – **program koretně vytvoří neuronovou síť na základě vstupních souborů a vrátí správný výsledek na konzoli**. Jsou zde samozřejmě místa, která by snesla nějaké to vylepšení, např. minimalizovat počet přístupů do souboru (textový soubor je přečtený dvakrát), což by zkrátilo dobu běhu programu zejména u velmi velkých sítí s velkým množstvím hran. Také řešení některých nepředpokládaných situací (zejména chyby ve vstupních souborech nebo problémy s alokací paměti) by se dalo provést robustněji, ale při korektním použití programu by s tímto neměl být problém.

Co se týče rychlosti průběhu programu, během testování se jevila jako uspokojivá. Na počítači, kde byl program vyvíjen – dvoujádrový procesor **Core 2 Duo na taktu 3,5GHz**, **4GB DDR2** paměti, běžný magnetický disk s **7200 ot./min** a operační systém **Windows** - byla typická doba provedení všech operací pro poskytnutá testovací data v průměru kolem **1,2 sekundy**.

Program by šel samozřejmě dále upravit a optimalizovat pro nižší spotřebu paměti nebo pro větší univerzálnost, ale i tak dokáže dobře řešit zadanou úlohu.