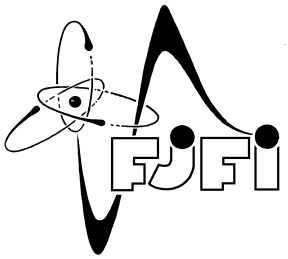
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA JADERNÁ A FYZIKÁLNĚ INŽENÝRSKÁ
Katedra matematiky



Výzkumný úkol:

Paralelní algoritmy ve zpracování dat

Bc. Jan Hofta

Školitel: Ing. Tomáš Oberhuber 2006/2007
Zaměření: Softwarové inženýrství

Čestné prohlášení: Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl veškerou použitou literaturu.

……..……

Bc. Jan Hofta

Obsah

Úvod

Hlavním cílem této práce je pomoc při rozvoji projektu NNSU vedeném panem Ing. Františkem Haklem, CSc. z Ústavu informatiky AV ČR. Projekt NNSU se zaobírá vývojem univerzálního separačního nástroje založeného na aplikaci neuronových sítí optimalizovaných genetickými algoritmy. Představen je ve třetí kapitole této práce. Před tím práce popisuje teoretický podklad, nutný pro pochopení činnosti tohoto projektu. V první kapitole se tak můžeme seznámit s principem fungování genetických algoritmů a druhá kapitola nabízí základní přehled problematiky neuronových sítí.

Konkrétním úkolem na projektu NNSU byla paralelizace kódu genetického algoritmu. K paralelizaci bylo použito rozhraní MPI, o kterém se lze dočíst také ve třetí kapitole. Na jejím konci je popsána navržená paralelizace.

Jelikož projekt NNSU se již nějakou dobu vyvíjí, nebylo snadné proniknout do jeho tajemství. Naštěstí mi byli velmi nápomocni stávající členové vývojového týmu. Moc bych chtěl poděkovat především panu Ing. Haklovi za seznámení s projektem jako celkem a za korekturu teoretických textů. Velice děkuji i panu Ing. Romanu Kalousovi za všestrannou pomoc s pochopením pluginu GA, který neustále rozvíjí.

V neposlední řadě děkuji i svému školiteli, panu Ing. Tomáši Oberhuberovi, za neustálou a vydatnou pomoc při tvorbě této práce. Ať už šlo technické problémy při programování nebo shánění vhodné literatury, pan Ing. Oberhuber byl vždy příkladně vstřícný.

1. Genetické algoritmy

1.1. Kde se vzaly

1.1.1. Úvod

Vývoj vědy je skutečně obdivuhodný. Jako lidstvo jsme začali s objevem ohně. Pokračovali jsme s nalézáním tisíců různých věcí – zbraně, chov zvířat, matematika, míchání barev, psychologie… Dokud nebylo nového vědění příliš mnoho, mohl všechny obory obsáhnout jeden dostatečně vzdělaný člověk, který je v sobě dokázal spojit. Jenže objevů stále přibývalo a lidé se museli začít specializovat. Vznikli historikové, kteří nevěděli nic o psychoanalýze, biologové, nemající tušení o statice staveb, fyzici, kteří nerozeznali kočku od psa. Ve svém oboru kapacity, v mnoha jiných nuly.

Proto je vždy nanejvýš zajímavé, když se nynější oddělené různé obory navzájem ovlivňují. Výsledek bývá často potřebným oživením pro dané odvětví. Z fascinace spisovatelů technikou vzniklo sci-fi. Porozumění radioaktivitě se blahodárně odrazilo ve vývoji lékařství. A kvůli spojení genetiky a matematiky vznikly v sedmdesátých letech minulého století genetické algoritmy.

Americký matematik John Holland se svými kolegy z michiganské univerzity dokázal rozvést nápady svých předchůdců a spojit dohromady dvě tak rozdílné věci, jako je Darwinova evoluční teorie a hledání extrému funkce. Ve své knize *Adaptation in Natural and Artificial Systems* (1975) poprvé systematicky popisuje systém používání populací, mutací, křížení a reprodukce, který dovede řešit ryze matematické problémy.

1.1.2. Darwinova evoluční teorie

Z čeho Holland a jeho předchůdci vycházeli? Jejich inspirací byl vývoj samotné přírody popsaný v 19. století slavným biologem Charlesem Darwinem. Jeho stěžejní dílo *O vzniku druhů přírodním výběrem* (1859) vychází z myšlenky, že vývoj probíhá pomocí nepatrných změn jednotlivců v následujících generacích. Které změny se provedou, záleží na výběru tvorů, kteří se budou rozmnožovat. Darwin předpokládá, že čím lepší vlastnosti má daný rodič, tím více bude mít potomků a tím více se také tyto jeho dobré vlastnosti rozšíří v dalších generacích. Uveďme si jednoduchý příklad:

Dejme tomu, že na Zemi je jedna velká zelená louka. Na ní žijí jednak nazelenalé srnky, které se živí trávou, a jednak lvi, kteří se živí srnkami. Představme si, že přišla doba ledová a louku pokryl sníh. Nechť se náhodou v tutéž dobu narodilo několik srnek šedých. Nechť pro rozmnožování srnek platí, že pokud se rozmnožují nazelenalá a šedá srnka, může vzniknout i zvláštní bílá a pokud se rozmnožují dvě srnky stejné barvy, barva potomka zůstává zachována[1].

Být šedý je nyní pro srnku dobrá vlastnost, být bílý ještě lepší, neboť lev na sněhu vidí (a tedy spíš sežere) nejlépe srnky nazelenalé, potom srnky šedé a nejhůř srnky bílé. Proto se podle Darwina v další generaci objeví víc bílých a šedých srnek než nazelenalých – ty budou postupně sežrány lvi. Později, až už nebudou k dispozici žádné nazelenalé, budou žrát lvi především šedé srnky. Z populace nazelenalých srnek se tak během několika generací stane populace bílých.

Na tomto jednoduchém příkladu můžeme vidět všechny tři základní principy, které využijeme pro naše genetické algoritmy.

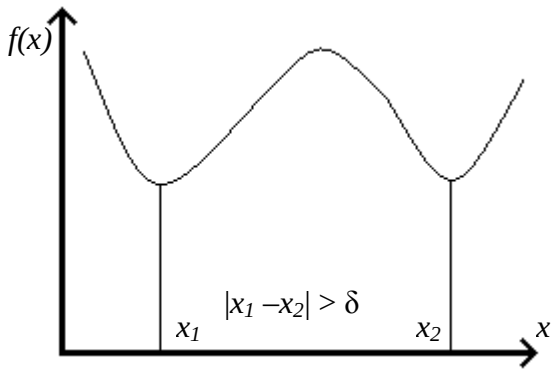
[1] Popis tohoto rozmnožování pomocí jednotlivých genů možná ani neexistuje, ale to není pro tento příklad podstatné.

1. přirozený výběr – jedinci, kteří mají dobré vlastnosti, tedy tzv. velké fitness, přenášejí své kvality do další generace s mnohem větší pravděpodobností než jedinci s malým fitness. Je mnohem pravděpodobnější, že do dospělosti doroste bílá srnka bílých rodičů, než nazelenalá od nazelenalých.
2. náhodná genetická mutace – některé nové vlastnosti u jednotlivců se objeví náhodně. Darwin to mohl jen tušit, ale my už dnes víme, že za vlastnosti jednotlivců zodpovídá jejich genetický kód, který se občas náhodně změní, tedy zmutuje. Mutace může přinést novou pozitivní vlastnost, která se dál prosadí, nebo negativní, která zapadne. V našem příkladu byla pozitivní genetickou mutací změna barvy několika nazelenalých srnek na šedou. Pokud by gen pro barvu srsti zmutoval na červenou barvu, je pravděpodobné, že by takovéto srnky dlouho nepřežily.
3. reprodukční proces – potomek vzniká z rodičů kombinací jejich vlastností, které jsou opět určeny pomocí genů. Toto tzv. křížení může do populace přinést nové a lepší vlastnosti, vzniklé kombinací starých. Bílá barva srnek je toho příkladem.

1.2. Prostředky a cíle použití genetických algoritmů

1.2.1. Použití genetických algoritmů

Genetické algoritmy jsou především efektivním nástrojem k nalezení optimální hodnoty pro daný problém. Typicky je používáme k hledání lokálního minima (nebo maxima) dané funkce. Tato funkce musí splňovat dva základní předpoklady: musí být dobře vypočitatelná, tedy pro každé x z definičního oboru musíme umět najít s požadovanou přesností funkční hodnotu, a vzdálenost argumentů stejných lokálních extrémů musí být větší než daný parametr δ. V dalším textu se omezíme na hledání lokálního minima, maximum nalezneme analogicky.



Obrázek 1: Ilustrace druhé podmínky kladené na funkci f

Mnohý z vás si jistě řekne, že k hledání extrémů funkce známe mnoho jiných přímých i nepřímých postupů – od klasického vyšetřování průběhu funkce pomocí derivací, přes horolezecké algoritmy (*hill climbing algorithms, viz použitá literatura)* po simplexovou metodu lineárního programování. Jenže všechny tyto postupy mají velmi mnoho různých požadavků na vyšetřovanou funkci. Fungují pouze lokálně, jsou málo robustní. Genetické algoritmy se od nich odlišují čtyřmi hlavními věcmi:

- nepracují přímo s parametry dané funkce, ale s jejich zakódovanými hodnotami; s tímto kódováním se seznámíme v následující kapitole
- optimum se hledá z více míst, nejen z jednoho bodu
- používáme ohodnocení jednotlivých bodů, nikoli derivace nebo jiné další znalosti o průběhu funkce
- výpočet není deterministický, ale pravděpodobnostní

1.2.2. Binární kódování reálné proměnné

Abychom dokázali nalézt minimum reálné funkce reálné proměnné pomocí genetických algoritmů, musíme být schopni převést reálnou proměnnou na binární vektor. Nejprve mějme binární vektor α délky k.

$$\alpha = (\alpha_1, \alpha_2, ..., \alpha_k) \in \{0,1\}^k$$

Ten můžeme klasicky reprezentovat jako celé číslo:

$$\text{int}(\alpha) = \sum_{i=1}^{k} \alpha_i 2^{k-i} = \alpha_1 2^{k-1} + \alpha_2 2^{k-2} + ... + \alpha_{k-1} 2 + \alpha_k$$

Reálné číslo $x \in \langle a, b \rangle$ potom můžeme aproximovat následujícím způsobem:

$$x \approx real(\alpha) = a + \frac{b-a}{2^k - 1}\text{int}(\alpha)$$, inverzní transformace: $$\text{int}(\alpha) = \left\lfloor \frac{x-a}{b-a}(2^k - 1) \right\rfloor$$

Koeficient $b - a / 2^k - 1$ udává přesnost naší aproximace. Hranaté závorky potom znamenají celou část. Pro hodnoty *k*=3, *a*=0, *b*=1 zobrazuje vzájemné převody tabulka 1:

α	000	001	010	011	100	101	110	111
int(α)	0	1	2	3	4	5	6	7
real(α)	0	1/7	2/7	3/7	4/7	5/7	6/7	1

Tabulka 1: Převod binárních, celých a reálných čísel

Nevýhodou tohoto klasického binárního kódování je to, že blízká čísla mohou mít velmi odlišný kód. Například trojka (011) a čtyřka (100) se liší ve všech znacích, ale jdou hned po sobě. Tento nedostatek odstraňuje tzv. Grayovo kódování, které je založené na tom, že dvě po sobě jdoucí čísla mají kód odlišný právě v jednom znaku. Nechť α je kódování čísla ve standardním kódu se složkami α_1 až α_k a nechť $\tilde{\alpha}$ je kódování čísla v Grayově kódu o složkách $\tilde{\alpha}_1$ až $\tilde{\alpha}_k$. Potom pro vzájemný převod obou kódování platí:

$\alpha_1 = \tilde{\alpha}_1$	$\tilde{\alpha}_1 = \alpha_1$
$\alpha_2 = \tilde{\alpha}_1 \oplus \tilde{\alpha}_2 = \alpha_1 \oplus \tilde{\alpha}_2$	$\tilde{\alpha}_2 = \alpha_1 \oplus \alpha_2$
$\alpha_3 = \tilde{\alpha}_1 \oplus \tilde{\alpha}_2 \oplus \tilde{\alpha}_3 = \alpha_2 \oplus \tilde{\alpha}_3$	$\tilde{\alpha}_3 = \alpha_2 \oplus \alpha_3$
…	…
$\alpha_k = \tilde{\alpha}_1 \oplus \tilde{\alpha}_2 \oplus ... \oplus \tilde{\alpha}_k = \alpha_{k-1} \oplus \tilde{\alpha}_k$	$\tilde{\alpha}_k = \alpha_{k-1} \oplus \alpha_k$

Operace $\oplus$ představuje binární sčítání (1+1 = 0+0 = 0, 1+0 = 0+1 =1). Představme si například, že máme ve standardním binárním kódování zakódované číslo 12 jako 1100. Grayův kód potom bude: $\tilde{\alpha}_1$ = 1, $\tilde{\alpha}_2$ = 1+1 = 0, $\tilde{\alpha}_3$ = 1+0 = 1, $\tilde{\alpha}_4$ = 0+0 = 0, tedy celkově 1010. Z definice Grayova kódování je také vidět, že vždy, když budeme chtít zakódovat nebo dekódovat reálné či celé číslo v Grayově kódu, budeme ho muset nejprve přetáhnout přes

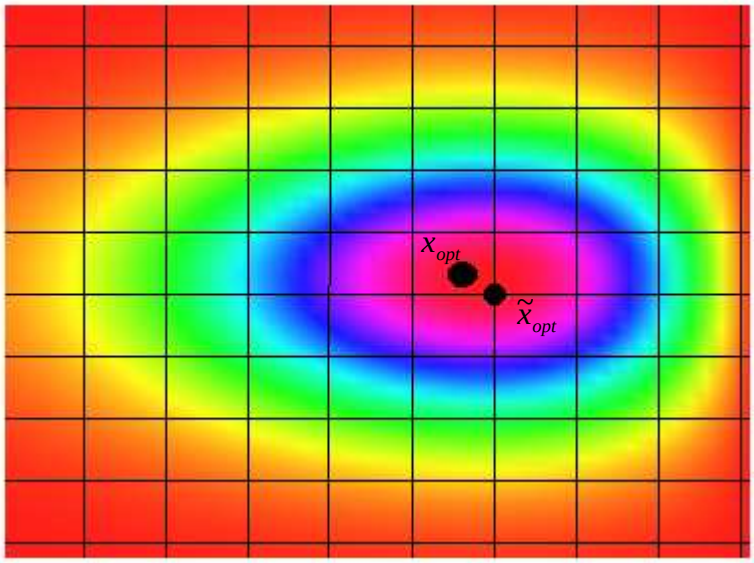
mezistupeň standardního binárního kódování. To je také jeho nevýhoda – při práci s ním narůstá potřebný čas na zpracování.

Nyní, když už víme, jak převést na binární vektor jedno reálné číslo, ukážeme si, jak přetransformovat spojitý problém. Nechť funkce *f* má *n* proměnných. Hodnotu každé proměnné dovedeme převést shora uvedeným způsobem na binární vektor délky *k*. Hodnoty všech proměnných bude potom reprezentovat binární vektor délky *nk*, vzniklý napsáním jednotlivých vektorů za sebe.

S jak velkou přesností nahrazujeme reálná čísla binárními vektory, popisuje délka jednotlivých vektorů k. Musí platit:

$$\delta >> \frac{b_i - a_i}{2^k - 1} \text{ pro } \forall i \in \hat{n}$$

kde δ je parametr, určující vzdálenost argumentů lokálních extrémů z kap. 1.2.1. Tento předpoklad je velmi důležitý – zaručuje, že graf hledané funkce proložíme dostatečně hustou mřížkou z bodů tak, abychom o každém bodu dokázali rozhodnout, zda se optimální hodnota funkce nalézá v jeho okolí a ne okolí jeho sousedů. Ilustruje to obrázek 2. Hodnota x_{opt} je skutečné minimum funkce, hodnota $\tilde{x}_{opt}$ je námi nalezený nejbližší bod na mřížce.



Obrázek 2: K velikosti parametru k

1.3. Populace a operátory

1.3.1. Nástin algoritmu

V předchozí kapitole jsme se seznámili s hlavním cílem genetických algoritmů – hledáním minima reálné spojité funkce. Také jsme si ukázali, jak reprezentovat reálnou spojitou funkci pomocí binárních vektorů. V této kapitole si popíšeme, jak tyto dvě věci spojit.

Hrubý nástin algoritmu je následující. Na začátku sestavíme populaci, tedy skupinu přípustných argumentů dané funkce reprezentovaných binárními vektory. Těmto vektorům budeme říkat chromozomy. Každý chromozom dostane ohodnocení, jak moc je dobrý, tzv. fitness. Podle hodnoty fitness se vyberou dvojice chromozomů, které se spolu budou moci

reprodukovat (tvořit potomky). Rozmnožování probíhá pomocí operátoru křížení, který způsobí výměnu částí kódu. Při reprodukci se může projevit také operátor mutace – některé části kódu se změní náhodně. Nově vzniklé chromozomy potom vytvoří další generaci populace, na které postup opakujeme, dokud nezískáme požadované výsledky.

Tento náčrt si nyní podrobně rozebereme.

1.3.2. Populace a fitness

Na začátku algoritmu vybereme náhodně skupinu binárních vektorů stejné délky *k*, jejichž reálná interpretace odpovídá přípustným argumentům vyšetřované funkce. Každý takovýto vektor se nazývá chromozom, celá skupina pak populace chromozomů. Populaci v j-tém kroku algoritmu nazýváme j-tou generací. Nyní na počátku mluvíme o nulté generaci.

Na velikosti nulté generace, tedy počtu chromozomů, které populace na začátku obsahuje, závisí rychlost konvergence celého algoritmu. Proto se snažíme mít ji co největší. Omezením shora je výkon výpočetní techniky.

Než začneme chromozomy reprodukovat, musíme každému přiřadit ohodnocení jeho vlastností, tedy jak moc dobře splňuje naši úlohu. Tuto hodnotu nazýváme fitness. Podle hodnoty fitness se bude určovat pravděpodobnost vstupu chromozomu do reprodukčního cyklu.

Jak se fitness určí? Existuje pro to více způsobů. Nechť $g : R^n \to R$ je funkce, jejíž minimum hledáme. Buď dále $f : \{0,1\}^k \to R$, pro kterou platí, že $f(\alpha) = g(x)$ pokud α je binárním kódováním vektoru *x*. Naším cílem je tedy nalezení

$$\alpha_{opt} = \arg\min_{\alpha \in \{0,1\}^k} f(\alpha).$$

Z této úvahy plyne, že pro hodnotu fitness $F : \{0,1\}^k \to R^+$ by měla platit následující podmínka:

$$\forall \alpha_1, \alpha_2 \in \textit{populace}\, P : f(\alpha_1) \le f(\alpha_2) \Rightarrow F(\alpha_1) \ge F(\alpha_2) \ge 0$$

Toho lze docílit více způsoby. Nejsnazší je definovat fitness následujícím způsobem:

$$\begin{aligned} F(\alpha) &= C_{\max} - f(\alpha) && \textit{pokud } f(\alpha) < C_{\max} \\ &= 0 && \textit{jinak} \end{aligned}$$

Hodnotu konstanty $C_{\max}$ lze volit různě – např. jako doposud nejvyšší hodnotu *f*, jako nejvyšší hodnotu *f* v dané generaci nebo v posledních *j* generacích apod.

Pokud ji přejeme mít hodnotu fitness mezi hodnotami $F_{\min} = \varepsilon$ a $F_{\max} = 1$, můžeme použít vzorec:

$$F(\alpha) = \frac{1}{f_{\min} - f_{\max}}\left((1-\varepsilon) f(\alpha) + f_{\min}\varepsilon - f_{\max}\right), \quad f_{\max} = \max_{\alpha \in P} f(\alpha),\ f_{\min} = \min_{\alpha \in P} f(\alpha)$$

Tento vzorec se v praxi používá, jestliže se hodnoty funkce *f* příliš nemění (jsou stále stejného řádu). Když se řádově liší, vede vzorec k neopodstatněné preferenci chromozómů s malou funkční hodnotou. Proto se spíš využívá následující cesta. Nechť bez újmy na obecnosti pro funkční hodnoty funkce *f* na všech chromozomech populace P platí:

$$f(\alpha_1) \le f(\alpha_2) \le \ldots \le f(\alpha_{|P|})$$

Potom fitness definujeme vztahem:

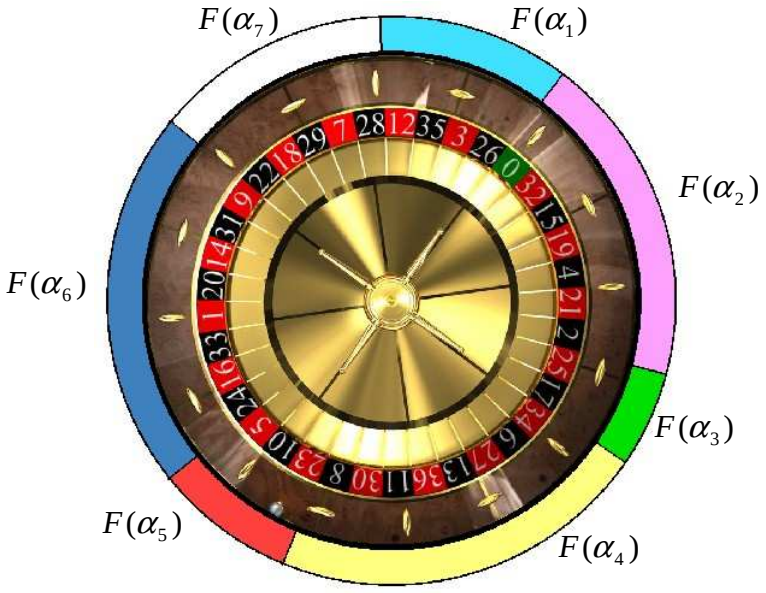
$$F(\alpha_i) = \frac{1}{1-|P|}\left((1-\varepsilon)i + \varepsilon - |P|\right) \quad pro\ \forall i = 1,2,\ldots,|P|$$

Nevýhodou tohoto vzorce je, že chromozomům, které dávají stejnou funkční hodnotu funkce *f*, přiřazuje různé fitness. Proto je ho dobré upravit tak, že pokud tato situace nastane, přiřadí se fitness stejné, tedy:

$$f(\alpha_{i-1}) = f(\alpha_i) \Rightarrow F(\alpha_i) = F(\alpha_{i-1}) \quad pro\ \forall i = 2,3,\ldots,|P|$$

1.3.3. Výběr chromozomů pro reprodukci

Už jsme si prozradili, že výběr chromozomů, které dají vzniknout potomkům, záleží na jejich fitness – čím vyšší fitness, tím větší šance na vstup chromozomu do reprodukčního cyklu. V praxi se chromozomy nejsnáze vybírají pomocí tzv. nevyvážené rulety. Intuitivně ji zobrazuje následující obrázek:



Obrázek 3: Nevyvážená ruleta

Délka přiřazeného oblouku odpovídá části fitness daného chromozomu v celkovém součtu $\sum_{i=1}^{|P|} F(\alpha_i)$ a udává pravděpodobnost výběru tohoto chromozomu pro rozmnožování. Nyní vždy, když chceme vybrat vhodný chromozom pro reprodukci, stačí roztočit ruletu. Například pravděpodobnost výběru chromozomu α_3 z našeho obrázku je 2/37 = 5,4% nebo chromozomu α_6 8/37=21,6%.

1.3.4. Reprodukce

Ke každému rozmnožování potřebujeme vybrat dva chromozomy α a β (nechť mají délku *k*), což učiníme pomocí nevyvážené rulety z minulé podkapitoly. Mohou být klidně stejné. Poté náhodně vybereme číslo od nuly do jedné. Pokud je toto číslo vyšší než stanovená

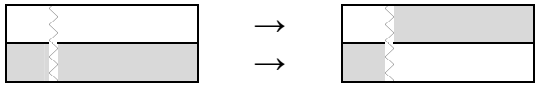
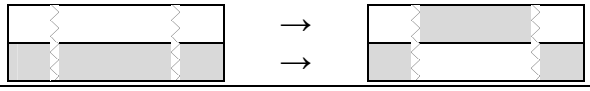
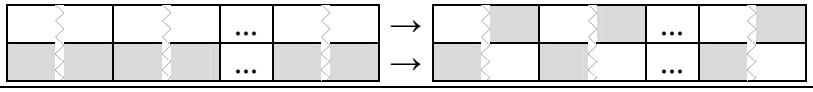
pravděpodobnost reprodukce P_{repro}, k reprodukci nedochází a výsledkem operace jsou původní chromozomy. V opačném případě nastupuje reprodukční algoritmus.

Reprodukce probíhá pomocí operátorů křížení a mutace.

Operátor křížení vytvoří z původních chromozomů α a β dva nové chromozomy α' a β' následujícím způsobem:

- náhodně vybereme číslo $c \in \{1, 2, ..., k-1\}$
- chromozom α' bude tvořen popořadě znaky $\alpha_1, \alpha_2, ..., \alpha_c, \beta_{c+1}, \beta_{c+2}, ..., \beta_k$
- podobně β' bude sestávat z $\beta_1, \beta_2, ..., \beta_c, \alpha_{c+1}, \alpha_{c+2}, ..., \alpha_k$

Tomuto křížení říkáme jednobodové. Pokud bychom si přáli mít u nových chromozomů stejný začátek a konec jako u jejich předchůdců a měnit jen prostředek, použili bychom analogické dvoubodové. Stejně tak pokud je náš chromozom tvořen kódy *n* proměnných dané funkce, křížíme mezi sebou jednotlivé části. Ilustrativně to zachycuje následující tabulka:

Jednobodové křížení	
Dvoubodové křížení	
Více proměnných	

Tabulka 2: Různé typy křížení

Druhou částí reprodukce je použití operátoru mutace. Ten použijeme na oba nově vzniklé chromozomy α' a β'. Definován je následujícím způsobem:

$$\forall i \in \widehat{k} \qquad M(\alpha_i) = \begin{cases} 1-\alpha_i & c_i \le P_{mut} \\ \alpha_i & jinak \end{cases}$$

α_i jsou jednotlivé symboly z řetězce chromozomu α, c_i jsou navzájem nezávislá náhodná čísla z intervalu <0,1) generovaná rovnoměrně a P_{mut} je pravděpodobnost mutace. Ta bývá obvykle menší než 0,1. Prakticky provádí operátor mutace to, že projde celý řetězec chromozomu a s určitou pravděpodobností převrátí jeho znaky na opačnou hodnotu.

Po použití operátoru mutace na oba chromozomy α' a β' získáme nové chromozomy α'' a β'', které jsou konečným výsledkem celé reprodukce.

1.3.5. Zastavení algoritmu

Nyní víme, jak vznikají nové populace chromozomů. Otázkou této podkapitoly je, jak dlouho mají vznikat, tedy kdy zastavit tvorbu nových generací. V literatuře jsou zmiňované tři možnosti.

Nejjednodušší je zastavit algoritmus po určitém počtu generací nezávisle na tom, jaké má výsledky.

Druhou možností je nalézt v každé generaci chromozom s nejvyšším fitness a porovnat počet chromozomů, které tohoto fitness dosahují, s jejich celkovým počtem. Pokud

je to více než určitá prahová hodnota (80 až 95%), je malá pravděpodobnost, že se v příští generaci objeví chromozomy s vyšším fitness. Tudíž nastává správný čas zastavit algoritmus.

Třetí možností, jak poznat, že máme zastavit, je sledování uspořádanosti chromozomů. Toho docílíme tak, že stanovíme pro každou populaci $P = \{\alpha^{(1)}, \alpha^{(2)}, ..., \alpha^{(p)}\}$, $\forall i \in \hat{p}\ \ \alpha^{(i)} = (\alpha_1^{(i)}, \alpha_1^{(i)}, ..., \alpha_1^{(i)})$, náhodný pravděpodobnostní vektor $\boldsymbol{w}$ o složkách definovaných:

$$\forall j \in \hat{k} \quad w_j = \frac{1}{p} \sum_{i=1}^{p} \alpha_j^{(i)}$$

Jednotlivé složky tohoto vektoru udávají, jaká je pravděpodobnost výskytu jedničky na tom kterém místě v chromozomu. Dále stanovíme parametr χ:

$$\chi(w) = \frac{4}{k} \sum_{j=1}^{k} (w_j - 0{,}5)^2$$

Hodnoty tohoto parametru jsou z intervalu <0,1>. Nuly nabývá pro náhodnou populaci velmi se lišících chromozomů, jedničky pro populaci sestávající z totožných vektorů. Při průběhu algoritmu se jeho hodnota zvyšuje a pokud překročí určitou hranici (hodně chromozomů má na stejných pozicích nuly a jedničky), můžeme algoritmus ukončit.

1.4. Schémata

1.4.1. Co jsou to schémata

Kolem genetických algoritmů samozřejmě můžeme zkoumat mnoho rozličných teoretických problémů. Vzhledem k tomu, že v rámci této práce jsou genetické algoritmy použity pouze jako nástroj a nikoli cíl, budeme se zde věnovat pouze okrajově tomu nejdůležitějšímu, kterým je studium výskytu schémat v rámci průběhu genetického algoritmu.

Co jsou tedy schémata?

Mějme množinu chromozomů $M \subseteq \{0,1\}^k$, potom schématem nazveme řetězec $\xi \in \{0,1,*\}^k$, kde symbol * označujeme jako volný symbol. Řekneme, že chromozom $\alpha \in M$ je příkladem schématu ξ, pokud pro všechna $i \in \hat{k}$ platí: jestliže *i*-tý symbol schématu ξ je nula, resp. jedna, pak také *i*-tý symbol chromozomu α je nula, resp. jedna.

Jinak řečeno, schéma definuje množiny chromozomů, které mají na předepsaných místech předepsané znaky a na ostatních pozicích jsou znaky libovolné. Pokud bychom měli například schéma $\xi = 10{*}{*}1{*}$, pak jeho ukázkami jsou třeba chromozomy $\alpha = 101111, \beta = 100010, \gamma = 101010$, ale už ne chromozom $\delta = 100000$.

Dále se nám budou ještě hodit následující pojmy.

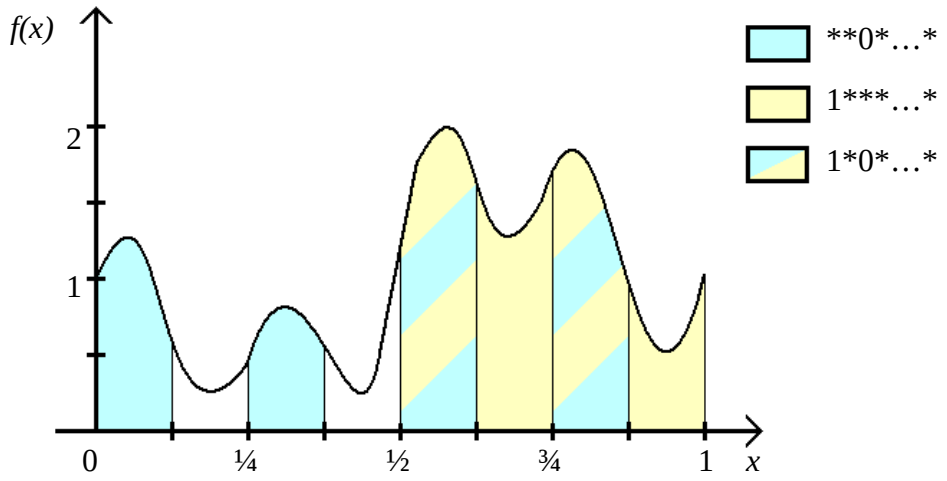
Symbolem $I(\xi)$ budeme označovat množinu všech příkladů schématu ξ. Řádem schématu ξ rozumíme počet nul a jedniček v daném schématu a označujeme ho $o(\xi)$. Symbol $\Delta(\xi)$ znamená délku schématu ξ a je definován jako rozdíl čísel pozic prvního a posledního binárního symbolu. Pomocí symbolu $N(\xi) = |I(\xi) \cap P|$ značíme počet příkladů schématu ξ v populaci *P*. Pro již zmiňované schéma $\xi = 10{*}{*}1{*}$ mají tyto pojmy následující hodnoty: $I(\xi) = \{100010, 100011, 100110, 101010, 100111, 101011, 101110, 101111\}$, $o(\xi) = 3$, $\Delta(\xi) = 5 - 1 = 4$, $N(\xi)$ záleží ne konkrétní populaci.

1.4.2. Využití schémat

V této podkapitole si osvětlíme, k čemu jsou schémata užitečná.

Genetické algoritmy používáme především tehdy, pokud hledáme globální extrém funkce s mnoha lokálními extrémy a s velkým definičním oborem. Prohledávání celého definičního oboru by mohlo být zdlouhavé a výpočetně náročné. Proto by se nám hodilo náš algoritmus nějak vylepšit, aby prohledával pouze „slibné“ části definičního oboru. Slovy genetických algoritmů si přejeme, aby se v populacích objevovaly hlavně takové chromozomy, o kterých víme, že obsahují na konkrétních pozicích hodnoty, dávající společně s hodnotami z ostatních pozic (a nezávisle na nich) vždy lepší než průměrný výsledek. Jinak řečeno, hledáme vhodná schémata.

Jako ilustraci si představme jednorozměrnou funkci *f(x)* s definičním oborem <0,1>. Nechť hodnoty *x* jsou kódovány chromozomy délky 20 standardním binárním kódováním. Potom schéma 1**…* popisuje všechny hodnoty proměnné *x* ležící mezi jednou polovinou a jedničkou a schéma **0*…* kóduje $x \in <0,1/8) \cup <1/4,3/8) \cup <1/2,5/8) \cup <3/4,7/8)$. Spojení těchto dvou schémat, tedy schéma 1*0*…*, potom definuje hodnoty *x* ležící v průniku hodnot kódovaných předchozími dvěma schématy. Pokud by funkce *f(x)* měla průběh jako na obrázku 4, byly by chromozomy definované schématem 1*0*…* velmi výhodné pro hledání maxima této funkce.



Obrázek 4: Schémata a jednorozměrná funkce

Pokud uvažujeme vícerozměrnou funkci, situace se příliš nezmění – jednotlivé části chromozomu reprezentující různé proměnné můžeme chápat jako samostatné jednorozměrné funkce.

1.4.3. Přežití schémat

Zajímavým problémem je zkoumání toho, jaká je pravděpodobnost, že při reprodukci vznikne chromozom se stejným schématem jako měl jeho předchůdce, tedy pravděpodobnost přežití schématu.

Jak už dobře víme, reprodukce sestává z křížení a mutace. Výpočet pravděpodobnosti, že dané schéma zůstane zachované po mutaci, je snadný. Pravděpodobnost zachování hodnoty na libovolné pozici je $(1-P_{mut})$, kde P_{mut} je pravděpodobnost mutace. Jelikož po mutaci musí zůstat stejných $o(\xi)$ pozic s předepsanými nulami a jedničkami, je pravděpodobnost přežití schématu po aplikování operátoru mutace $(1-P_{mut})^{o(\xi)} \approx 1-o(\xi)P_{mut}$.

S přežitím schématu po aplikaci operátoru křížení je to poněkud složitější. Nejprve uvažujme, že se mezi sebou kříží chromozomy s rozdílnými schématy. Potom bude pravděpodobnost zániku jednoho ze schémat $P_{cross}\Delta(\xi)/k-1$, kde P_{cross} je pravděpodobnost, že dojde ke křížení, a *k* je délka chromozomu. Dále je jasné, že pokud se mezi sebou kříží dva chromozomy se stejným schématem, schéma nezaniká. Z toho plyne, že zničení schématu může způsobit pouze část populace *P* daná podílem $1-N(\xi)/|P|$. Celkem je tedy pravděpodobnost přežití schématu po aplikaci operátoru křížení rovna

$$1-P_{cross}\frac{\Delta(\xi)}{k-1}\left(1-\frac{N(\xi)}{|P|}\right)\approx 1-P_{cross}\frac{\Delta(\xi)}{k-1}$$

Dále můžeme uvažovat o tom, kolik chromozomů bude mít dané schéma ξ po provedení celé reprodukce. K tomu si zavedeme následující veličiny. Střední fitness schématu ξ definujeme jako

$$F(\xi)=\frac{1}{N(\xi)}\sum_{\alpha\in I(\xi)\cap P}F(\alpha)$$

kde $F(\alpha)$ značí fitness chromozomu α. Střední fitness populace *P* určuje vztah

$$\overline{F}=\frac{1}{|P|}\sum_{\alpha\in P}F(\alpha)$$

Pokud bychom při tvorbě nové generace chromozomů vůbec neaplikovali reprodukci (v algoritmu by platilo, že P_{repro} = 0, viz 1.3.4._), bude počet příkladů schématu ξ v nové populaci roven

$$N_{new}(\xi)=N_{old}(\xi)\frac{F_{old}(\xi)}{\overline{F}_{old}}$$

kde indexy *new* a *old* odkazují na novou a starou populaci.

Jestliže nyní zapojíme operátory mutace a křížení, dostaneme se k výslednému vztahu

$$N_{new}(\xi)=N_{old}(\xi)\frac{F_{old}(\xi)}{\overline{F}_{old}}\left(1-P_{cross}\frac{\Delta_{old}(\xi)}{k-1}\left(1-\frac{N_{old}(\xi)}{|P|}\right)\right)(1-P_{mut})^{o_{old}(\xi)}\approx$$

$$\approx N_{old}(\xi)\frac{F_{old}(\xi)}{\overline{F}_{old}}\left(1-P_{cross}\frac{\Delta_{old}(\xi)}{k-1}-P_{mut}o_{old}(\xi)\right)$$

Celé naše snažení z této podkapitoly potom shrnuje následující věta. Odvodil ji v roce 1975 John Holland a představuje jeden z hlavních teoretických výsledků genetických algoritmů:

Nechť pro populaci P_{old} platí, že $N_{old}(\xi)$ je počet chromozomů obsahujících schéma ξ v této populaci a $F_{old}(\xi)$ je střední hodnota jejich fitness. Nechť dále $\overline{F}_{old}$ je střední hodnota fitness populace P_{old}. Potom počet chromozomů $N_{new}(\xi)$ obsahujících schéma ξ v nové populaci splňuje následující nerovnost:

$$N_{new}(\xi) \geq N_{old}(\xi)\frac{F_{old}(\xi)}{\overline{F}_{old}}\left(1 - P_{cross}\frac{\Delta_{old}(\xi)}{k-1} - P_{mut}o_{old}(\xi)\right)$$

2. Neuronové sítě

2.1. Úvod

2.1.1. Historie

Na začátku povídání o genetických algoritmech jsem zdůrazňoval, jak důležité může být pro lidstvo, pokud začnou spolupracovat odborníci ze zcela odlišných oborů. V této kapitole si nastíníme, co vznikne, když spojí své síly neurobiolog a statistik.

Neurobiologem byl v našem případě Warren McCulloch a statistikem Walter Pitts. Tito dva pánové vydali roku 1943 článek[1], ve kterém navrhli jednoduchý matematický model nervové buňky, neboli neuronu. Tím položili základ nové vědní disciplíny, studiu a tvorbě umělých neuronových sítí. Jejich inspirací byla nervová síť živých organizmů, o které si něco povíme v následující podkapitole.

Tento článek odstartoval vlnu bouřlivého výzkumu. Už v roce 1949 publikoval Donald Hebb zákon učení neuronových sítí[2], neuronovými sítěmi se tehdy zabýval také např. John von Neuman. Další zajímavé výsledky přinesl v roce 1958 Frank Rosenblatt[3]. Zobecnil tehdy základní McCullochův a Pittsův model neuronu a vytvořil tak umělou neuronovou síť zvanou perceptron, kterou si později podrobněji popíšeme. Z perceptronu následně vyšlo mnoho dalších modelů – např. v roce 1959 síť ADALINE (Adaptive Linear Neuron, jeden neuron s několika vstupy a doplňkovým jednotkovým signálem) a mnohé další.

Zásadní obrat, který málem pohřbil celou tuto disciplínu, nastal v roce 1969. Tehdy publikovali matematici Marvin Minsky a Seymour Papert svou práci *Perceptrons*[4]. V ní došli k závěru, že umělé neuronové sítě mají jen velmi omezené možnosti, jelikož perceptron nedovede realizovat logickou funkci XOR. Problém prý sice zvládá síť vícevrstevný perceptron, pro ten ale zas není znám učící algoritmus a parametry sítě se musí nastavovat ručně. Proces je to velmi komplikovaný a z toho pánové vyvodili, že učící algoritmus ani neexistuje.

Tento závěr byl ve své době vzhledem k tehdejší úrovni výzkumu oprávněný. Avšak Minsky s Papertem si neuvědomili, že výzkum může pokročit dál. Díky jejich práci se však zkoumání neuronových sítí na více jak deset let téměř zastavilo, a tak to vypadalo, že nakonec měli pravdu.

Jenže co se nestalo. Na počátku osmdesátých let minulého století se opět objevily obrovské investice do výzkumu umělých neuronových sítí. Stalo se to v USA a zapříčinili to vědci ze skupiny DARPA (Defence Advance Research Project Association). Díky těmto penězům se v roce 1986 podařilo nezávisle na sobě LeCunuovi a týmu D. Rumelharta, G. Hintona a R. Williamse[5] vyzkoumat hledaný, údajně neexistující, algoritmus. Nazvali ho algoritmus zpětného šíření chyby (*Error Backpropagation of Gradient – BPG*) a dodnes platí za jeden z nejpopulárnějších algoritmů pro učení neuronových sítí.

Poté se do výzkumu v této oblasti opět vrhlo mnoho vědců z Ameriky, Evropy i Japonska. Za všechny jmenujme třeba fyzika J. J. Hopfielda a jeho práce popisující souvislost modelů neuronových sítí s fyzikálními modely magnetických materiálů[6] nebo T. Kohonena

[1] Warren McCulloch, Walter Pitts: A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biology, 1943
[2] Donald Hebb: The Organisation of Behavior, Whiley, New York, 1949
[3] Frank Rosenblatt: The perceptron: a probabilistic model for information storage and organization in the brain, Psychol Rev, 1958
[4] Marvin Minsky, Seymour Papert: Perceptrons; an Introduction to Computational Geometry, MIT Press, 1969
[5] D. Rumelhart, G. Hinton, R. Williams: Learning internal representations by error propagation, MIT Press, 1986
[6] např. J. J. Hopfield: Brain, neural networks, and computation, Reviews of Modern Physics, 1999

z Finska, který rozvinul Hebbovu myšlenku a vytvořil tak nový typ dvouvrstevné neuronové sítě, kde jedna vrstva má strukturu dvojrozměrné matice[7].

V současnosti výzkum neuronových sítí probíhá téměř po celém světě. Jeho aplikace můžeme najít v mnoha vědních disciplínách od techniky přes společenské vědy až k medicíně. Tím se ovšem uzavírá pomyslný kruh – umělé neuronové sítě vyšly od lékařských výzkumů nervů živých organizmů a dnes pomáhají lékařům léčit ta samá těla, která dala impuls k jejich vzniku.

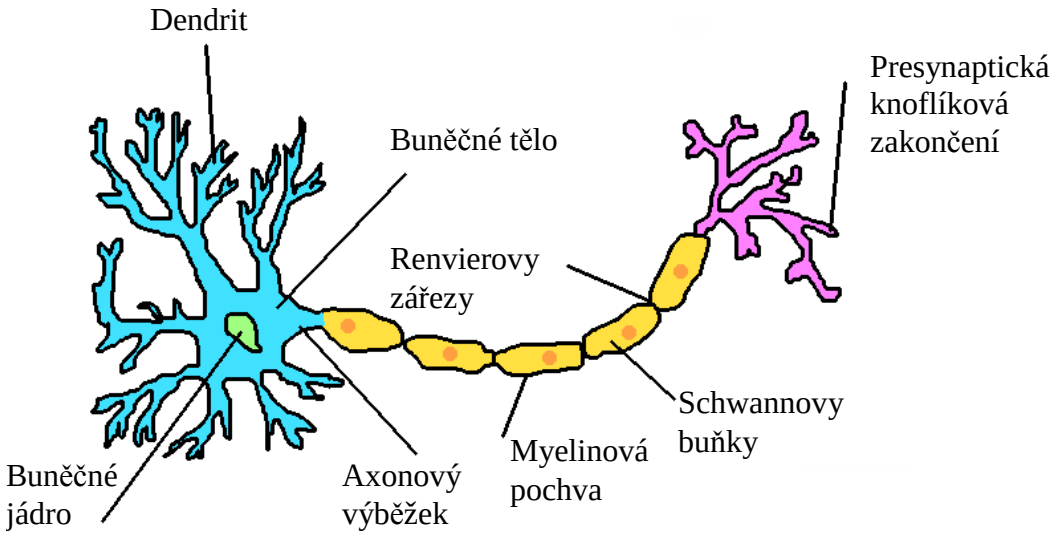
2.1.2. Biologické neuronové sítě

Jak už jsem se zmínil, inspirací pro umělé neuronové sítě byla nervová soustava vyšších živočichů, kterou bychom mohli také nazývat biologickou neuronovou sítí. V této podkapitole si trochu objasníme její stavbu a principy fungování, které nám následně pomohou lépe pochopit neuronové sítě umělé.

Cílem nervové soustavy je zachycení a zpracování podnětů působících na organizmus a zajištění odpovídající reakce na ně. Nervových soustav zná biologie několik typů, od rozptýlené přes žebříčkovitou a gangliovou až po centrální nervovou soustavu vlastní obratlovcům. Ta je tvořena mozkem, míchou a periferním nervstvem.

Jak všichni dobře víme, živé organizmy jsou tvořeny buňkami. Stejně tak je buňkami tvořena i jejich nervová soustava. Nervové buňky nazýváme neurony. Termín neuron je souborný název pro tělo buňky (soma, perikaryon) a její výběžky: dendrity a axon (osový válec, nervové vlákno). Dendrity jsou krátké, tenké, obecně je jich mnoho a přivádí vzruch do buňky. Naopak axon je dlouhý, tlustší, je jen jeden a odvádí vzruch z neuronu pryč. U člověka tvoří nervovou soustavu 40 až 100 miliard nervových buněk.

Neurony mají v jednotlivých částech těla různý tvar i velikost. Například v lidském mozečku měří tělo buňky jen 5 μm, naopak některé neurony v míše mají somu velkou až 120 μm. K tomu se ještě přidává axon, s velikostí od několika mikrometrů až po 90 cm. Jak průměrný neuron vypadá, můžeme vidět na obrázku 5:



Obrázek 5: Struktura neuronu

Ne všechny neurony jsou stejné. Z hlediska jejich funkce je můžeme rozdělit na tři skupiny: aferentní, eferentní a interneurony. Aferentní neurony jsou uzpůsobené k tomu, aby získávaly od smyslových orgánů (receptorů) nějakou informaci (vzruch) a předávali ji dál do

[7] T. Kohonen: Self-organization and associative memory, Springer-Verlag New York, Inc., New York, 1989

nervové sítě interneuronům. Ty si předávají vzruch jen mezi sebou a při tom ho nějak vyhodnocují. Nakonec se vzruch dostane až k eferentním neuronům, které ho pošlou ven z nervové soustavy nějakému svalu nebo žláze (efektor). Zapamatujme si toto rozdělení, neboť umělé neuronové sítě ho používají také.

Vzruch se mezi buňkami předává pomocí elektrických impulzů při takzvaných synapsích. Na konci axonu na buněčné membráně vzniká díky rozdílné koncentraci iontů K^+, Na^- a dalších iontů potenciál o hodnotě –50 až –90 mV. Pokud dojde ke změně potenciálu (depolarizaci membrány), která je vyšší než prahová hodnota, posílá neuron vzruch dál. Prahová hodnota není konstantní, ale závisí na konkrétní situaci.

Neurony v rámci neuronové sítě mají jen jednu možnost, jak navzájem komunikovat – pro daný vzruch se spojit do správné dráhy, která umí na daný podnět dobře zareagovat. Při učení tedy metodou pokus-omyl hledáme správné neurony, které by na danou potřebu reagovaly vytvořením a posílením patřičných propojení. A velmi podobně fungují i umělé neuronové sítě, o kterých si budeme povídat dál. Protože k biologickým neuronovým sítím se už příliš vracet nebudeme, v dalším textu rozumějme pod pojmem neuronová síť tu umělou.

2.2. Definice umělých neuronových sítí

2.2.1. Neuron

Neuronové sítě lze nadefinovat více způsoby, od těch méně formálních až po striktní matematické definice. Nejjednodušeji můžeme neuronovou sítí nazývat ohodnocený graf, jehož vrcholy jsou neurony. Tuto jednoduchou definici však budeme muset ještě rozvést, abychom se dostali k něčemu smysluplnému. K tomu potřebujeme vědět, co si představit pod pojmem neuron.

Stejně jako u biologických neuronových sítí jsou neurony základním stavebním kamenem i zde. Jenže v tomto případě se nejedná o buňky, nýbrž o určitý výpočetní model. Formálně bychom neuron mohli definovat následovně:

Buďte $n, m \in N$, $w \in W^n$, *kde* W *je množina přípustných ohodnocení hran grafu reprezentujícího neuronovou síť. Nechť dále* $x \in R^n$, $y \in R^m$, $\theta \in R$, $f : R^n \times W^n \times R \to R^m$, $f(x, w, \theta) = y$. *Pak uspořádanou čtveřici (f, θ, n, m) nazveme neuron a funkci f přechodová funkce neuronu.*

Jak vypadá neuron konkrétně, si můžeme ukázat na příkladu původního McCulloch-Pittsova neuronu. Vektor *x* je v tomto případě vstupní vektor se složkami obsahujícími data z předcházejících neuronů v dané síti. Vektor *w* představuje modifikátor dat z jednotlivých předcházejících neuronů. Nazýváme ho vektorem synaptických vah a právě díky jeho úpravám můžeme naši neuronovou síť něco naučit. Výstupní vektor *y* má všechny složky stejné a jejich počet je roven počtu neuronů následujících v další vrstvě neuronové sítě.

Přechodová funkce *f* je složena ze dvou funkcí – obvodové funkce *u* a aktivační funkce *g*. Obvodová funkce slouží pro kombinaci vstupního vektoru *x* a vektoru synaptických vah *w*. Zároveň v sobě obsahuje další parametr neuronu – tzv. práh neuronu *θ*. Ten udává, kdy bude neuron posílat něco dál a kdy zůstane nečinný. Tento práh je analogický prahové hodnotě u biologického neuronu. Obvodová funkce mívá nejčastěji tvar:

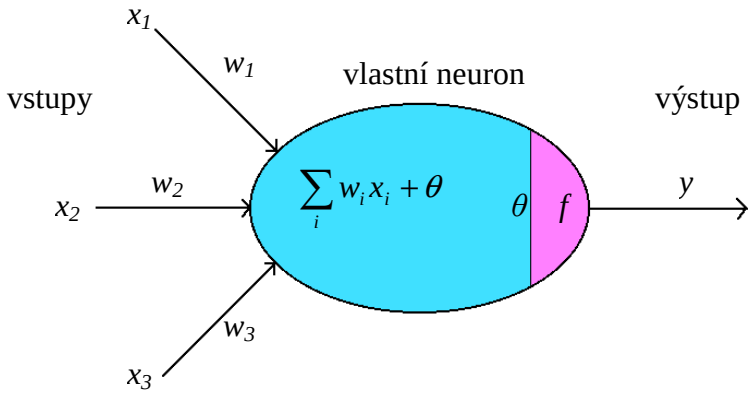
$$u(x, w, \theta) = \sum_{i=1}^{n} w_i x_i + \theta$$

ale může mít i jiný.

Aktivační funkce *g* slouží jako modifikátor výsledku obvodové funkce *u*. Nejčastěji se používají funkce:

$$f(u) = \frac{1}{1+e^{-u/T}},\quad f(u) = \tanh\left(\frac{u}{T}\right),\quad f(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases} \text{ apod.}$$

Jak takovýto neuron vypadá, můžeme vidět na následujícím obrázku. Významy jednotlivých symbolů jsou stejné, jako v předchozím textu.



Obrázek 6: McCulloch-Pittsův neuron

2.2.2. Neuronová síť

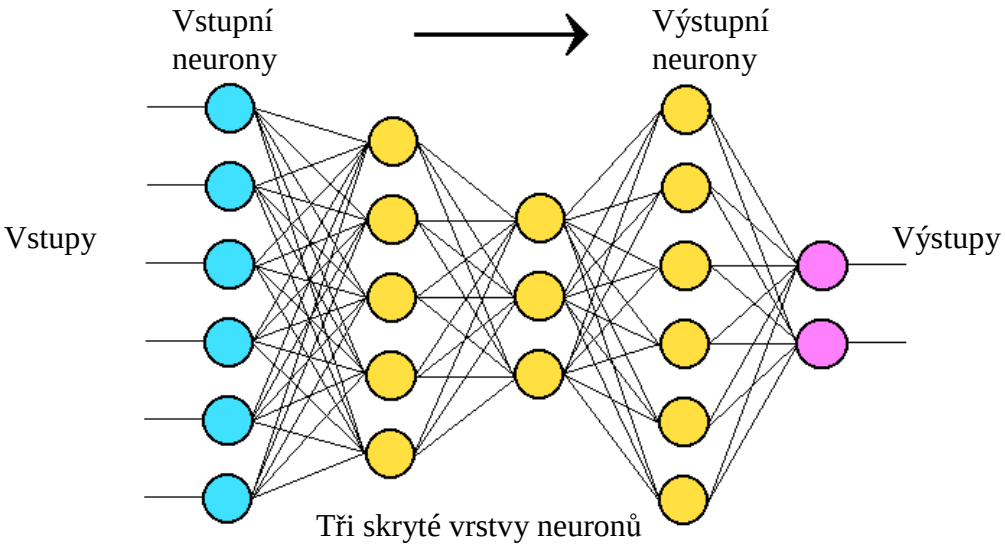
Když už víme, jak si představit neuron, můžeme rozvést definici neuronové sítě a formalizovat ji.

Buď $U = \{(f_1, \theta_1, n_1, m_1), (f_2, \theta_2, n_2, m_2), \ldots, (f_l, \theta_l, n_l, m_l)\}$ *množina neuronů, označme* $u_i = (f_i, \theta_i, n_i, m_i)$ *i-tý neuron. Buď dále* $G = (I \cup V \cup O, E, w)$ *orientovaný ohodnocený graf, kde I, V a O jsou disjunktní množiny vrcholů grafu G,* $E \subset (V \times V) \cup (I \times V) \cup (V \times O)$, $\forall v \in O\ d(v) = 1$ *a* $w : E \to W$ *je ohodnocení grafu. Nechť existuje bijektivní zobrazení* $b : V \to U$ *, pro které platí* $\forall i \in \hat{I}\ \forall v \in V\ b(v) = u_i \Rightarrow ind(v) = n_i \wedge outd(v) = m_i$ *, kde ind(v) označuje počet hran vstupujících do vrcholu a outd(v) počet hran vycházejících z vrcholu.*

Potom uspořádanou šestici $NN = (V, I, O, G, U, b)$ *nazveme neuronovou sítí, množinu vrcholů I, resp. O, vstupy, resp. výstupy, neuronové sítě, čísla* $|I|$ *, resp.* $|O|$ *, vstupní, resp. výstupní, dimenze neuronové sítě. Graf G označujeme jako strukturu neuronové sítě.*

Tato formální definice nepopisuje nic jiného, než co jsme si už intuitivně pod pojmem neuronové sítě představovali. Na začátku máme |*I*| vstupů, které předáme některým neuronům. Tyto neurony nazýváme vstupní a jsou analogické biologickým aferentním neuronům. Neurony vstupy zpracují pomocí svých přechodových funkcí a pošlou je přes odpovídající výstupní hrany další množině, resp. vrstvě neuronů. Informace se opět upraví a předá dál. Vrstvy těchto neuronů se nazývají skryté a odpovídají interneuronům v živých organizmech. Poslední vrstva neuronů, zvaná výstupní a analogická s biologickými eferentními neurony, vydá konečné výstupy neuronové sítě. Tento poměrně obecný typ

neuronové sítě se nazývá vícevrstvá neuronová síť a její příklad můžeme vidět na následujícím obrázku:



Obrázek 7: Struktura vícevrstevné neuronové sítě

Neuronovou síť můžeme definovat i abstraktněji jako systém sestávající z architektury, dynamiky a úlohy. Úlohou v tomto případě rozumíme zobrazení realizované mezi množinou vstupů a výstupů. Konkrétně to může být například separace tříd, aproximace daného zobrazení či interpolace množiny. Architekturou nazýváme orientovaný graf totožný s orientovaným grafem z předchozí definice společně s popisem přechodových funkcí jednotlivých neuronů. Konečně dynamika znamená změny parametrů přechodových funkcí a grafu v čase. Rozlišujeme aktivní dynamiku (vybavování), při které pouze prochází informace ze vstupů sítí a na jednotlivých neuronech se modifikuje, a adaptivní dynamiku (učení), při které se mění ohodnocení hran grafu a prahové hodnoty jednotlivých neuronů. Pokud se nad touto definicí zamyslíme, uvidíme, že definuje totéž, co předchozí. Navíc se tu ovšem objevují i dynamické vlastnosti neuronových sítí – co umí síť řešit a proces změn sítě.

2.3. Typy neuronových sítí

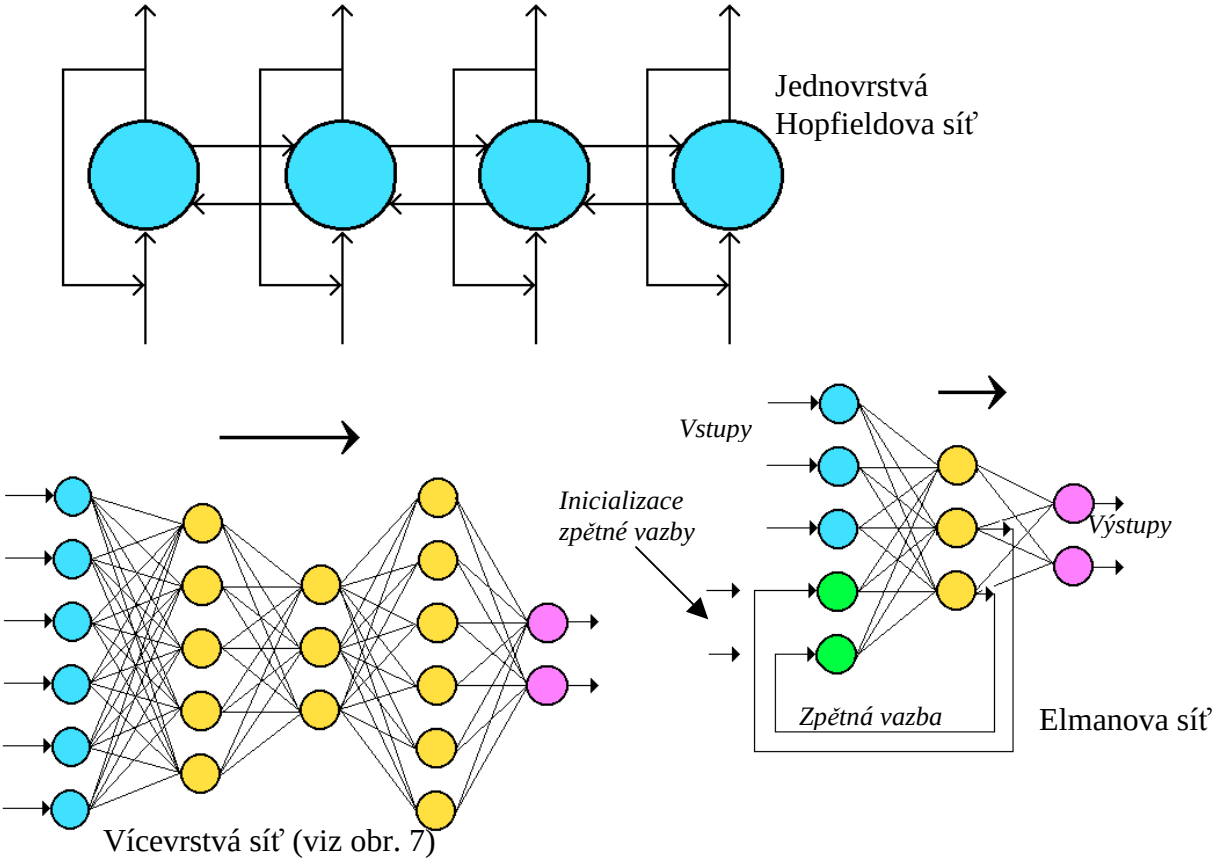
2.3.1. Rozlišení jednotlivých typů neuronových sítí

Jak jsme si popsali v minulé kapitole, důležitou charakteristikou každé neuronové sítě je orientovaný graf popisující její strukturu. Tento graf může být různý – měnit lze počty vrstev, propojení mezi jednotlivými vrstvami sítě nebo v rámci těchto vrstev, lze přidávat zpětné vazby z vrstvy blíže k výstupu do některé předcházející a podobně. Pomocí všech těchto postupů můžeme odvozovat stále nové a nové typy neuronových sítí. Zároveň lze různě měnit přechodové funkce neuronů a způsoby nastavování vah spojnic mezi nimi. Tak vznikají další spousty různých mutací.

Jelikož cílem této práce není přinést jejich přehled, uvedeme si pouze nákresy několika příkladů různých struktur neuronových sítí. V dalších podkapitolách potom podrobněji popíšeme jeden z nejprobádanějších typů neuronových sítí zvaný perceptron a neuronové sítě s přepínacími jednotkami, které se přímo vztahují k praktické části této práce.

Příkladem jednovrstvé sítě je tzv. Hopfieldova síť. Obsahuje neurony pouze v jedné vrstvě, kde je spojen každý s každým. Typickou vícevrstevnou síť jsme si popsali na

předchozí straně. Konečně jako příklad neuronové sítě se zpětnou vazbou můžeme uvést tzv. Elmanovu síť. Strukturu všech těchto tří typů porovnává obrázek 8.



Obrázek 8: Příklady různých typů neuronových sítí

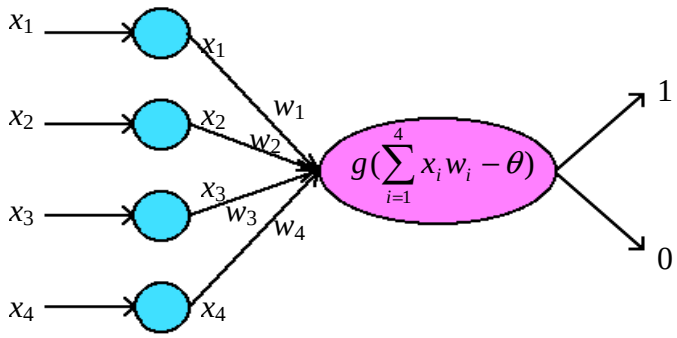
2.3.2. Perceptron

Nejprobádanější a zároveň velmi jednoduchou neuronovou sítí je síť zvaná perceptron. Jedná se o dvouvrstvou síť. Ve vstupní vrstvě nalezneme *n* neuronů, které pouze posílají to, co dostanou, do další vrstvy. Tu tvoří jediný neuron, který na základě výpočtu rozhodne, zda bude výstup nula nebo jedna[8]. Formálně se výstup perceptronu y se vstupy $\{x_1, x_2, \ldots, x_n\} \in R^n$ spočítá jako:

$$y = g\left(\sum_{i=1}^{n} x_i w_i - \theta\right), \quad g(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases}$$

kde w_i jsou složky vektoru synaptických vah od výstupního neuronu a θ jeho prahová hodnota. Situaci shrnuje následující obrázek.

[8] V některé literatuře lze najít, že může přiřadit i něco mezi nulou a jedničkou. Záleží na tom, zda jako aktivační funkci budeme vždy vyžadovat tzv. tvrdou nelinearitu ($g(u)=0 \quad u<0$, $g(u)=1$, $u\geq 0$) nebo nám postačí i sigmoidální funkce (spojitý přechod od nuly k jedné).



Obrázek 9: Schéma perceptronu

Vhodně nastavený perceptron je tedy díky své stavbě předurčen především k separaci konvexních disjunktních množin – prvkům z množiny A bude přiřazovat jedničku a prvkům B nulu.

Jak probíhá „nastavení“, neboli učení, perceptronu?

Nejprve je třeba zdůraznit, co můžeme na perceptronu měnit. Jsou to hodnoty w_i složek vektoru synaptických vah a prahová hodnota θ, celkem tedy n+1 parametrů.

Učení perceptronu probíhá pomocí tzv. perceptronového trénovacího (učícího) algoritmu (Perceptron learning rule), které patří mezi chybová učení:

- na vstup pustíme učící vektor x, síť vrátí výstup y
- výstup y porovnáme s cílovou hodnotou c_x, které chceme pro daný vektor x dosáhnout, definujeme chybu $e = c_x - y$
- změníme parametry neuronu:
  - $w_i(t+1) = w_i(t) + ex_i$
  - $\theta(t+1) = \theta(t) + e$
- pokračujeme s dalším učícím vektorem od začátku

Jak si můžeme snadno rozmyslet, tento algoritmus má tu vlastnost, že pokud si přejeme získat $c_x = 0$ a získáme $y = 1$, změní se hodnoty parametrů tak, aby v následující iteraci vzrostla pravděpodobnost výsledku 0 a naopak. Dále lze také vidět, že velikost chyby závisí na velikosti vstupů. Proto je vhodné nerovnoměrně velké učící vstupní vektory normalizovat.

2.3.3. Neuronové sítě s přepínacími jednotkami

Jak jsme se již zmínili v úvodním historickém přehledu, hlavním nedostatkem jednoduchého perceptronu je, že nedovede provést logickou operaci XOR, tedy rozdělit rovinu přímkou tak, aby v jedné polorovině ležely body (0,0) a (1,1) a v druhé (0,1) a (1,0). Také jsme si řekli, že vícevrstevná neuronová síť složená z perceptronů to dokáže. Problémem je, že učení takovéto sítě trvá příliš dlouhou dobu na to, aby se dala použít v aplikacích pracujících v reálném čase.

Způsob, jak tento problém obejít, navrhla skupina českých vědců v roce 1994. Jsou jím neuronové sítě s přepínacími jednotkami (NNSU). Jelikož právě s tímto typem neuronových sítí se setkáme v následující části této práce, bude dobré si ho podrobněji představit.

NNSU patří mezi vrstevnaté sítě. Obsahuje tři druhy jednotek (uzlům grafu nejsou přiřazeny pouze klasické neurony, ale i jiné jednotky) – přepínací jednotky, výpočetní jednotky a výstupní jednotku. Jednotlivé typy si nyní popíšeme.

Prvním typem jsou tzv. přepínací jednotky. Signál, který jimi prochází, není nijak modifikován, je pouze směrován do jednoho z jejich potomků. Formálně je můžeme vyjádřit pomocí nespojité přepínací funkce *S*:

$$S : R^d \times U \rightarrow \{1, 2, \ldots, m\}, \quad S = S(x, u), \quad x \in R^d, \quad u \in U,$$

kde vektor *x* představuje signál přicházející z *d* předků přepínací jednotky, *u* je vektor parametrů z daného parametrického prostoru *U* a číslo *m* udává počet potomků přepínací jednotky. Jednoduše řečeno, přepínací jednotka funguje jako výhybka na nádraží pouštějící vlaky na jednotlivá nástupiště.

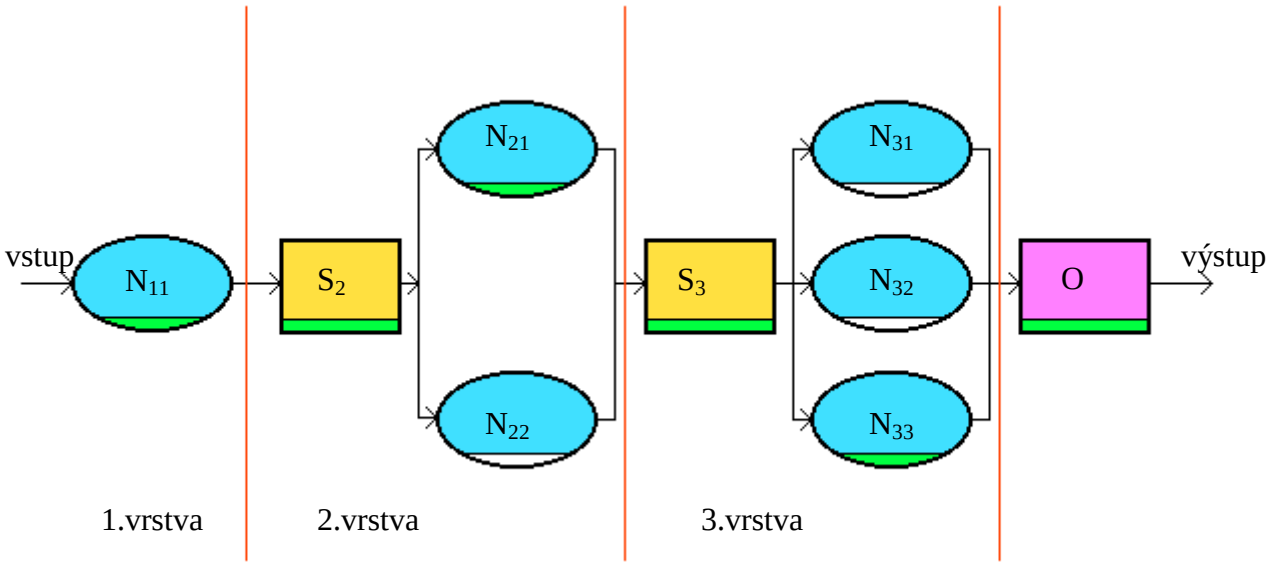
Druhý typ, výpočetní jednotky, jsou klasické neurony představené v předchozím textu (viz podkapitola 2.2.1.).

Poslední typ jednotek NNSU představuje výstupní jednotka. Ta je v celé síti pouze jedna a to těsně před výstupem. Jedinou její činností je projekce *d*-dimenzionálního výstupu do nižší nebo stejné dimenze. Formální zápis je následující:

$$O : R^d \rightarrow R^h, \quad O = O(x), \quad x \in R^d, \quad h \leq d$$

Můžeme ji například využít jako rozhodovací modul při separaci množin.

Nyní známe všechny stavební kameny, můžeme si tedy ukázat, jak jsou pospojovány dohromady. Příklad ukazuje následující obrázek:



Obrázek 10: Příklad neuronové sítě s přepínacími jednotkami

Můžeme vidět, že NNSU na obrázku je trojvrstvá. N_{ii} označují výpočetní jednotky, S_i přepínací jednotky a O jednotku výstupní. Všimněme si struktury jednotlivých vrstev. Každá (s výjimkou první, kde to není třeba, neboť první vrstva obsahuje pouze jeden neuron) se skládá z přepínací jednotky a jejích potomků, kterými jsou výpočetní jednotky. Zelené spodní části jednotek ukazují možný průběh signálu označenými uzly. Charakteristické je, že signál prochází všemi přepínacími jednotkami a právě jednou výpočetní jednotkou v každé vrstvě. Je tedy modifikován tolikrát, kolik vrstev síť obsahuje.

Pomocí přepínacích jednotek v závislosti na jejich parametrech u dovedeme rozdělit množinu přípustných signálů na tolik disjunktních množin, kolik různých cest vede mezi vstupem a výstupem. Tento počet je zřejmě roven součinu počtů výpočetních jednotek ve všech vrstvách (tedy pro příklad NNSU z našeho obrázku by to bylo 1.2.3 = 6 cest a tedy šest disjunktních množin). Tvar množin určují parametrické vektory u jednotlivých přepínacích jednotek.

Při učení NNSU můžeme volit počet vrstev a počty výpočetních jednotek v jednotlivých vrstvách a nastavovat hodnoty vektorů synaptických vah w_{ij} od všech neuronů a parametrické vektory u_i od přepínacích jednotek. Učení probíhá postupně ve směru od vstupu k výstupu. Výhodou je, že můžeme nastavovat každou jednotku zvlášť nezávisle na ostatních. V neuronové síti z našeho příkladu bychom tedy nejprve učili výpočetní jednotku N_{11}, následně přepínací jednotku S_2 atd. Postupovali bychom systematicky, jednotku za jednotkou, vrstvu za vrstvou. Vlastní postup nastavování jednotlivých parametrů lze nalézt v literatuře.

2.3.4. Zobecnění NNSU

Hlavní výhodou neuronových sítí s přepínacími jednotkami je vysoká rychlost učení v porovnání s učením vícevrstvých perceptronů. Z tohoto důvodu byly NNSU vybrány jako vhodný prostředek pro vývoj aplikace, k jejímuž rozvoji by měla přispět i tato práce. V tomto rozsáhlém programu, o kterém si něco řekneme v příští kapitole, se vyskytují velká množství neuronových sítí a proto je velmi důležité, abychom je dokázali učit rychle.

V průběhu vývoje aplikace se lidé snažili základní model NNSU popsaný v předchozí kapitole ještě vylepšit a zobecnit.

Prvním pokusem bylo zavedení libovolné acyklické struktury sítě. To vedlo k například tomu, že mohly vést hrany mezi přepínacími jednotkami nebo že výpočetní jednotka mohla mít více rodičů. Při takovémto zobecnění však mohou nastat problémy s nejednoznačností vstupní dimenze výpočetní jednotky, do níž vedou hrany ze dvou přepínacích jednotek. Pokud by totiž posílaly signály o dimenzích n a m obě naráz, byla by velikost vstupní dimenze $n+m$. Když by však signál přicházel pouze od jedné, byla by jeho dimenze n, resp. m, což je nejednoznačné.

Z tohoto důvodu se přistoupilo na další omezení, která tento nedostatek eliminovala. Řeklo se, že:

- potomkem přepínací, resp. výpočetní, jednotky může být pouze výpočetní, resp. přepínací, jednotka
- každá výpočetní jednotka má jediného rodiče
- potomci přepínací jednotky mají společné potomky a shodné typy

Výsledkem byly neuronové sítě schopné řešit méně komplikované úlohy. Pro složitější případy sice vycházely smysluplné výsledky, které však nebyly dostačující. Proto se při dalším výzkumu vrátilo zpět k původním NNSU, které se pomalu dále rozšiřují. Více lze nalézt v použité literatuře.

3. Projekt NNSU

3.1. Představení projektu

3.1.1. Co je to projekt NNSU

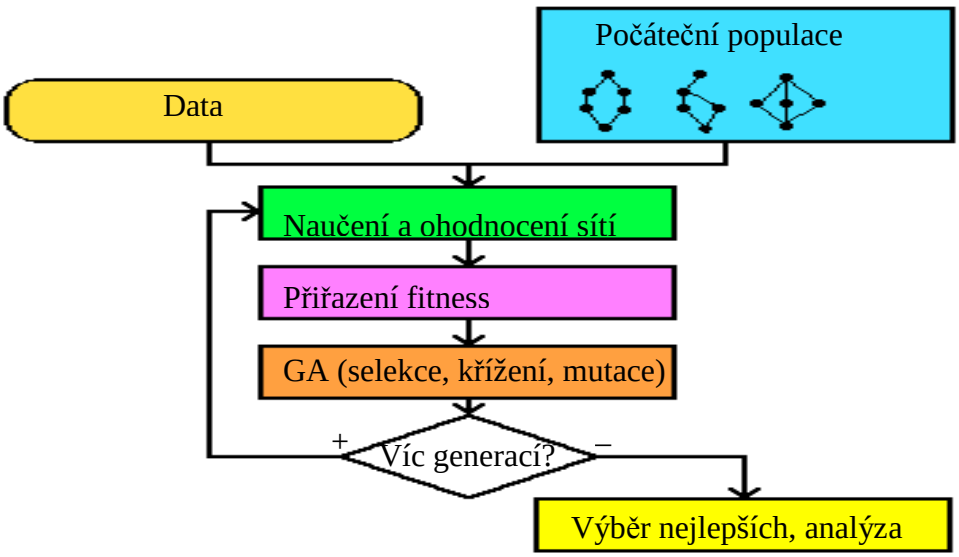
Jedním z cílů této práce je pomoc dalšímu rozvoji projektu vývoje univerzálního separačního nástroje založeného na aplikaci neuronových sítí optimalizovaných genetickými algoritmy. Projekt se pracovně označuje NNSU a v této kapitole se s ním seznámíme. Poté zde předvedu i své výsledky.

V současné době je projekt NNSU již poměrně rozsáhlý. Jeho duchovním otcem a stálou hlavní hybnou silou je pan Ing. František Hakl, CSc. z Ústavu informatiky AV ČR. Kolem pana Ing. Hakla se postupně vytvořil celý tým, složený především ze studentů a doktorandů katedry matematiky naší fakulty. V průběhu let se složení týmu různě mění a tak v současnosti je jeho členy již mnoho lidí, kteří jsou specialisty na konkrétní části problému.

Cílem projektu NNSU je vyvinout program, který dovede separovat dvě množiny dat. Hlavních úspěchů bylo dosaženo při detekci Higgsova bosonu v simulovaných datech stavěného LHC (*Large Hadron Collider*) detektoru v CERNu. Tento problém se převedl na klasický problém rozpoznávání vzorů řešitelný pomocí neuronových sítí. Zbývalo rozhodnout, jaký typ neuronových sítí použít a jak je optimalizovat.

Tým Ing. Hakla se rozhodl pro neuronové sítě s přepínacími jednotkami popsané v předchozí kapitole. Jako prostředek jejich optimalizace byly vybrány genetické algoritmy.

Jak tedy program zhruba funguje? Na začátku dostane nějaká data, která jsou rozdělena na učící a testovací. Vytvoří se počáteční populace jedinců, z nichž každý představuje neuronovou síť s přepínacími jednotkami. Těmto jedincům se pomocí učících dat nastaví parametry a následně se na ně pošlou testovací data. Podle toho, jak úspěšně si s nimi jedinci poradí, jsou ohodnoceni nějakou hodnotou fitness. Pak nastupuje genetický algoritmus popsaný v první kapitole. Díky němu vznikají další a další generace jedinců (tedy neuronových sítí). Jejich fitness se získává obdobně jako u první generace a stále narůstá. Obvykle se pracuje s desítkami jedinců v dané generaci a stovkami generací. Po skončení běhu genetického algoritmu se vyberou ty nejlepší neuronové sítě. Schématicky je to znázorněno na následujícím obrázku:



Obrázek 11: Schéma NNSU

3.1.2. Implementace programu

Současná verze implementace NNSU je už třetí. Z původního jednoduchého procedurálního návrhu se vyvinul velmi komplexní a abstraktní program.

Základem je virtuální jádro, se kterým může většina členů týmu pracovat jako s černou skříňkou, neboť nemá na jejich konkrétní úkoly přímý vliv. K tomuto jádru se připojují jednotlivé moduly s částmi projektu, zvané pluginy. Existuje plugin s grafickým rozhraním, plugin s genetickými algoritmy, plugin pro neuronové sítě a mnohé další. Parametry pro pluginy se zadávají pomocí konfiguračních souborů ve formátu XML.

Tvorba a připojení nového pluginu není příliš obtížná (viz dále). To dává vývojářům možnost pracovat na svém pluginu, aniž by museli vědět, co se děje v ostatních modulech. Vše je zpracované tak, aby se při změně pluginu či jeho parametrů nemusel přestavovat celý program. Je to velmi efektivní a funkční, avšak daní za tyto výhody je snížená možnost porozumění celému kódu.

Celý program se spouští pomocí podprogramu Starter. Pokud se Starter spustí bez parametrů, objeví se přívětivé grafické rozhraní, vhodné pro uživatele. Druhým módem je spouštění s parametrem, kterým je XML soubor. V něm si můžeme vybrat konkrétní spustitelný plugin a následně tam také zadáme požadované hodnoty jeho proměnných.

3.1.3. Současný vývoj

V současnosti NNSU funguje vždy na jednom konkrétním počítači, na kterém se nainstaluje. K dispozici je verze pro Linux i Windows. Rychlost zpracování je přibližně 5000 jedinců za hodinu. V nejbližší době se však uvažuje o tom, že by se aplikace upravila tak, aby mohla běžet na nějakém centrálním serveru a uživatelé se k ní mohli připojovat přes internet. Výhodné by to bylo z několika důvodů:

- nemusely by se dohledávat a instalovat aktuální verze knihoven[1]
- urychlení výpočtů v závislosti na serveru
- velmi perspektivní pro použití paralelních výpočtů

Jako o serveru se uvažuje o klastru v Ústavu informatiky s 24 procesory, ve Fyzikálním ústavu s asi stovkou procesorů nebo dokonce o výpočetním systému v CERNu s několika sty procesory, což by rychlost zpracování velmi urychlilo. Právě díky tomu by byla paralelizace kódu velmi výhodná.

3.1.4. Přidání nového pluginu

Abychom alespoň částečně ukázali, jak se při vývoji programu nyní postupuje, popíšeme na tomto místě postup pro přidání nového pluginu k aplikaci.

Plugin nazveme Hello a jediné, co bude umět, je, že po spuštění Starteru s příslušným XML souborem jako parametrem vypíše v terminálu obligátní „Hello, World!“.

Vytvořit k tomu bude třeba jeden nový adresář a šest souborů. Adresář se bude jmenovat stejně jako plugin, tedy *Hello*. Umístit ho musíme k adresářům s ostatními pluginy do složky **/NNSU/src/plugin/*. V tomto adresáři můžeme pro větší přehlednost vytvořit adresáře *include* pro hlavičkové soubory a *src* pro zdrojové, není to ale nutné.

Prvním souborem, který vytvoříme v našem novém adresáři, bude hlavičkový soubor *hello.h*. Po spuštění našeho pluginu *Hello* se vytvoří objekt, odvozený ze základního *basicinterface.h*. Metody *Hello* budou čtyři, kromě konstruktoru a destruktoru to budou ještě *start* a *classId*. Metoda *start* není nutná přímo k funkčnosti pluginu, ale pokud chceme plugin spustit pomocí XML parametru, přítomna být musí. Bez ní se nám objeví hláška o nemožnosti

[1] Což není vůbec triviální záležitost. Uvést NNSU do provozu pro neznalého uživatele je práce na několik dní.

pluginu *Hello* být starterem. Pomocí *classId* se náš plugin registruje do hlavního programu a tato metoda je tedy nutná. Zdrojový kód bude vypadat následovně:

hello.h:

```
#ifndef HELLO_H
#define HELLO_H

#include "basicinterface.h"

class Hello:public BasicInterface{

	public:
		Hello(buildId zBuildId);
		virtual ~Hello();
		void start(CoreStringList* errors);
	protected:
		const char* classId() const;
};

#endif
```

Dalším souborem, který vytvoříme v našem adresáři, je *hello.cpp* s popisem metod. V tomto nejminimálnějším pojetí obsahuje pouze dva includované soubory, obvykle jich však bývá mnohem víc. V projektu je totiž připravena spousta různých pomocných věcí, jako např. *messager* v hlavičkovém souboru *useful.h* pro výpis zpráv ve stylu C++ příkazu *cout* apod. V našem příkladu se konstruktor se dědí po *BasicInterface*, objektu, z kterého jsou odvozeny všechny ostatní pluginy. Metoda *classId* pouze vrací hodnotu HELLO_ID, nutnou pro registraci pluginu (viz dále). Konečně pomocí *startu* udělá plugin požadovanou činnost, tedy vypíše na obrazovku „*Hello World!!!*“.

hello.cpp:

```
#include "hello.h"
#include "hello.typ"

Hello::Hello(buildId zBuildId):BasicInterface(zBuildId){}

Hello::~Hello(){}

const char* Hello::classId() const{
	return HELLO_ID;
}

void Hello::start(CoreStringList* errors){
	printf("Hello World!!!\n");
}
```

Třetí soubor v adresáři *Hello* se nazývá *create.cpp*. Pomocí tohoto souboru se vytvoří instance objektu *hello*. Vlastní tvorbu má na starosti *basicinterface*. Užití *extern “C“* je zde proto, že C++ si ke jménu funkce přidává informace o typech argumentu, což by mohl být při překladu problém, neboť v dynamické knihovně je funkce vázána na jméno. Protože samotné C si nic nepřipojuje, používá se zde ono. V konstrukci *if* se srovnává hodnota parametru *type* od metody *classId* z *hello.cpp* s hodnotou HELLO_ID, popsanou v souboru *hello.typ* (viz dále).

create.cpp:

```
#include "hello.h"
#include "hello.typ"
#include <cstring>

extern "C" BasicInterface* create(const char* Type,buildId BuildId){

	if (!strcmp(Type,HELLO_ID)){
		return new Hello(BuildId);
	}
	return 0;
}
```

Soubor *hello.typ*, čtvrtý, který je nutno vytvořit, je obyčejný textový soubor, mající funkci céčkového hlavičkové souboru. Na rozdíl od předchozích se neukládá do adresáře *Hello*, ale spolu s ostatními soubory pluginů **.typ* do **/NNSU/src/interface/*.

hello.typ:

```
#define HELLO_ID "Hello"
```

Pátým souborem nutným pro překlad[2] je *makefile*. Ukládá se opět do adresáře *Hello*. Pro všechny pluginy vypadá téměř stejně, jen se upravuje cesta. Jeho kód zde uvádět nebudu, jelikož jsem pouze použil univerzální šablonu z jiného pluginu.

Posledním, šestým, souborem je *hello.xml* pro spuštění pluginu. Jelikož nebudeme zadávat žádné parametry, je jeho struktura jednoduchá. Umístění není podstatné, jen si musíme zapamatovat cestu k tomuto souboru:

hello.xml:

```
<?xml version="1.0" ?>
  <Root type="Hello" />
```

Nyní, když už máme připraveny všechny soubory, můžeme plugin zprovoznit. Před vlastním spuštěním ještě musíme upravit soubor *Makefile* v adresáři **/NNSU/src* – je třeba do něj připsat, že se má překládat i náš plugin. Následuje už vlastní překlad, instalace a spuštění:

- přepneme se do adresáře **/NNSU*
- zadáme nejprve *make* a poté *sudo make install*
- instalační program aktualizuje (tedy přeloží jen nové nebo změněné části programu) program NNSU a připojí náš nový plugin
- přepneme se do adresáře **/NNSU/NNSU/bin/*
- spustíme program zadáním *./starter cesta_k_xml_souboru/hello.xml*

Pokud vytvoříme plugin popsaným způsobem, získáme výpis uvedený na následujícím obrázku. Potěšit by nás měl předposlední řádek s nápisem *„Hello World!!!“* značící, že náš plugin funguje.

[2] Při práci na systému Linux

```
hofta@ubuntu1:~/nnsu/nnsu/bin$ ./starter /home/hofta/nnsu/src/plugin/hello/hello.xml
looking for plugins in pattern:*.typ
dir:../plugin/
factory: registering plugin experiment.typ
factory: registering plugin randomnnlearn.typ
factory: registering plugin starters.typ
factory: registering plugin nn.typ
factory: registering plugin netgenerator.typ
factory: registering plugin costfunctions.typ
factory: registering plugin representation.typ
factory: registering plugin gui.typ
factory: registering plugin ga.typ
factory: registering plugin test.typ
factory: registering plugin evaluation.typ
factory: registering plugin visual.typ
error while loading plugin: Factory:error loading plugin:../plugin/visual.plg
libdotneato.so.0: cannot open shared object file: No such file or directory
factory: registering plugin fitness.typ
factory: registering plugin hello.typ
Hello World!!!
hofta@ubuntu1:~/nnsu/nnsu/bin$
```

Obrázek 12: Výpis programu NNSU po přidání pluginu Hello

3.2. Plugin GA

3.2.1. Funkce pluginu GA

Jak jsem se již zmínil, cílem této práce je vnést do projektu NNSU paralelizaci. A co by šlo paralelizovat lépe, než genetický algoritmus? Rozhodli jsme se provést paralelizaci tak, že vytvoříme vlastní plugin s paralelní verzí genetického algoritmu. Než jsme k tomu mohli přistoupit, museli jsme poznat, jak fungují genetické algoritmy v současnosti.

Nyní zprostředkovává běh genetických algoritmů v aplikaci plugin GA, jehož autorem a stálým vývojářem je pan Ing. Roman Kalous. Ve spolupráci s ním a díky jeho diplomové a rozpracované disertační práci se nám podařilo porozumět fungování jeho pluginu. O hrubý nástin, nutný k pochopení další části práce, se pokusíme v této kapitole. Rádi bychom objasnili způsob kódování neuronových sítí pro genetický algoritmus, metodu jejich ohodnocování a přidělování fitness a systém, jakým jsou realizovány operace genetického algoritmu (selekce, křížení, mutace).

3.2.2. Používané typy neuronových sítí s přepínacími jednotkami

Jak už jsme se několikrát zmínili, v projektu NNSU se používají neuronové sítě s přepínacími jednotkami. Do genetických algoritmů vstupují dva základní typy.

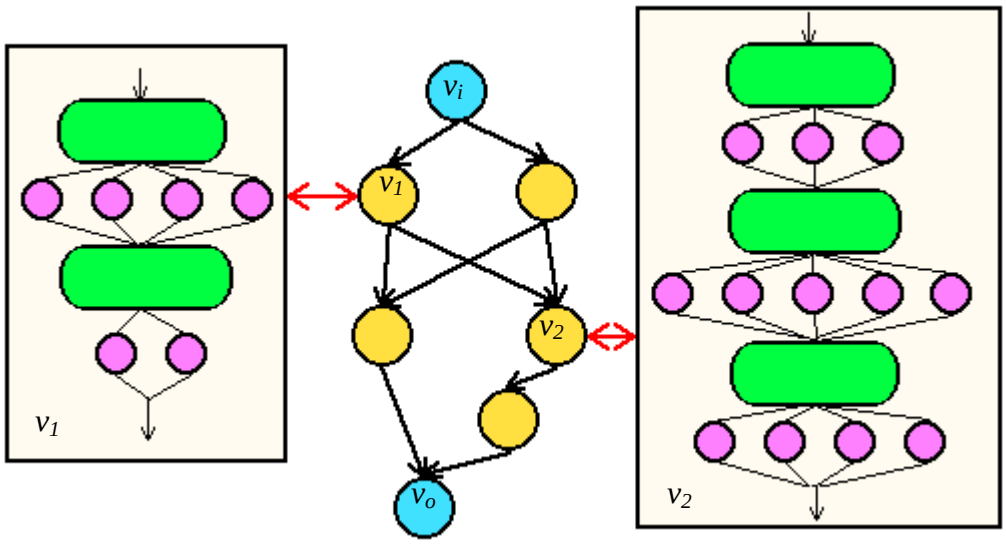
První z nich jsou „klasické“ NNSU popsané na konci kapitoly o neuronových sítích. Jsou charakteristické tím, že v jejich stavbě se pravidelně střídají přepínací jednotka s množinou výpočetních jednotek, které jsou jejími potomky a zároveň předky následující přepínací jednotky. Díky jejich řetězcovitému tvaru jim budeme dále říkat řetězcové NNSU nebo CHAIN NNSU a značit je CH_NNSU. Skupinu tvořenou přepínací jednotkou a jejími potomky budeme nazývat blok.

Druhá skupina neuronových sítí s přepínacími jednotkami vzniká zobecněním první kategorie. Označujeme ji jako CCH_NNSU. Kromě výpočetních a přepínacích jednotek zde budeme ještě potřebovat tzv. separátory datového toku. Ty nijak neupravují hodnoty vstupních signálů, pouze mění jejich dimenzi, tedy vybírají složky vstupního signálu, které se mají poslat dál. Pokud je vstupní dimenze např. $q = 5$ a separátor má výstupní dimenzi $r = 3$, může separátor dále poslat třeba signály x_1, x_3 a x_5. Jsou to tedy vlastně jakési „zužovače“.

Když víme, co jsou separátory datového toku, můžeme pomocí formální definice zavést třídu neuronových sítí CCH_NNSU:

Mějme orientovaný souvislý acyklický graf G = (V,E), pro jehož vrcholy platí, že $\exists v_i \in V : ind(v_i) = 0; \exists v_o \in V \setminus \{v_i\} : outd(v_o) = 0; \forall v \in V \setminus \{v_i, v_j\} : ind(v) \geq 1 \wedge outd(v) \geq 1,$ *kde symbolem ind(v) označujeme počet hran vstupujících do vrcholu v a symbolem outd(v) počet hran z něj vystupujících. Přiřaďme bijektivně každému* $v \in V \setminus \{v_i, v_o\}$ *libovolný řetězec tvořený bloky na jehož začátek vložíme takový separátor datového toku, aby vstupní dimenze řetězce odpovídala dimenzi dané řetězci přiřazenými předcházejícím vrcholům grafu a strukturou grafu. Takto definovanou strukturu nazýváme třídou neuronových sítí CCH_NNSU.*

Příklad toho, co definice popisuje, můžeme vidět na obrázku. Uprostřed je struktura celé sítě, po stranách můžeme vidět řetězce odpovídající vrcholům v_1 a v_2. Červené šipky zachycují bijekci, zeleně jsou označeny přepínací jednotky, fialově výpočetní. Podobné řetězce jsou přiřazeny i ostatním žlutým uzlům grafu. Separátory datového toku nejsou na obrázku zakresleny, berme je jako technickou redukci dimenzí nutnou, aby všechno fungovalo.



Obrázek 13: Příklad neuronové sítě z třídy CCH_NNSU

3.2.3. Kódování CCH_NNSU

Nyní víme, jak „fyzicky“ vypadají jedinci, potřební do genetického algoritmu. Jenže, algoritmus nepracuje s jedinci, ale s jejich chromozomy. Jak tedy předělat neuronovou síť z předcházející podkapitoly na řetězec znaků, vhodný ke genetickému algoritmu?

Začneme kódováním řetězcových NNSU. Jejich parametry jsou dvojího druhu – vnitřní parametry jednotlivých jednotek a parametry řetězce jako celku (délka, počty výpočetních jednotek v jednotlivých vrstvách). Jelikož vnitřní parametry jednotek se nastavují pomocí učících algoritmů, pracují genetické algoritmy pouze s parametry řetězce. Ty zakódujeme poměrně snadno. Pro řetězcovou NNSU s n přepínacími jednotkami takovou, že za i-tou přepínací jednotkou následuje vždy n_i výpočetních jednotek, budeme klást její reprezentaci $\alpha = n_1 n_2 \ldots n_n$. Vznikají nám tedy chromozomy proměnlivé délky s jednotlivými

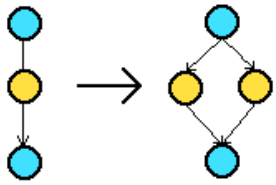
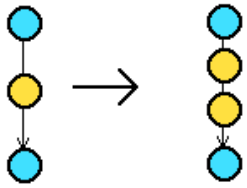
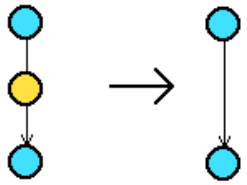
znaky z množiny $\{1,2,\ldots \max(n_i \mid i \in \hat{n})\}$. Například řetězec odpovídající vrcholu v_2 na obrázku na předchozí straně by se zakódoval jako 354. Jedná se o určité rozšíření genetického algoritmu popsaného v první kapitole, ale poté, co si popíšeme realizaci jednotlivých operací GA, uvidíme, že to není problém.

Kódování neuronových sítí typu CCH_NNSU je díky jejich nelineární struktuře poněkud obtížnější. V současné době je vytvořené tak, že podle generovaného kódu se vytváří instrukční strom a podle něj se staví neuronová síť. V opačném pořadí tato procedura nefunguje. Jak jednotlivé části tohoto postupu si probíhají, si nyní popíšeme.

Začneme popisem instrukčních stromů. Instrukční strom je strom[3], pro který platí:

- strom obsahuje pevně zvolený vrchol, zvaný kořen, jehož stupeň je roven dvěma, pokud počet vrcholů stromu je aspoň dva
- pokud leží dva vrcholy spojené hranou na cestě z kořene do listu, pak vrchol bližší ke kořeni nazýváme otcem vrcholu vzdálenějšího, který nazýváme synem vrcholu bližšího
- každý otec má žádného (je-li listem[4]) nebo dva syny (jinak)
- každému vrcholu, který není listem, je přiřazen prvek z množiny instrukcí $\{'S','P','D'\}$, pokud vrchol listem je, přiřazujeme mu instrukci $'I'$

Než si ukážeme, že pomocí takovéhoto stromu můžeme vytvořit požadovanou neuronovou síť, musíme si ještě osvětlit významy jednotlivých instrukcí. Shrnuje je následující tabulka 3. V příkladech si za modrými uzly můžeme představit libovolný počet uzlů spojených hranami se všemi žlutými v následující vrstvě.

Instrukce	Význam	Příklad
S	Vloží uzel vedle existujícího	
P	Vloží uzel za existující	
D	Smaže uzel	
I	Ukončí vývoj uzlu a přiřadí mu řetězcovou NNSU	

Tabulka 3: Instrukce pro tvorbu NNSU

Jak tedy tvorba neuronové sítě probíhá? Jednoduše se od kořene prochází instrukční strom a podle instrukcí přiřazených jeho vrcholům vzniká neuronová síť. Každý list instrukčního stromu pak odpovídá jednomu vrcholu grafu. Příklad si ukážeme později.

[3] souvislý graf, který neobsahuje kružnici
[4] vrcholem stupně jedna

Nyní si vysvětlíme, jaký kód popisuje instrukční strom. Každý instrukční strom s n vrcholy je popsán stejným počtem kódových slov tvaru $(a, INST, b)$, kde

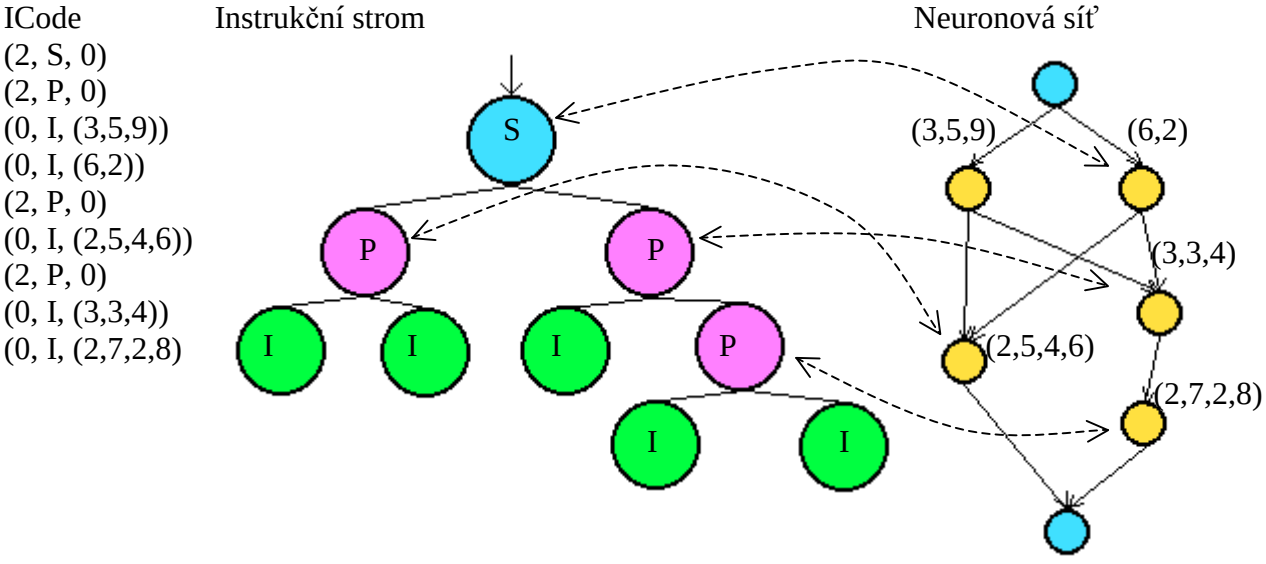
$$a \in \{0, 2\},\ INST \in \{'S','P','D','I'\},\ b = \begin{cases} 0 & INST \neq I \\ (n_1, \ldots n_m) & INST = I \end{cases}$$

Každé kódové slovo popisuje jeden vrchol instrukčního stromu, v pořadí od kořene dále vždy do hloubky přes nejlevější podgraf. Tedy popisujeme tak dlouho levé syny, dokud to jde a když už není co popisovat, vrátíme se v podgrafu o úroveň blíže ke kořeni, popíšeme pravého syna a zase následuje popis levého podgrafu. Parametr a v kódovém slově udává, kolik má příslušný vrchol synů, následuje instrukce svázaná s daným vrcholem. Proměnná b říká u všech listů, jak má vypadat řetězová NNSU přiřazená v konečné neuronové síti danému vrcholu.

Tento kód jako celek budeme nazývat ICode. Pořadí hodnot parametru a v ICodu není zcela libovolné. Aby kód dával dobrý smysl, musí hodnoty a jednotlivých slov tvořit tzv. grafovou posloupnost, neboli musí splňovat (nechť je slov celkem q):

$$\sum_{i=1}^{m} a_i \geq m,\ m = 1,2,\ldots,q-1 \quad \sum_{i=1}^{q} a_i = q-1$$

Posloupnost hodnot a nazýváme Readovým kódem a setkáme se s ní v podkapitole o reprezentaci jednotlivých operací genetického algoritmu. Na závěr této podkapitoly si ještě ukážeme na konkrétní příkladu, jak vypadá kód, instrukční strom a neuronová síť. Čárkované šipky ukazují, který vrchol instrukčního stromu může za vznik každého uzlu sítě. Uzel s parametry (3,5,9) je iniciální a přítomen vždy.



Obrázek 14: Přiřazení kódu, instrukčního stromu a neuronové sítě

3.2.4. Ohodnocování neuronových sítí

Nyní již víme, s jakými neuronovými sítěmi algoritmus pracuje a jak je pro své použití kóduje. V této podkapitole trochu osvětlíme, jak se pozná, že daná neuronová síť má dobré,

respektive špatné, výsledky. Jinými slovy naznačíme, podle jakého klíče probíhá přiřazení hodnoty fitness neuronové síti.

Tento proces má dvě části. První z nich je učení neuronové sítě, neboli nastavení vnitřních parametrů pomocí učících dat. Pro stanovení hodnoty fitness není mechanizmus této části důležitý, neboť na volbu hodnoty fitness nemá bezprostřední vliv. Můžeme si dovolit brát ho jako černou skříňku – do algoritmu vstoupí nenaučená neuronová síť a učící data a vystoupí neuronová síť s plně nastavenými vnitřními parametry.

Mnohem důležitější je druhá část. Naučené neuronové síti předložíme množinu testovacích dat a ta se pokusí separovat je do dvou stejně velkých disjunktních podmnožin. Jedna podmnožina se nazývá pozadí a označíme ji D_{BG}^T, druhá signál s označením D_{SG}^T. Podle míry úspěšnosti této separace pak přiřadíme neuronové síti hodnotu fitness.

Aby to nebylo tak jednoduché, výstupem neuronové sítě pro daný vstup není číslo podmnožiny, ale reálné hodnoty z intervalu $\langle -1{,}5; 1{,}5\rangle$ udávající, jak moc vstup patří to té které podmnožiny. Výstupy pro kompletní množinu testovacích dat se následně vynášejí do tzv. grafu výsledků sítě NNSU. Tvorba a analýza grafu je poměrně složitá a blíže se jí zaobírá ve svých pracích kolega Kalous. Na základě tohoto grafu také odvozuje následující optimální ohodnocení dané neuronové sítě:

Nechť je dána populace naučených neuronových sítí P a testovací data $D^T = D_{SG}^T \cup D_{BG}^T$, $D_{SG}^T \cap D_{BG}^T = \theta$. *Pro každou neuronovou síť označme funkce reprezentující hustotu rozdělení výsledků testování vzorků z* D_{SG}^T *a* D_{BG}^T *jako* $SG : \langle -1{,}5; 1{,}5\rangle \to R^+$ *a* $BG : \langle -1{,}5; 1{,}5\rangle \to R^+$. *Buď dále I libovolný interval* $I \subset \langle -1{,}5; 1{,}5\rangle$. *Pak definujeme proměnné SG2BG(I) a BG2SG(I), vyjadřující poměry počtů vzorků typu SG a BG, které mají ohodnocení v intervalu I, následujícím vztahem:*

$$SG2BG(I) = \frac{\int_I SG(x)dx}{\int_I BG(x)dx} \quad a \quad BG2SG(I) = \frac{1}{SG2BG(I)}$$

Dále definujme:

$$BGSG(I) = \int_I (BG(x) + SG(x))dx$$

Zvolme nyní intervaly $BW, SW \subset \langle -1{,}5; 1{,}5\rangle$ *jako okolí očekávaných výstupů pro data typu BG a SG. Nazýváme je okno pozadí a okno signálu.*

Pro danou neuronovou síť $A \in P$ *definujeme evaluační funkci* $e : P \to R$ *a hodnotu její fitness* $f : P \to \langle 0; 1\rangle$ *jako:*

$$e(A, D^T) = BG2SG(BW).BGSG(BW).SG2BG(SW).BGSG(SW)$$

$$f(A, D^T) = \frac{e(A, D^T)}{\max\{e(A, D^T) \mid A \in P\}}$$

Tyto hodnoty fitness jsou přiřazovány všem neuronovým sítím z dané populace a podle nich se vybírají jedinci do vlastního genetického algoritmu, jak si popíšeme v následující podkapitole.

3.2.5. Operace genetického algoritmu

Jak už dobře víme, genetický algoritmus používá tři základní operace – selekci, křížení a mutaci. V této podkapitole si se znalostmi z předchozích podkapitol popíšeme, jak jsou tyto operace konkrétně realizovány v pluginu GA.

Nejsložitější je operace selekce, tedy vybrání správných jedinců pro křížení. Klasická nevyvážená ruleta popsaná v kapitole o genetických algoritmech zde nedává příliš kvalitní výsledky. Jedinců je takové množství s tak podobnými fitness, že délky oblouků pomyslné kružnice jsou příliš podobné a dochází k neopodstatněnému preferování nepříliš kvalitních jedinců. Proto se zde používá tzv. úrovňový výběr. Při něm se pravděpodobnosti výběru počítají pomocí vztahu:

$$s_j = \frac{\rho_j}{\sum_{j=1}^{R} \rho_j} \quad j \in \widehat{R}$$

kde R značí počet jedinců a předpokládáme, že fitness jsou seřazeny od nejmenší pro $j=1$ po největší pro $j=R$. Rostoucí funkce ρ_j může být definována různě, používají se především:

$$\rho_j^{\log} = \log(1+\log(1+\frac{j}{R}))$$

$$\rho_j^{lin} = \frac{j}{R}$$

$$\rho_j^{\exp} = e^{\frac{j}{R}} - 1$$

Takto zvolené pravděpodobnosti výběru jsou daleko odlišnější a proto vhodnější. Nad nimi již nevyváženou ruletu můžeme roztočit.

V předchozí operaci jsme museli znát pouze hodnoty fitness. Pro křížení a mutaci musíme rozumět kódování neuronové sítě pro algoritmus. Křížení a mutace probíhají na prvních členech Icodu, tedy na Readově kódu (viz 3.2.3.). Operátor mutace realizujeme tak, že v této posloupnosti nalezneme část odpovídající podstromu instrukčního stromu neobsahujícímu kořen a tu vyměníme za novou, která se generuje náhodně. V případě křížení si jedinci vymění část této posloupnosti, reprezentující určité náhodné podstromy.

Jak poznáme, že daný kousek Readova kódu reprezentuje podstrom? Náhodně zvolíme k-tý člen posloupnosti Readova kódu a postupujeme po dalších členech dokud nesplníme podmínku, že daná vybraná část je grafová posloupnost, tedy že platí:

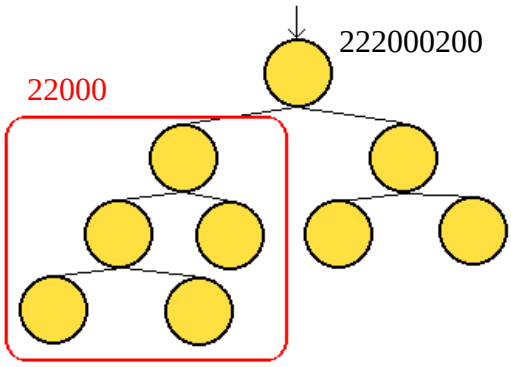
$$\sum_{i=1}^{m} a_i \geq m,\ m = 1,2,\ldots,q-1 \quad \sum_{i=1}^{q} a_i = q-1$$

Jako příklad mějme kód 2**2**2000200 a náhodně zvolme dvojku na druhé pozici. Postup ověřování shrnuje následující tabulka:

q	m	Dostupné členy posloupnosti a_i	$\sum_{i=1}^{m} a_i$	$\sum_{i=1}^{q} a_i$	$\sum_{i=1}^{m} a_i \geq m$?	$\sum_{i=1}^{q} a_i = q-1$?
1	0	$a_1=2$	1	2	ano	ne
2	1	$a_1=2$, $a_2=2$	2	4	ano	ne
3	1, 2	$a_1=2$, $a_2=2$, $a_3=0$	2, 4	4	ano	ne
4	1, 2, 3	$a_1=2$, $a_2=2$, $a_3=0$, $a_4=0$	2, 4, 4	4	ano	ne
5	1, 2, 3, 4	$a_1=2$, $a_2=2$, $a_3=0$, $a_4=0$, $a_5=0$	2, 4, 4, 4	4	ano	ano

Tabulka 4: Nalezení podstromu v Readově kódu

Z tabulky můžeme vidět, že kód 22000 v kódu 2**22000**200 definuje podstrom. Prakticky nám to ukáže následující obrázek, na kterém můžeme vidět tento instrukční strom i vybraný podstrom (v červeném rámečku).



Obrázek 15: K výběru podstromů

Nyní již známe všechny náležitosti fungování genetického algoritmu v pluginu GA a můžeme postoupit k jeho paralelizaci.

3.3. Paralelizace v projektu NNSU

3.3.1. Komunikační rozhraní MPI

Jak už jsem se několikrát zmínil, jedním z cílů této práce je provést paralelizaci části kódu v projektu NNSU. Pod pojmem paralelizace kódu rozumíme rozdělení instrukcí na jednodušší úlohy, které mohou zpracovávat samostatné procesy souběžně. Cílem paralelizace je urychlení programu.

Jako nástroj pro tuto činnost jsme zvolili komunikační rozhraní MPI (*Message Passing Interface*). MPI je nejrozšířenějším rozhraním používaným pro tvorbu paralelních programů s distribuovanou pamětí. Obsahuje řadu funkcí, pomocí kterých mezi sebou mohou jednotlivé procesy komunikovat prostřednictvím posílaných zpráv. V této podkapitole si ho trochu představíme.

Myšlenka vzniku MPI jako univerzálního komunikačního rozhraní paralelního programování vznikla v roce 1992 na konferenci *Supercomputing 92*. Do té doby existovalo mnoho různých podobných rozhraní, které však byly vyvinuty pouze pro konkrétní účel. Každý, kdo chtěl něco paralelizovat, si vyráběl vlastní a tak vznikalo stále dokola totéž. První standard, zvaný MPI-1, byl představen v roce 1994. Aktualizovaná a rozšířená verze, MPI-2, byla uvedena v roce 1998, ale její první implementace přišla až v listopadu 2002. Vedlejším

důsledkem tvorby standardu MPI-2 bylo upřesnění původního MPI-1, čímž vzniklo MPI-1.2. Vývoj MPI se poté zastavil. Ke znovuoživení došlo až koncem roku 2006 a jeho důsledkem je opravený standard MPI-2.1 z 9.února 2007.

V současnosti vedle sebe existují dvě podporované verze MPI, a to MPI-1.2 a MPI-2.1. MPI-2 přináší oproti svému předchůdci různá vylepšení, navíc především dynamickou správu procesů (možnost startovat nové procesy za běhu), jednostrannou komunikaci procesů pomocí sdílené paměti, práci se vstupními a výstupními soubory a mnoho dalšího. MPI-1.2 popisuje zhruba sto třicet funkcí, MPI-2.1 více než pět set. Důležité je, že funkce MPI-1.2 jsou v novějším standardu zachovány (nebo vhodně předělány), a proto programy využívající pouze tyto funkce fungují i v implementacích MPI-2.1. To je možná jedním z důvodů, proč se novinky ze standardu MPI-2.1 příliš nerozšiřují a mnoho programátorů stále zůstává u MPI-1.2. Navíc k vytvoření fungujícího paralelního programu není třeba znát desítky funkcí, stačí jen zhruba pětadvacet základních. Proč se tedy namáhat s dalšími 475?

Standard MPI je nezávislý na konkrétním programovacím jazyce (existují implementace pro Fortran, C, C++, Python, Javu a další, dokonce i MATLAB). Nejrozšířenější implementace jsou však připraveny pro tři programovací jazyky – C, C++ a Fortran. V současné době patří mezi nejznámější MPICH (MPI-1.2) a MPICH 2 (MPI-2.1) od Argonne National Laboratory a OpenMPI (MPI-2) od rozsáhlejší skupiny společností. OpenMPI přímo navazuje na dřívější projekt LAM/MPI. Pro svou práci jsem se rozhodl používat OpenMPI.

3.3.2. Funkce MPI

Nyní si představíme několik nejdůležitějších funkcí MPI. Funkce budeme popisovat ve formě pro použití v programovacím jazyce C/C++.

Ve funkcích se často objevuje argument typu MPI_Comm. Je to identifikátor skupiny procesů, v rámci které probíhá vzájemná komunikace, tzv. komunikátoru. Jeden proces může patřit i do více komunikátorů. Pokud má komunikace probíhat mezi všemi procesy spuštěnými pomocí MPI, přiřazujeme mu hodnotu MPI_COMM_WORLD.

Dalším typem proměnných je MPI_Datatype. Jelikož MPI jako takové nezávisí na programovacím jazyku, musí mít vlastní specifikaci toho, co si mezi sebou procesy vlastně posílají. V následující tabulce vidíme vzájemné přiřazení MPI_Datatype a datových typů jazyka C.

MPI_Datatype	*Datový typ v C*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	viz dál
MPI_PACKED	viz dál

Tabulka 5: Srovnání MPI_Datatype a datových typů jazyka C

Kromě těchto typů můžeme samozřejmě posílat i pole, stačí zadat adresu jejich prvního prvku, typ prvků z tabulky a počet posílaných prvků. Mimo běžných céčkovských typů může používat MPI ještě dva další typy. MPI_BYTE znamená, že se posílá jeden bajt neboli osm bitů dat. To může být nepraktické, jelikož vždy nevíme, kolika bity je zrovna daný typ kódován. Na druhou stranu nám to umožní posílat i neznámá data. MPI_PACKED značí skupinu dat, která vznikla z nesouvislých dat zabalením.

Možná jste si všimli, že pomocí takto definovaných MPI_Datatype nejde zdaleka poslat všechno. Třeba instance nějakého objektu by se posílala dost těžko. Naštěstí MPI dovoluje odvozovat další datové typy s použitím řady svých příkazů. Zatím však vystačíme se základními typy.

A teď už přichází slibovaný přehled nejdůležitějších funkcí MPI:

- int MPI_Init(int *argc, char ***argv)
  - účel: inicializace MPI, používá se právě jednou, nutné volat před všemi[5] ostatními funkcemi MPI
  - argc – počet argumentů programu při jeho spouštění z příkazové řádky
  - argv – pole s texty těchto argumentů
  - podle standardu MPI-2 lze místo obou argumentů psát nulový ukazatel NULL
- int MPI_Finalize()
  - účel: ukončení MPI
- int MPI_Comm_Size(MPI_Comm comm, int *size)
  - účel: do proměnné *size* uloží počet procesů v daném komunikátoru
  - comm – jméno komunikátoru, pro který funkci volám, hodnota MPI_COMM_WORLD znamená, že pro všechny procesy
  - size – počet procesů v daném komunikátoru
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - účel: do proměnné *rank* uloží číslo procesu, který funkci volá
  - comm – jméno komunikátoru, pro který danou funkci volám, hodnota MPI_COMM_WORLD znamená, že pro všechny procesy
  - rank – číslo procesu v daném komunikátoru
- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - účel: volající proces odešle data procesu *dest*
  - buf – kde jsou data uložená
  - count – kolik dat typu datatype se odesílá
  - datatype – typ dat v buf
  - dest – číslo cílového procesu
  - tag – číslo zprávy, pro odlišení
  - comm – jméno komunikátoru, kde se procesy nacházejí
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - účel: pro příjem dat od procesu *source*
  - buf – kam se data uloží
  - count – počet dat typu datatype, který se má přijmout
  - datatype – typ dat v buf
  - source – od kterého procesu se má přijímat, lze MPI_ANY_SOURCE – přijmou se data daných parametrů od libovolného procesu
  - tag – číslo, které má mít přijatá zpráva, pro libovolné lze MPI_ANY_TAG

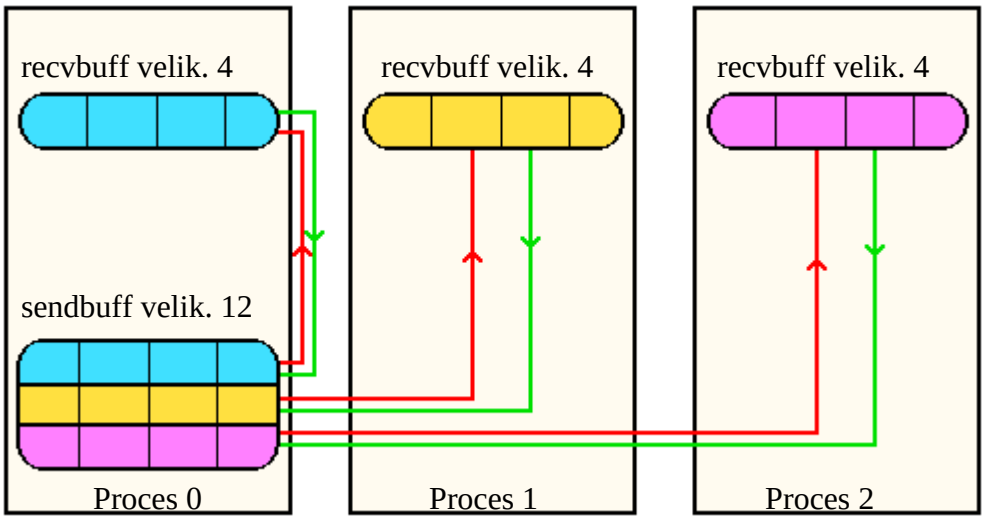
[5] Existují asi tři funkce, pro které to neplatí, viz www.mpi-forum.org

    - comm – jméno komunikátoru, kde se procesy nacházejí
    - status – struktura, obsahující zprávy o průběhu operace a případná chybová hlášení
- int MPI_Barrier(MPI_Comm comm)
    - účel: zastaví běh procesů v daném komunikátoru, dokud funkci nezavolají všechny (pro synchronizaci)
    - comm – jméno komunikátoru, kterého se to týká
- int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)
    - účel: proces *source* odešle všem ostatním procesům v komunikátoru stejná data z jeho *buf* do jejich *buf*
    - buf – odkud data pocházejí, resp. kam se ukládají
    - count – počet dat typu datatype
    - datatype – typ přenášených dat
    - source – číslo vysílajícího procesu
    - comm – jméno komunikátoru, kterého se to týká
- int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm)
    - účel: všechny procesy z daného komunikátoru včetně procesu *target* pošlou procesu *target* zprávu, ty se uloží do *recvbuf* v pořadí podle čísel procesů za sebou
    - sendbuf – odkud se data posílají
    - sendcount – počet odesílaných dat (od všech procesů stejný)
    - senddatatype – typ odesílaných dat (od všech procesů stejný)
    - recvbuf – kam se data na přijímajícím procesu uloží, pro vysílající procesy může být NULL
    - recvcount – počet přijímaných dat od každého procesu (stejný, jako počet odesílaných)
    - recvdatatype – typ přijímaných dat (stejný, jako typ odesílaných)
    - target – číslo přijímajícího procesu
    - comm – jméno komunikátoru, kterého se to týká
- int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)
    - účel: jako MPI_Gather, jenže opačným směrem, proces *source* posílá každému procesu z jeho komunikátoru jinou zprávu, které jsou na začátku uložené za sebou v *sendbuf*
    - parametry – jako u předchozího s tím, že NULL může mít sendbuf a ne recvbuf

To je tedy přehled několika základních funkcí MPI a přehled takových funkcí, které jsou pro budoucí paralelizaci GA potřebné. Pro lepší pochopení funkcí MPI_Gather a MPI_Scatter slouží následující obrázek. Červené čáry znázorňují vysílání při MPI_Scatter a zelené příjem zpráv při MPI_Gather. Odesílací (přijímací) buffer na nultém procesu má velikost dvanáct a každému procesu se posílá (od každého se přijímá) čtyři jednotky. Pro vysílání by kód jednotlivých procesů mohl být třeba následující:

Proces 0: MPI_Scatter(&sendbuff, 4, MPI_INT, &recvbuff, 4, MPI_INT, 0, MPI_COMM_WORLD)

Procesy 1 a 2: MPI_Scatter(NULL, 4, MPI_INT, &recvbuff, 4, MPI_INT, 0, MPI_COMM_WORLD)



Obrázek 16: MPI_Scatter a MPI_Gather

3.3.3. Návrh paralelizace pluginu GA

V tuto chvíli známe všechny potřebné prostředky i cíle k tomu, abychom mohli navrhnout způsob paralelizace pluginu GA.

Časově jednoznačně nejnáročnější je naučení jednotlivých sítí a jejich následné ohodnocení hodnotou fitness. Proto jsme se rozhodli rozdělit tuto práci mezi více procesů. Ostatní části algoritmu by bez velké časové ztráty měl zvládat jeden proces.

Předpokládejme nyní, že máme k dispozici $npes \in N$ procesů. Nultý proces budeme dále nazývat master, ostatní slave. Master bude práci vytvářet a přidělovat, úlohou slavů bude naučit a ohodnotit dodané neuronové sítě a vrátit masterovi jejich fitness.

Popišme si nyní tento úkol podrobněji.

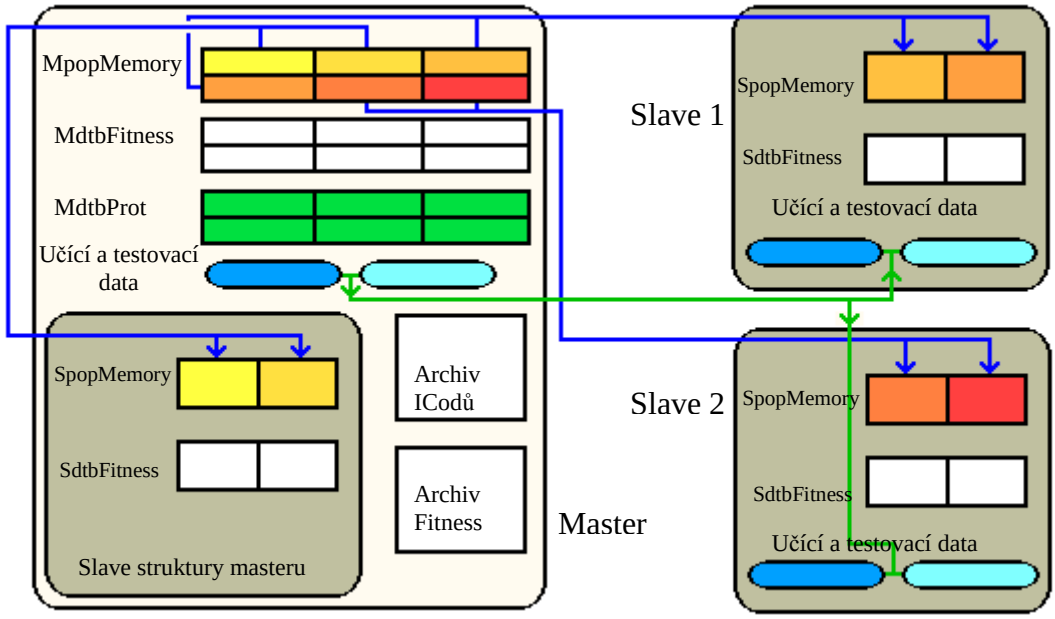
Na začátku získají všechny procesy základní parametry, jako je počet generací, velikost populace apod., z konfiguračního XML souboru. Master následně načte a rozešle slavům učící a testovací data, nutná k naučení a ohodnocení sítě. Využívá při tom funkci MPI_Bcast. Poté se na masteru vytvoří nultá generace ICodů. Ta bude po dobu běhu nulté iterace uložena v proměnné MpopMemory. Z ní posílá master každému slavu stejnou poměrnou část ICodů tak, že se mezi slavy rozdistribuují všichni jedinci. Toto vysílání probíhá pomocí funkce MPI_Scatter. Jelikož během učení a ohodnocování by proces master čekal, rozhodli jsme se, že bude také dostávat svou porci ICodů a kromě práce mastera bude plnit i úkoly slavu. Master si následně ještě poznamená, na kterém slavu a kde se nalézá daný jedinec.

Slavy přijímají ICody do své SpopMemory. Z ní se jednotlivé sítě vybírají, učí a ohodnocují fitness. Ta se ukládá do SdtbFitness, odkud se následně najednou posílají fitness všech sítí mastru. Slouží k tomu funkce MPI_Gather.

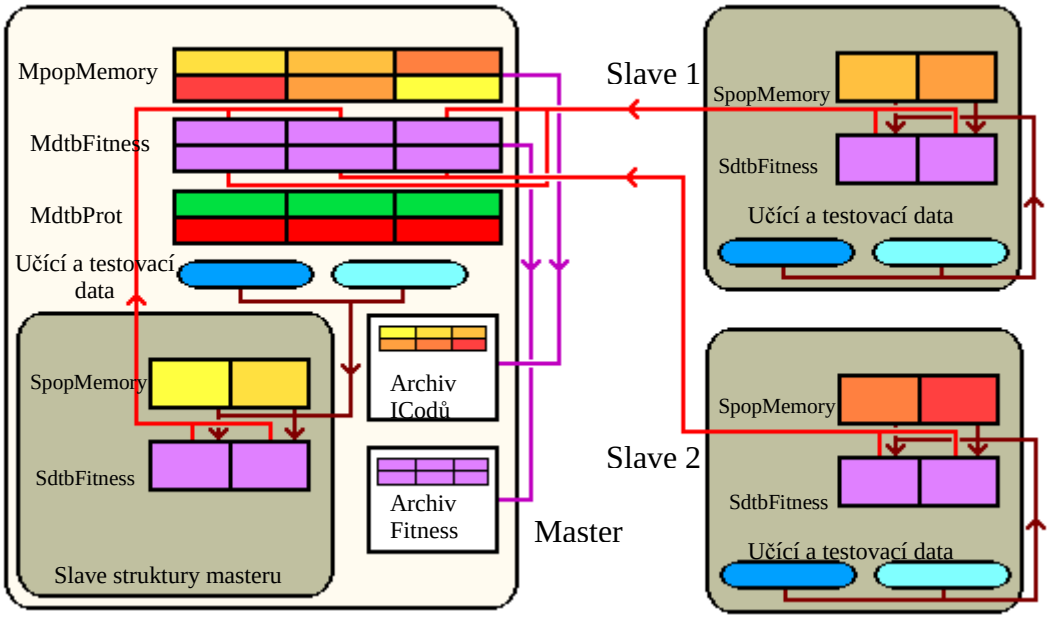
Master přijme všechny fitness do proměnné MdtbFitness. Hned poté je uloží společně s ICody z MpopMemory do archivních matic (každý řádek odpovídá jedné generaci), které mohou sloužit pro další analýzu dat. Následuje srovnání všech databází podle hodnoty fitness tak, aby byly sítě s nejvyšší fitness na začátku. Prvních *k* sítí dostane status „protected“, což znamená, že při tvorbě nové generace se na jejich místě v MpopMemory nebudou tvořit nové ICody a navíc se tyto sítě nebudou ani posílat slavům k ohodnocení. Aby toho bylo docíleno, přesunou se jejich fitness a ICode na konce příslušných databází.

Nyní přichází evoluční cyklus. Začíná tvorbou nové generace. Pomocí MpopMemory, která stále obsahuje ICody předchozí generace, a díky mechanizmům genetických algoritmů se vytvoří a do MpopMemory uloží nové ICody. Je jich právě tolik, aby společně s „protected“ sítěmi dávaly stejný počet jedinců, jako měla nultá generace. Poté, co jsou vytvořeny, se nové ICody posílají slavům k ohodnocení a následuje už stejný postup jako pro nultou generaci, s tím rozdílem, že máme o něco méně jedinců.

Celou navrženou paralelizaci můžeme schématicky vidět na následujících obrázcích. Na prvním z nich je zachycena situace zhruba uprostřed průběhu nulté iterace. Zelené čáry zobrazují posílání učících a testovacích dat z masteru jednotlivým slavům (sám sobě je master neposílá). Modrými čarami je znázorněna distribuce jednotlivých Icodů (zobrazeny v odstínech oranžové).



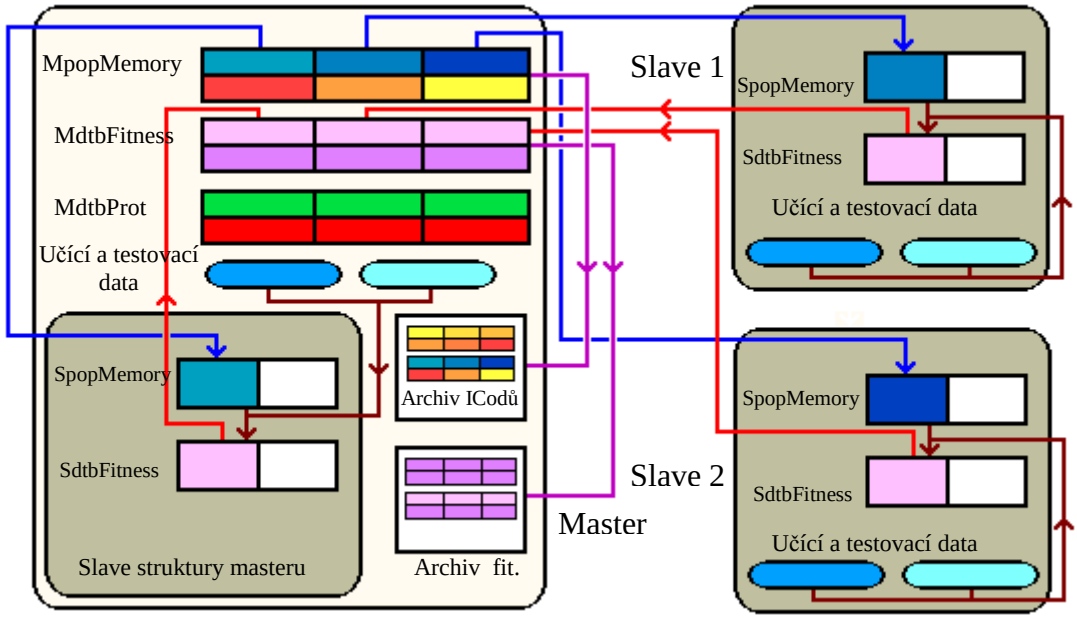
Obrázek 17: První část nulté iterace průběhu paralelního algoritmu



Obrázek 18: Druhá část nulté iterace průběhu paralelního algoritmu

Na druhém obrázku je znázorněna druhá část nulté iterace. Slavy naučily a ohodnotily na základě učících a testovacích dat dodané jedince (hnědé čáry a fialová barva SdtbFitness) a poslaly hodnoty fitness masteru (červené čáry). Ten uložil do archivu ICody a fitness nulté generace, což zachycují fialové čáry. Následně se všechny databáze masteru zpermutují tak, aby nejprve byly sítě s nejvyšším fitness na začátku, kde se prvních *k* označí jako „protected“ a tyto se následně přesunou na konec databáze (červené prvky v MdtbProtected a přeházení MpopMemory).

Začátek další iterace můžeme vidět na posledním obrázku. Na místech, která nemají status „protected“ vznikají nové ICody, které se opět distribuují jednotlivým slavům a celý cyklus začíná nanovo.



Obrázek 19: První iterace průběhu paralelního algoritmu

Závěr

Tato práce přináší základní přehled problematiky genetických algoritmů a neuronových sítí, nutný k dalšímu pochopení projektu NNSU. Dále představuje samotný projekt NNSU včetně několika praktických rad ohledně zprovoznění nových pluginů. Podrobněji se věnuje pluginu GA. Nalézá se zde i úvod do paralelizace pomocí komunikačního rozhraní MPI.

Vyvrcholením práce je návrh paralelizace genetických algoritmů, vytvořený autorem. V textu se nachází jeho teoretický popis, prakticky je tento návrh implementován jakožto nový plugin nazvaný GAPAR na přiloženém CD. Naprogramována je funkční simulace problému paralelizace, která věrně zachycuje komunikační průběh operací, a zjednodušeně simuluje vlastní činnosti genetických algoritmů. Stejně jako u jiných pluginů, lze i u tohoto zadávat parametry pomocí konfiguračního XML souboru.

V budoucnu bych rád ve spolupráci s ostatními členy vývojového týmu projektu NNSU docílil toho, aby můj plugin dokázal komunikovat s ostatními pluginy a především využívat metody, které jsou v nich naprogramovány. Projekt NNSU by poté mohl fungovat rovněž paralelně, aniž by došlo k duplicitě kódu.

Pevně doufám, že navržená paralelizace je pro projekt NNSU přínosem, který povede k dalšímu rozvoji tohoto programu.

Seznam použité literatury

Genetické algoritmy:

1/ D. E. Goldberg:
Genetic Algorithms in Search, Optimization, and Machine Learning, Addison Wesley, 1989

2/ J. H. Holland:
Adaptation in Natural and Artificial Systems, MIT, 1992

3/ V. Kvasnička, J. Pospíchal, P. Tiňo:
Evolučné algoritmy, STU, Bratislava, 2000

Neuronové sítě:

4/ P. Bitzan, J. Šmejkalová, M. Kučera:
Neural Network with Switching Units, Neural Network World, 4:515-526, 1994

5/ F. Hakl, M. Holeňa:
Úvod do teorie neuronových sítí, ČVUT, Praha, 1997

6/ M. Hlaváček:
Návrh a analýza neuronové sítě s přepínacími jednotkami vhodné pro studium procesů rozpadu elementárních částic s využitím možnosti genetické optimalizace, diplomová práce, Katedra matematiky, ČVUT-FJFI, 2002

7/ J. Tučková:
Úvod do teorie a aplikací umělých neuronových sítí, ČVUT, Praha, 2005

8/ S. Wright:
Klinická fyziologie, Avicenum – zdravotnické nakladatelství, Praha, 1970

9/ www.wikipedia.org

Projekt NNSU:

10/ A. Grama, A. Gupta, G. Karypis, V. Kumar:
Introduction to Parallel Computing, Addison Wesley, 2003

11/ R. Kalous:
Návrh, implementace a paralelizace genetických algoritmů pro optimalizaci neuronových sítí s hierarchickou strukturou, diplomová práce, Katedra matematiky, ČVUT-FJFI, 2002

12/ R. Kalous:
Evolutionary Optimization of Neural Networks Architectures Using Genetic Algorithm, disertační práce, Katedra matematiky, ČVUT-FJFI, dosud nezveřejněno

13/ Z. Meglicky:
The History of MPI, beige.ucs.indiana.edu/I590/node54.html, 2004

14/ www.mpi-forum.org

15/ www.open-mpi.org