Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Teoretické Informatiky

Bakalářská práce

Návrh komprimace výstupů z digitální evoluce SVET

Vladislav Jásek

Vedoucí práce: RNDr. Jiřina Scholtzová, Ph.D.

12. května 2015

Poděkování

Děkuji RNDr. Jiřině Scholtzové, Ph.D. za vedení této práce, věnovaný čas a ochotu při konzultacích.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2015 .....

Odkaz na tuto práci

Jásek, Vladislav. *Návrh komprimace výstupů z digitální evoluce SVET*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem této bakalářské práce je navrhnout a naimplementovat optimální kompresi z výstupu systému digitální evoluce SVET. Navrhl jsem kompresi šitou na míru digitální evoluci SVET, výsledkem je program, který komprimuje původní textový výstup do vlastního formátu, ze kterého lze snadno získat informace o průběhu evoluce, jako je rodokmen, preparace jednotlivých organismů, či výpis určitého časového úseku evoluce.

Klíčová slova digitální evoluce, komprese

Abstract

The primary purpose of this thesis is to propose and implement optimal compression of outputs from digital evolution system SVET. The result of this thesis is a program, from which you can easily get information about the course of evolution, such as family tree, preparation of individual organisms or extract certain time period of evolution.

Keywords digital evolution, compression

Obsah

Seznam obrázků

Úvod

Evoluce (přesněji biologická evoluce) je teorie popisující dlouhodobý a samovolný proces, v jehož průběhu se rozvíjí a diverzifikuje pozemský život. Lidé se odpradávna zajímali o původ a vývoj nejen lidstva, ale celé živé přírody.

První teorie byly založeny na víře v nadpřirozeno, většinou propagované šiřiteli náboženství, jejichž společným znakem je považování životních forem za neměnné a přisuzování vzniku života zásahu vyšší moci.

Za první ucelenou vědeckou evoluční teorii byl považován lamarckismus, jejíž autor Jean Baptiste Lamarck publikoval v roce 1809 dílo Philosophie zoologique. Vychází z myšlenky, že rodiče vlastnosti získané za života předají svému potomstvu. Například potomci ptáka, který vylouskal nejvíc ořechů budou mít nejsilnější zobáky.

Tuto teorii odmítl jeho současník Georges de Cuvier, který změny fauny a flory připisoval hromadným zánikům při katastrofách a následnému vzniku nových životních forem, tzv. teorie katastrof.

V roce 1859 přichází Charles Robert Darwin se svojí, dnes obecně nejpřijímanější teorií, darwinismem. Oproti lamarckismu se liší především v tom, že na populaci se uplatňuje selekce, která vychází ze schopnosti jedince se adaptovat, a tím získat pro sebe co největší množství omezených, pro život důležitých, zdrojů. Pokud bychom opět použili paralelu s ptáky, pak ptáci, kteří mají nejsilnější zobák, vylouskají nejvíc ořechů, takže budou mít nejvíc síly k množení se.

Evoluce dnes známých životních forem je velmi složitý a zdlouhavý proces, který současná věda nedokáže pozorovat přímo. Překotný vývoj výpočetní techniky v posledních desetiletích však otevřel vědcům cestu, kterou se dá evoluce nasimulovat. Mluvíme o evoluci digitální. Digitální evoluce je počítačová simulace, určená ke studiu principů evoluce v rozumném čase. Vychází z toho, že na sekvenci DNA lze nahlížet stejně jako na skupinu instrukcí programu.

Počátky digitální evoluce se datují do roku 1990, kdy se Steen Rasmussen nechal inspirovat počítačovou hrou „Core War“ [11]. Tato hra probíhala jako soutěž krátkých prográmků, které se v běhovém prostředí navzájem přepiso-

valy a snažily se přežít. Nejúspěšnější byly programy, které se snažily množit a zachovat své kopie v případě svého poškození jiným programem. Evoluce v pravém slova smyslu však nemohla proběhnout, jelikož v systému nebylo umožněno křížení a mutace jednotlivých programů. Rasmussen do "Core War" přidal instrukci kopírování, která však čas od času místo pevně dané instrukce zapsala instrukci náhodnou, tudíž fungovala jako mutace. Projekt nebyl úspěšný, jelikož se systém velmi rychle dostal do neživého stavu z důvodu vyčerpání paměti.

Prvním úspěšným systémem digitální evoluce tak byl až systém Tierra [6], který vyvinul Thomas Ray v roce 1991. V něm již organismy musely požádat o přidělení paměti, a pokud byla zaplněna, tak byl systémem "zabit" některý ze starých organismů.

V roce 1992 započal vývoj evolučního systému Avida [12]. Organismy, které byly schopné vyřešit předem určený úkol, například úspěšně sečíst či vydělit dvě čísla, byly odměněny přídavným časem na procesoru, který mohly dále využít například k množení se, čímž na ně byl aplikován selekční tlak. Navíc oproti systému Tierra běžely organismy paralelně.

V roce 2009 byl vyvinut systém digitální evoluce SVET v rámci diplomové práce Lenky Vláčilové na VŠCHT [11], kde je také testován. Oproti předchozím systémům se odlišuje značně volnějším popisem organismu, který je určen pouze ukazatelem do hlavní paměti a stavem registrů. Celý systém je implementován v jazyce C. SVET je stále ještě ve vývoji a chybí mu některé součásti, které umožní příjemnou práci s aplikací. Jednou z nich je optimální ukládání výstupů z proběhlé evoluce.

Hlavním cílem této práce je navrhnout a realizovat optimální kompresi výstupu ze SVET. Velikost současných výstupů totiž dosahuje stovek megabytů dat i přesto, že evoluce běží jen pár minut. Navíc se z těchto výstupů některé informace nedají dohledat buď vůbec, nebo jen velmi složitě a neefektivně.

Je tedy potřeba vybrat parametry, které budou z hlediska evoluce podstatné a navrhnout, jak je optimálně ukládat.

Pro snadnou práci s komprimovanými daty bude dále třeba navrhnout základní kolekci přídavných funkcí. Mezi takové operace patří například zobrazení průběhu dané evoluce pouze v určitém časovém úseku, vyhledávání a porovnávání nebo kompletní zobrazení konkrétních organismů.

Kapitola 1

Metody komprese

V této kapitole vysvětlím základní pojmy týkající se komprese, a následně rozeberu několik nejčastěji používaných základních algoritmů komprese bezztrátové. Uvedu příklady jejich použití v praxi.

1.1 Základní pojmy

Abeceda, slovo, symbol a jazyk

- **Abecedou** Σ definuji libovolnou konečnou neprázdnou množinu. **Symboly** s_i nazvu jednotlivé její prvky; $\Sigma = \{s_1, s_2, \ldots, s_n\}$.
- **Slovo** nad abecedou Σ poté definuji jako konečnou posloupnost symbolů z dané abecedy; $w = s_1 s_2 \cdots s_n$, kde $s_i \in \Sigma$.
  - **Délka slova** $|w|$ je rovna počtu symbolů v posloupnosti.
  - Slovo, pro které platí, že $|w| = 0$ se označuje ϵ a říká se mu **prázdné slovo**.
  - **Prefixem slova** $w = s_1 s_2 \cdots s_n$ nazvu jakékoliv slovo $p = s_1 s_2 \cdots s_m$ pro které platí, že $m \leq n$, kde $m, n \geq 0$.
- **Množina všech slov** nad danou abecedou se označuje Σ^+. Pokud se za element této množiny považuje i prázdné slovo, pak se množina všech slov značí Σ^*.
- **Jazyk** L je definován jako podmnožina slov nad danou abecedou Σ; $L \subseteq \Sigma^*$.

Entropie a redundance, obsažnost jazyka

Kvalita kódovacích metod a kompresních algoritmů se často posuzuje podle toho, jak si poradí s redundancí daného jazyka. K zadefinování tohoto pojmu z teorie informace použiji následující pojmy:

- Základní jednotka množství informace se nazývá **bit** (z anglického binary digit). Značí se b. Může nabývat jednu ze dvou logických hodnot, nejčastěji je značíme 0 a 1. Osm bitů tvoří jeden **byte** (B).

- **Entropie** jazyka L je střední hodnota informace obsažené v symbolu abecedy Σ jazyka L. Vyjadřuje průměrný počet bitů nezbytných k zakódování symbolu z Σ při použití optimálního kódování [9]. Spočítá se jako
$$H(L) = -\Sigma_{i=0}^{n} p(s_i) \cdot \log_2(p(s_i)),$$
kde $p(s_i)$ je pravděpodobnost výskytu symbolu $s_i \in \Sigma$ v jazyce L.

- **Obsažnost** jazyka pro slova délky N se rovná průměrné entropii znaku slova dlouhého N:
$$R_N = \frac{H(L)}{N}.$$
Obsažnost jazyka vzhledem k jednomu symbolu se spočítá jako
$$r = \lim_{N \to \infty} R_N.$$
Absolutní obsažnost jazyka D vyjadřuje průměrnou entropii symbolu, kdyby všechny symboly a všechna slova byla stejně pravděpodobná. Je definována vztahem
$$D = \log_2(|\Sigma|),$$
kde $|\Sigma|$ je počet symbolů abecedy.

- **Redundance** jazyka vzhledem k jednomu symbolu vyjadřuje, jaké množství informace je v jednom symbolu abecedy daného jazyka nadbytečných a je dána výrazem
$$R = D - r.$$

Kód, kódování

Jako **kód** [3] definuji předpis, pomocí kterého lze data převést do jiné reprezentace, neboli uspořádanou trojici

$$c = \langle A, C, f \rangle,$$

kde A je zdrojová abeceda, C je kódová abeceda a f značí prosté zobrazení symbolů z abecedy A do prostoru slov v abecedě C,

$$f : A \to C^+.$$

Každému symbolu zdrojové abecedy tedy odpovídá jedno slovo z kódového jazyka, tzv. **kódové slovo**.

- **Kódování** je aplikace kódu c na konkrétní data. Pokud $C = \{0, 1\}$, což je v době počítačů nejčastější kódová abeceda, pak se jedná o **binární kódování**. V této práci se nadále budu zabývat pouze metodami binárního kódování.
- **Blokový, prefixový a optimální prefixový kód**
  - **Blokový kód** definuji jako kód, ve kterém mají všechna kódová slova stejnou délku.
  - **Prefixový kód** je takový kód, v kterém žádné kódové slovo není prefixem jiného kódového slova, tudíž jej lze jednoznačně dekódovat bez speciálního symbolu, který by jednotlivá slova odděloval.
  - **Optimální** (též minimální) **prefixový kód** je takový prefixový kód, který obsahuje minimální možný počet symbolů pro kódování jednotlivých slov.

Komprese

Komprese je zakódování dat s cílem zmenšit jejich objem [5], při současném zachování informací v těchto datech obsažených, oproti původní reprezentaci. Dělí se podle mnoha různých kritérií, některá z nich zmíním v této sekci.

Kompresní poměr a úspora místa

- **Kompresní poměr** (CR, compression ratio) se spočítá jako

$$CR = \frac{S_{in}}{S_{out}},$$

  kde S_{in} je velikost dat v bitech před aplikací dané kompresní metody a S_{out} je velikost zkomprimovaných dat.
- Procentuální **úspora místa** (SS, space savings) se spočítá jako

$$SS = 1 - \frac{S_{in}}{S_{out}}.$$

Ztrátová a bezztrátová komprese

- **Bezztrátová komprese** je definována jako komprese z jejíhož výstupu lze pro každý výstup získat původní vstup. Je nutné ji použít tam, kde vstupní data pro cílové použití již neobsahují žádnou redundanci, například pro binární soubory, v kterých data mohou být znehodnocena již po záměně či vynechání jediného bitu.
- **Ztrátová komprese** se vyznačuje tím, že ve výsledných komprimovaných datech není stoprocentně zachována původní informace, takže původní vstup z těchto dat nelze úplně zrekonstruovat. Většinou se skládá ze dvou fází:

– zahození dat nerelevantních pro další použití,

– bezztrátová komprese zbylých dat.

Z čehož vyplývá, že dosahuje oproti bezztrátové kompresi lepšího kompresního poměru. Uplatnění nachází tam, kde má smysl tuto částečnou ztrátu tolerovat, typicky třeba pro výstupy, které budou dále zpracovány už jen nedokonalými lidskými smysly. Nejrozšířenějším zástupcem takto komprimovaných výstupů je video a hudba.

Komprese statická, semi-statická a adaptivní

- Komprese se nazývá **statická**, pokud je kód předem určený pro všechna slova zdrojového jazyka, takže data lze zpracovat jediným průchodem.

- Při **adaptivní** kompresi daný kód není jednoznačně určen již před začátkem komprese, ale může se během jejího průběhu dynamicky měnit v závislosti na charakteristice vstupních dat.

- **Semi-adaptivní** komprese znamená, že kód je vytvořen podle vzorku vstupních dat, v průběhu samotné komprese se ale již nemění, čímž se liší od komprese adaptivní. Data je tedy potřeba projít dvakrát, prvním průchodem je spočítána četnost dat a stanoven kód. Během druhého průchodu jsou tato data zakódována.

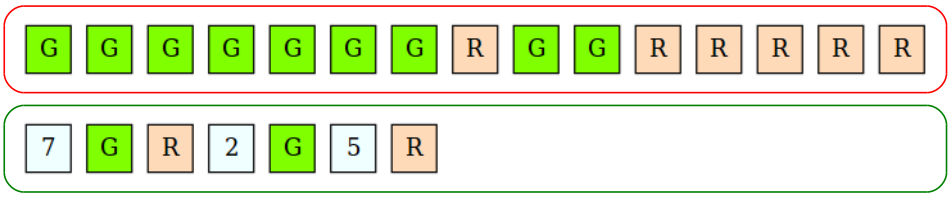
1.2 Kompresní algoritmy

Nyní přejdu k nejpoužívanějším metodám bezztrátové komprese. Představím metody, které byly v rámci této práce zkoumány, a uvedu příklady jejich použití v praxi. Považuji za nutné zmínit, že v dnešní době téměř každý program pro bezztrátovou kompresi dat kombinuje více různých kompresních algoritmů a přístupů k nim. Čerpal jsem z těchto zdrojů [10, 15, 14, 1].

1.2.1 RLE

RLE (run length encoding [10]) je velice jednoduchá, statická metoda komprese dat, která transformuje každou posloupnost stejných symbolů vstupní abecedy ve vzorku na uspořádanou dvojici $\langle C, S \rangle$, kde C značí počet symbolů v posloupnosti a S je symbol. Výhodou je velmi jednoduchá implementace.

V praxi se využívá pro data, která obsahují dlouhé posloupnosti stejných symbolů, typicky bitmapy obrázků se souvislými jednobarevnými plochami. S RLE jako hlavní kompresní metodou se setkáme například ve formátu PCX, známém z programu MS Paintbrush. Příklad aplikace této kompresní metody je na obrázku 1.1.



Obrázek 1.1: Vzorek dat “GGGGGGGRGGRRRRR” před (červený rámeček) a po RLE kompresi (zelený rámeček). Každá posloupnost složená ze stejných symbolů byla nahrazena jedním výskytem tohoto symbolu, kterému předchází délka této posloupnosti.

1.2.2 Huffmanovo kódování

V roce 1952 vymyslel David A. Huffman algoritmus, kterým lze sestavit optimální prefixový kód [15]. Délka kódu pro jednotlivé znaky se určuje z pravděpodobnosti jejich výskytu v kódovaném textu, původní verze Huffmanova kódování je tedy semi-statická metoda.

Využívá se například jako poslední aplikovaná kompresní metoda pro formát MP3, hlavní metoda komprese pro formát JPEG a jako jedna z použitelných metod v téměř každém současném programu pro kompresi dat.

Sestavení stromu

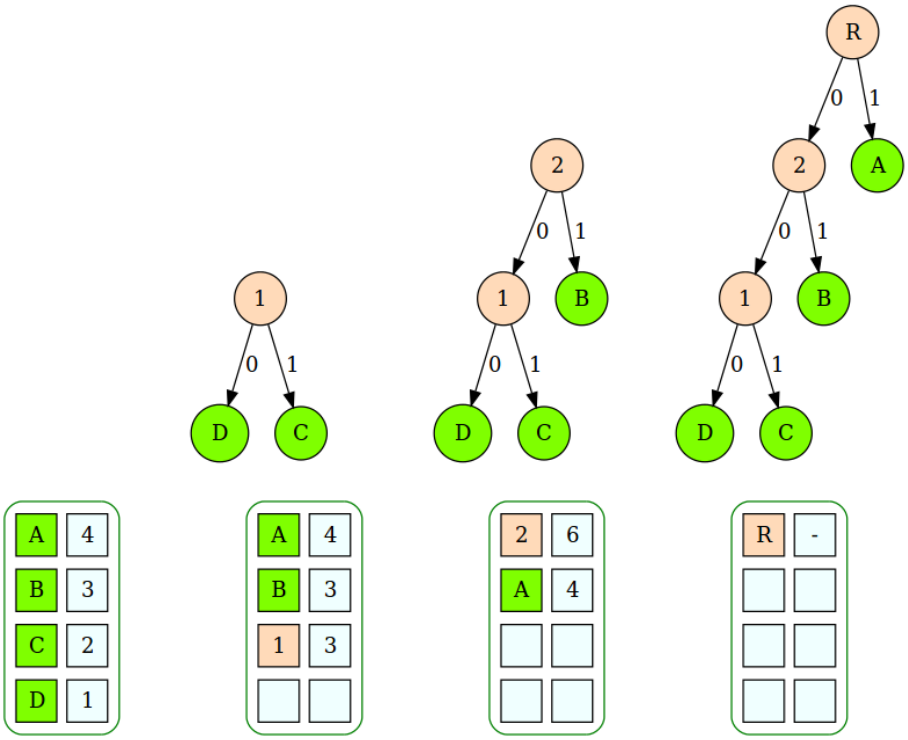
Při prvním průchodu je sestavena tabulka četností jednotlivých symbolů, z kterých se stanou listové uzly výsledného stromu. Z tabulky postupně vytváříme strom tak, že z ní jsou vyjmuty vždy dva symboly s nejnižší četností, jsou prohlášeny za potomky nově vytvořeného uzlu, který je do tabulky vložen a jehož četnost je součtem četností obou jeho potomků.

Proces opakujeme, a skončíme tehdy, když v tabulce zůstane jen jediný uzel. Směrem zespoda nahoru tak vzniká binární strom, jehož kořenem je poslední prvek v tabulce. Příklad je na obrázku 1.2.

Komprese a dekomprese

Samotná komprese proběhne tak, že se vytvoří převodní tabulka, v níž bude každý symbol kódován posloupností bitů, odpovídajících cestě do jeho listu z kořene Huffmanova stromu, přičemž přechod do levého/pravého potomka bude reprezentován jedničkovým bitem, přechod do druhého potomka nulovým. Následně každý symbol substituujeme za jemu příslušný kód.

Při dekompresi se prochází stromem do levého/pravého potomka podle nejvyššího nezpracovaného bitu. Pokud se narazí na listový uzel, symbol je nahrazen za jemu příslušný kód a dekomprese dalšího symbolu začíná opět z kořene stromu.



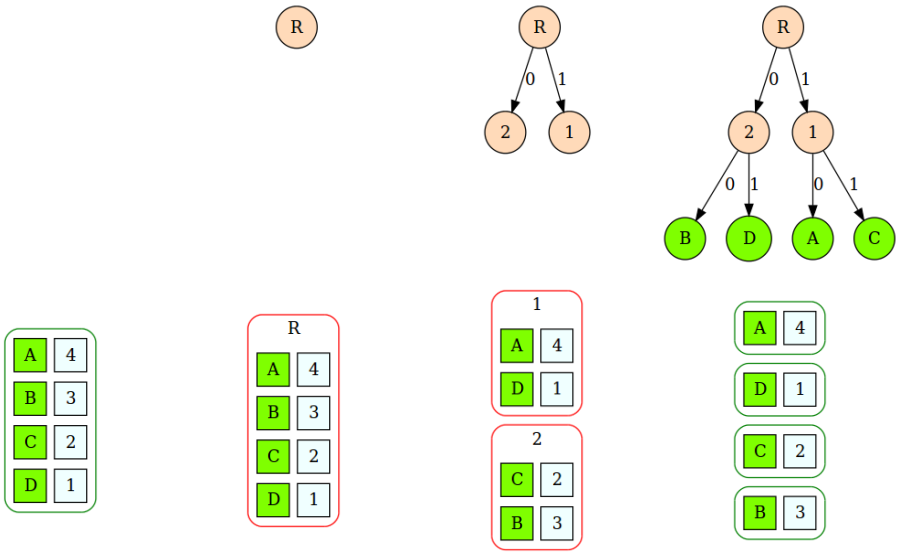
Obrázek 1.2: Vytvoření Huffmanova stromu z tabulky četností výskytů pro vstup "AABABCABCD". V každém kroku jsou z tabulky vyjmuty dva symboly s nejnižší četností. Ty jsou prohlášeni za potomky nově vytvořeného uzlu, který je do tabulky vložen a jehož četnost je součtem četností obou jeho potomků. Výsledné přidělené kódy se získají tak, že se zřetězí hodnoty všech hran v cestě z kořene stromu do odpovídajícího listu.

Nejednoznačnost

Huffmanův kód není určen jednoznačně, platné Huffmanovy kódy můžeme získat například rotací jednotlivých podstromů nebo záměnou jedniček za nuly. Tudíž je potřeba uchovávat použitý kódovací strom společně s komprimovanými daty. U malých vstupů může uložený strom zabrat více místa než samotná komprimovaná data, z čehož vyplývá, že tento způsob komprese je vhodný pro větší množství dat.

Adaptivní verze

Adaptivní verze dokáže zpracovávat data jedním průchodem, pro začátek se předpokládá četnost všech symbolů nulová [4]. Tabulka četností a strom se mění dynamicky, jsou upraveny po každém načteném symbolu. Z toho vyplývá, že stejnému znaku mohou být v průběhu kódování postupně přidělovány



Obrázek 1.3: Stavba Shannon-Fanova stromu pro vstup “AABABCABCD”. Začíná se vytvořením kořene, který reprezentuje množinu všech symbolů. V jednom kroku algoritmu je každá množina uzlů rozdělena na dvě tak, aby rozdíl součtu četností obou množin byl co nejmenší. Za každou rozdělenou množinu jsou potom přidáni potomci uzlu, který reprezentoval množinu původní. Skončí se ve chvíli, kdy každá množina obsahuje právě jeden symbol. Kódy odečteme stejně jako u Huffmanova stromu.

odlišné kódy. Při dekompresi je tedy nutné strom stavět a upravovat průběžně, což snižuje její rychlost.

Vzhledem k tomu, že není třeba ukládat kódovací strom, tak může tato metoda oproti semi-statické dosáhnout lepšího kompresního poměru pro malé množství dat. Jelikož přesné poměrné zastoupení symbolů je během komprese neznámé, pro velké množství dat bude dosahovaný kompresní poměr statisticky horší.

1.2.3 Shannon-Fanovo kodování

Algoritmus se velmi podobá Huffmanovu kódování. Zásadní rozdíl je ve stavbě kódovacího stromu směrem shora dolů [14]. Daný kód je sice prefixový, ale není optimální, tudíž výsledný kompresní poměr je statisticky horší než u Huffmanova kódování.

Využívá se jako jedna z kompresních metod programu PKZIP [13]. Příklad je na obrázku 1.3.

Sestavení stromu

Tabulka četností se vytvoří stejně jako u Huffmanova kódu. Strom se staví tak, že se tabulka rozdělí na dvě poloviny, které budou mít co nejmenší rozdíl součtu svých výskytů. Každá polovina dostane odlišný jednobitový prefix ke svému kódu. Pokračuje se rekurzivně pro každou polovinu zvlášť.

Komprese a dekomprese

Probíhá stejně jako u Huffmanova kódování.

1.2.4 Lempel-Ziv-Welch 84

Je algoritmem, který používá dynamicky tvořený slovník slov, který některým slovům vstupního jazyka přiřazuje číselný index. Ten je na začátku naplněn všemi symboly abecedy vstupního jazyka [1].

Ve vstupu se potom najde takový nejdelší prefix, který je ještě obsažen ve slovníku a na výstupu je substituován za svůj slovníkový index. Do slovníku je jako nové slovo přidán původní prefix, zřetězený s následujícím znakem na vstupu. Prefix je poté ze vstupu odstraněn.

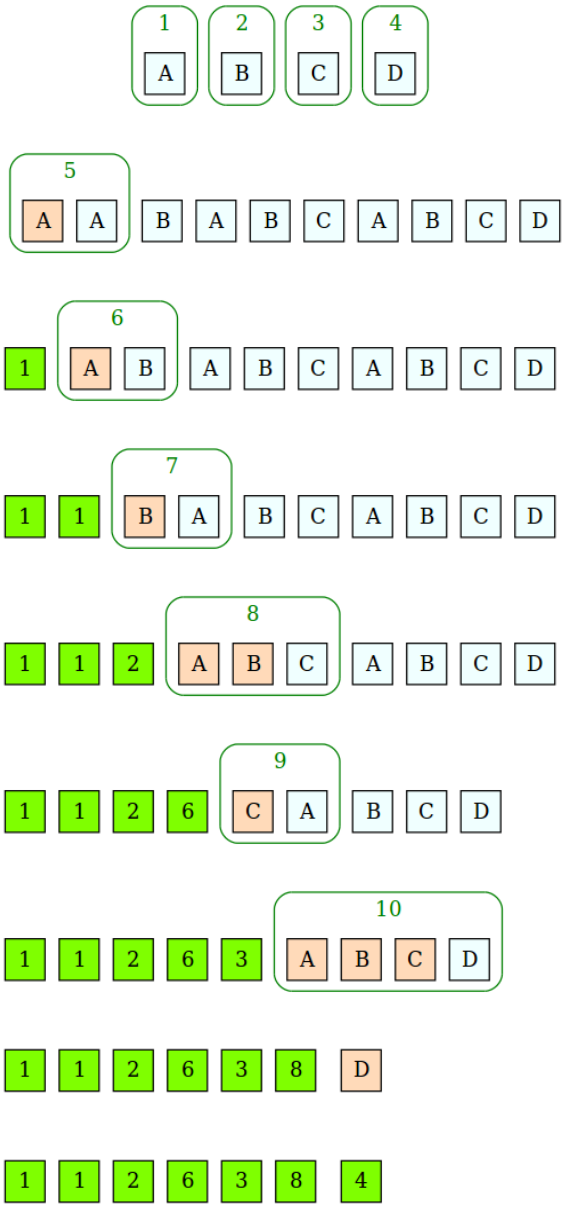
Algoritmus ve zbylém vstupu opět najde nový nejdelší prefix, postup se opakuje až do konce vstupu. Jedná se o adaptivní kompresní metodu. Ukázka použití této metody je na obrázku 1.4.

Dekomprese

Na začátku je slovník opět inicializován všemi symboly abecedy vstupního jazyka.

Pokud je na vstupu index, který se již nachází ve slovníku, odstraní se a na výstup se přidá slovo odpovídající danému indexu ve slovníku. Jako nové slovo je do slovníku přidáno slovo stávající, zřetězené s následujícím znakem na vstupu.

Pokud je ovšem na vstupu index, který se ve slovníku nenachází, vloží se znovu na výstup poslední slovo, zřetězené se svým prvním znakem. Celý tento řetězec je potom vložen do slovníku jako nové slovo.



Obrázek 1.4: Ukázka běhu algoritmu LZW pro vstup “AABABCABCD”. Na začátku je slovník naplněn všemi symboly abecedy jazyka. V každé iteraci se na vstupu najde nejdelší prefix obsažený ve slovníku – vyznačeno červeně. Do slovníku se poté přidá zřetězený s následujícím symbolem a odpovídajícím indexem – zelený rámeček. Původní prefix je v následující iteraci substituován za svůj slovníkový index. Po zpracování vstupu obsahuje slovník všechna slova v zelených rámečcích.

Kapitola 2

Analýza

V této kapitole se zabývám analýzou stávajícího systému digitální evoluce SVET a popisem jeho výstupů. Kompletní dokumentace k systému SVET, včetně seznamu všech instrukcí, je v dispozici v magisterské práci Lenky Vláčilové [11].

2.1 Popis systému SVET

SVET je systém pro digitální evoluci, ve kterém spolu organismy soupeří o přežití v prostředí s limitovanými zdroji. Tyto organismy jsou schopny spolu vzájemně interagovat, mutovat a množit se.

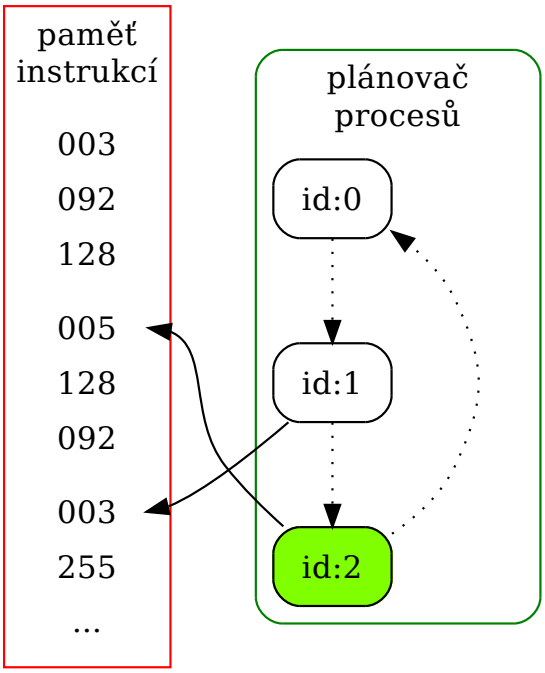
Prostředí SVETa představuje uzavřený prostor, na kterém jsou náhodně zapsány různé instrukce. Na začátku je část prostředí nastavena do určitého počátečního stavu a do této sekce je vypuštěn první organismus, který je předkem všech ostatních. Tento organismus poté "putuje" po SVETě, přičemž vykonává instrukce, na které při svém putování narazí. Vykonané instrukce mají vliv na budoucí podobu SVETa.

Pokud organismus při svém putování narazí na instrukci množení se, vytvoří se někde v prostředí SVETa jeho potomek. V případě, že narazí na neplatnou instrukci, "zemře".

Běhové prostředí

Běhové prostředí má charakter jednorozměrného pole, které slouží jako paměť instrukcí i dat, sdílená všemi procesy. Dále je k dispozici několik globálně přístupných registrů. Plánovač procesů je implementován jako cyklický jednosměrně zřetězený seznam. Pokud vznikne nový proces, je přiřazen na začátek seznamu, v další iteraci se tedy k vykonání instrukce dostane jako první proces, který vznikl poslední.

Organismus (proces) je v aktuální verzi reprezentován jako struktura, skládající se z ukazatele do paměti (který je ve SVETě pojmenován jako "program



Obrázek 2.1: Schéma systému pro digitální evoluci SVET. Proces s identifikátorem 2 se právě dostal v plánovači na řadu, takže provede instrukci číslo 5 (která může mít vliv například na sdílenou paměť) a adekvátně změní svůj ukazatel do paměti. Poté se dostane na řadu proces 0, který inkrementuje čas a plánovač předá řízení následujícímu procesu.

counter") a vlastního registrového pole. Procesy jsou v prostředí fixně plánovány, tudíž provádějí instrukce vždy ve stejném pořadí. V systému existuje speciální proces s identifikátorem 0, jehož jediným úkolem je inkrementace času. Zjednodušené schéma systému je znázorněno na obrázku 2.1.

Instrukce se skládají z 8bitového operačního znaku, za kterým může následovat jeden až dva operandy, které mohou mít 8, nebo 16 bitů. SVET podporuje následující adresní módy: registrový, přímý operand a registrový nepřímý.

Původní výstup

V původní verzi jsou výstupy reprezentovány velmi jednoduchým textovým výpisem. Jeho řádky jsou seřazeny vzestupně podle času, sekundárním řadícím klíčem je pořadí organismů v plánovači procesů. Ve výpisu jsou některé věci redundantní (např. ukazatel rámce), jiné naopak chybí; operandy některých instrukcí nejsou správně dekódovány. Výsledný výstup je velký a nedají se

```
cas:    4098    proc    3       pc=2272 inst:   69      240     255
cas:    4098    proc    2       pc=2269 inst:   19      0       1
cas:    4098    proc    1       pc=48   inst:   69      240     255
cas:    4099    proc    15      pc=2260 inst:   2       244     99
cas:    4099    proc    14      pc=2263 inst:   12      248     1
cas:    4099    proc    13      pc=2263 inst:   12      248     1
cas:    4099    proc    12      pc=2257 inst:   3       99      0
cas:    4099    proc    5       pc=2266 inst:   12      244     1
cas:    4099    proc    4       pc=2257 inst:   3       99      0
cas:    4099    proc    3       pc=2257 inst:   3       99      0
cas:    4099    proc    2       pc=2272 inst:   69      240     255
cas:    4099    proc    1       pc=33   inst:   3       99      0
cas:    4100    proc    15      pc=2263 inst:   12      248     1
cas:    4100    proc    14      pc=2266 inst:   12      244     1
cas:    4100    proc    13      pc=2266 inst:   12      244     1
cas:    4100    proc    12      pc=2260 inst:   2       244     99
cas:    4100    proc    5       pc=2269 inst:   19      0       1
cas:    4100    proc    4       pc=2260 inst:   2       244     99
cas:    4100    proc    3       pc=2260 inst:   2       244     99
cas:    4100    proc    2       pc=2257 inst:   3       99      0
```

Obrázek 2.2: Ukázka původního výstupu ze SVET. Na prvním řádku proces číslo 3 vykonal instrukci 69, která se nacházela na adrese 2272. Poslední dva sloupce reprezentují pouze stav následujících paměťových buněk – 2273 a 2274. Na čtvrtém řádku se změnil čas, a na řadu se dostal proces, který je v plánovači na první pozici.

v něm rychle a jednoduše vyhledávat zajímavé informace o tom, jak evoluce probíhala.

2.2 Statistická analýza

Abych získal informace o výstupu aplikace SVET, které by bylo možné využít k lepšímu návrhu samotné komprese, provedl jsem jednoduchou statistickou analýzu výstupů. Zaměřil jsem se na poměrné zastoupení jednotlivých instrukcí.

Pro změření četnosti jednotlivých instrukcí jsem upravil program tak, aby vypisoval všechny provedené instrukce do zvláštního souboru.

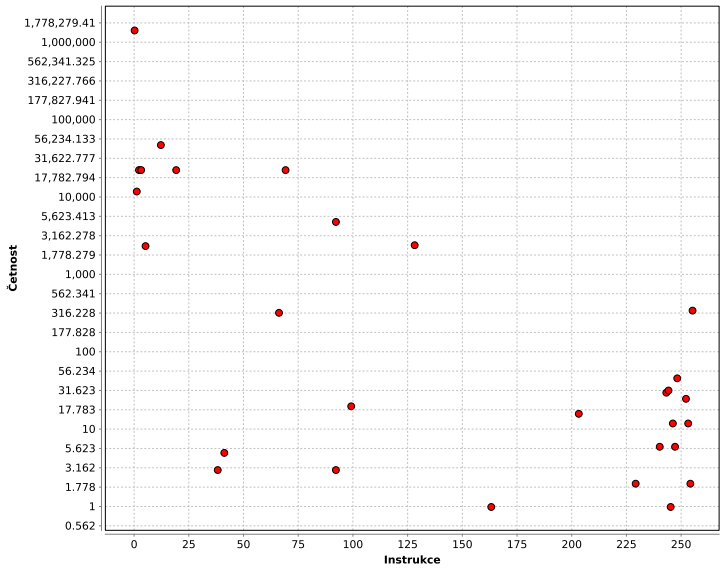
Veškeré provedené instrukce jsou závislé na prvotním nastavení prostředí systému SVET, které jsem měl k dispozici pouze jedno. Je pravděpodobné, že s jiným nastavením se procentuální zastoupení instrukcí může změnit.

Krátká evoluce

Spustil jsem třicet krátkých instancí evoluce (limit 10 000 jednotek času). Z nasbíraných dat vyplývá, že aktuální verze systému využívá pouze zlomek instrukcí, které jsou zastoupeny velmi nerovnoměrně. V krátkých cyklech evoluce je zdaleka nejčastější instrukce číslo 0. Výsledky jsou vizualizovány na obrázku 2.3.

Dlouhá evoluce

Poté jsem spustil jednu velmi dlouhou instanci evoluce, která běžela cca 4 hodiny reálného času. Po zaplnění paměti již není nejčastější instrukce 0 a celkový poměr instrukcí je více vyrovnaný. Také si lze všimnout, že tato dlouho běžící instance provedla podstatně méně různých instrukcí. Obrázek 2.4 interpretuje nasbíraná data.



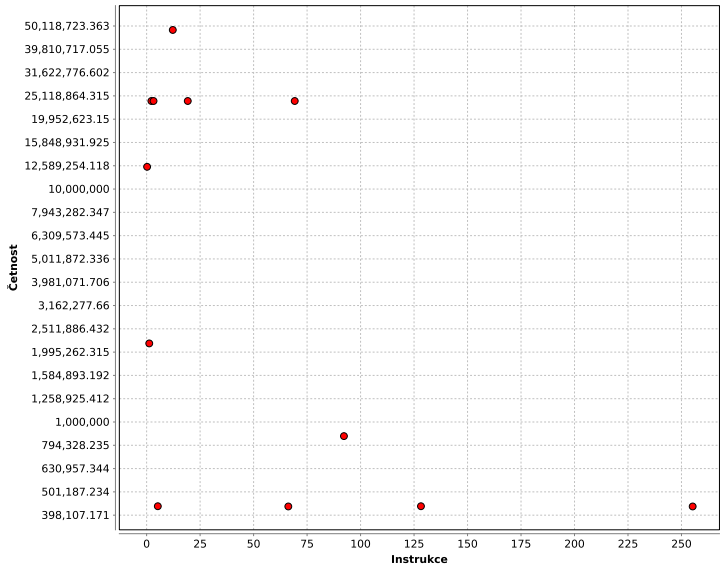
Obrázek 2.3: Četnost instrukcí ve třiceti krátkých cyklech, osa Y má logaritmické měřítko.

2.3 Analýza požadavků

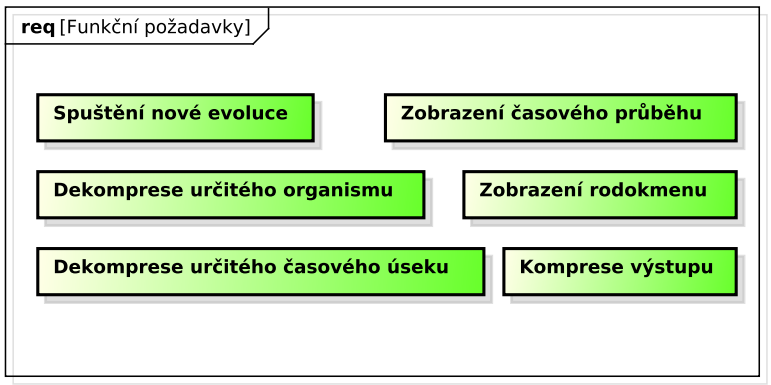
V průběhu vývoje aplikace vyvstalo několik požadavků, které bylo třeba do aplikace zakomponovat.

Funkční požadavky

- **Start nové evoluce**
  Z prostředí aplikace bude možné spustit novou instanci evoluce SVET.
- **Komprese a dekomprese výstupu**
  Výstupy z evoluce bude možné převádět z původní, člověkem čitelné formy, do formy komprimované.
- **Výběr časového úseku a jednotlivých organismů**
  Z výstupu již proběhlé evoluce půjde vybrat pouze určitý časový úsek i pouze určité organismy. Tyto výstupy poté půjdou samostatně uložit.



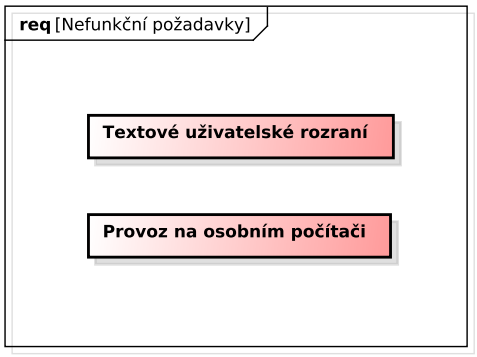
Obrázek 2.4: Četnost instrukcí v jednom velmi dlouhém cyklu, osa Y má opět logaritmické měřítko.



Obrázek 2.5: Funkční požadavky

- **Vizualizace rodokmenu a časového průběhu**
  Aplikace vygeneruje data pro proběhlou evoluci ve formátu DOT, jenž budou zachycovat rodokmen všech organismů. Taktéž vygeneruje data ve formátu CSV (Comma Separated Values, hodnoty oddělené čárkami), která budou zachycovat existenci organismů v závislosti na čase.

Nefunkční požadavky



Obrázek 2.6: Nefunkční požadavky

- **Textové uživatelské rozhraní**
  Aplikace bude nabízet jednoduché uživatelské textové rozhraní pomocí command line interface (CLI) neboli příkazové řádky.
- **Provoz na PC**
  Aplikaci bude možné provozovat na osobním počítači, bez nutnosti jakéhokoliv speciálního hardwaru či softwaru.

Kapitola 3

Návrh

V této kapitole předkládám návrh, jak řešit nedostatky nastíněné v rámci předchozí kapitoly. Navrhnu, jak upravit základní textový výstup, a poté metodu, jak tento výstup optimalizovat.

3.1 Použité technologie

Programovací jazyk – C/C++

Vzhledem k tomu, že původní program byl psán v jazyku C (norma ANSI), jsem se rozhodl, že pro zaznamenávání důležitých dat dodělám do původní aplikace modul a její výstup upravím. Komprese, dekomprese a selekce dat z nasbíraných záznamů následně budou realizovány pomocí samostatné aplikace, která bude psána v jazyce C++, s použitím nejnovějšího standardu C++11 a využitím rozšiřující knihovny Standard Template Library (STL).

Formáty použité pro výstup

- **CSV**
  Comma Separated Values je jednoduchý textový formát souboru, mezi jeho nejčastější využití patří přenos tabulkových dat mezi jinak nekompatibilními programy. Požadavky na formát CSV jsou následující [8]:
  - Celý soubor se skládá ze záznamů, oddělených koncem řádky. Poslední záznam konec řádky obsahovat nemusí.
  - Každý záznam se skládá z polí, oddělených jedním speciálním znakem. Typicky se používá mezera, čárka, nebo středník.
  - Pořadí polí v jednotlivých seznamech je v každém záznamu v rámci souboru stejné.
- **DOT**
  DOT je jazyk pro sémantický popis grafů s lidsky čitelnou syntaxí. Může

být použit jak pro popis orientovaných tak neorientovaných grafů. Existuje velké množství programů, které dokáží z DOT souboru vygenerovat grafický výstup, například volně dostupný Graphviz nebo OmniGraffle.

3.2 Komprese původního výstupu

cas:	3626	proc=	10	pc=	1182	inst:	92
cas:	3626	proc=	9	pc=	3526	inst:	0
cas:	3626	proc=	8	pc=	1657	inst:	128
cas:	3626	proc=	7	pc=	1906	inst:	3
cas:	3627	proc=	16	pc=	2907	inst:	0

Obrázek 3.1: Navrhovaná komprese původního výstupu: Červené sloupce budou zahozeny, každý zelený sloupec bude zakódován pomocí separátního Huffmanova kódovacího stromu. Sloupec modrý obsahuje rostoucí posloupnost, jejíž členy se liší maximálně o jedničku. Každý modrý řádek tedy bude zakódován jako jediný bit, značící, zdali došlo ke změně oproti předchozímu řádku.

Původním záměrem bylo zachování stávajícího výstupu. K jeho kompresi jsem chtěl použít statické Huffmanovo kódování. Sloupce obsahující textové popisky výstupu by byly vynechány.

Vzhledem k tomu, že časový údaj v prvním sloupci roste monotónně a má počátek v nule, měl být tento sloupec nahrazen za jednobitový praporek, indikující změnu oproti předchozímu stavu. Pro každý zbylý sloupec by se následně vygeneroval separátní Huffmanův kód, kterým by byl zakódován.

3.3 Nová forma výstupu

V průběhu implementace předchozího řešení bylo zjištěno, že některé požadované věci z aktuální formy výstupu není možné vůbec dohledat. Rozhodl jsem se tedy, že výstup z původního programu upravím do lépe zpracovatelné a komprimovatelné podoby.

Organismus

V nové formě výstupu bude mít každý organismus vlastní soubor, v kterém budou zaznamenány vykonané instrukce ve formátu CSV. Tím se znatelně

Tabulka 3.1: Schéma výstupních souborů, obsah prvního souboru tvoří metadata (červeně). Všechny tři soubory jsou uloženy v binární podobě.

Sekce	Obsah
Hlavička	Seznam organismů, jejich rodičů a časy vzniku/zániku.
Rejstřík	Ukazatele na bity v sekci data.
Kódový strom	Strom, pomocí kterého byl tento soubor zakódován.
Data	Zakódované instrukce jednotlivých procesů.

zjednoduší operace nad organismy oproti původnímu řešení, navíc lze data tohoto typu jednoduše zpracovávat libovolným programem pro statistické výpočty.

Globální data evoluce

Ve zvláštním souboru se bude ukládat pro každý proces jeho rodič, čas jeho vzniku a zániku. Tyto informace jsou typicky nejzajímavější pro proběhlou evoluci a vyplatí se je tedy zachovat i v zkomprimované formě, jako hrubý náhled na celkovou instanci evoluce. Taktéž je možné z těchto informací později jednoduše vygenerovat požadované grafy.

Zásah do původní aplikace

Do aplikace bude přidán rozšiřující modul, který se postará o sbírání dat a jejich zapisování do souborů. V průběhu evoluce budou jednotlivé organismy a jejich soubory organizovány v hashovací tabulce s uzavřeným adresováním. Jako vstup hashovací funkce se použije identifikační číslo organismu.

Všechny stávající vlastnosti aplikace zůstanou zachovány, včetně výstupu v původní podobě, který je využit pro porovnání výsledků.

3.4 Komprese nového výstupu

Jako kompresní metodu jsem zvolil statické Huffmanovo kódování, jelikož lze v datech v takto komprimované formě relativně snadno implementovat vyhledávání, což algoritmus LZW neumožňuje. Adaptivní forma Huffmanova kódování by zase měla statisticky horší kompresní poměr.

Obsah výsledných souborů

Celý výstup má ve své komprimované formě podobu tří binárních souborů. První slouží k indexaci, zároveň obsahuje globální informace o průběhu evo-

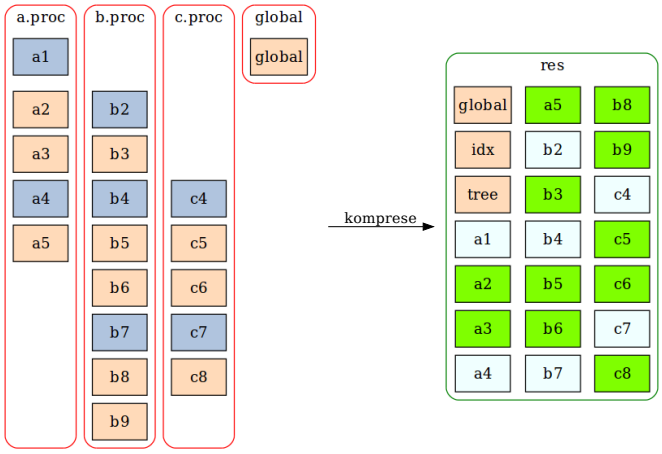
luce. Dále je v samostatném souboru uložen kódovací strom. V posledním souboru jsou uložena samotná komprimované data, viz tabulka 3.1.

- **Hlavička**
  Na začátku hlavičky je uložen počet organismů vzniklých v této evoluci. Po něm následuje odpovídající počet záznamů, popisujících organismy. Každý záznam se skládá z následujících položek:
  - identifikátor procesu
  - identifikátor rodiče
  - časový okamžik vzniku
  - časový okamžik zániku
- **Rejstřík**
  Na začátku této sekce je opět uloženo číslo, které odpovídá počtu indexů v rejstříku. Za ním je uložen shodný počet těchto indexů. Každý index má následující podobu:
  - identifikátor procesu
  - časová značka, na kterou tento index odkazuje
  - odpovídající bit v zkomprimovaném souboru
- **Kódovací strom**
  Kódovací strom je v souboru zapsán ve formátu pre-order. Za každý navštívený uzel se do souboru zapíše jeden bit. Za vnitřní uzel zapíše se bit s hodnotou 0, za listový uzel bit s hodnotou 1, následovaný hodnotou kódovaného čísla.

  Zpětné načítání takto uloženého stromu je rekurzivní. Rekurzivní funkce načte vždy jeden bit ze souboru. Pokud načte bit, jehož hodnota je 0, vytvoří vnitřní uzel a rekurzivně se zavolá pro levý a pravý potomek tohoto uzlu. Pokud načte bit jedničkový, pak načte ještě následující znak a vytvoří pro něj listový uzel.
- **Metadata**
  Jako metadata budu označovat pomocné, redundantní informace, sloužící pro pohodlnější orientaci v samotných datech evoluce – data obsažená v sekcích Hlavička a Rejstřík.

Komprese

Vstupem kompresní funkce je soubor s globálními daty evoluce a konstanta N, která udává, na jaké násobky času budou odkazovat indexy v rejstříku. Komprese bude provedena následovně:



Obrázek 3.2: Komprese nového výstupu pro N=3. Vlevo je výstup z aplikace SVET. V průběhu evoluce vznikly tři procesy – a, b, c a jeden soubor s globálními daty evoluce. Při kompresi tohoto výstupu se budou indexovat pozice instrukcí vykonaných organismy v čase dělitelném třemi – modrá políčka. Taktéž je třeba zaindexovat první vykonanou instrukci každého organismu. Komprimovaný soubor je ve schématu zakreslen po sloupcích. Globální informace o evoluci jsou uloženy jako první soubor, společně s vytvořeným rejstříkem (blok "idx"), který v sobě obsahuje pozice všech bílých bloků. V druhém souboru je uložen kódovací strom (blok "tree"). Následují samotná data evoluce ve třetím souboru.

Krok 1 Soubor s globálními daty evoluce bude načten do paměti, tím se získají čísla všech organismů v dané evoluci. Tím se získá informace o souborech, které budou komprimovány do výsledného.

Krok 2 Spočítá četnost výskytu jednotlivých instrukcí a operandů v těchto souborech. Nad instrukcemi a operandy se vytvoří jeden Huffmanův strom pro celou evoluci.

Krok 3 Podle stromu z kroku 2 se postupně komprimují jednotlivé soubory každého organismu, přičemž výsledky komprese se připisují do jednoho výsledného souboru.

Krok 4 Výsledkem komprimace jsou soubor s komprimovanými daty a soubor s odpovídajícím kódovým stromem.

Indexace

Z důvodu zefektivnění vyhledávání v zkomprimovaných datech jsem navrhl následující formu indexace. Pokaždé, když se zkomprimuje instrukce, která

byla provedena v čase dělitelným N, přidá se do rejstříku ukazatel na pozici posledního bitu. Stejně tak se ukazatel přidá ve chvíli, kdy se začne komprimovat soubor dalšího organismu.

Ke zjištění, která instrukce byla vykonána v jakém čase, se využije faktu, že každý organismus vykoná za jednotku času právě jednu instrukci. Navíc je v metadatech uložen čas vzniku každého organismu.

Dvojice indexu a ukazatele jsou ve výsledném souboru uchovávány v rejstříku.

Dekomprese

Rejstřík a kódovací strom jsou načteny do paměti. Poté je v rejstříku vyhledána pozice prvního a posledního bitu pro každý organismus. Pro všechny organismy je potom sekvenčně dekomprimována celá jejich sekce a každý z nich je uložen do samostatného souboru.

Dotazy a operace nad komprimovaným výstupem

Pro snadnou práci s komprimovanými daty navrhuji několik přídavných funkcí, které mají za cíl zefektivnit interpretaci těchto dat pro nejčastěji zkoumané vlastnosti evoluce.

- **Rodokmen a časový průběh**
  Rodokmen bude z důvodu větší efektivity generován rovnou z metadat, bez nutnosti jakoukoliv část souboru dekomprimovat. Z hlavičky souboru se postupně načtou identifikátory všech obsažených organismů a jejich rodičů.

  Tato data budou následně transformována do univerzálního grafového formátu DOT a vypsána do výstupního souboru. Lze je tudíž vizualizovat jakýmkoliv programem kompatibilním s DOT.

  Graf časového průběhu se taktéž generuje přímo z metadat. Výsledkem je CSV soubor, který může být zobrazen prakticky libovolným programem pro statistické výpočty.

- **Preparace organismů**
  K získání popisu jednotlivých organismů se v rejstříku vyhledají indexy ukazující na začátek daného a následujícího organismu. Poté se takto ohraničená část souboru dekomprimuje. Výsledkem je popis jednoho organismu.

- **Preparace časového úseku**
  V rejstříku se vyhledají pro každý organismus, který existoval v daný časový interval, indexy, jejichž časové značky tento interval překrývají. Poté jsou části odpovídající části souboru dekomprimovány. Výsledky jsou po dekompresi případně ještě oříznuty, aby začínaly a končily v požadovaný časový okamžik.

Kapitola 4

Implementace

V této kapitole popíši způsob, kterým jsem naimplementoval mnou navržená řešení.

4.1 Huffmanovo kódování

Zde popíši třídy, které jsem implementoval, aby bylo možné převádět data ze standardní reprezentace do Huffmanova kódu, a tyto data poté nadále zpracovávat.

Bitové buffery

V Huffmanově kódování mohou mít jednotlivé kódy různou délku, která navíc nemusí být vyjádřitelná pomocí celých bajtů. Jazyk C/C++ dokáže pracovat pouze s celými bajty. Jedinou možností jak zpřístupnit konkrétní bit jsou operátory bitového posunu a logické bitové operace na atomických typech. Bylo proto potřeba implementovat dvě třídy, které operace s jednotlivými bity v souboru zpřístupní.

Třída `BitReader`

Je třída, která dokáže ze souboru číst postupně jednotlivé bity. V paměti je načtená část souboru reprezentována jako staticky alokované pole typu char.

`BitWriter` je inicializován pomocí metody `Init`, která si vyžádá ukazatel na soubor. Eventuálně lze pomocí parametru předat pozici v souboru, na které se má začít číst.

Třída umožňuje přečíst následující bit a následující integer ze souboru.

Třída `BitWriter`

Je určena k zapisování sekvence bitů do souboru. Opět je pomocí metody `Init` přiřazena souboru. Jednotlivé bity jsou před zapsáním reprezentovány pomocí

STL kontejneru bitset.

Pro zapsání je určena metoda `Add`, která si vyžádá jako parametr ukazatel do paměti a počet bitů, které mají být zapsány. Pokud je po metodě `Add` v bitsetu počet bitů dělitelný osmi, jsou zapsány do souboru.

Aby všechny operace proběhly korektně, je třeba po skončení práce zavolat metodu `Flush`, která zapíše všechna zbývající data v bitsetu.

Třída `Huffman`

Pro realizaci statického Huffmanova kódování jsem naimplementoval třídu `Huffman`. Ta umí uložit a načíst strom, který vytvoří na základě analýzy vstupních dat. Tímto stromem potom dokáže komprimovat/dekomprimovat soubory. Jako symboly vstupního jazyka jsem použil hodnoty atomického typu integer.

- **Sestavení stromu**
  Metoda `countOcc` spočítá výskyty jednotlivých symbolů v daném souboru jedním průchodem, výstup je reprezentován pomocí kontejneru STL map, kde klíčem je daný symbol a hodnota jeho četnost. Tato mapa je přetransformována na multimapu, záměnou klíče a hodnoty.

  Ta je potom použita metodou `buildTree` ke stavě kódovacího stromu. Po jeho sestavení jsou všechny kódy symbolů pro potřeby komprese ještě uloženy do převodní mapy.

- **Zapsání stromu**
  Kódovací strom je do souboru zapsán ve formátu pre-order. Za každý navštívený uzel se do souboru zapíše jeden bit. Za vnitřní uzel zapíše se bit s hodnotou 0, za listový uzel bit s hodnotou 1, následovaný hodnotou kódovaného čísla.

- **Zakódování souboru**
  Pro zakódování souboru se volá metoda `compress`, která jako parametr potřebuje ukazatel na vstupní soubor a `BitWriter`, napojený na výstupní soubor. Podle převodní tabulky každý symbol vstupního souboru nahradí za odpovídající kód v převodní mapě a zapíše do `BitWriteru`.

- **Dekódování souboru**
  Pro dekódování souboru je potřeba metodě `decompress` předat ukazatel na výstupní soubor a `BitReader`, inicializovaný na soubor vstupní. Poté je na základě jednotlivých načtených bitů rekurzivně procházen strom.

  Pokud se narazí na listový uzel, je jeho hodnota zapsána do výsledného souboru a v další iteraci se strom opět prochází od kořene.

4.2 Komprese původního výstupu

Předzpracování dat

Pro přeformátování původního vstupu a zahození redundantních popisků jsem implementoval jednoduchou funkci `preprocess`, která tyto popisky zahodí a každý sloupec uloží do zvláštního souboru. Funkce `postprocess` má přesně opačný účel, ze sloupců opět sestaví původní výstup.

Třída `Diff`

Je použita pro kompresi času. Jelikož čas v systému běží vždy od okamžiku 0 a zvětšuje se mezi dvěma výstupy maximálně o jedničku, bylo by zbytečné pro tento údaj vytvářet zvláštní kódovací strom. Tudíž jsem každý časový údaj zakódoval jako bit 0, pokud se od předcházející časové značky nezměnil, a jako bit 1, pokud oproti předchozí časové značce vzrostl.

4.3 Nový výstup

Úprava výstupu aplikace SVET

Do SVETa jsem přidal hashovací tabulku, v které je pro každý organismus ve struktuře `TLog` uchován čas jeho vzniku, rodič a čas zániku. Dále obsahuje ukazatel na soubor, do kterého se zapisují všechny provedené instrukce tohoto procesu. Pro manipulaci se záznamy v této tabulce jsem naimplementovat několik funkcí, `logBirth` pro přidání nového organismu do tabulky, `logDeath` pro odebrání zaniklého organismu a `logInst` pro zapsání instrukce provedené daným organismem.

Vzhledem k tomu, že procesů je v systému maximálně omezené množství, jsem tuto tabulku implementoval s uzavřeným hashováním, jako staticky alokované pole spojových seznamů. Jako hashovací funkci jsem použil operátor modulo, jejím vstupem je identifikátor organismu.

Aby bylo možné evoluci kdykoliv přerušit, naprogramoval jsem `handler` signálu SIGTERM, který zajistí validitu nasbíraných informací a korektní uzavření všech souborů.

Upravení třídy `Huffman`

Po požadavku na změnu musela být tato třída přepracována tak, aby dokázala komprimovat více souborů a navíc při kompresi provádět jejich indexaci.

- **Rejstřík a metadata**
  Do této třídy byl přidán rejstřík a metadata. Obojí je reprezentováno STL množinou odpovídajících struktur – `TIndex` a `TOrganism`.

Tato data lze načítat a ukládat ze souboru pomocí funkcí `loadMeta` a `saveMeta`.

- **Generování rejstříku**
  Metodu `compress` jsem upravil, jejím vstupem je nyní navíc ještě množina organismů a konstanta, která říká, na jaké násobky času mají indexy v rejstříku ukazovat. Vždy, když se zkomprimuje instrukce v odpovídajícím čase, je do rejstříku uložena její bitová pozice ve výsledném souboru.

- **Dekomprese**
  Přepracoval jsem rozhraní metody `decompress`, nyní je jejím vstupem identifikátor organismu a časový interval, který bude dekomprimován.

  Pro dekompresi je v rejstříku nalezena nejbližší nižší časová značka organismu, od tohoto místa jsou dekomprimovány všechny instrukce až po konec časového intervalu. Instrukce, které neleží v časovém intervalu jsou zahozeny.

  Pro dekompresi celé evoluce je k dispozici přetížená verze, které volá původní verzi v cyklu.

- **Generování grafů**
  Přidal jsem metody `generateLineage` a `generateLifetime`, které z metadat generují požadované grafy do požadovaného souboru.

  Pokud je v systému přítomen Gnuplot a Graphviz, pak jsou tyto grafy rovnou vizualizovány jako vektorový obrázek ve formátu SVG.

4.4 Instalace a spuštění

Aplikace byla vyvinuta pro x86_64 GNU/Linux. Verze pro Microsoft Windows zatím realizována nebyla.

- Instalace se provede příkazem **“make install”**.
- Pro spuštění aplikace SVET se použije příkaz:
  “./svet [VÝSTUP]”.
- Pro práci s takto zkomprimovaným výstupem slouží:
  “./zverak [VÝSTUP]”.
- K odinstalaci slouží příkaz **“make clean”**.

Kapitola 5

Testování

Aplikace byla odladěna pro x86_64 GNU/Linux. Během vývoje jsem ji průběžně testoval.

Valgrind

Valgrind je nástroj pro ladění a profilování spustitelných souborů na operačním systému LINUX. Dokáže detekovat chyby související s pamětí, například neinicializované proměnné programu nebo přístup k nealokované části paměti. Použil jsem ho během celého vývoje aplikace.

Jednotkové testy

Po dokončení každé větší části programu jsem tuto část otestoval.

- **Test upraveného výstupu SVETa**
  Upravený výstup jsem ručně porovnával s původním. Kontroloval jsem také paměťovou korektnost všech provedených operací s hashovací tabulkou.

- **Testy bitových bufferů**
  Pro bitové buffery jsem provedl několik testů, spočívajících v zapisování a čtení různých dat, oříznutých na určité počty bitů. Takto získané výstupy jsem porovnával s očekávanými.

- **Test stavby Huffmanova stromu**
  Třída `Huffman` v několika iteracích stavěla strom nad různými daty, průběžným výpisem jsem kontroloval jeho validitu. Taktéž jsem otestoval zapsání tohoto stromu na disk, následné načtení a porovnání s původním stromem.

- **Komprese a dekomprese** Pro otestování jsem komprimoval náhodně generované sekvence čísel různých délek. Výstupy jsem poté dekomprimoval a porovnával se vstupy.

Systémový test

Na několika výstupech z aplikace jsem otestoval funkčnost celého systému, včetně generování grafů a dekomprese pouze určitých částí výstupu.

Kapitola 6

Výsledky

Tabulka 6.1: Výsledky pro osm různých běhů evoluce. Všechny velikosti jsou uvedeny v kB.

původní výstup	původní výstup – zip	nový výstup – data	nový výstup – strom	nový výstup – indexace	nový výstup – celkem
1230	**166**	69	51	1	**121**
155	**21**	8	9	1	**18**
1730	**239**	96	51	2	**147**
989	**141**	54	51	1	**106**
4831	**637**	280	135	3	**418**
5607	**700**	309	52	4	**365**
16481	**2070**	694	51	8	**753**
18901	**2218**	1054	96	10	**1160**

Výsledky ukazují, že pro tyto běhy evoluce dosáhl můj program lepšího kompresního poměru než ZIP. Navíc lze v těchto výstupech vyhledávat bez nutnosti dekomprese celého souboru.

Závěr

Cílem této bakalářské práce bylo navrhnout a naimplementovat optimální kompresi z výstupu systému digitální evoluce SVET.

V rámci analýzy jsem zjišťoval míru redundance výstupů a rozvržení instrukcí, kvůli volbě vhodného kompresního algoritmu. Představil jsem několik kompresních algoritmů, které by vyhovovaly pro zpracování těchto dat a vybral jsem z nich statické Huffmanovo kódování pro jeho dobrý kompresní poměr a snadné vyhledávání v zkomprimovaných datech.

Navrhl jsem kompresi šitou na míru digitální evoluci SVET, která textový výstup zmenší minimálně na desetinu původní velikosti.

Výsledkem je program, který komprimuje původní textový výstup do vlastního formátu, ze kterého lze snadno získat informace o průběhu evoluce, jako je rodokmen, preparace jednotlivých organismů, či výpis určitého časového úseku evoluce.

Výsledný výstup jsem porovnal se standardní metodou komprese ZIPem. Můj program dosáhl lepších výsledků.

Současná aplikace je konzolová, s jednoduchou nabídkou těchto možností – výpis vybraného časového úseku, podrobný výpis konkrétního organismu, výpis celé evoluce a rodokmen.

Do budoucna bych navrhl dodělat příjemnější uživatelské rozhraní, rozšířit nabídku generovaných grafů, případně rozšířit samotný SVET o nové vlastnosti organismů.

Literatura

[1] BHAT, S. *LZW Data Compression* [online]. Duke University [cit. 12.3.2015]. Dostupné z `https://www.cs.duke.edu/csed/curious/compression/lzw.html`

[2] FLEGR, J. *Úvod do evoluční biologie* 2. vyd. Praha : Academia, 2007. ISBN 978-80-200-1539-6.

[3] KOPECKÝ, M. *Dokumentografické informační systémy* [online]. [cit. 19.3.2015]. Dostupné z: `http://www.ms.mff.cuni.cz/~kopecky/vyuka/dis/11/dis11_v1.html`

[4] LOW, J. *Adaptive Huffman Coding* [online]. Duke University [cit. 28.3.2015] Dostupné z: `https://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html`

[5] MAIER, Z. *Ztrátová a bezztrátová komprese: Znáte rozdíl?* [online]. Softwarový magazín SWMag [cit. 29.12.2014]. Dostupné z: `http://www.swmag.cz/869/ztratova-a-bezeztratova-komprese-znate-rozdil/`

[6] RAY, T. *Tierra Home Page* [online]. Tierra Home Page [cit. 24.4.2015]. Dostupné z: `http://life.ou.edu/tierra/`

[7] SCHOLTZ, V. *Digitálna evolúcia* [online]. Aldebaran bulletin [cit. 17.12.2014]. Dostupné z: `http://www.aldebaran.cz/bulletin`

[8] SHAFRANOVICH, I. *RFC 4180* [online]. [cit. 10.4.2015]. Dostupné z: `https://tools.ietf.org/html/rfc4180`

[9] SHANNON, C. *A mathematical theory of information* 27. vyd. [s.n.]

[10] TIŠNOVSKÝ, P. *Implementace komprimace RLE* [online]. [cit. 7.3.2015]. Dostupné z: http://www.root.cz/clanky/pcx-prakticky-implementace-komprimace-rle

[11] VLÁČILOVÁ, L. *Digitální evoluce*. Diplomová práce, VŠCHT, 2010.

[12] DEVOLAB *Digital Evolution Lab* [online]. [cit. 24.4.2015]. Dostupné z: http://devolab.msu.edu/

[13] PKWARE Inc., *.ZIP File Format Specification* [online]. [cit. 10.4.2015] Dostupné z: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[14] Prague Stringology Club *Shannon-Fanovo kódování* [online]. [cit. 1.3.2015].Dostupné z: http://www.stringology.org/DataCompression/sf/index_cs.html

[15] Prague Stringology Club *Statické Huffmanovo kódování* [online]. [cit. 1.3.2015]. Dostupné z: http://www.stringology.org/DataCompression/sh/index_cs.html

Příloha A

Uživatelská příručka

A.1 Instalace a spuštění

Aplikace byla vyvinuta pro x86_64 GNU/Linux.

- Instalace se provede příkazem **“make install”**.
- Pro spuštění aplikace SVET se použije příkaz:
  “./svet [VÝSTUP]”.
- Pro práci s takto zkomprimovaným výstupem slouží:
  “./zverak [VÝSTUP]”.
- K odinstalaci slouží příkaz **“make clean”**.

A.2 Hlavní nabídka

```
What you want?
1. Show data
2. Generate lineage graph
3. Generate lifetime graph
4. Show organism in time
5. Show Huffman tree
6. Exit
```

A.2.1 Show data

Zobrazí hlavičku souboru, která ukazuje počet organismů vzniklých v načteném souboru evoluce, časy života a předky jednotlivých organismů.

A.2.2 Generate lineage graph

Vygeneruje DOT soubor, který zobrazuje rodokmen všech organismů. Pokud je v systému nainstalován Graphviz, pak je tento graf rovnou vizualizován ve formátu SVG.

A.2.3 Generate lifetime graph

Vyexportuje časy života všech organismů ve formátu CSV. Pokud je v systému přítomen Gnuplot, tato data vizualizuje jako krabicový graf (boxplot) ve formátu SVG.

A.2.4 Show organism in time

Dekomprimuje vybraný organismus ve zvoleném čase. Pokud je jako identifikátor organismu uvedena 0, je dekomprimován celý soubor.

A.2.5 Show Huffman tree

Zobrazí Huffmanův strom, který byl použit k zakování dat této evoluce. Strom je vypsán ve formátu preorder, kódy jsou vypsány v binárním formátu.

A.2.6 Exit

Ukončí program.

Příloha B

Seznam použitých zkratek

ANSI American National Standards Institute

CLI Command Line Interface

CSV Comma Separated Vales

GNU GNU's Not Unix!

PCX PC Paintbrush File Format

STL Standard Template Library

SVG Scalable Vector Graphics

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bin` ..... adresář se spustitelnou formou implementace
  - `svet` ..... aplikace SVET s upraveným výstupem
  - `zverak` ..... adresář pro kompresi výstupů ze SVET
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `Makefile` ..... Makefile
    - `svet.c` ..... zdrojový kód aplikace SVET
    - `svet_asm.h` ..... instrukce aplikace SVET
    - `svet_log.c` ..... modul pro výstup aplikace svet
    - `svet_log.h` ..... modul pro výstup aplikace svet
    - `buffer.h` ..... hlavičkový soubor bitových bufferů
    - `buffer.C` ..... třídy bitových bufferů
    - `huffman.h` ..... definice třídy pro huffmanovo kódování
    - `huffman.C` ..... třída s huffmanovým kódováním
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS