Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Vizualizace protisměrného vyhledávání ve stromech

Josef Rypáček

Vedoucí práce: Ing. Jan Trávníček

9. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Janu Trávníčkovi za poskytnutí zajímavého a prospěšného tématu, za rady ohledně způsobu zpracování a náplně práce.

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 9. května 2015 .....

Odkaz na tuto práci

Rypáček, Josef. *Vizualizace protisměrného vyhledávání ve stromech.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato bakalářská práce se zabývá vizualizací protisměrného vyhledávání vzorků ve stromech. Popisuje návrh a požadované vlastnosti vizualizace. Pojednává o výběru technologií a na základě požadavků a analýzy implementuje vizualizaci. V práci jsou porovnávány způsoby implementace a jejím výsledkem je webová aplikace pro vizualizaci algoritmu.

Klíčová slova protisměrné vyhledávání, vyhledávání vzorků, vyhledávání ve stromech, vizualizace algoritmu, JavaScript

Abstract

This thesis deals with visualization of backward tree pattern matching algorithm. It describes design and features of required visualization. It implements visualization based on requirements and analysis. The thesis compares methods of implementation and its result is web application for algorithm visualization.

Keywords backward matching, pattern matching, tree matching, algorithm visualization, JavaScript

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Stromy jsou důležitou datovou strukturou používanou v mnoha počítačových programech. Mezi důležité vlastnosti datových struktur patří také schopnost rychle vyhledávat v uložených datech. Mezi programy využívající stromy ke své činnosti patří například kompilátory a parsery. V této práci se zaměříme právě na vyhledávání vzorků ve stromových strukturách a vizualizaci jednoho z algoritmů. Právě výběr správného algoritmu je klíčový pro efektivitu celého programu.

Tato práce se zaměřuje na vizualizaci nového algoritmu vymyšleného na Katedře teoretické informatiky, Fakultě informačních technologií, Českého vysokého učení technického v Praze. Vizualizace tohoto algoritmu umožní snazší pochopení všech součástí algoritmu protisměrného vyhledávání ve stromech a může sloužit jako podpora při výuce studentů.

Cíle práce

- Představení vyhledávání ve stromech
- Vysvětlení protisměrného vyhledávání ve stromech
- Návrh možných způsobů vizualizace
- Vytvoření webové aplikace umožnující vizualizaci protisměrného vyhledávání včetně pomocných struktur
- Otestování vizualizace a porovnání jednotlivých způsobů

Struktura práce

V první kapitole se seznámíme se základními pojmy potřebnými pro porozumění této práci, představíme základy vyhledávání a určíme požadavky na vizualizaci algoritmu.

Druhá kapitola pojednává o výběru implementační technologie a návrhu jednotlivých částí vizualizace.

Třetí kapitola se věnuje implementaci navhrnutého řešení využitých postupech.

V poslední kapitole se budeme věnovat testování časových a pamětových nároků implementace a jednotlivé způsoby mezi sebou porovnáme.

Kapitola 1

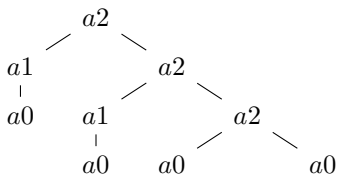
Analýza

1.1 Základní pojmy

Abeceda

Abeceda je konečná neprázdná množina symbolů. Ohodnocená abeceda je taková, kde má každý ze symbolů určenou nezápornou aritu. Symboly arity 1, 2, 3, ..., p jsou nazývány nulární, unární, binární, ternální, ..., p-ární. V naší práci budeme aritu symbolu značit číslem za symbolem samotným. Například $a1$ je unární symbol a, $b3$ je ternální symbol b. Jako příklad uvádíme ohodnocenou abecedu $\mathcal{A} = \{a2, a1, a0\}$.

Strom



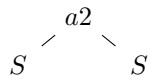
Obrázek 1.1: Strom s_1 z ohodnocené abecedy $\mathcal{A}$

Na obrázku 1.1 vidíme příklad stromu s_1 jako datové struktury. Přesněji jde o zakořeněný strom, který je definován jako strom s jedním význačným vrcholem (kořenem). Kořen jako jediný má vstupní stupeň roven 0. Hrany vedou vždy směrem od kořene k jeho potomkům (uzlům). Všechny uzly, kromě kořene mají vstupní stupeň 1. Uzly, které nemají další potomky

se nazývají listy a jejich výstupní stupeň je 0. Strom je acyklický, nesmí tedy obsahovat žádné smyčky. S tímto stromem budeme dále pracovat a vysvětlíme si všechny pojmy potřebné pro pochopení této práce.

Vzorek

Pro vyhledávání budeme potřebovat také vzorek, speciální případ zakořeněného stromu z abecedy $\mathcal{A} \cup \{S\}$. Vzorek může obsahovat zástupný symbol S symbolizující jakýkoli podstrom daného uzlu. Podstromem může být i list, uzel tedy nemusí mít žádné potomky.



Obrázek 1.2: Vzorek v_1

Řetězec

Řetězec o délce i je sekvence i po sobě jdoucích symbolů z abecedy $\mathcal{A}$.

Linearizace stromu

Pro algoritmus budeme potřebovat linearizovanou podobu vstupního stromu i vyhledávaného vzorku. Linearizace znamená převod stromu na řetězec podle určitých pravidel. Za tímto účelem je nutné definovat prefixovou ohodnocenou barovou notaci [1].

Notaci pref_ranked_bar(s) stromu s definujeme jako:

1. pref_ranked_bar(S) = S $|S$
2. pref_ranked_bar(a) = $a0$ $|0$, pokud a je list
3. pref_ranked_bar(s) = an pref_ranked_bar(b_1) pref_ranked_bar(b_2) ... pref_ranked_bar(b_n) $|n$, kde a je kořen stromu s, n je arita kořenu a $b_1, b_2, \ldots, b_n$ jsou potomky kořene a.

Pokud aplikujeme výše definovanou prefixovou ohodnocenou barovou notaci na ukázkový strom s_1, dostaneme pref_ranked_bar(s_1) = $a2$ $a1$ $a0$ $|0$ $|1$ $a2$ $a1$ $a0$ $|0$ $|1$ $a2$ $a0$ $|0$ $a0$ $|0$ $|2$ $|2$ $|2$. Pro vyhledávaný vzorek v_1 platí pref_ranked_bar(v_1) = $a2$ S $|S$ S $|S$ $|2$.

Při využití barové notace je potřeba rozšířit abecedu o barové symboly. V tomto případě $\mathcal{A} = \{a2, a1, a0, |2, |1, |0\}$.

1.2 Vyhledávání v datových strukturách

Vyhledávání ve stromech

Současné algoritmy pro vyhledávání vzorků ve stromech často využívají konečných automatů, některé jsou založeny na protisměrném vyhledávání v řetězcích, žádný ale nevyužívá linearizace stromu i vyhledávaného vzorku [2]. To nám umožňuje náhodný přístup k symbolům, jelikož linearizovaný strom může být implementován pomocí pole.

Protisměrné vyhledávání v řetězcích

Při protisměrném vyhledávání se vzorek testuje s řetězcem v opačném směru než se vzorek posouvá. Tedy vzorek se umístí na začátek textu (vlevo), ale začíná se testovat pravý znak vzorku. To často umožní při neshodě posunout vzorek o více než jeden znak a tím urychlit vyhledávání. Pro výpočet posunu se používá Boyer-Moore-Horspool algoritmus a vypočtená data se ukládají do tabulky bad character shift (BCS) [3].

Protisměrné vyhledávání ve stromech

Jak již bylo zmíněno v úvodu práce, budeme se zabývat algoritmem protisměrného vyhledávání ve stromech a jeho vizualizací. Tento algoritmus je založený na protisměrném vyhledávání v řetězcích, má však své odlišnosti, které je potřeba při implementaci vzít v úvahu. Především je potřeba ošetřit případy, kdy vyhledávaný vzorek obsahuje zástupný symbol S. Pro efektivní přeskakování podstromů identifikovaných zástupným symbolem S je použita tabulka subtree jump table (SJT).

Algoritmus 1 ConstructBCS [1] přijímá jako vstupní parametr vyhledávaný vzorek v prefixové ohodnocené barové notaci o velikosti m. Jeho výstupem je vytvořená tabulka BCS.

Vytvoření tabulky SJT zajišťuje algoritmus 2 ConstructSJT [1]. Jeho vstupním parametrem je strom délky n, index aktuálního uzlu $rootIndex$ a reference na prázdnou tabulku SJT. Výstupem je $index$ a vytvořená tabulka SJT.

```
1  begin
2      s := m
3      for i := 1 to m do
4          if pref_ranked_bar(pattern)[i] = S then s = m − i;
5      end
6      foreach x ∈ A do BCS[x] = m;
7      foreach x ∈ A do
8          if x ∉ A↑ then shift := s + Arity(x) ∗ 2;
9          else if s >= 2 then shift := s − 1;
10         else shift := s;
11         if BCS[x] > shift then BCS[x] := shift;
12     end
13     for i := 1 to m − 1 do
14         if pref_ranked_bar(pattern)[i] ∉ {S, ↑S} and
           BCS[pref_ranked_bar(pattern)[i]] > (m − i) then
           BCS[pref_ranked_bar(pattern)[i]] := m − i ;
15     end
16 end
```

Algoritmus 1: Vytvoření tabulky BCS

```
1  begin
2      index := rootIndex + 1
3      for i = 1 to Arity(pref_ranked_bar(t)[rootIndex]) do
4          index :=
           ConstructSJT(pref_ranked_bar(t), index, SJT(pref_ranked_bar(t)))
5      end
6      index := index + 1
7      SJT(pref_ranked_bar(t))[rootIndex] = index
8      SJT(pref_ranked_bar(t))[index − 1] = rootIndex − 1
9      return index
10 end
```

Algoritmus 2: Vytvoření tabulky SJT

Algoritmus 3 Protisměrné vyhledávání ve stromech [1] jako argumenty požaduje: strom v prefixové ohodnocené barové notaci velikosti n, vyhledávaný vzorek v prefixové ohodnocené notaci velikosti m, tabulku BCS a SJT. Výstupem jsou pozice nalezených shod vzorku se stromem.

```
1  begin
2      i := 0
3      while i <= (n − m) do
4          j := m
5          position := i + j
6          while j > 0 and position > 0 do
7              if pref_ranked_bar(subject)[position] =
               pref_ranked_bar(pattern)[j] then
8                  position := position − 1
9              else if pref_ranked_bar(pattern)[j] = ↑S and
               pref_ranked_bar(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked_bar(subject))[position]
11                 j = j − 1 {Subtree skip}
12             else break;
13             j := j − 1
14         end
15         if j = 0 then output(position + 1);
16         i := i + BCS[pref_ranked_bar(subject)[i + m]]
17     end
18 end
```

Algoritmus 3: Protisměrné vyhledávání vzorků ve stromech

1.3 Vstupní data

Vizualizace bude přijímat vstupní data, která bude dále zpracovávat. Konkrétně se jedná o:

- Vstupní strom
- Vyhledávaný vzorek
- Abeceda

Strom i vzorek budou zadávány v linearizovaném formátu určeném prefixovou ohodnocenou barovou notací. Abeceda pak bude představovat všechny

povolené symboly, oddělovačem bude mezera. Velikost vstupních dat není algoritmem omezena, pro účely vizualizace jsou však vhodné stromy v linearizovaném formátu do délky 50 symbolů pro nejběžnější monitory s FullHD rozlišením. Pro následné měření časové a paměťové náročnosti budou použity vstupní data většího rozsahu, aby bylo možné přesněji určit výsledné hodnoty.

1.4 Očekávané výstupy

Naše práce by měla po dokončení umožňovat vizualizaci následujících částí:

- Vstupní strom a vzorek v grafickém formátu
- Tabulky BCS a SJT s možností označování jednotlivých buněk
- Výstup algoritmu – nalezené výsledky
- Vizualizaci porovnávání vzorku se stromem a správné zarovnání symbolů
- Možnost krokování po jednotlivých řádcích
- Označení aktuálně prováděného řádku
- Výpis několika vybraných proměnných algoritmu

Kapitola 2

Návrh

Při návrhu vizualizace potřebujeme rozhodnout, v jakém programovacím jazyce provedeme následnou implementaci. Důležitá je pro nás multiplatformnost a spuštění bez nutnosti instalace. Pro účely vizualizace algoritmu protisměrného vyhledávání ve stromech tyto podmínky splňuje skriptovací jazyk JavaScript ve spojení se značkovacím jazykem HTML5 a kaskádovými styly CSS. Tato kombinace nám zajistí funkčnost ve všech moderních webových prohlížečích nezávisle na operačním systému.

Dále si musíme uvědomit, co všechno a jakým způsobem budeme vizualizovat. S použitím zvolených technologií máme dvě možnosti. Prvním způsobem jsou HTML elementy v kombinaci s kaskádovými styly. To je způsob, kterým je tvořena většina webových stránek a který podporuje každý internetový prohlížeč. Pomocí CSS lze vytvořit všechny běžné tvary, barvy i přesně definovat velikosti jednotlivých elementů. Druhou možností, jak vizualizaci implementovat je použití HTML5 Canvas. Element <canvas> je určen pro kreslení grafiky na webových stránkách. Lze ho použít pro zobrazení tvarů, přechodů, textu i obrázků [4].

HTML5 Canvas přináší výhody při potřebě zobrazovat složité tvary, animace, herní grafiku nebo video. V našem případě není zapotřebí vykreslování složitých tvarů, proto si vystačíme bez HTML5 Canvas a všechny zobrazované informace vyjádříme pomocí běžných HTML elementů, jejichž CSS atributy budeme měnit pomocí JavaScriptu. Pro snazší interakci mezi JavaScriptem a HTML použijeme velmi známou a rozšířenou knihovnu jQuery [5].

Návrh vizualizace můžeme rozdělit na 4 části:

- Získání a kontrola vstupů
- Vykreslení vstupů ve stromovém formátu
- Výpočet a vykreslení pomocných struktur a výsledku vyhledávání
- Krokování algoritmu

2.1 Získání a kontrola vstupů

Jako vstupní bod naší aplikace bude sloužit HTML formulář, který bude následně zpracován JavaScriptem. Formulář bude také obsahovat sadu předpřipravených vstupních dat vhodných pro ukázku. Po zadání vstupů a potvrzení formuláře proběhne jejich kontrola. Načteme abecedu a zjistíme, jestli obsahuje pouze platné znaky. Dále načteme vstupní strom a zkontrolujeme, zda obsahuje pouze symboly z abecedy a zahájíme převod na stromovou strukturu, během kterého kontrolujeme správnost syntaxe. Stejný postup opakujeme i s vyhledávaným vzorkem s tím rozdílem, že abeceda $\mathcal{A}$ je automaticky rozšířena o zástupné symboly S $|S$. Při zjištění jakékoli chyby zobrazíme varovnou hlášku a přerušíme další zpracování dat.

2.2 Vykreslení vstupů ve stromovém formátu

Pokud jsme zadali platné vstupy, proběhne jejich jednoduché vykreslení ve stromovém formátu pro okamžitou kontrolu. Tento výstup neslouží k dalším účelům, dále se již nebude se stromovou strukturou vstupů pracovat, neboť algoritmus protisměrného vyhledávání ve stromech požaduje linearizovaný vstup.

Pro vykreslení stromů můžeme zvolit z několika metod. Lze použít, případně upravit některou z dostupných knihoven pro vizualizaci grafů. Například vis.js [6] nebo Dracula Graph Library [7]. Další možností je napsání vlastní knihovny pro vizualizaci grafů nebo přímo stromů využívající HTML5 Canvas. Tím bychom však nezískali výhodu oproti již zmíněným knihovnám a vlastní implementace by byla časově náročná. Poslední možností je využití základních HTML elementů a zobrazení stromu vlastním způsobem, trochu odlišným od běžně používaného způsobu. Vzhledem ke skutečnosti, že vykreslení vstupů ve stromovém formátu je pouze okrajová

část naší vizualizace a pro pochopení algoritmu není důležitá, přiklonili jsme se k možnosti použití běžných HTML elementů. Tato metoda je rychlejší a výstup je méně náročný na obrazový prostor.

2.3 Vykreslení pomocných struktur a výsledku vyhledávání

Další požadovanou částí, kterou musíme vizualizovat jsou pomocné struktury algoritmu a jeho výstup – tedy nalezené podstromy. Zahájíme výpočet tabulek *bad character shift* a *subtree jump table*. Ty zobrazíme a následně je použijeme pro běh algoritmu protisměrného vyhledávání ve stromech. Po dokončení vyhledávání zobrazíme pozice, na kterých byly nalezeny shody se zadaným vzorkem.

Tabulky budou vytvořeny pomocí HTML elementu <table> v horizontálním směru. Jednotlivé buňky v tabulce bude možné označovat a vizualizovat tak přístup do datových struktur, které tyto tabulky znázorňují. Výsledek vyhledávání bude zobrazen formou čísel, která značí pozici ve vstupním stromě, na které byla nalezena shoda. Jednotlivé výsledky bude možné zvýraznit.

2.4 Krokování algoritmu

Hlavní částí vizualizace, kterou se budeme zabývat je krokování algoritmu po jednotlivých řádcích. To můžeme dále rozdělit na dva logické celky. Prvním je diagram znázorňující zadaný strom a pod něj zarovnaný vyhledávaný vzorek. Při krokování bude vzorek překreslován podle toho, v jaké fázi se algoritmus právě nachází. Tato část je nejdůležitější a simuluje porovnávání uzlů stromu a vzorku. Druhou částí je zobrazený pseudokód algoritmu, ve kterém je možno označovat právě prováděný řádek. U podmínek je barevně označeno její splnění, nebo nesplnění. Dále jsou vypisovány hodnoty důležitých proměnných algoritmu. V několika případech proběhne také označení buněk v tabulkách BCS, SJT a výsledku vyhledávání. Obě části vizualizace jsou mezi sebou synchronizované. Ovládání krokování bude možné jak myší (pomocí odkazů), tak pomocí klávesnice a bude shodné pro všechny navržené metody.

Důležitý je také způsob, jakým bude krokování probíhat. Standardní krok je nastaven na zpracování jednoho řádku pseudokódu algoritmu. Takzvaný velký krok simuluje vybrané řádky algoritmu, při kterých se mění výstup nebo potvrzuje shoda vzorku se vstupním stromem. Jedná se o řádky

7, 9 a 12, kde je porovnáván strom se vzorkem v podmínkách a řádek 15 algoritmu 3, který ověřuje shodu.

Jelikož se jedná zároveň o nejnáročnější část, provedeme podrobnější návrh možných řešení. V úvahu připadaní následující způsoby:

- Přepočítávané krokování
- Předpřipravené krokování
- Reverzibilní krokování

2.4.1 Přepočítávané krokování

V této jednoduché metodě vizualizace a krokování nepoužijeme další pomocné struktury a nebudeme ukládat zjištěné informace pro další použití. Při každém kroku vždy spustíme výpočet znovu od počátku do požadovaného kroku. Tím získáme minimální paměťovou náročnost, ale celková časová náročnost může být až nepřiměřeně vysoká. Pro velikost dat, která se hodí pro vizualizaci však předpokládáme časy zpracování v řádu jednotek milisekund. Proto tuto metodu implementujeme a následně otestujeme na velkých testovacích datech, kde očekáváme projevení vyšší časové náročnosti.

2.4.2 Předpřipravené krokování

Druhá metoda je opakem přepočítávaného krokování. Všechny výsledky, které jednou vypočítáme si uložíme a vícekrát nebude třeba výpočet spouštět. Všechny informace podstatné pro vizualizaci budeme načítat z datové struktury. Časovou náročnost se nám podaří snížit na minimum, bude však nutné použít pole struktur pro uložení všech vizualizovaných informací. Při testování budeme u tohoto způsobu sledovat především paměťovou náročnost, která bude s přibývající velikostí vstupů narůstat. U dat o velikosti vhodné pro vizualizaci předpokládáme maximální velikost pole v řádu stovek kilobajtů, což v dnešní době považujeme za zanedbatelné. Tuto metodu budeme implementovat a následně ji srovnáme s metodou přepočítávaného krokování. Na velkých testovacích datech očekáváme nízkou časovou náročnost, za cenu vyššího využití paměti.

Do pomocné struktury budeme ukládat následující položky:

- Číslo aktuálního kroku a informaci, zda-li se jedná o velký krok
- Buňku k označení v tabulce BCS a SJT

- Položku výstupu, která má být zvýrazněna
- Číslo řádky pseudokódu algoritmu a barva označení
- Vygenerovaný a zarovnaný vzorek – hlavní část vizualizace
- Hodnoty proměnných i, j, *position*

2.4.3 Reverzibilní krokování

Metodu, která si neukládá všechny vypočítané informace, ale přesto ji není nutné vždy spouštět znovu od počátku nazýváme reverzibilní. Pomocí ukládání několika nejdůležitějších proměnných by bylo možné optimalizovat metodu a rychle se dostat k požadovanému kroku. Tato metoda by byla na implementaci nejvíce náročná a její rychlost a paměťové nároky se pohybují mezi dvěma předchozími metodami. Hlavním problémem jsou však kroky zpět, které musí vizualizace podporovat. Ne všechny operace v algoritmu vyhledávání jsou reverzibilní a bylo by nutné ukládat velké množství dodatečných informací.

Při reverzibilní metodě je obecně problém s přiřazením do proměnných, naopak jejich inkrementace či dekrementace se dají snadno zpětně získat. V algoritmu protisměrného vyhledávání ve stromech jsou všechny hlavní proměnné, které jsou zároveň řídícími proměnnými v cyklech (i, j, *position*) přepsány novou hodnotnou. Přesto by bylo možné reverzibilní metodu realizovat s pomocnou strukturou, do které by byly hodnoty těchto proměnných uloženy v případě, že aktuální operaci není reverzibilní. Celkem jsou v algoritmu čtyři operace přiřazení, které nejsou reverzibilní a tři dekrementace, které lze snadno vrátit zpět. Tento poměr operací pro nás není výhodný a dále se již touto metodou nebudeme zabývat.

Kapitola 3

Implementace

3.1 Struktura projektu

Základem projektu je HTML soubor obsahující statický obsah, formulář pro zadávání vstupů a cestu ke vloženým souborům. Jedním z připojených souborů je CSS soubor s kaskádovými styly určující vzhled stránky a další předvolené styly použité při vizualizaci. Poslední a největší součástí této práce jsou soubory s kódem v jazyce JavaScript. Jedná se o knihovnu jQuery a další soubory vytvořené v rámci této bakalářské práce.

3.1.1 HTML

HTML je značkovací jazyk určený pro tvorbu webových stránek [8]. Hlavním souborem projektu je index.html. Jedná se o textový soubor obsahující HTML značky, kterými je definována struktura a obsah stránky. V hlavičce stránky je stále viditelný formulář pro zadání stromu, vyhledávaného vzorku a abecedy. K dispozici je rovněž několik ukázkových vstupů, které můžeme zvolit. Po odeslání formuláře se zobrazí další části stránky. Jedná se o zobrazení stromu, vzorku, tabulek BCS a SJT, výsledku vyhledávání a krokování algoritmu. Index.html také připojuje do projektu kaskádový styl style.css a několik JavaScriptových souborů.

3.1.2 CSS

Pomocí kaskádových stylů můžeme definovat vzhled stránky odděleně od obsahu. Předvolený styl můžeme aplikovat na více místech bez duplicitního definování parametrů. Nacházejí se zde styly pro celkový vzhled stránky, výpisu tabulek BCS a SJT, zobrazení stromu a vyhledávaného vzorku pomocí

překrývání poloprůhledných elementů, řádkování pseudokódu algoritmu a barvy určené pro zvýrazňování během krokování.

3.1.3 JavaScript

JavaScript je objektově orientovaný jazyk a jeho možnosti nezaostávají za ostatními vyspělými objektově orientovanými jazyky. Přesto se pojetí objektového přístupu liší od jiných jazyků a má své omezení. Hlavní příčinou je samotný typ jazyka. Jedná se o interpretovaný skriptovací jazyk, který je spuštěn a zpracován na straně klienta. Klient musí mít vždy k dispozici zdrojový kód aplikace, která je spouštěna. Z toho důvodu není JavaScript vhodný na ověřování přihlašovacích údajů, připojování k databázi a dalším operacím, které by bylo možné zneužít. Podobné úkony je nutné provádět pomocí jiných technik.

Další významnou vlastností plynoucí z typu jazyka je nemožnost zabránit úpravám kódu a zobrazení obsahu proměnných. Smysl částečně postrádají i privátní metody, přestože lze JavaScriptový kód navrhnout tak, aby privátní metody obsahoval. Využívání takových metod není v JavaScriptu doporučeno [9] a proto i naše práce bude implementovat metody standardní formou – pomocí JavaScript prototype. Javascript je také specifický tím, že třídu definujeme klíčovým slovem function, stejně jako funkci nebo metodu. Za konstruktor se považuje obsah funkce, kterou jsme třídu definovali a její metody je vhodné přidat pomocí prototype, speciálního objektu přidruženého ke každé funkci.

Nyní si na krátké ukázce kódu 1 ukážeme, jak vytvořit třídu Tree s konstruktorem s metodu getName(). Následně si vytvoříme novou instanci a vypíšeme výsledek vrácený zmíněnou metodou.

```
1 var Tree = function(name) {
2     this.name = name;
3 };
4 Tree.prototype.getName = function () {
5     return this.name;
6 };
7 var myTree = new Tree("oak");
8 alert(myTree.getName());
```

Ukázka 1: Základní práce s třídou v JavaScriptu

3.2 Získání a kontrola vstupů

Projekt obsahuje vloženou knihovnu jQuery ve verzi 1.11.1, což byla v době implementace poslední stabilní verze. Logika celé vizualizace je rozdělena do tří samostatných JavaScriptových souborů.

Prvním souborem, který jsme vytvořili je tree.js a obsahuje kontrolu a vizualizaci vstupních dat. Dále se jedná o tbpm.js, který implementuje algoritmus protisměrného vyhledávání a jeho vizualizaci. Poslední soubor s názvem main.js obsahuje kód pro ovládání uživatelského rozhraní a zajišťuje spuštění a ovládání vizualizace.

Vstupy jsou uživatelem zadány do formulářových polí s názvem subject. pattern a alphabet. Pomocí JavaScriptu vytvoříme handler, speciální funkci pro obsluhu událostí. Ta bude automaticky zavolána po odeslání formuláře. Náš handler (ukázka 2) převezme zpracování a zruší výchozí odeslání formuláře, což by zapříčinilo znovunačtení stránky a ztrátu dat. V obsluze události dále nalezneme získání a kontrolu vstupů, jejich vykreslení, zobrazení pomocných struktur a spuštění krokování algoritmu.

```
1 $(function () {
2     $("#form").submit(function (event) {
3         event.preventDefault();
4         // zpracování dat
5     });
6 });
```

Ukázka 2: Obsluha odeslání formuláře v JavaScriptu a jQuery

V jQuery se $ používá jako speciální znak pro rychlý přístup k funkcím knihovny. Na řádku 1 je definovaná funkce, která bude zavolána v okamžiku načtení stránky. Následně je nastaven handler na odeslání formuláře. V obsluze této události zrušíme výchozí zpracování formuláře (řádek 3) a provedeme inicializaci vizualizace.

Vstupy si nejprve načteme z formuláře a uložíme do proměnných, jak je uvedeno v ukázce 3. Následně vytvoříme novou instanci objektu *Tree* a zahájíme zpracování vstupů metodou parse(). Během zpracování probíhá zároveň kontrola vstupu i sestavení stromové stuktury, která bude následně vykreslena. Metoda parse() v případě chybného vstupu vrací false a handler přeruší další zpracování. V případě úspěchu se vykreslí stromová struktura.

```
1 // získáme data z formuláře
2 var fSubject = $("input[name=subject]").val();
3 var fAlphabet = $("input[name=alphabet]").val();
4 // vykreslíme vstupní strom
5 var subject = new Tree();
6 var retVal = subject.parse(fSubject, fAlphabet, false);
7 if (retVal) {
8     $("#subject").html(subject.draw());
9 } else {
10     return; // metoda parse() vrátila chybu, přerušíme zpracování
11 }
```

Ukázka 3: Získání vstupů z formuláře

3.3 Vykreslení vstupů ve stromovém formátu

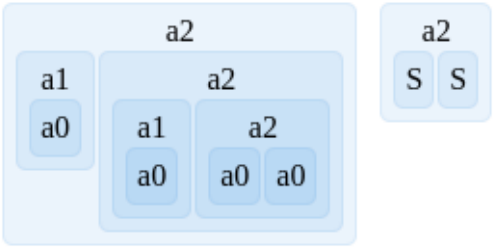
Jak již jsme zmínili, vykreslení vstupů proběhne po úspěšné kontrole správnosti vstupu. K oběma účelům slouží objekt Tree a jeho metody.

Konstruktor nám umožní vytvořit prázdný uzel stromu nebo mu ihned přiřadit hodnotu. Metoda parse() se volá po získání uživatelských vstupů a zahájí kontrolu a načtení vstupu. Nejprve rozdělí vstupy podle mezer, zkontroluje platnost symbolů v abecedě a začne převádět vstup na stromovou strukturu. Během převodu se kontroluje platnost symbolů podle abecedy a zároveň hlídáme syntaxi vstupu. Takto postupujeme, dokud nepřečteme poslední požadovaný symbol. Na závěr ověříme, zda-li nebyly zadány nějaké další znaky, které do vstupu nepatří. Za běhu se rekurzivně volá metoda parseNode(), která zpracovává podstromy. Dále metody readNode() a readBar(), které zpracovávají otevírací a uzavírací symbol v linearizovaném formátu stromu.

Metoda draw() má za úkol vykreslit zadaný vstupní strom a vzorek ve stromovém formátu. Rekurzivně volá metodu drawChildren(), která zajišťuje vykreslení podstromů. K vizualizaci používáme HTML elementy s poloprůhledným barevným pozadím. Elementům s třídou node je pomocí CSS nastavena barva s průhledností 90 %. Postupným zanořováním a současným částečným překrýváním se barva ztmavuje s narůstající hloubkou uzlu. Příklad vygenerovaného HTML kódu je uveden v ukázce 4. Výsledek zobrazuje obrázek 3.1.

```
<div id="subject">
    <div class="node">
        <div class="data">a1</div>
        <div class="children">
            <div class="node">
                <div class="data">a0</div>
            </div>
        </div>
    </div>
</div>
```

Ukázka 4: HTML struktura vykresleného stromu



Obrázek 3.1: Vstupní strom a vzorek

Vstupy máme zkontrolované a můžeme přejít na zpracování dalších částí vizualizace. Algoritmus požaduje vstupní data v linearizovaném formátu, proto získanou stromovou strukturu již dále nevyužijeme.

3.4 Vykreslení pomocných struktur a výsledku vyhledávání

Implementovaný kód pro vyhledávání a jeho vizualizaci se nachází v souboru tbpm.js. Dále zde najdeme definici objektu TBPM_Map (ukázka 5), neboť JavaScript zatím nemá nativní podporu datové struktury mapa. Naše implementace umožňuje navíc data vypsat do přehledné tabulky a označovat v ní konkrétní buňky tak, jak požadujeme při vizualizaci. Implementace v jazyku JavaScript je poměrně jednoduchá, neboť každý objekt je implementován pomocí asociativního pole. Kromě níže uvedených základních metod objekt dále obsahuje metodu pro vykreslení a zvýraznění dat.

```
1  var TBPM_Map = function (htmlId) {
2      this.values = new Object();
3      this.htmlId = htmlId;
4  };
5  TBPM_Map.prototype.has = function (key) {
6      return this.values.hasOwnProperty(key);
7  };
8  TBPM_Map.prototype.get = function (key) {
9      return this.values[key];
10 };
11 TBPM_Map.prototype.set = function (key, value) {
12     this.values[key] = value;
13 };
```

Ukázka 5: Implementace datové struktury mapa

Jako parametr je konstruktoru předána hodnota id HTML elementu, která bude použita při vykreslení. Ta jednoznačně identifikuje tabulku na stránce a později ji použijeme při zvýrazňování buněk. Při vytváření pomocných tabulek algoritmu použijeme hodnoty BCS, SJT.

Komplexnějším objektem je TBPM, který má dvě hlavní funkce. Implementaci protisměrného vyhledávání ve stromech a vizualizaci téhož algoritmu. Podívejme se, jak náš handler postupuje po ověření vstupních dat (ukázka 6).

```
1 var tbpm = new TBPM(fSubject, fPattern, fAlphabet);
2 tbpm.constructBCS();
3 $("#BCS_wrapper").html(tbpm.printBCS());
4 tbpm.constructSJT();
5 $("#SJT_wrapper").html(tbpm.printSJT());
6 $("#OUTPUT").html(tbpm.printSearchResults());
```

Ukázka 6: Vytvoření tabulek BCS, SJT a výsledku vyhledávání

Nejdříve je vytvořena instance objektu TBPM. Následně vypočítáme a zobrazíme obě zmíněné pomocné struktury jako HTML tabulky. S jejich využitím spustíme algoritmus protisměrného vyhledávání ve stromech

Tabulka 3.1: Tabulka BCS pro vzorovou abecedu $\mathcal{A}$

a2	a1	a0	|2	|1	|0
5	5	2	1	1	1

Tabulka 3.2: Tabulka SJT pro vzorový vstupní strom s_1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	6	5	2	1	18	11	10	7	6	17	14	11	16	13	10	5	0

a výsledek vypíšeme. Tabulky 3.1 a 3.2 jsou vytvořené pro vzorový vstupní strom s_1 a vyhledávaný vzorek v_1.

Výsledkem vyhledávání je v tomto případě 11, 6, 1. Nejprve algoritmus nalezl shodu začínající na 11. pozici, poté zbylé dvě. Nalezení shody vzorku se stromem na 6. pozici je zobrazeno na obrázku 3.2.

3.5 Krokování algoritmu

Vizualizace krokování algoritmu je nejdůležitější část a její spuštění je provedeno rovněž prostřednictvím handleru, čímž jeho úloha končí. Je zavolána metoda preparedStepAnimateInit() respektive realtimeStepAnimateInit() a jako parametr je předáno id HTML elementu, do kterého se bude vizualizace vykreslovat (ukázka 7).

```
1 tbpm.preparedStepAnimateInit("STEPANIMATE");
```

Ukázka 7: Inicialicace krokování

Implementovat budeme dva rozdílné způsoby, které následně porovnáme. Obě metody budou mít podobnou strukturu a jejich výsledkem bude stejný výstup. Inicializační metoda připraví neměnné části vizualizace, které není třeba překreslovat a zobrazí je. Další metody se postarají právě o vykreslení a správné zarovnání vyhledávaného vzorku, který se bude při krokování měnit.

Společný bude i způsob ovládání. Jedním způsobem je čtveřice odkazů, druhou možností jsou klávesové zkratky. Odkazy jsou při každém kroku upraveny. Mění se parametry určující následující krok a je možné je nezaktivnit, například na prvním nebo posledním kroku.

Využití klávesových zkratek je implementováno v souboru main.js pomocí druhého handleru (ukázka 8). Posun o jeden krok je namapován na klávesové šipky, velký krok je proveden při současném podržení klávesy

Shift. V souvislosti s touto vlastností bylo nutné vyřešit chování, pokud je uživatel kurzorem v textovém poli formuláře (tzv. focus). V tomto případě nebylo možné se posouvat v textu pomocí klávesových šipek. Při ošetření tohoto případu mohl nastat opačný problém v případě, že uživatel formulář odeslal klávesou Enter. Focus zůstal stále ve formuláři a krokování pomocí klávesnice nebylo možné. To se nám podařilo vyřešit příkazem v handleru formuláře, který zaměří focus na odesílací tlačítko. V obsluze události je třeba zrušit výchozí akci pro klávesy, které používáme pro krokování, jinak by mohlo docházet k posouvání stránky do stran.

```
1  $(document).keydown(function (e) {
2      // Zrušení obsluhy události, pokud je uživatel v textovém poli
3      if ($("input[type=text]").is(":focus")) {
4          return;
5      }
6      switch (e.which) {
7          case 37: // vlevo
8              var prev = $("a#STEPPREV");
9              if (prev.length) {
10                 var datastep = parseInt(prev.attr("data-step"));
11                 stepClick(datastep, -1);
12             }
13         break;
14         // Zkráceno o zpracování dalších kláves...
15     }
16     // Zabránění provedení výchozí akce – posouvání stránky
17     e.preventDefault();
18 });
```

Ukázka 8: Handler pro ovládání pomocí klávesnice

3.5.1 Přepočítávané krokování

První metodou, kterou jsme implementovali bylo přepočítávané krokování. Krokování algoritmu lze rozdělit na 3 části, které jsou implementovány pomocí metod. Rozdělení na části nám umožnilo oddělení samotného výpočtu od změny ovládacích prvků, výpisu proměnných a inicializační části.

První část implementovaná metodou realtimeStepAnimateInit() je inicializační. Jejím parametrem je id HTML elementu, do kterého bude vizualizace vykreslena. Dále jsou zde definovány třídní proměnné, které budou

využity v dalších částech krokování. Poslední úlohou inicializace je vytvoření statické části vizualizace krokování. Jedná se o čísla pozic symbolů ve vstupním stromě, a výpis vstupního stromu. Pro vyhledávaný vzorek je pouze vytvořeno místo, zobrazen bude až později.

Další metodou, kterou jsme implementovali je realtimeStepAnimatePrinter(). Tato metoda je zavolána vždy, když uživatel požaduje vykreslit nový krok. Nejprve provedeme nastavení výchozích parametrů, pokud nebyly uvedeny. Jedná se o číslo kroku a informaci, zda-li jde o velký krok. Následně je spuštěn výpočet a po jeho dokončení provedeme aktualizaci údajů na stránce. Nejprve vykreslíme správně zarovnaný vyhledávaný vzorek. Následně upravíme odkazy pro ovládání krokování. Pokud jsou odkazy aktivní, mají uvedený parametr data-step a událost onclick pro přechod na další krok. Parametr data-step slouží jako informace pro ovládání klávesovými zkratkami. Poslední operací je vypsání aktuálních hodnot proměnných.

Hlavní část celého krokování je implementována v metodě realtimeStepAnimateWorker(). Tato metoda je velmi podobná samotnému algoritmu protisměrného vyhledávání ve stromech. Obsahuje stejný kód doplněný o kontrolní mechanizmy nutné pro krokování. Po každém řádku z pseudokódu provedeme inkrementaci čítače aktuálního kroku a ověříme, zda-li se jedná o krok, který byl požadován. Pokud ano, vypočteme zobrazení vizualizace a provedeme označení řádku a dalších případných hodnot. Jak již bylo řečeno, kód je podobný vyhledávacímu algoritmu. Pro možnost vizualizace bylo nutné upravit podmínky a ošetřit správnou vizualizaci cyklů. Při vizualizaci je požadováno, aby byla podmínka označena i v případě, že nebyla splněna a to jinou barvou. Toho se nám podařilo dosáhnout přidáním příkazu else za každou podmínku. U podmínky else if musela být část podmínky vnořena do bloku else, který zároveň vizualizoval neúspěch první části podmínky (ukázka 9).

3.5.2 Předpřipravené krokování

Metoda předpřipraveného krokování pracuje na principu ukládání všech vypočtených dat do paměti. Vypočtení probíhá při prvním spuštění a následující žádosti o zobrazení dalšího kroku jsou vyřízeny okamžitě. Tuto metodu jsme rozdělili rovněž na tři části s podobnou funkcionalitou.

Metoda inicializační pojmenovaná jako preparedStepAnimateInit() opět přijímá argumentem id HTML elementu, do kterého budeme vizualizaci vykreslovat. Předpřipravené krokování nepotřebuje používat třídní proměnné

```
1  // Původní struktura podmínek
2  if (condition1) {
3  } else if (condition2) {
4  } else {
5  }

6  // Nově vytvořená struktura
7  if (condition1) {
8  } else {
9      // Řádek, na kterém je if(condition1) jako neprovedený
10     if (condition2) {
11         // Řádek, na kterém je else if (condition2) jako provedený
12     } else {
13         // Řádek, na kterém je else if (condition2) jako neprovedený
14         // Řádek, na kterém je else, jako provedený
15     }
16 }
```

Ukázka 9: Změna struktury podmínek

v takovém rozsahu, neboť všechny důležité informace budou uloženy do pomocné struktury. V inicializační části zavoláme metodu pro připravení vizualizovaných dat, vykreslíme neměnné části vizualizace a vykreslíme první krok.

Vykreslení požadovaného kroku zajišťuje metoda preparedStepAnimatePrinter(). Jako argumenty lze zadat aktuálně zobrazený krok, typ kroku a směr, ve kterém požadujeme další krok. Tento přístup byl zvolen v rámci zjednodušení počítání obou typů kroků a následného předávání údajů při generování ovládacích prvků. Nyní je předáváno pouze číslo aktuálního kroku a požadovaný krok je určen podle příznaků před jeho vykreslením právě v rámci této metody. Pokud se nejedná o velký krok, jsou požadované údaje přímo dostupné v poli pod indexem požadovaného kroku. Při požadavku na velký krok je nutné určit jeho pozici. Zvolili jsme strategii, která nepotřebuje ukládání dalších dat – odkazů na nejbližší velký krok, ale pomocí cyklu iterativně najde nejbližší velký krok. Vzhledem k tomu, že velké kroky jsou od sebe vzdáleny nejvýše pět standardních kroků, nebude při tomto postupu žádné znatelné zpomalení. Po jeho nalezení opět provedeme vykreslení a výpis dat.

Další optimalizací v rámci předpřipraveného algoritmu je ukládání zarovnaného vzorku pouze v krocích, kdy nastala změna. Tento přístup snížení paměťových nároků opět vyžaduje cyklus, který nalezne nejbližší krok

s potřebným výstupem. Tento výstup mají řádky, na kterých jsou zároveň zaznamenávány velké kroky. Po vykreslení dat potřebujeme aktualizovat ovládací prvky a vypsat stavy proměnných pomocí stejného principu, jako v předchozí metodě.

Poslední metoda preparedStepAnimateWorker() zajišťuje prvotní výpočet a připravení dat pro vizualizaci. Tato metoda je během vizualizace spuštěna pouze jednou. Po vykonání každého řádku pseudokódu je do pole vložena nová struktura s požadovanými údaji. Především číslo kroku a řádku, hodnoty proměnných a v některých případech i HTML kód zarovnaného vzorku. Na konci je uloženo do třídních proměnných i číslo posledního kroku. Tato hodnota poslouží pro ukazatel aktuálního kroku a celkového počtu kroků. Uloženy jsou i čísla prvního a posledního velkého kroku, které usnadní generování odkazů pro ovládání vizualizace.

Data pro vizualizaci jsou ukládána do speciální struktury, která byla za tímto účelem vytvořena. Všimněme si, že v JavaScriptu se definice struktury provádí stejně, jako jiného objektu a že definice může být zároveň i konstruktorem.

```
1  var StepAnimateStucture = function (step, bigStep, SJT, BCS,
   line, lineColor, pattern, i, j, position, output) {
2      this.step = step;
3      this.bigStep = bigStep;
4      this.SJT = SJT;
5      this.BCS = BCS;
6      this.line = line;
7      this.lineColor = lineColor;
8      this.pattern = pattern;
9      this.i = i;
10     this.j = j;
11     this.position = position;
12     this.output = output;
13 };
```

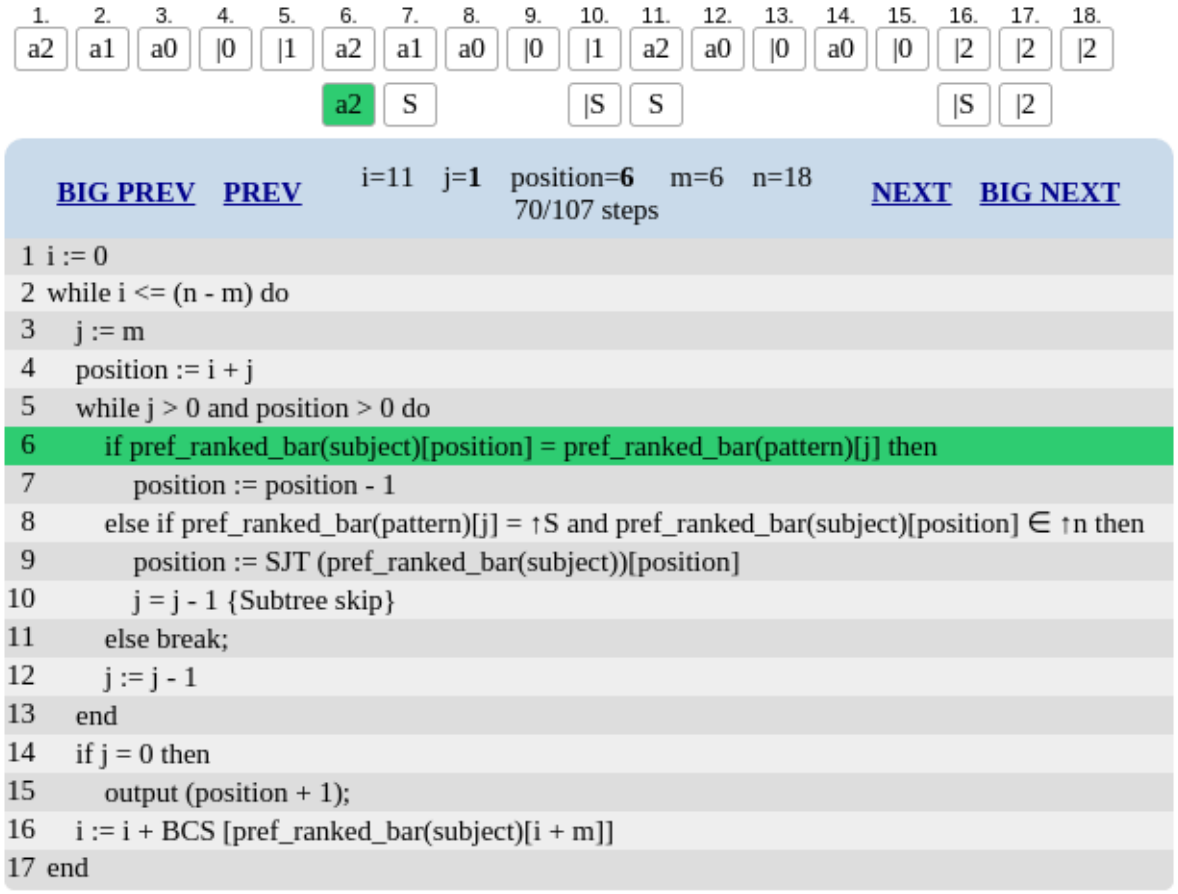
Ukázka 10: Definice struktury pro ukládání vizualizovaných dat

Tato metoda vizualizace krokování byla během implementace zvolena za hlavní. Za výhody považujeme přehlednější kód výpočetní metody, možnost zobrazení ukazatele kroků bez nutnosti dalších výpočtů a nižší časovou náročnost, kterou ověříme během testování. Pro tuto metodu byl vylepšen výpis hodnot proměnných tak, aby změněné hodnoty byly vyznačeny tučným písmem. Této funkcionality bylo dosaženo čistě kontrolou aktuálně zobrazených údajů s novými údaji pomocí JavaScriptu.

3.5.3 Společné metody

Přepočítávané krokování i předpřipravená varianta vizualizace používá další metody k dosažení výsledku. Tyto metody jsou společné pro obě varianty. Získání HTML kódu pro vykreslení vyhledávaného vzorku umožňují dvě metody. Pro vytvoření nezarovnaného a zarovnaného vzorku používáme metodu animatePatternUnaligned() respektive animatePatternAligned(). Za zarovnaný vzorek považujeme takový, který bude ve vstupním stromě nalezen a je ho tedy možné přesně zarovnat pod strom zobrazený v linearizovaném formátu. Pokud dopředu víme, že vzorek nemůže být nalezen, zavoláme metodu pro vykreslení nezarovnaného vzorku přímo. V ostatních případech voláme metodu druhou, která se pokusí vzorek zarovnat.

Poslední společnou metodou je stepAnimateHighlight(). Ta zajišťuje zvýraznění aktuální řádky pseudokódu definovanou barvou. Také je zrušeno případné zvýraznění buněk v tabulkách pomocných struktur a výsledku vyhledávání. Tato metoda se volá při každém požadavku na vizualizaci kroku.



Obrázek 3.2: Nalezená shoda při krokování algoritmu

Kapitola 4

Testování

V rámci této práce jsme implementovali vizualizaci algoritmu pomocí dvou odlišných metod krokování. V této kapitole se budeme věnovat testování těchto metod pomocí velkých vstupních dat. Pro testování jsme vygenerovali stromy o různých délkách dvou typů. První testovací data obsagují n uzlů arity 1 a jeden list. Tato speciální data zajištují ve spojení se správným vzorkem n shod a jsou náročnější na vizualizaci. Druhé testování proběhlo na náhodně vygenerovaných stromech s maximální aritou 7. Během testování jsme se zaměřili na využití paměti a časové nároky jednotlivých částí vizualizace.

Testování proběhlo na počítači s operačním systémem Linux, procesorem Intel Core i5-3317U se základní frekvencí 1,7 GHz a pamětí RAM 8 GB. Naměřené hodnoty byly získány pomocí vývojářských nástrojů prohlížeče Chromium 41. Využito bylo profilování JavaScriptu k získání časových údajů a dále volba Heap Snapshot pro získání přehledu o využití paměti jednotlivými objekty.

4.1 Testování dat s velkým počtem shod

Velkého množství nalezených shod jsme docílili návrhem testovacíh dat. Pro vstupní strom o délce 22 znaků a hloubce stromu 10 jsme zvolili vzorek, který nalezne shodu na každé pozici kromě jednoho listu.

Strom: $a1\ a1\ a1\ a1\ a1\ a1\ a1\ a1\ a1\ a1\ a0\ |0\ |1\ |1\ |1\ |1\ |1\ |1\ |1\ |1\ |1\ |1$.
Vyhledávaný vzorek: $a1\ S\ |S\ |1$.
Abeceda: $a1\ a0\ |1\ |0$.

Tabulka 4.1: Společné hodnoty pro data s velkým počtem shod

Hloubka	Délka	Nálezů	Kroků	Strom [B]	BCS [B]	SJT [B]
10	22	10	253	736	864	384
31	64	31	785	1 496	864	696
100	202	100	2 533	4 168	864	2 280
310	622	310	7 853	13 784	864	6 120
1 000	2 002	1 000	25 333	37 624	864	18 544
3 100	6 202	3 100	78 533	253 672	864	65 576

V rámci testování jsme se zaměřili na několik sledovaných parametrů. Hloubka stromu je definována jako délka cesty od kořene do nejvzdálenějšího listu. V tomto případě existuje pouze jediný list a hloubku označíme d. Délka vstupu značí celkový počet symbolů vstupního stromu. V tomto případě $2 * d + 2$. Mezi společné hodnoty patří počet nálezů a počet kroků. Tyto hodnoty jsou určeny algoritmem protisměrného vyhledávání ve stromech. Obě varianty mají také stejnou velikost vstupního stromu a tabulek BCS a SJT.

Dále sledujeme celkový čas, který je potřeba k obsloužení události odeslání formuláře. V této události jsou načteny a zpracovány vstupy, vykresleny pomocné struktury, připraveno krokování a zároveň je vygenerován první krok.

Nejdůležitější jsou však časy potřebné pro zobrazení prvního a posledního kroku. Tyto údaje nám dávají informaci právě o časové náročnosti jednotlivých metod. U zobrazení kroků sledumeme jak celkový čas, tak čas potřebný pro získání dat a pro jejich vykreslení. Hodnoty jsou uvedeny ve formátyu: *celkem (získání/zobrazení)*. U předpřipravené metody nás bude dále zajímat čas přípravy dat a velikost pomocné struktury.

Z naměřených hodnot uvedených v tabulkách 4.2 a 4.3 je patrný narůstající rozdíl mezi způsoby implementace krokování. Předpřipravená varianta potřebuje k inicializaci znatelně více času. Zobrazení prvního kroku je u obou variant stejně náročné a nepozorujeme zde významné rozdíly. S požadavkem na zobrazení dalších kroků rychle roste časová náročnost přepočítávané metody. Předpokládaná efektivita předpřipraveného krokování se potvrdila, zobrazení posledního kroku je výrazně rychlejší než u první varianty. Čas přesto není konstantní a se zvětšujícími se vstupy mírně roste. Z detailnějších údajů uvedených v závorkách lze vidět, že roste především čas potřebný ke změně HTML kódu a vykreslení nového zarovnaného vzorku.

Tabulka 4.2: Přepočítávané krokování pro data s velkým počtem shod

Hloubka	Struktura [kB]	Celkem [ms]	Příprava [ms]	1. krok [ms]	Poslední krok [ms]
10	0	20	0	5 (1/4)	8 (2/6)
31	0	25	0	8 (2/6)	8 (2/6)
100	0	26	0	6 (2/4)	16 (4/12)
310	0	54	0	6 (2/4)	37 (12/25)
1 000	0	140	0	8 (4/4)	172 (101/71)
3 100	0	1180	0	17 (11/6)	1 280 (1160/120)

Tabulka 4.3: Předpřipravené krokování pro data s velkým počtem shod

Hloubka	Struktura [kB]	Celkem [ms]	Příprava [ms]	1. krok [ms]	Poslední krok [ms]
10	107	14	1	5 (1/4)	6 (1/5)
31	666	21	4	6 (2/4)	8 (2/6)
100	5 655	50	6	6 (2/4)	12 (2/10)
310	50 591	106	41	6 (2/4)	27 (3/24)
1 000	503 359	692	242	11 (4/7)	77 (6/71)
3 100	-	-	-	-	-

Tabulka 4.4: Společné hodnoty pro náhodná data

Délka	Nálezů	Kroků	Strom [B]	BCS [B]	SJT [B]
22	1	101	896	3 160	512
64	3	347	1 880	3 160	912
202	24	1 187	5 960	3 160	2 256
622	39	3 247	18 808	3 160	5 480
2 002	49	8 877	54 648	3 160	18 240
6 202	303	27 729	253 608	3 160	62 760

4.2 Testování náhodných dat

Jako druhá testovací data jsme zvolili náhodně vygenerované stromy, které více odpovídají skutečnému použití. V tomto případě již neuvádíme hloubku stromu, data byla generována na stejnou délku jako v prvním případě.

Strom: $a2\ a3\ a0\ |0\ a1\ a1\ a0\ |0\ |1\ |1\ a0\ |0\ |3\ a3\ a0\ |0\ a0\ |0\ a0\ |0\ |3\ |2$.
Vyhledávaný vzorek: $a2\ S\ |S\ S\ |S\ |2$.
Abeceda: $a7\ a6\ a5\ a4\ a3\ a2\ a1\ a0\ |7\ |6\ |5\ |4\ |3\ |2\ |1\ |0$.

Náhodná testovací data obsahují řádově méně shod vzorku se stromem. Z toho důvodu algoritmus potřebuje při stejně dlouhých vstupních datech pouze přibližně třetinu kroků, jak je uvedeno v tabulce 4.4. Dle tabulek 4.5 a 4.6 je pamětová náročnost předpřipravené varianty poloviční oproti testovacím datům s velkým počtem shod. Klesla i časová náročnost, což je viditelné především u nejdelších testovacích dat.

4.3 Výsledky testování

Dle naměřených časových údajů je předpřipravená metoda výrazně efektivnější, především pokud potřebujeme zobrazit více kroků. Doba potřebná pro vykreslení kroků není konstantní, neboť stále roste čas potřebný ke změně HTML kódu na stránce. Také čas označení buňky v tabulce SJT roste s velikostí vstupního stromu.

Výhoda předpřipraveného kokování je založena na potřebě pomocné struktury pro ukládání dat. V tabulkce 4.3 a 4.6 je uvedena velikost této pomocné struktury. Při měření jsme zjistili, že velikost struktury nabývá vysokých hodnot a pro vstupní data o délce 6 202 symbolů nebylo možné hodnoty naměřit.

Tabulka 4.5: Přepočítávané krokování pro náhodná data

Délka	Struktura [kB]	Celkem [ms]	Příprava [ms]	1. krok [ms]	Poslední krok [ms]
22	0	11	0	4 (1/3)	6 (2/4)
64	0	23	0	3 (1/2)	6 (1/5)
202	0	32	0	4 (2/2)	11 (2/9)
622	0	52	0	4 (2/2)	26 (3/23)
2 002	0	78	0	7 (3/4)	103 (26/77)
6 202	0	453	0	11 (6/5)	477 (220/257)

Tabulka 4.6: Předpřipravené krokování pro náhodná data

Délka	Struktura [kB]	Celkem [ms]	Příprava [ms]	1. krok [ms]	Poslední krok [ms]
22	61	26	1	5 (2/4)	5 (2/4)
64	367	24	1	7 (3/4)	9 (3/6)
202	2 771	37	10	8 (3/5)	12 (3/9)
622	22 565	71	16	6 (2/4)	18 (3/15)
2 002	203 146	170	84	11 (4/7)	66 (3/63)
6 202	-	-	-	-	-

Ve sloupci příprava je uvedený čas, který potřebuje vizualizace k sestavení pomocné struktury. Tento čas neroste lineárně, jak bychom mohli očekávat podle algoritmu protisměrného vyhledávání ve stromech. Vyšší časová náročnost je dána potřebou generovat vzorky zarovanné pod zobrazený vstupní strom.

Jak již bylo řečno v kapitole 1, pro vizualizaci jsou vhodná data o délce stromu 50 symbolů. U naměřených hodnot pro délky 22 a 64 lze pozorovat velmi podobné časové nároky obou variant do velikosti desítek milisekund. Pamětové nároky předpřipravené varianty nepřekročí 1 MB. Pro vizualizaci je takové řešení dostatečné, přesto z testování vyplynulo několik poznatků, které by mohly být použity pro snížení nároků předpřipravené varianty pro velká vstupní data.

Jednou z možností jak snížit pamětové nároky a čas nutný k přípravě hodnot je změna ukládaných informací do struktury. Namísto již vygenerovaného vzorku je možné ukládat další celočíselné proměnné, ze kterých by byl vzorek vygenerován až v případě zobrazení daného kroku. Tato změna by vedla ke snížení pamětových nároků a snížení času nutného na přípravu. Naopak čas potřebný k zobrazení libovolného kroku by vzrostl o vygenerování zarovnaného vzorku.

Druhou možností je příprava pouze několika desítek prvních kroků. Při požadavku na další krok, který není v paměti by byly připraveny další kroky a ty nepotřebné by mohly být smazány. Nevýhodou takové varianty je nutnost znovu generovat kroky, které byly před chvílí smazány v případě krokování zpět.

Během testování jsme pro rychlý přechod na první nebo poslední krok v rámci velkých testovací dat použili následující příkaz, který lze zadat do adresního řádku prohlížeče. Za m dosadíme celkový počet kroků.

```
1 javascript:stepClick(0, 1) // První krok
2 javascript:stepClick(m − 1, 1) // Poslední krok
```

Ukázka 11: Rychlý přechod na požadovaný krok

Závěr

V rámci této práce jsme popsali způsob, jakým je vyhledáváno ve stromových strukturách. Analyzovali jsme požadavky vizualizace a zvolili jsme technologii, v níž byla vizualizace implementována. Navrhli jsme strukturu vizualizace a její jednotlivé části. Zaměřili jsme se na způsoby, jakými mohou být data zobrazena.

Nejvější důraz jsme kladli na porovnání metod krokování a výběr té správné pro vizualizaci. Implementovali jsme dvě odlišné metody, z nichž jsme jednu zvolili jako upřednostňovanou. Jedná se o variantu předpřipraveného krokování, která byla dále rozšířena o počítadlo kroků a zvýrazňování změněných hodnot proměnných. Každou část implementace jsme důkladně popsali a uvedli důvody, proč jsme tak postupovali.

V rámci testování jsme připravili dva druhy testovacích dat, na kterých jsme ověřili časové a pamětové nároky obou variant. Naměřené hodnoty jsme porovnali a diskutovali možné příčiny vysokých hodnot a způsoby optimalizace vedoucí k jejich snížení.

Požadavky definované v zadání práce byly splněny a vizualizace je plně funkční.

Literatura

[1] Trávníček, J.; Janoušek, J.; Melichar, B.; aj.: Backward Linearised Tree Pattern Matching. *LATA 2015*, 2015: s. 599–610.

[2] Hoffmann, C. M.; O'Donnell, M. J.: Pattern Matching in Trees. *Journal of the ACM*, ročník 29, č. 1, January 1982: s. 68–95.

[3] Horspool, R. N.: Practical fast searching in strings. *Software Practice and Experience*, ročník 10, č. 6, 1980: s. 501–506.

[4] W3Schools: *HTML5 Canvas [online]*. [cit. 2015-04-21]. Dostupné z: `http://www.w3schools.com/html/html5_canvas.asp`

[5] The jQuery Foundation: *jQuery JavaScript library [online]*. [cit. 2015-02-10]. Dostupné z: `https://jquery.com`

[6] *Vis.js visualization library [online]*. [cit. 2015-03-15]. Dostupné z: `http://visjs.org`

[7] *Dracula Graph Library [online]*. [cit. 2015-03-15]. Dostupné z: `http://www.graphdracula.net`

[8] Mozilla Foundation: *HTML [online]*. [cit. 2015-04-05]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTML`

[9] Mozilla Foundation: *Introduction to Object-Oriented JavaScript [online]*. [cit. 2015-03-25]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript`

Příloha A

Seznam použitých zkratek

BCS Bad character shift

CSS Cascading Style Sheets

HTML HyperText Markup Language

JS JavaScript

SJT Subtree jump table

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `test-data` ..... data použitá při testování
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF