Diplomová práce

Algoritmy řádkového a stránkového zlomu v počítačové sazbě

Jan Pazdziora

Fakulta informatiky
Masarykovy univerzity v Brně

Diplomová práce

Algoritmy řádkového a stránkového zlomu v počítačové sazbě

Jan Pazdziora

Vedoucí diplomové práce
RNDr. Petr Sojka

Brno, duben 1997

Zadání

▪ Seznamte se s problematikou algoritmů řádkového a stránkového zlomu z dostupné literatury. ▪ Proveďte jejich srovnání a analýzu časové a paměťové náročnosti. ▪ Charakterizujte nevýhody a omezení stávajících implementací a navrhněte možná zlepšení. ▪ Navrhněte a implementujte prototypový systém, který demonstruje navržená rozšíření. ▪ Diskutujte alternativní postupy a chování tohoto systému pro různé typy vstupních textů.

Děkuji především vedoucímu své diplomové práce, RNDr. Petru Sojkovi, za všestrannou pomoc, připomínky a odkazy na zdroje informací, kterými mě zásoboval, a prof. Donaldu Knuthovi za typografický systém TEX a knihy, které o počítačové sazbě napsal.

Děkuji také svým blízkým, obzvlášť Petře, za shovívavost a trpělivost, kterou se mnou v době tvorby této práce měli. To se týká samozřejmě i kolegů z CVT.

Tato práce byla vysázena programem TEX ve formátu plain, s použitím DC-fontů a balíku maker pro práci s PostScriptovými příkazy PSTricks, na strojích s operačním systémem Un*x.

Obsah

1. Úvod

Od svého vzniku v roce 1978 a zvláště po zásadních a konečných změnách v roce 1982 se sázecí systém TEX stal pevnou součástí akademických institucí při sazbě textů vysoké kvality. Za tuto svou pozici vděčí vlastnostem, které od svého autora, profesora Donalda Knutha ze Stanford University, dostal do vínku: kvalitní algoritmy pro lámání textu, propracované možnosti při sazbě matematických vzorců, mocný makrojazyk a systém formátů, které dovolují téměř nekonečné rozšiřování, a také otevřenost systému, který byl zveřejněn včetně všech zdrojových textů a podrobné dokumentace.

TEX se stal výkonným a stabilním nástrojem, vzniklo množství maker pro nejrůznější úkoly, ale kupodivu málo lidí využilo výzvy prof. Knutha po dalším bádání v oblasti algoritmů, které tvoří jádro programu. Přesto se ale občas setkáváme s případy, kdy výsledný dokument nesplňuje přesně naše požadavky a občas musíme vynaložit relativně hodně „ručního“ úsilí, abychom dosáhli vyhovujícího výsledku.

Typickým problémem je odstraňování parchantů z dokumentu. Můžeme nastavit hodnotu `\widowpenalty` a `\clubpenalty` a tím říci, že má algoritmus hledat takovou výslednou podobu dokumentu, ve které se vdovy s sirotci nebudou vyskytovat. Tato metoda ale bude úspěšná pouze v textu, kde je hodně variablilního místa mezi řádky, jako jsou například matematické přednášky s mnoha rovnicemi a proměnným místem mezi odstavci, definicemi či důkazy. Právě pro takový typ dokumentů byl TEX původně vytvořen a zde se ukáže jeho síla. Hladký text s pevným řádkovým rejstříkem, například beletrie, dává algoritmu mnohem méně možností alternativních řešení při odstraňování nevhodného zlomu stránky. Jedním z nich je zkrácení strany o řádek, případně musí uživatel programu pomoci ručním vložením `\looseness` do předchozích odstavců.

Jiný problém představuje sazba do zrcadla s nestejnou šířkou. Délky řádků můžeme totiž v TEXu specifikovat pouze relativně k právě zpracovávanému odstavci (pomocí maker `\parshape` nebo `\hangindent` a `\hangafter`), větší změny již ale uživatel musí provést ručně sám.

Tato práce se zabývá algoritmy počítačové sazby, s důrazem na algoritmy použité v řádkovém a stránkovém zlomu. Uvedeme několik možných přístupů a hlavní pozornost zaměříme na algoritmy použité v sázecím systému TEX. Jejich kvalita a existence zdrojových textů z nich dělají vhodné kandidáty pro další výzkum. Označíme jak přednosti TEXovských postupů, tak jejich úzká místa, a nastíníme možnosti, jak tato odstranit bez velkých dopadů na efektivitu výpočtů a při zachování vysoké kvality výsledných dokumentů.

Popíšeme implementaci těchto zlepšených algoritmů a chování prototypového systému při sazbě reálných dokumentů. Zamyslíme se nad postupy, které by v prototypovém systému šly dále zlepšit.

V kapitole 2 popíšeme typický pracovní cyklus sázecího systému, od vstupu dat přes vnitřní zpracování po výstup. Ve třetí až páté kapitole se zaměříme na používané algoritmy řádkového zlomu, v textovém a grafickém režimu. Při tom se budeme opírat o model box-glue-penalty, na němž je založen i sázecí systém TEX. Kapitola 6 pojednává o možných přístupech ke zlomu stránkovému.

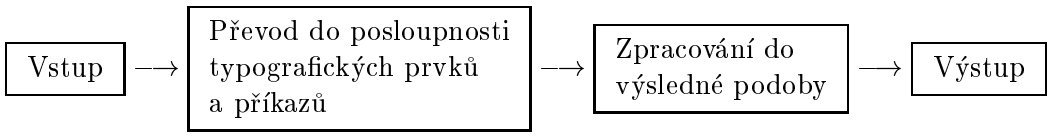
V další kapitole se podrobněji zaměříme na postupy použité v TEXu — jednak na možnosti, které jsou přístupné uživateli prostřednictvím maker, jednak na vnitřní implementaci tohoto systému. Zde se také zastavíme u omezení, která z TEXovských přístupů plynou.

Návrh, jak některé z nich odstranit, přináší kapitola osmá. Základní úpravou je změna řízení toku dat v sázecím systému a rozšíření modelu o nový datový typ. V kapitole 9 pak popíšeme, jak byla tato rozšíření využita při tvorbě prototypového systému.

Pro výpisy algoritmů byl zvolen formát, který kombinuje slovní popis prováděných činností s přehlednou strukturou zápisu jazyka C či Perl. Cílem bylo dosáhnout co nejvyšší vyjadřovací schopnosti. Proto jsou například proměnné v algoritmech označeny písmeny pouze tam, kde je to užitečné, jinde je použito čitelnější slovní vyjádření.

2. Zpracování dokumentu systémem počítačové sazby

Úkolem systému pro počítačovou sazbu je převést uživatelův vstup do popisu rozmístění obsahu dokumentu na dvourozměrné ploše stránek. Podle tohoto popisu je pak možné zobrazit dokument na obrazovce či vytisknout na papír. Schematicky můžeme tento postup zapsat následovně:



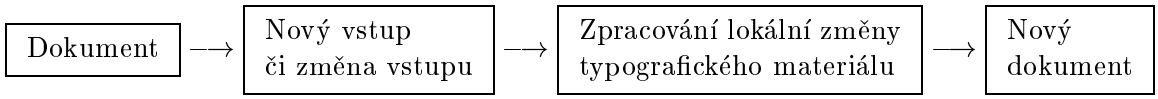
Vstupem zde může být například posloupnost stisků kláves nebo data čtená ze souboru. Vstup je pak převeden do více či méně složité vnitřní reprezentace, která popisuje tvar dokumentu. Tato reprezentace se použije pro výpočet umístění jednotlivých prvků dokumentu na stránkách. Takto popsaný dokument je potom předán zpět uživateli. Buď se stránka zobrazí přímo na monitoru počítače, nebo jsou data dále zpracována tiskárnou či jinými softwarovými systémy.

2.1. Vstup dat v sázecích systémech

Počítačové sázecí systémy se z pohledu uživatele liší nejvíce právě způsobem vstupu dat. Z tohoto hlediska je rozdělujeme na interaktivní a dávkové.

Interaktivní systémy se také často označují jako WYSIWYG — co_vidíš, to_dostaneš. Zde píšeme dokument a ten je přímo při psaní formátován a zobrazován, změny se promítají rovnou do výsledného tvaru dokumentu. Kurzor označuje místo na obrazovce, kam vkládáme nový text a kde provádíme změny, stisk klávesy vyvolá okamžitou akci, ať již je to přidání písmene či mezery k textu nebo například změna velikosti písma. Důležitým pomocníkem při práci s tímto typem systémů je také myš, kterou můžeme přímo do dvourozměrného obrazu stránky zasáhnout, vložit obrázek, zvětšit mezeru mezi řádky, přesunout část textu, a podobně.

U těchto systémů velmi záleží na okamžité odezvě sázecího systému, aby uživatel viděl výsledek své práce co nejdříve na obrazovce. Pokud bychom po každé akci uživatele zopakovali pro celý dokument výše naznačený postup od vstupu po výstup, nebyl by systém interaktivní, protože bychom se v reálném čase nedočkali odpovědi. Proto se tyto systémy zaměřují spíše na způsob, jak změnit výslednou podobu dokumentu, pokud se změní vstup (například uživatel napíše další písmeno slova):



Stisk klávesy nebo akce provedená myší se zde přímo transformuje na typografický příkaz, například: připoj písmeno „p“ ke třetímu odstavci, změň atribut typ písma daného slova na kurzívu, začni nový odstavec. Dochází zde tedy ke změnám v interní reprezentaci dokumentu a sázecí systém pak provádí jenom ty výpočtu, které jsou nutné jako reakce na nové změny vyvolané uživatelem.

Lokálnost výpočtů je důležitým rysem interaktivních systémů. Uživatele při psaní zajímá pouze to, co vidí na obrazovce, což může být jen velmi malá část celého dokumentu. Sázecí systém může tohoto rysu využít a také pracovat pouze s tou částí dokumentu, která je právě zobrazená. Výpočty změn pozic písmen a řádků před a za touto oblastí nejsou po většinu času nutné.

Pro **dávkové zpracování** (*batch processing*) je typické načítání vstupu ze souboru, případně z proudu dat, který je výstupem jiného programu. Zde nejprve vytvoříme text dokumentu spolu s pokyny, jak má být typograficky upraven [Strong 88], [Knuth 84]. Například pracujeme-li se systémem TEX, napíšeme text v obyčejném (ASCII) editoru a použijeme v něm příkazy jako `\par` či `\vskip`. Sázecí systém potom čte náš vstup, provádí expanzi příkazů (maker) a převádí text do vnitřní reprezentace dokumentu. Tuto vnitřní reprezentaci následně použije pro zformátování dokumentu.

Podobně pracují i HTML prohlížeče (Netscape, Mosaic), které čtou text obsahující značky `<H1>`, `<A HREF=...>` nebo `<PRE>` a formátují podle nich dokument do výsledné podoby, kterou posléze zobrazí uživateli.

Dávkové systémy jsou typické tím, že v dokumentu nedochází v průběhu zpracování ke změnám. Data načítaná ze vstupu reprezentují stav dokumentu a pokud uživatel změní text na vstupu, musí se znovu spustit celý proces zpracování nad novými daty.

Oba typy sázecích systémů mají své výhody a nevýhody. Interaktivní systémy dávají uživateli okamžitý obraz toho, jak dokument vypadá. Jelikož zde ale požadujeme odezvu v reálném čase, platíme za to obvykle nižší typografickou kvalitou výsledného dokumentu či jiným kompromisem. Dávkové systémy, kde není doba zpracování hlavním kritériem, obvykle zpracují dokument lépe. Algoritmy použité zde mohou být složitější a tudíž kvalitnější. Čeho se zde naopak vzdáváme je flexibilita okamžité odpovědi.

Proto můžeme do našich úvah zahrnout i hybridní systémy kombinující oba přístupy. Například dávkový systém, který bude mít uživatelsky přívětivé rozhraní pro zadávání tvaru a jiných parametrů textu myší. Nebo interaktivní program, který v normálním režimu nabídne pouze náhled dokumentu v nižší typografické kvalitě a pouze na požádání či při delší odmlce uživatele provede kompletní přeformátování textu se všemi výhodami dávkového zpracování. Většina běžně používaných interaktivních systémů vůbec dávkový způsob práce nepodporuje, uživatel před klávesnicí počítače je jejich nezbytnou součástí. To se ale při tisku stovek stránek hromadných dopisů, pozvánek nebo přehledů může stát limitujícím faktorem, a pak se projeví síla takového propojení.

2.2. Vnitřní reprezentace dokumentu

Tvar, do kterého sázecí systém převádí uživatelův vstup, je velmi závislý na specifickém účelu sázecího systému a na použitých algoritmech. Jinou vnitřní reprezentaci bude používat program `fold`, který pouze zalomí dlouhé řádky, jinak bude se vstupem zacházet editor `T602`, který dokáže zarovnat okraje textu a rozdělit slova na vhodných místech, ale používá neproporcionální písma, a úplně jiný přístup použije sázecí systém jako je TEX, kde musíme uvažovat kromě základní sazby nejrůznějšími typy a velikostmi písma i sazbu tabulek, matematických výrazů nebo například chemických vzorců.

V jednodušších systémech, které předpokládají výstup na terminál v textovém režimu, obvykle vystačíme s přístupem „co písmeno, to jeden bajt“. V textovém režimu totiž dokument formátujeme do matice, kde každý znak bude umístěn do vyhrazené pozice jednotné výšky a šířky. Hodnota znaku v daném kódování pak určuje písmeno, které bude na terminálu zobrazeno. I takovýto systém může být schopen splnit například požadavek na

zobrazení části textu tučně či inverzně. Takovýto atribut textu musí sázecí systém správně zaznamenat a posléze zpracovat.

Složitější systémy se více zaměřují na strukturu dokumentu a vlastní text je pak pouze jedním z atributů [Zabala 82]. Například kapitola textu má nadpis vysázený výrazným typem písma, před ním je uvedeno číslo kapitoly, za ním následují jednotlivé odstavce. Odstavec textu je složen ze slov a mezislovních mezer, matematický výraz jako $f = \sum_{i=0}^{n} f_i(x)$ můžeme rozložit na levou a pravou stranu a rovnítko, pravou stranu dále na sumu, její meze a argument, atd. Jiným příkladem jsou systémy pro sazbu tabulek, kde systém musí vzít do úvahy všechny řádky ještě před tím, než udělá definitivní rozhodnutí o šířce sloupce.

Pro každý z prvků dokumentu může být kromě zobrazovaného textu podstatná celá řada specifických atributů, jako jsou rozměry, použitý font, barva, a samozřejmě také vztah vůči okolním prvkům dokumentu. Atributy se mohou navzájem ovlivňovat, například šířka slova vysázeného tučným řezem písma je obvykle větší než stejný text v polotučné podobě stejného fontu. Tyto informace a vztahy mezi nimi je třeba vhodným způsobem zahrnout to vnitřní podoby dokumentu, aby se s nimi dalo efektivně pracovat. Konkrétní paměťové struktury jsou ale velmi ovlivněny jak návaznými algoritmy, tak použitým programovacím jazykem.

Při hledání vhodné reprezentace je navíc nutné zaměřit se pouze na relevantní informace. Pro systém, který pro výpočty používá rozměry znaků, je zbytečné zabývat se tvarem výsledné kresby písmene, systém pro textový režim vůbec rozměry znaků nemusí brát v úvahu, protože jsou všechny stejné. Podobně pokud sázecí systém musí zajistit správné obtékání obrázku textem, je pro jeho správnou činnost podstatný pouze tvar a rozměry, nikoli již bitová mapa, která obrázek tvoří. Rozhodnutí, které parametry sázeného dokumentu jsou důležité, se může velmi odrazit na složitosti a tím i rychlosti výpočtu, a proto je nutno při něm brát v úvahu účel a požadované funkce navrhovaného systému.

2.3. Zpracování do výsledné dvourozměrné podoby

Vnitřní reprezentace dokumentu je základem pro jeho zformátování. Cílem je získat popis rozmístění prvků dokumentu na straně či stranách. Typickým médiem jsou listy papíru, stále častěji se ale setkáváme i s elektronickým publikováním, kdy je dokument zobrazen prohlížečem a tisk na papír je jen vedlejší funkcí. I zde je ale rozmístění prvků dokumentu na straně velmi důležité, protože může velmi přispět k čitelnosti (ale samozřejmě i k nečitelnosti) textu.

Systém se při formátování řídí jednak příkazy uživatele, jednak svými vestavěnými algoritmy a parametry, které definují optimální vzhled výsledného dokumentu. Možností, jak vyhovět zadání uživatele, může být velmi mnoho a úkolem sázecího systému je najít v rozumném čase za použití rozumného výpočetního výkonu takové řešení, které splní požadavky uživatele nejlépe. V následujících odstavcích vyjmenujeme alespoň některé z algoritmů, které se při hledání optimální podoby dokumentu uplatní. Ve specializovaných systémech mohou existovat i další a naopak, sázecí systém může být užitečný, i když některé z uvedených funkcí nebude obsahovat.

Základním pilířem sázecího systému je algoritmus pro **řádkový zlom** (*line-breaking*). Jeho úkolem je rozdělit odstavec na řádky dané délky. Algoritmus prochází slova a ostatní prvky odstavce a hledá místa, kde je možné ukončit řádek. Takové místo nazýváme **místo zlomu** (*breakpoint*). Možnými kandidáty pro zalomení jsou nejčastěji mezery mezi slovy, a také místa, kde je možné slova rozdělit (*hyphenation point*). Délka výsledného zalomeného řádku většinou není přesně rovna požadované šířce odstavce. Přebytečné či chybějící bílé

místo pak může být rozmístěno mezi prvky řádku, například do mezislovních mezer. Tak dostaneme odstavec zarovnaný zleva i zprava. Můžeme samozřejmě chtít i sazbu na pravý nebo levý praporek, kdy se rozdíl v délce vyrovná volným místem na pravém či levém okraji řádku.

Algoritmus řádkového zlomu, ale i ostatní, o kterých budeme hovořit, by měl sledovat jeden hlavní cíl: být co nejméně patrný. Šeď na stránkách má být stejnoměrná, zlom řádků nesmí čtenáře rušit. Při ruční sazbě se většinou rozhodujeme mezi více variantami a podobná rozhodnutí dělá i počítač. Důležitá je potom volba kritéria, podle něhož se algoritmus řídí. Toto kritérium by mělo zachytit krásu a vhodnost dokumentu v jednotkách, které jsou počítačem uchopitelné a zpracovatelné.

Důležitou funkcí moderních systémů pro počítačovou sazbu je algoritmus pro **dělení slov** (*hyphenation*). Dovoluje najít další místa zlomu v odstavci a tak zmenšit bílé místo, které se vkládá do řádků, pokud se jejich délka liší od požadované. Pokud například ve vstupním textu napíšeme slovo „neuchopitelný“, je na systému, aby slovo rozdělil (neuchopitelný), pokud by to jinak vedlo k výrazně horšímu řádku. Vzhledem k tomu, že pravidla pro dělení slov se jazyk od jazyka liší, závisí správná funkce tohoto algoritmu na kvalitním popisu povolených míst dělení; ten je nutné vytvořit pro každý jazyk zvlášť. Jednou z možností je pracovat se seznamem všech slov jazyka, ve kterém budou vyznačeny dělící body. Takové řešení by ale bylo velmi paměťově i časově náročné, a proto se častěji používají **vzory dělení** (*hyphenation patterns*), které popisují vhodná místa pomocí pravidel či malých podčástí slov.

Stránkový zlom (*page-breaking*) má podobný úkol jako zlom řádkový, pouze ve vertikálním smyslu. Hledá možná místa zlomu mezi řádky a jinými prvky dokumentu, kde potom bude začínat další strana dokumentu. U hladké sazby, která obsahuje pouze text, například beletrie, je často požadován pevný **řádkový rejstřík**, kdy vzdálenost **účaří** (*baseline*) řádků je pevná. Zde je potom situace jednoduchá: pokud víme, jak velká bude vzdálenost mezi účařími řádků a známe výšku stránky, je snadné spočítat počet řádků na stranu. Jiný případ nastane, pokud nepožadujeme, aby vertikální vzdálenost mezi elementy dokumentu byla pevná. Mezi řádky si většinou nic takového nepřejeme, pokud jsou ale části textu psány jinou velikostí písma, pokud text obsahuje tabulky, grafy či rovnice, nebo když použijeme poznámky pod čarou či obrázky, vznikne v dokumentu množství vertikálního bílého místa, jehož rozměr nemusí být pevně zadán. Pak je třeba opět počítat optimální místo pro ukončení aktuální strany a začátek další a vybírat při tom z více možností.

Často požadovanou funkcí je **sazba tabulek**, kdy zadáváme obsah jednotlivých řádků a sloupců bez toho, aniž bychom specifikovali přesně rozměry jednotlivých políček tabulky. Úkolem systému je nalézt co nejvyváženější podobu tabulky, při níž musí vzít v úvahu všechny souvislosti mezi jednotlivými prvky. Podobný úkol vzájemných vztahů mezi částmi řeší i algoritmy pro **sazbu matematických výrazů** či **chemických vzorců**. I zde je nutné najít optimální pozice částí výrazu či rovnice, zde navíc musí program respektovat historické tradice tiskařů, které vedly ke složitým pravidlům pro zápis matematiky. Ve všech těchto případech je vhodné, aby měl uživatel možnost ručně poopravit výsledek práce programu a dosáhnout tak co nejvyšší kvality dokumentu.

V předchozím textu jsme se zmínili o poznámkách pod čarou. Pro elementy podobného typu existuje výstižný název: **plovoucí objekty** (*floating objects*). Jejich pozice ve výsledném dokumentu totiž není pevně daná už na vstupu, ale je nalezena až v průběhu výpočtu. Pokud totiž chceme použít poznámku pod čarou, která se objeví pod čarou dole na stránce, bylo by velmi nepraktické umisťovat ji na správné místo již ve vstupním textu. Jeho změnou se totiž změní i místa zlomů mezi stranami a my bychom si pozici poznámek museli

stále hlídat a opravovat ji. Proto se tento problém řeší tak, že se ve vstupním textu uvede pouze text poznámky a místo, odkud je na ni odkazováno — to bývá označeno například hvězdičkou nebo číslem. Sázecí systém pak sám zajistí, aby se text poznámky na spodním okraji strany objevil na stejné straně, ze které je odkazována.

Podobně je tomu s velkými obrázky či tabulkami. Pokud bychom je zařadili přímo do hlavního textu a v tomto místě by došlo ke zlomu stránky, zůstalo by na konci předchozí stránky velké bílé místo. Pokud takový objekt označíme jako plovoucí, zajistí sázecí systém jeho umístění na co nejvhodnějším místě tak, aby to nenarušilo zbytek dokumentu. Různé systémy dovolují různé typy **ukotvení** (*anchor*) plovoucích objektů: vůči řádku, odstavci, sloupci nebo stránce, dovolují také specifikovat absolutní či relativní pozici vzhledem ke zbytku stránky.

Plovoucí obrázek, který vložíme do textu, může zabírat třeba jen třetinu šířky stránky a my můžeme chtít zbytek strany vyplnit textem. Systém pak zajistí správné **obtékání obrázků** tím, že změní požadovanou délku řádků, které padnou ve výsledném dokumentu vedle takovéhoto objektu a i při změně dokumentu přepočítá správně délky řádků na patřičném místě.

Dokumenty technického či vědeckého charakteru jsou většinou děleny na kapitoly, části a oddíly, obsahují tabulky a schémata, definice, výčty případů. Pro pořádek je vhodné tyto části textu číslovat. Z praktických důvodů je ale často dobré nechat sázecí systém, aby provedl **číslování** (*enumeration*) za nás. Můžeme potom přidat do textu další tabulku či kapitolu bez toho, abychom museli projít celý zbytek textu a všude zvyšovat pořadová čísla o jedničku.

Další funkce, která velmi souvisí s proměnlivostí zpracovávaného dokumentu, jsou **odkazy v textu** (*cross-reference*). Tím, že při psaní předem nevíme, na kterou stranu který odstavec padne, nemůžeme odkaz na jiné místo v textu napsat přímo slovy absolutně, jako třeba „viz. Věta na straně 25“. Po změně a přeformátování dokumentu by takováto čísla téměř jistě nebyla platná. Místo toho vložíme do textu pouze značku, na kterou chceme ukázat, a systému řekneme, že má do odkazu doplnit vždy správné číslo strany, na které se značka nachází. Často bychom také chtěli napsat odkaz typu „případ, o němž se zmiňujeme v kapitole 7“ nebo „jak je vidět z tabulky 6.3“. I tato čísla může sázecí systém doplnit za nás, pokud jsme použili nějakou formu automatického číslování těchto objektů.

Do podobné kategorie patří i **sazba obsahů, rejstříků, literatury**. Také zde necháme na počítači, aby vyhodnotil odkazy, čísla stránek a seznamy sám. Do dokumentu pak zařadí již jen hotový obsah či rejstřík a postará se o to, že uvedené informace budou stále platné.

V každém systému je velmi důležité funkční propojení všech částí. Ze třeba se zaměřit nejen na kvalitu vlastních algoritmů, ale také na jejich spolupráci a vzájemnou vyváženost. Jádro této diplomové práce tvoří algoritmy řádkového a stránkového zlomu. Zaměříme se na existující postupy a navrhneme jejich rozšíření, obzvláště co se týče jejich vzájemné interakce.

2.4. Výstup sázecího systému

Zformátovaná podoba dokumentu začne mít pro uživatele smysl až tehdy, kdy se objeví na výstupu. Výstupním zařízení interaktivních systémů je obvykle monitor počítače. Výstupní část sázecího systému vykreslí dokument na obrazovce za pomoci volání funkcí operačního systému pro grafický či textový výstup. Tyto funkce jsou ale pro každou platformu jiné, a proto bývá nutné výstupní část takovéhoto sázecího systému pro každý nový systém naprogramovat zvlášť.

Všechny sázecí systémy proto dovolují i uložení výsledku v některém z formátů pro popis strany, dávkové systémy toto považují za jediný způsob výstupu. Z nejpoužívanějších formátů jmenujme alespoň PostScript (PS, EPS) a Portable Document Format (PDF), které vyvinula firma Adobe, či DeVice Independent formát (DVI), používaný systémem TEX. Existují samozřejmě i mnohé další, většinou se ale jedná o proprietární formáty komerčních firem, které nejsou dokumentovány a mění svůj tvar s každou novou verzí softwarového produktu. A samozřejmě, že za jistý druh formátu popisujícího vzhled stránky je třeba považovat i čistý text zformátovaný pro zobrazení v textovém módu.

3. Algoritmy pro řádkový zlom v textovém režimu

Řádkový zlom hledá mezi slovy a ostatními prvky odstavce možná místa zlomu, která jsou kandidáty pro ukončení aktuálního řádku. Algoritmus se řídí hlavně požadovanou šířkou odstavce, tedy délkou jednotlivých řádků. Při práci je nutné vzít v úvahu především rozměry znaků ve slovech a velikosti mezer mezi nimi. Algoritmus musí umět spolupracovat s algoritmem pro dělení slov, který dovolí nalézt další místa zlomu uvnitř slov.

Ukážeme postupy používané pro řešení tohoto úkolu. V této kapitole se zaměříme na ty nejjednodušší, které pracují v textovém režimu, v dalších uvedeme ty, které se používají v digitální sazbě při přípravě knih.

3.1. Program fold pro zalomení textu

Tento příkaz známý ze systému Unix zarovná textová data tak, aby žádný řádek nebyl delší než zadaná mez, například 80 znaků. Delší rozdělí na více řádků.

```
fold:
    nuluj proměnnou uchovávající délku aktuálního řádku;
    while (je ještě nějaký znak na vstupu)
        {
        načti ho;
        if (je to znak nového řádku || délka aktuálního řádku je rovna požadované)
            {
            zapiš na výstup znak nového řádku;
            nuluj délku aktuálního řádku;
            }
        if (nejde o znak nového řádku)
            {
            zapiš na výstup načtený znak;
            zvyš délku aktuálního řádku o jedna;
            }
        }
```

Program čte vstupní text znak po znaku a zapisuje je na výstup. Při tom si pamatuje délku aktuálního řádku a pokud by měla překročit danou šířku textu, odřádkuje, typicky příkazem `print "\n"`.

Časová složitost tohoto algoritmu je lineární, paměťová náročnost konstantní. Rozhodnutí o dalším průběhu výpočtu se děje hned po načtení dalšího znaku ze vstupu. Text není uchováván v paměti, okamžitě ho posíláme dále na výstup.

Algoritmus vypadá velmi jednoduše, ale při delším použití objevíme některé nedostatky. Z nich hlavní je ten, že předpokládáme, že všechny jednobajtové znaky, které se objeví na vstupu, budou mít na obrazovce šířku jeden znak. To je pravda například u písmen či číslic, ale už nikoli u řídících znaků, tedy těch, jejichž ASCII hodnota je nižší než 32. Ty jsou určeny pro řízení výstupních zařízení (terminálu, tiskárny) a na výstupu se projeví například pípnutím, smazáním předchozího znaku, a podobně. Činnost jiných je ještě výraznější — řídící znak tabelátor (`\t`) způsobí přesun aktuální pozice na další tabelační zarážku, znak uvozující tzv. **escape** sekvenci může být následován libovolným počtem argumentů,

které určí výslednou akci, například přepnutí na inverzní písmo, do jiné znakové sady, atd. Program by měl s těmito alternativami počítat: buď je korektně ošetřit, nebo takovéto znaky z textu odstranit.

3.2. Program fold se zalomením mezi slovy

Verze uvedená výše je vhodná pro zalomení výpisů programů, kde nám jde pouze o to neztratit ty části textu, které by jinak zmizely za pravým okrajem. Při tisku textů a počítačových manuálů bychom se rádi více přiblížili běžně používané podobě. Druhým úkolem bude tedy ořezání textu na danou šířku s tím, že rozdělení řádku povolíme pouze v mezislovní mezeře.

Algoritmus můžeme popsat takto:

```
fold-space:
    nuluj proměnnou uchovávající délku aktuálního řádku;
    nuluj pomocnou paměť pro uchování vstupního slova;
    while (jsou na vstupu nějaká data)
        {
        načti slovo nebo mezeru či konec řádku;
        if (je to znak nového řádku || délka aktuálního řádku plus délka načteného slova
                a mezery je větší než požadovaná)
            {
            zapiš na výstup znak nového řádku;
            nuluj délku aktuálního řádku;
            if (bylo vstupem slovo)
                {
                zapiš je na výstup;
                zvyš délku aktuálního řádku o délku zapsaného slova;
                }
            }
        else
            {
            zapiš na výstup mezeru a načtené slovo;
            zvyš délku aktuálního řádku o jejich délku;
            }
        }
```

Rozdíl oproti předchozímu programu je v tom, že ze vstupu nečteme po znacích, ale po slovech. Potřebujeme tedy již paměť pro uchování řetězce znaků. Po načtení porovnáme, zda se celé slovo ještě vejde na aktuální řádek. Pokud ano, přidáme ho na výstup, jinak řádek ukončíme a začneme nový.

Asymptotická složitost tohoto algoritmu je stejná jako u původního programu `fold`, při vlastní implementaci ale musíme zvážit, jak velkou paměť vyhradíme jako buffer pro načítané slovo. Další komplikace, na kterou narazíme až při vlastním programování, je skryta pod příkazem „načti slovo“. Co se totiž stane, pokud je načítaný text delší než zadaných 80 znaků? Zřejmě se na aktuální řádek nemůže celý vejít a zde se musíme předem rozhodnout, zda v tomto nouzovém případě zalomíme řádek i jinde než na mezeře, a nebo slovo prostě ořízneme.

3.3. Práce s odstavci — program pack

Při zpracování informací v textovém režimu je často vhodné pracovat s odstavci, které jsou od sebe odděleny jedním či více prázdnými řádky. Uprostřed odstavce se mohou vyskytovat znaky nového řádku, které ale program nahradí mezerami a na výstup dá nově zarovnaný text.

Pro takový úkol můžeme použít předchozí algoritmus, kde změníme test přítomnosti znaku nového řádku na test na alespoň dva takové znaky a při čtení ze vstupu převádíme znaky `\n` na mezery. Časová i paměťová složitost zůstává stejná a stejný je i přístup k problému — v dané chvíli uvažujeme pouze právě načtené slovo.

3.4. Oboustranné zarovnání textu — uchování aktuálního řádku

Konečně se dostáváme k úkolu, který bude již vyžadovat komplexnější přístup. Chceme mít na výstupu text, v němž budou všechny řádky (pochopitelně kromě posledního řádku odstavce) končit na dané pozici, budou zarovnány zleva i zprava. Zřejmě tedy musíme pozdržet zápis dat na výstup nejméně do chvíle, kdy bude zřejmé, která slova se na zpracovávaný řádek vejdou a kolik mezer musíme do řádku vložit na jeho zarovnání.

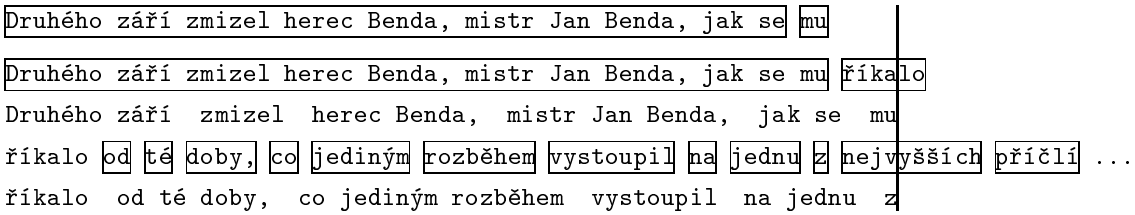
Níže uvedený kód pracuje nad jedním odstavcem textu. Na zpracování celého dokumentu bychom použili ještě jeden vnější `while` cyklus.

```
line-buffer:
    nuluj buffer pro uchování aktuálního řádku;
    nuluj buffer pro načítané slovo;
    while (jsou na vstupu nějaká data odstavce)
        {
        načti slovo až po další možné místo zlomu;
        znaky nového řádku převeď na mezery;
        if (délka aktuálního řádku plus načteného slova a mezery > šířka odstavce)
            {
            vypočti rozdíl (požadovaná délka − délka aktuálního řádku);
            tolik mezer vlož do aktuálního řádku k již existujícím mezerám;
            zapiš na výstup zarovnaný řádek;
            nuluj buffer aktuálního řádku;
            }
        přidej do bufferu načtené slovo, od předchozího odděl mezerou;
        }
    if (buffer obsahuje nějaká data z posledního řádku)
        {
        zapiš je na výstup;
        }
```

Zde si v pomocné paměti uchováváme právě zpracovávaný řádek. Přidáváme do ní slova čtená ze vstupu, dokud celková délka řádku nepřekročí zadanou mez. Potom slova v této paměti zarovnáme doplněním chybějících mezer do vhodných míst a celý řádek najednou pošleme na výstup.

Paměťová i časová složitost je stále stejná jako u předchozích algoritmů, i když komplexnost řešení se zvýšila. Máme zde dvě pomocné paměti — kromě bufferu na právě načtené slovo zde je i paměť na data celého aktuálního řádku. Teprve až známe celý řádek a víme jeho délku, můžeme provést zarovnání a tisk.

Predveďme si příklad výpočtu pomocí algoritmu `line-buffer` pro šířku odstavce 64 znaků. Používáme zde dvě pomocné paměti, první pro aktuální řádek a druhou pro právě načtené, testované slovo:

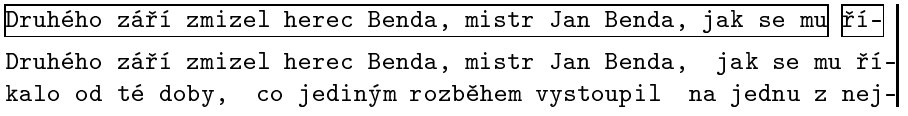


Způsob, jakým přebytečné mezery do textu vložíme, může také přispět k čitelnosti textu. Je například lepší vložit mezeru za čárku než za neslabičnou předložku a následující slovo. Přesným stanovením pravidel se zde ale nebudeme zabývat, navíc je nutné je definovat pro každý jazyk zvlášť.

Řádek načtený v paměti umožní i jiné úpravy než oboustranné zarovnání. Místo rozložení „výplňových“ mezer do řádku je můžeme vytisknout na výstup ještě před vlastním řádkem a dostaneme tak text zarovnaný zprava, na levý praporek. Pokud bychom mezery vložili za řádek, dostali bychom stejný výsledek jako u programu `pack`, snadno dosáhneme i centrovaného textu. Data aktuálního řádku uložená v paměti jsou primitivní vnitřní reprezentací dokumentu — můžeme s nimi dělat další manipulace, nejen je přímo okopírovat na výstup.

Na začátku cyklu jsme použili formulaci „načti slovo až po další možné místo zlomu“. Možným místem zlomu je pro nás především mezislovní mezera, ale při načítání slova, které by již překročilo zadanou mez, v něm můžeme ještě navíc hledat místa pro rozdělení. Poté, co k takové části slova přidáme rozdělovník, vznikne další možné místo zlomu. Opačný postup můžeme použít například u krátkých českých předložek: z pravopisného hlediska je chybné rozdělit větu „šel v lese s kládou“ za jednopísmennými předložkami. Pro ošetření takovýchto případů je vhodné upravit algoritmus tak, aby mezeru za jednopísmenným slovem nepovažoval za možné místo pro zlom.

Takto by vypadala předchozí věta, pokud bychom rozdělili slovo „říkalo“ a místo mezislovní mezery za předložkou „z“ zvolili za místo zlomu bod dělení ve slově „nejvyšších“:



Přidáním dalších bodů zlomu uvnitř slov se vzhled řádků výrazně zlepšil, například místo pěti mezer bylo nutné do prvního řádku vložit pro zarovnání textu pouze jednu a odstranili jsme zlom stránky za neslabičnou předložkou. Daní za to jsou dva po sobě jdoucí řádky končící rozdělovníkem.

Vidíme, že i u takovýchto jednoduchých úkolů je vhodné zvolit přístup, který nám dovolí abstrahovat od detailní implementace a spíše se zaměřit na podstatu řešení. Termín místo zlomu je zde označení přesnější než mezislovní mezera.

Ve všech předchozích příkladech jsme předpokládali vstup dat po jednotlivých elementech, znacích. V moderní programovacích jazycích je ale běžné provádět vstupní a výstupní operace po větších celcích. Dostupnost paměti přestává být problémem a z hlediska efektivnosti je lepší přečíst najednou celý řádek osmdesáti znaků než volat 80 krát čtení jednoho. Manipulace se vstupem se pak změní na práci s indexy do pole znaků. Podstata algoritmů ale zůstává stejná.

Tímto jsme uzavřeli výčet postupů, které použijeme, pokud sázíme text pro zobrazení na textovém terminálu či tisk na tiskárně s neproporcionálními fonty. Jejich hlavním rysem je, že vůbec neuvažují šířku jednotlivých znaků — všechny (včetně mezer) ji mají stejnou. V dalším textu se již podíváme na postupy, které počítají s výstupem na grafické zařízení a s texty, které hýří fonty nejrůznějších typů a velikostí.

4. Používané modely a terminologie

Dříve než se začneme zabývat proporcionální sazbou, uvedeme jednu z možných terminologií pro popis typografických prvků, se kterými budeme pracovat.

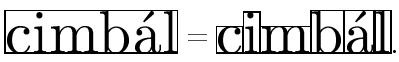
4.1. Znaky a mezery

Černé znaky a bílé mezery mezi nimi tvoří základní stavební kameny dokumentu. Úkolem sázecího systému je nalézt jejich optimální rozložení na ploše, aby výsledek splňoval estetická hlediska, posuzovaná čtenářem. I když se v tiskovinách okolo sebe můžeme setkat i s jinými elementy, jako jsou obrázky, čáry, použití barev a různých efektů, jsou tyto dva typy — znaky a mezery — tím nejdůležitějším.

Znak popisuje tvar a další vlastnosti jednoho symbolu ve fontu. Je definován několika základními atributy: svou kresbou, rozměry **ohraničujícího obdélníka** (*bounding box*) a vztahem ke svému okolí. Pro sázecí systém většinou kresba znaku (to, jak nakonec vypadá na tiskárně) není podstatná. Sázecí algoritmy se o vlastní zobrazení nestarají, to je úkolem až výstupní části systému, případně navazujících programů.

Důležitější je informace tom, jaké místo znak zabírá. Podívejme se, jaký je ohraničující obdélník písmene c:

Důležité je také umístění znaku vůči účaří a ostatním znakům. Proto se pro znak definuje **referenční bod** (*reference point*), který je u písmene „c“ v levém dolním rohu. Obdélník, v němž je znak umístěn, má dva rozměry: **výšku** a **šířku**. Pokud má sázecí systém systém vysázet slovo „cimbál“, poskládá vedle sebe těchto šest písmen tak, že každé následující připojí těsně za předchozí:



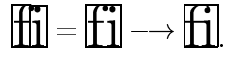
Vzdálenost referenčních bodů je rovna právě šířce jednotlivých znaků. Můžeme si také všimnout, že kresby písmen se vzájemně nedotýkají a že písmena s kulatými tahy mírně přesahují vymezený prostor. Výška a šířka ohraničujícího obdélníka spolu s umístěním referenčního bodu je jediným vodítkem, podle kterého se sázecí algoritmus řídí.

Při zpracování znaků, které zasahují i pod účaří, se ukazuje být užitečné zavést ještě třetí rozměr, **hloubku**. Stále zůstaneme na dvourozměrné ploše, hloubkou vyjádříme rozměr znaku ve směru pod účaří. Ve slově

jsou pak všechny referenční body v jedné rovině na účaří.

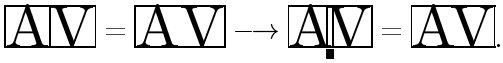
Řekli jsme, že algoritmus při sázení znaků vedle sebe bere v úvahu šířku jejich obrysových obdélníků. To je sice pravda, ale v některých případech je vhodné do takového postupu zasáhnout, protože třeba některé dva znaky nevypadají vedle sebe pěkně, anebo je vhodné vzdálenost mezi nimi mírně poopravit. K tomu slouží **slitky** (*ligature*) a **mezipísmenný proklad** (*kerning*), které jsou doplňujícími informacemi o znacích ve fontu.

Ukažme si typickou ligaturu. Znaky f a i, pokud se objeví vedle sebe, je vhodné spojit v jeden:



Znaky se spojily a vytvořily celek, který vypadá při tisku lépe. Poznamenejme, že takovouto transformaci musí každý sázecí systém, hodný toho jména, provést sám. Existují i programy, kde uživatel je nucen spustit sám „náhradu všech dvojic znaků f, i za znak ligatury fi“, ale slušné systémy se o tento úkol postarají samy. Občas nastane případ, kdy se dva znaky, pro něž je ve fontu definována ligatura, objeví na hranici částí složených slov či na jiném místě, kde by umístění ligatury bylo nevhodné či z pravopisného hlediska nesprávné. Protože je těžké tyto výjimky algoritmizovat, musí mít uživatel možnost, jak automatické náhradě ligatur zabránit. V češtině je typickým příkladem slovo „šéflékař“, kde je ligatura velmi nevhodná (šéflékař), protože nerespektuje způsob, jak bylo toto slovo vytvořeno.

Typickým kerningem je dvojice písmen AV, kde by díky jejich tvaru vznikla nepěkná velká mezera a proto jsou přisunuty blíže k sobě:



I tuto úpravu by měl sázecí systém zajistit sám bez zásahu uživatele.

Kerning znamená horizontální posun mezi znaky v textu, vložený doprostřed slova bez zásahu uživatele. Jiné posuny do textu zadává naopak vědomě sám uživatel. Nejčastější z nich jsou mezislovní mezery. Fakt, že se ve vstupním textu objeví znak s dekadickou hodnotou 32, který v ASCII tabulce označuje mezeru, se musí ve vnitřní reprezentaci systému projevit značkou pro horizontální posun, jehož velikost je závislá na aktuálním fontu a jeho velikosti. Například ve větě

Pane, to byl umělec, slyšíte?

jsou mezi pěti slovy čtyři mezislovní mezery. Tyto bílé mezery jsou jednak protiváhou černým znakům, jednak v nich sázecí systém může najít jistou flexibilitu při formátování textu. Pokud by algoritmus zjistil, že je nutné, aby předchozí věta zabrala více místa, byly by to právě mezislovní mezery, které by bylo možné roztáhnout. A protože nejsme limitováni žádnou maticí, do které bychom znaky měli poskládat a můžeme je proto po stránce libovolně posunovat, je nejpřirozenější nadbytečné místo rozdělit stejnoměrně mezi všechny čtyři mezery:

Podobně, pokud by bylo vymezené místo užší, systém by od každé mezery trochu ubral.

Kromě mezislovních mezer může uživatel zadat libovolný jiný typ posunu v textu. U některých je velmi důležité, aby se velikost posunu při řádkovém zlomu nijak neměnila. A také u proměnných mezer je vhodné stanovit jistá omezení na to, jak moc mohou být změněny. Takto zúžený text je již velmi nečitelný:

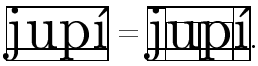
Pane, to byl umělec, slyšíte?

Řešením je ke každému posunu přidat informaci o tom, jak moc se může zvětšit a jak moc zmenšit. U pevných posunů nahoře nastavíme obě tyto hodnoty — **roztažitelnost** (*stretchability*) i **stažitelnost** (*shrinkability*) — na nulu a tím jejich velikost zafixujeme.

4.2. Model box-glue-penalty

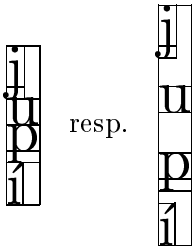
Výše uvedený princip znaků jako obdélníkových částí plochy se třemi rozměry a posunů, u kterých můžeme určit jejich roztažitelnost a stažitelnost, je v sázecím systému TEX rozšířen na model označovaný jako box-glue-penalty. Protože tato práce se bude zabývat především algoritmy použitými v TEXu a protože tento model je velmi názorný, uvedeme zde alespoň jeho stručný popis.

Box (někdy doslovně překládán jako krabice, my se přidržíme původního anglického termínu) je označení pro obdélník, který má tři rozměry — šířku, výšku a hloubku. Kromě rozměrů nese také informaci o tom, co je jeho obsahem, nebo-li co bude zobrazeno v ploše, kterou ohraničuje. Na rozdíl od předchozího textu, kde jsme vždy uvažovali pouze obdélník ohraničující písmeno, zde může být jeho obsah zcela libovolný, může obsahovat jakýkoli typografický materiál. Nejčastěji uvnitř boxu najdeme opět boxy. Například naše slovo jupí můžeme uzavřít do boxu, jehož obsahem budou vedle sebe čtyři boxy, nesoucí písmena:



Všimněme si, že tento box má opět šířku, výšku i hloubku, kde šířka je součtem šířek vnitřních boxů a výška a hloubka jsou maximem odpovídajících veličin z vnitřních boxů. Protože jsou vnitřní boxy uspořádány vedle sebe, nazývá se takovýto box **horizontální box**, zkráceně **hbox**.

Můžeme vytvořit i jeho vertikální ekvivalent, nebo-li **vbox**:



Šířka vboxu je maximální šířka vnitřních boxů, jeho výška je součet jejich vertikálních rozměrů mimo hloubky posledního. Hloubku definujeme rovnu hloubce posledního boxu. Výpočet rozměrů boxu je ve skutečné implementaci TEXu složitější, nám ale prozatím postačí tento intuitivní náhled.

Glue (mohli bychom přeložit jako lepidlo, ale i zde budeme používat anglický originál) znamená posun v textu, který má definovanou roztažitelnost a stažitelnost. Stejně jako box může mít dvě podoby — horizontální a vertikální. Horizontální glue se uplatní pouze pokud skládáme typografický materiál vedle sebe, vertikální pokud ho stavíme vertikálně pod sebe. Z praktických důvodů se v TEXu používají pro míry roztažitelnosti a stažitelnosti kromě klasických rozměrů jako cm nebo pt i míry v jednotkách nekonečna. Tyto se pak uplatní před tím, než by se použily rozměry ostatních (konečných) roz/stažitelností.

Penalta (zde ustoupíme od původního termínu penalty) je značka, která se použije při lámání typografického materiálu do řádků. Jak uvidíme níže, když lámací algoritmy posuzují vhodnost určitého zalomení, je jejich rozhodnutí založeno na hodnotě jisté (konkrétně definované) funkce. Penalta má jeden numerický atribut, kterým říká, jak se má změnit hodnota cenové funkce pro zalomení v daném bodě. Toho můžeme využít k označení míst, která se nám pro zalomení zdají vhodná a zlom by v nich vypadal dobře, například před nadpisem kapitoly, a naopak k potlačení zlomu v místech, kde to z typografických či pravopisných hledisek dobré není, v češtině třeba za neslabičnou předložkou.

Druhý argument určuje, jaký typografický materiál bude vložen do textu, pokud bude řádek zalomen právě na této penaltě. Toho můžeme s výhodou využít při dělení slov. Zadáme-li v možném bodě rozdělení slova penaltu, která má tento parametr nastaven na znak rozdělovníku, pak rozdělovník bude na konec řádku v případě rozdělení vložen automaticky. Dodejme, že toto vkládání penalt, které označují místa pro rozdělení slova, nedělá obvykle uživatel ručně. Je starostí algoritmu řádkového zlomu, aby ve vhodné chvíli zavolal funkci pro dělení slov a od ní získal potřebné informace.

S využitím modelu box-glue-penalty je možno popsat a vyřešit velké množství požadavků na tvar dokumentu. V [Knuth 80] i [Knuth 84] je ukázána řada aplikací při sazbě časopisů a knih, kde pomocí těchto tří prvků jsou implementována i taková zalomení textů, která by jinak vyžadovala náročnou ruční práci sazeče.

5. Algoritmy řádkového zlomu pro grafický výstup

Nejprve popíšeme tvar vstupu, se kterým budou následující algoritmy pracovat. Ten vychází z modelu box-glue-penalty a jeho použití nám pomůže nejen k vlastnímu popisu algoritmů, ale také ke srovnání jednotlivých řešení [Knuth 80].

Vstupem algoritmu pro řádkový zlom je posloupnost typografického materiálu, tvořená boxy, glue a penaltami. Stručně tuto posloupnost nazveme **horizontální seznam** (*horizontal list*). Algoritmus musí také znát požadované délky jednotlivých řádků, pro jednoduchost ale můžeme předpokládat, že chceme mít všechny řádky stejně dlouhé, a proto je parametrem pouze jedna hodnota. Výstupem je pak označení nejlepších míst zlomu, například jako posloupnost indexů do vstupního horizontálního seznamu. Vybrané prvky pak tvoří hranici řádků, prvky mezi nimi obsah jednotlivých řádků odstavce.

Boxy v tomto přístupu reprezentují znaky z fontu, ale i složitější konstrukce vytvořené uživatelem či sázecím systémem. Mají pevně danou šířku, která byla buď explicitně zadána uživatelem, nebo vypočtena z informací o rozměrech znaků ve fontu. Výška ani hloubka boxů nejsou pro účely řádkového zlomu podstatné. Glue, které se použije hlavně na vytvoření mezislovní mezery, má kromě šířky i hodnotu roztažitelnosti a stažitelnosti. Ty mohou být pochopitelně i nulové, a potom glue znamená prostě horizontální posun bez jakékoli variablility. Penalty označují vhodná a nevhodná místa pro zlom horizontálního seznamu, případně možná místa pro rozdělení slov. Ta mohou být vyhledána již před spuštěním lámacího algoritmu, nebo až na jeho žádost. Který způsob bude zvolen, záleží na konkrétní implementaci.

Všechny algoritmy, které jsme uvedli v části o textovém výstupu, mají svou podobu i při výstupu grafickém. Rozdíl spočívá v tom, že nyní nemůžeme předpokládat jednotnou šířku znaků, musíme pro každý načtený box zjistit jeho skutečné rozměry. Jinak ale můžeme postupovat obdobně: čteme vstup dokud není řádek naplněn, poté odřádkujeme.

Takto odvozený přístup ovšem nerespektuje fakt, že glue má kromě své **přirozené šířky** (*natural width*) i hodnotu roztažitelnosti a stažitelnosti. A právě tyto hodnoty je nutné vzít do úvahy při zarovnávání řádku na požadovanou šířku. Glue s hodnotou roztažitelnosti velkou by se mělo na roztažení řádku podílet více než glue, jehož roztažitelnost je menší. Pak teprve začnou mít tyto hodnoty smysl, protože budeme schopni rozlišit různé typy mezer. Například angličtina se sází se zvětšenými mezerami za interpunkčními znaménky. V našem modelu toho dosáhneme tak, že za čárku vložíme mírně širší mezeru (glue) s mírně větší mírou roztažitelnosti než mezi ostatní slova. Takové glue potom „roste“ rychleji než ostatní mezislovní mezery.

5.1. First fit

Tento algoritmus je rozšířením programu `line-buffer`, který jsme popsali v části o řádkovém zlomu pro textový výstup (viz. 3.4.). Bere vstup z horizontálního seznamu a testuje, zda se jedná o možné místo zlomu. Pokud na takové narazí, spočítá celkovou délku potenciálního řádku a hodnoty celkové roztažitelnosti a stažitelnosti v něm. Pro každé místo zlomu známe tedy tři hodnoty: šířku všech prvků v řádku w, roztažitelnost w^+ a stažitelnost w^-. Zároveň víme pro aktuální řádek jeho požadovanou šířku L.

Je-li řádek krátký ($w < L$), bylo by nutno pro dorovnání šířky textu v řádku glue roztáhnout, a to o hodnotu $L - w$. Toto zvětšení chceme přenést na všechna glue s nenulovou roztažitelností. **Poměr roztažení** r (*adjustment ratio*) řádku a tím i všech glue v něm je roven $(L - w)/w^+$. Konečná šířka každého glue v řádku bude $w_i + r \times w_i^+$. Naopak pro řádek, který zadanou šířku přesahuje ($w > L$), se uplatní hodnoty stažitelnosti. Výsledná šířka i-tého prvku se zmenší na $w_i + (L - w)/w^- \times w_i^-$, poměr $r = (L - w)/w^-$ je zde záporný. Pokud by se stalo, že místo zlomu vytvoří řádek, jehož šířka je přesně rovna L, žádné modifikace glue se pochopitelně neprovádí.

Algoritmus first fit se snaží zvolit za konec řádku takové místo zlomu, kde je absolutní hodnota r menší než 1. Tato podmínka znamená, že řádek byl změněn (roztažen či zkrácen) pouze do míry povolené hodnotami glue v řádku.

```
first-fit:
    začátek řádku = první prvek horizontálního seznamu;
    celková délka = celková roztažitelnost = celková stažitelnost = 0;
    while (je nějaký prvek v horizontálním seznamu)
        {
        if (je toto možné místo zlomu)
            {
            vypočti r pro řádek končící tímto prvkem;
            if (r ≤ 1)
                {
                pošli na výstup řádek končící v tomto prvku;
                nastav nový začátek řádku;
                nastav novou celkovou délku a roztažitelnost a stažitelnost na nulu;
                nuluj aktuální prvek;
                }
            }
        zvyš celkovou délku a roztažitelnost a stažitelnost o hodnoty aktuálního prvku;
        }
```

Program prochází prvky ve vstupním horizontálním seznamu a zaměřuje se na ty, které jsou možnými místy zlomu. Pro každý spočítá, jak dlouhý by byl řádek, kdyby v aktuálním prvku končil, a jaký by pak byl poměr roztažení (případně stažení) takto ukončeného řádku. Pokud najde takové místo zlomu, kde je řádek kratší než požadovaná délka a zároveň není kratší „o moc“ ($r \leq 1$), pošle ho na výstup.

Stejně tak pošle na výstup řádek, který je delší než zadaná mez (pro $r < 0$). Zde již ale není analogický test $r \geq -1$, z tohoto důvodu: pokud předpokládáme, že každé další místo zlomu v horizontálním seznamu je dále od počátku než předchozí, pak v případě, že již je řádek příliš dlouhý ($r < 0$), byl by v dalších bodech zlomu ještě delší. Potom je lepší vysázet první (nejkratší) z těchto dlouhých řádků, protože nemůžeme čekat, že bychom našli lepší místo. Řádek, u něhož $r < -1$ nazveme **přetečený** (*overfull line*) a vysázíme ho přesahující přes pravý okraj odstavce, s tím, že glue v něm budou zkráceny pouze o hodnotu své stažitelnosti, jakoby s $r = -1$. Kdybychom totiž zkrátili glue o více, než povolují, mohli bychom v extrémním případě dostat řádek se zápornými mezerami.

Pokud bychom algoritmus implementovali, museli bychom k této kostře přidat navíc několik zpřesnění a testů. Je například důležité definovat, co rozumíme pod pojmem možné místo zlomu. Přístup podobný tomu, který je použit v TEXu, považuje za možný bod zlomu buď penaltu s takovou hodnotou, která zlom nezakáže, nebo glue předcházené boxem. První možnost je typická pro rozdělení slov a pro implicitní pokyny uživatele sázecímu systému.

Glue se použije například v mezislovní mezeře; jeho rozměry se do celkové testované šířky a r nezapočítávají. Chceme totiž, aby řádek končil předcházejícím slovem (boxem), a nikoli mezerou. Pokud řádek na glue zalomíme, jsou všechna následující glue včetně aktuálního odstraněny, protože jinak by se jejich hodnoty projevily na začátku dalšího řádku, což uprostřed odstavce není žádoucí.

Další test by se měl týkat penalt, které mohou v některém místě textu zlom zakázat a v jiném vynutit. V prvním případě se vlastně nejedná o možné místo zlomu, ve druhém jde naopak o místo povinné. Pokud penaltou vynutíme zlom, může se stát, že musíme glue roztáhnout více, než jejich roztažitelnost dovoluje ($r > 1$). Vzniknou tak velmi velké mezery. Řádek se pak označuje jako **podtečený** (*underfull line*); ekvivalentní termín je nenaplněný. Takový řádek vysázet musíme, i když podmínka $r \leq 1$ není splněna — koneckonců byl tento zlom penaltou vynucen. Glue tedy mohou být zvětšeny více než je jejich roztažitelnost, ale nemohou být zkráceny více než o hodnotu své stažitelnosti.

Při výpočtu délky řádku můžeme s výhodou využít fakt, že vzdálenost dvou prvků horizontálního seznamu je rovna rozdílu jejich vzdáleností od počátku seznamu. Pro každý prvek stačí tedy uchovávat informaci o jeho vzdálenosti od počátku, vzdálenosti k ostatním elementům zjistíme odečtením.

Poslední důležitá věc, o kterou se musíme postarat, je korektní ukončení. To zajistíme například tak, že na konec horizontálního seznamu připojíme penaltu, která vynutí zlom, a tudíž ukončí řádek na posledním prvku odstavce. Tak data posledního řádku neztratíme. Protože bychom tím mohli snadno docílit podtečeného řádku, vložíme ještě před ni jinou penaltu, která zlom zakáže, a mezi ně glue s nulovou šířkou a hodně velkou roztažitelností. Toto glue vyplní bílé místo posledního řádku odstavce.

Stejně jako v algoritmu pro textový režim můžeme i zde řádek vysázet na pravý či levý praporek, místo aby byl zarovnaný z obou stran. To snadno uděláme tak, že při vlastní sazbě řádku doplníme na konec či začátek posloupnosti řádku glue s velkou roztažitelností, čímž „přebijeme“ roztažitelnost v glue uvnitř řádku.

Asymptotická časová složitost algoritmu je polynomiální vzhledem k délce vstupu a výstupu, paměťová složitost je polynomiální vzhledem k délce vstupu. V nejhorším případě se totiž celý vstup vejde na jeden řádek a potom je nutné ho po dobu výpočtu držet v paměti. To je rozdíl oproti algoritmům v řádkovém režimu, kde jsme díky pevným pozicím pro jednotlivé znaky mohli paměťové nároky algoritmu shora předem omezit. Jak ale uvidíme, často se při lámání textů pracuje pouze s indexy do pole typografických prvků a nedochází k manipulaci s elementy samotnými. Pak paměťové nároky algoritmu řádkového zlomu klesnou na konstantní.

Algoritmus je přímočarý: v každém kroku se rozhodne, zda uvažovaný prvek splňuje podmínky pro ukončení řádku, a buď ho vysází, nebo prvek přidá k aktuálnímu řádku. Je to vskutku first fit: bere se první řešení, které je nalezeno. V dalším ukážeme rozšíření, kdy již budeme pro každý řádek porovnávat více možností pro jeho ukončení a zvolíme tu, která bude nejvýhodnější.

5.2. Best fit

Při praktickém použití zjistíme, že algoritmus first fit často volí řádky s relativně hodně roztaženými mezerami, i když následuje krátké slovo, které by se na řádek ještě vešlo. Je to dáno především tím, že jako řešení akceptujeme první místo zlomu, kde je splněna mezní podmínka $r \leq 1$. Druhým důvodem je fakt, že při rozhodování jsou jediným kritériem pouze čisté hodnoty r. Metoda best fit oba tyto problémy odstraňuje.

Pro každý řádek spočítáme hodnotu, která bude charakterizovat, jak moc se liší jeho testovaná podoba od ideálu. Místo porovnávání hodnot r, které jsou lineární v závislosti na roztažení glue, zvolme **cenovou funkci**

$$b = 100 \times |r|^3,$$

jejíž hodnoty navíc pro $r < -1$ definujeme jako nekonečné.

Tato funkce vyjadřuje „**špatnost**“ (*badness*) řádku, mohli bychom také říci nehezkost či škaredost, cenu, kterou v estetických jednotkách platíme za konkrétní zlom. Pro hodnoty mezi -1 a 1 roste pomalu, posléze velmi rychle. Tím dobře vyjadřuje, co očekáváme od roztažitelnosti a stažitelnosti, na nichž je její hodnota založena. Měly by zabránit, aby se šířka glue změnila více, než je zadáno. Viděli jsme, že při podtečeném řádku je nutné glue roztáhnout více, ale tento řádek je již třeba považovat za velmi špatný. Zkrácení o více než stažitelnost vůbec nepovolíme, takový řádek je nekonečně špatný.

Program založený na přístupu best fit se bude snažit nalézt takový konec řádku, jehož badness bude minimální. Navíc zapojíme hodnotu penalt, které jsme v předchozím algoritmu použili pouze na vyznačení povinných a zakázaných míst zlomu (tedy přístup ano/ne). Zde již bude rozlišení jemnější: hodnotu penalty přičteme k badness řádku, a tím budeme schopni říci „toto je nevhodné místo pro zlom“. Uživatel takto bude moci vyjádřit svoje preference. Cenovou funkci rozšířenou o hodnotu penalt zde označujeme bp.

```
best-fit:
    začátek řádku = první prvek horizontálního seznamu;
    celková délka = celková roztažitelnost = celková stažitelnost = 0;
    prozatím nejlepší možné místo zlomu = undef;
    hodnota bp_s v prozatím nejlepším možném místě zlomu = ∞;
    while (je nějaký prvek v horizontálním seznamu)
        {
        if (je možným místem zlomu)
            {
            bp = badness b + případná penalta p;
            if (bp je nekonečné && |bp_s| konečné)
                {
                pošli na výstup řádek končící v prozatím nejlepším místě zlomu;
                nastav začátek dalšího řádku;
                nastav prozatím nejlepší místo zlomu na nedefinované;
                nastav bp_s na nekonečnou hodnotu;
                vypočti nové celkové hodnoty;
                }
            if (p vynucuje zlom)
                {
                proveď ukončení řádku na aktuálním prvku (kroky jako výše);
                }
            if (|bp| < |bp_s|)
                {
                prozatím nejlepší možné místo zlomu = tento prvek; bp_s = bp;
                }
            }
        zvyš celkovou délku a roztažitelnost a stažitelnost o hodnoty aktuálního prvku;
        }
```

Opět procházíme vstupní horizontální seznam. Pamatujeme si prozatím nejlepší místo pro ukončení aktuálního řádku a hodnotu bp_s, která udává krásu takto končícího řádku. Narazíme-li na lepší řešení ($|bp| < |bp_s|$), uložíme si ho. Pokud již takové místo není možno najít (bp je již nekonečné), vysázíme uložené nejlepší řešení. Pokud narazíme na penaltu vynucující zlom, vysázíme obdobně řádek ukončený na aktuálním prvku.

Stejně jako u first fit musíme tento stručný popis algoritmu rozšířit. Jednak definovat možné místo zlomu, jednak ošetřit přeskočení glue na začátku řádků. Také u tohoto algoritmu můžeme připojením glue na začátek či konec seznamu vysázet text na praporek vlevo či vpravo. V asymptotické složitosti algoritmu nedošlo oproti předchozímu first fit ke změně. Jediný rozdíl spočívá v přidání konstantní paměti pro uchování zatím nejlepšího řešení.

5.3. Optimum fit

Všechny algoritmy řádkového zlomu, které jsme dosud popsali, hledají v dané chvíli konec právě jednoho řádku. Poté, co je místo zlomu nalezeno, poznamenají si začátek následujícího řádku a opět k němu hledají co nejvhodnější konec. Řešením je pak posloupnost těchto „konců“, míst zlomu pro všechny řádky.

Nevýhoda tohoto přístupu spočívá právě v lokálnosti těchto rozhodnutí. Řešení, které jeden řádek vysází s nulovou hodnotou badness, může ztroskotat na následujícím řádku, zatímco kdyby se použilo řešení o málo horší, mohl by následující řádek vyjít mnohem lépe. A protože cílem sázecího procesu je dokument, v němž řádku i stránky působí stejnoměrně, je lepší takové zalámání, kde jsou všechny řádky mírně roztaženy, než to, v němž se střídají ideální řádky s řádky velmi zvětšenými či zmenšenými. Potřebujeme tedy algoritmus, který vybere nejlepší zlom celého odstavce. Rozumným kritériem je například minimální součet badness a penalt všech řádků v odstavci nebo minimální průměr této hodnoty.

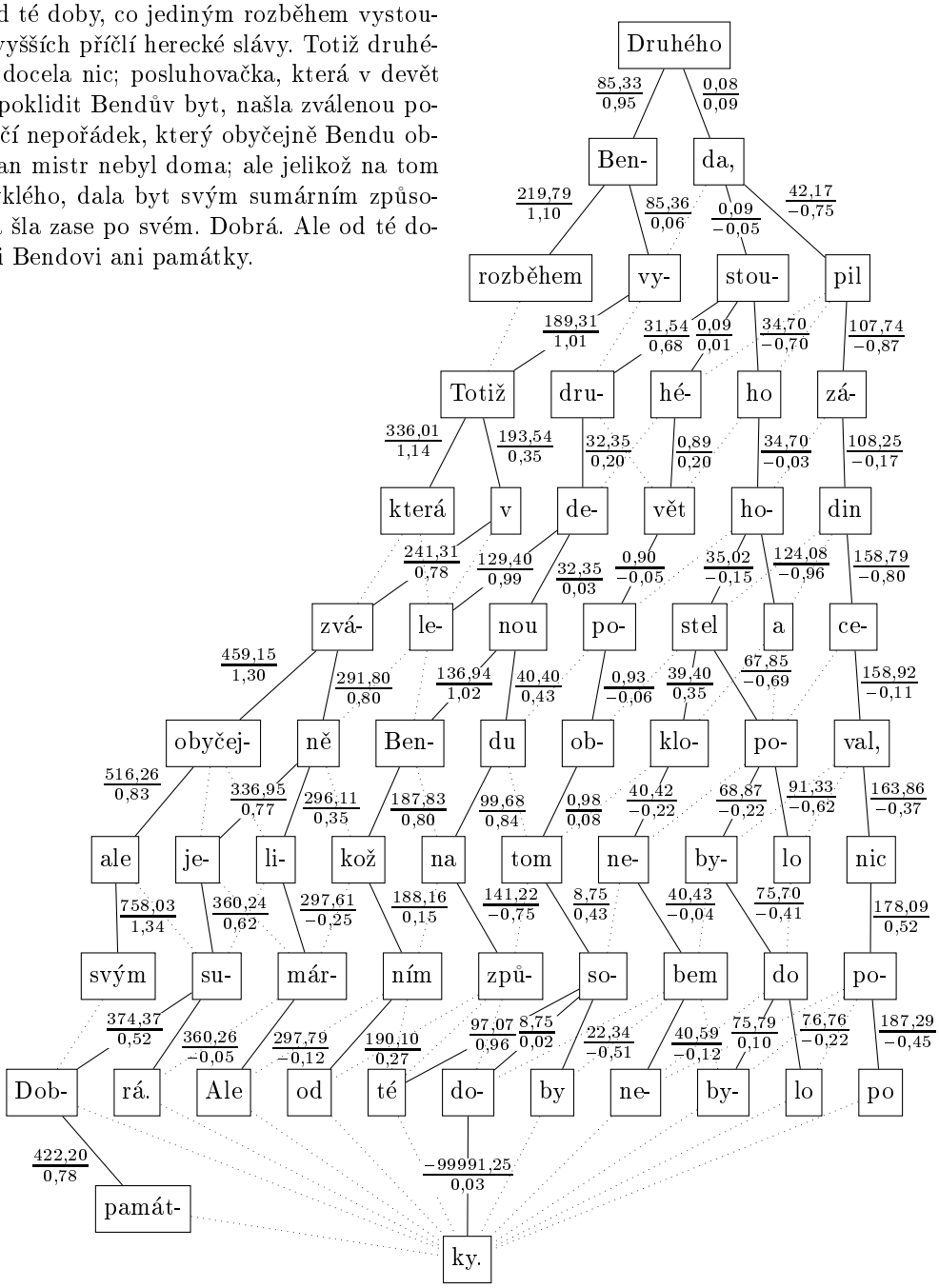
Algoritmus, který tuto hodnotu vypočte pro všechna možná zalámání odstavce, má exponenciální složitost, a proto se zaměříme na postup, jak vybrat z těchto 2^n řešení ta, která reálně mohou vést k rozumnému výsledku. Tento požadavek splňuje algoritmus optimum fit, jehož varianta je základem řádkového zlomu systému TEX.

```
optimum-fit:
    inicializuj seznam aktivních prvků;
    while (je nějaký prvek (x) v horizontálním seznamu)
        {
        if (x je možným místem zlomu)
            {
            for (všechny prvky a aktivního seznamu)
                {
                vypočti r, b a bp;
                if (r < −1 || p vynucuje zlom)
                    {   odstraň prvek a z aktivního seznamu;   }
                if (bp má přípustné hodnoty)
                    {   poznamenej si cestu z a do x;   }
                }
            if (byla nalezena alespoň jedna cesta z a do x)
                {   přidej x do aktivního seznamu s nejlepší cestou k předchůdci;   }
            }
        }
    vyber nejlepší cestu přes všechny zaznamenané body zlomu;
```

Při procházení míst zlomu ve vstupním horizontálním seznamu nehledáme konec pouze jednoho řádku, ale více řádků, jejichž začátky jsou uchovávány v **aktivním seznamu**. Nově nalezené konce řádků jsou do tohoto seznamu přidávány, protože jsou potenciálními začátky řádků dalších. Naopak z aktivního seznamu odstraňujeme ty prvky, které již nemohou nijak množinu řešení rozšířit, protože řádky v nich začínající by již bylo nutno zkrátit více, než dovoluje jejich stažitelnost ($r < -1$).

Výsledek činnosti algoritmu nejlépe ilustruje graf jeho řešení. Následující odstavec z knihy Karla Čapka Povídky z jedné kapsy [Čapek 64] byl vysázen desetibodovým $\mathcal{CS}$ písmem, šířka textu byla nastavena na 8,9 cm.

Druhého září zmizel herec Benda, mistr Jan Benda, jak se mu říkalo od té doby, co jediným rozběhem vystoupil na jednu z nejvyšších příčlí herecké slávy. Totiž druhého září se nestalo docela nic; posluhovačka, která v devět hodin ráno přišla poklidit Bendův byt, našla zválenou postel a celý ten kančí nepořádek, který obyčejně Bendu obklopoval, jenom pan mistr nebyl doma; ale jelikož na tom nebylo nic neobvyklého, dala byt svým sumárním způsobem do pořádku a šla zase po svém. Dobrá. Ale od té doby nebylo po herci Bendovi ani památky.



Uzly v grafu reprezentují možná místa zlomu, těmi jsou zde buď glue, kterým předchází slovo, nebo bod uvnitř rozděleného slova. Hrany mezi uzly označují potenciální řádky vytvořené mezi dvěma místy zlomu. Z každého uzlu vede nahoru jen jedna plná hrana, která je součástí plné cesty od počátku, horní číslo u hrany udává součet *bp* pro hrany v této cestě. Ta označuje nejlepší řešení, jak zalámat část odstavce od počátku k danému místu zlomu. Dolní hodnota u hrany znamená poměr roztažení či stažení r pro uvažovaný řádek.

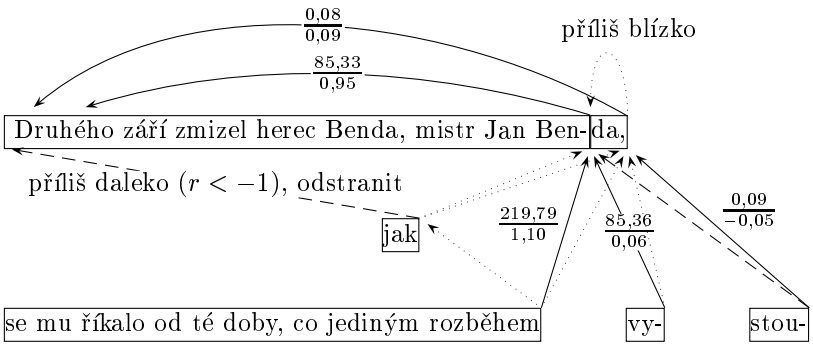
Řešením problému zalámání odstavce je cesta s nejmenším součtem *bp* od počátku k poslednímu elementu odstavce. Algoritmus vychází z metody **dynamického programování** (*dynamic programming*); postupně hledá řešení menších subproblémů a další jsou pak efektivně vypočteny jako rozšíření těchto výsledků. Pokud algoritmus nalezne nějaké subřešení, například místo zlomu za slovem „Totiž“ na začátku druhé věty, je tím již pevně dáno zalomení předchozích řádků. Zlom dalších nemůže žádným způsobem ovlivnit cestu od počátku k již zalámaným řádkům. Samozřejmě, že pokud nakonec vybereme zalomení třetího řádku uprostřed slova „druhé-/ho“, ztrácí řešení jdoucí přes „Totiž“ význam. Nicméně jeho subřešení je již známo a bylo využito při výpočtu zlomů za slovy „která“ a „v“.

Hrany vyznačené tečkovaně reprezentují další možné řádky, které ale byly zamítnuty, protože existují lepší řešení subproblému, která vedou k celkově lepšímu výsledku. Například řádek začínající za slovem „rozběhem“ není při dalším výpočtu uvažován, protože cesta jsoucí přes „vy-“ k „Totiž“ vykazuje nižší hodnotu cenové funkce *bp*. Tečkované hrany jsou možná řešení, která ale nejsou uchovávána pro další výpočet. Do množiny podřešení jsou uloženy ty částečné výsledky, které mohou generovat optimální výsledek celkový.

Při bližším zkoumání nás může zaujmout zlom „památ-“. Ten je na spojnici mezi „Dob-“ a koncem odstavce „ky.“ Zároveň ale vidíme, že existuje možný způsob, jak zalomit řádek za „Dob-“ a vytvořit jediný řádek od „rá“ až do konce odstavce. Nejenže zde existuje více možností, jak provést zalomení odstavce, existuje dokonce i více možností, co se počtu řádků týče. Pokud bychom totiž vybrali jako celkové řešení cestu jdoucí přes místo zlomu „památ-“, měl by odstavec 11 řádků místo stávajících deseti. Při sazbě delšího odstavce nebo do širšího zrcadla by těchto možností mohlo být samozřejmě ještě mnohem více.

Uzly jsou z aktivního seznamu odstraňovány v okamžiku, kdy je případný poměr stažitelnosti již menší než -1. Díky tomu je možno udržet teoreticky kvadratickou složitost na složitosti dn, kde d je počet prvků, které jsou v aktivním seznamu zároveň. V našem případě toto číslo nepřesáhlo hodnotu 12. Efektivitu výpočtu můžeme zvýšit i jinak: pokud je řádek z prvního prvku aktivního seznamu do aktuálního možného místa zlomu příliš krátký a b takového potenciálního řádku příliš vysoká, nemá smysl uvažovat ani tento řádek, ani řádky začínající v dalších prvcích horizontálního seznamu, protože ty by byly ještě kratší.

Na následujícím obrázku nastíníme výpočet pro několik prvních prvků:

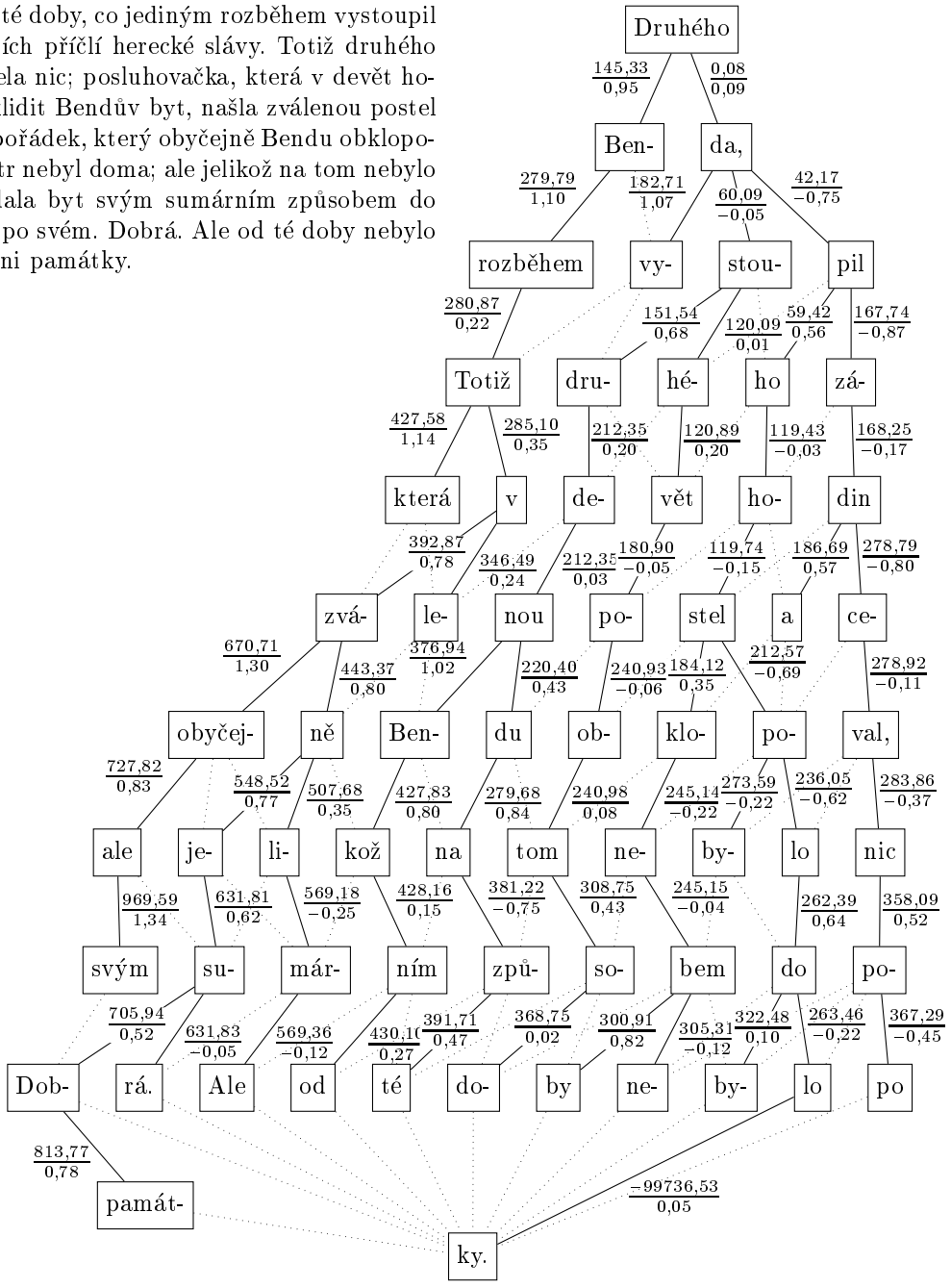


Nejprve je do aktivního seznamu vložen bod označující začátek odstavce. Vůči němu jsou porovnávána možná místa zlomu, ale až uvnitř slova „Ben-/da“ je nalezen takový potenciální

řádek, jehož hodnoty jsou přípustné — poměr roztažení je 0,95, badness je 85,33. Toto místo je přidáno do aktivního seznamu. Dalším možným místem zlomu je glue za „da,“ s poměrem roztažení vůči začátku odstavce 0,08, badness je 0,09. Místo za slovem „jak“ nepřidává další aktivní prvek, naopak počátek odstavce z aktivního seznamu odstraní, protože již nemůže generovat další subřešení (vyznačeno čárkovaně). Další změna v aktivním seznamu nastane až za slovem „rozběhem“, kdy přidáváme velmi roztažený řádek s poměrem 1,1. Místo dělení ve slově „vystou-/pil“ jednak přidává nové řešení s hodnotami $0{,}09/-0{,}05$, jednak odstraňuje z aktivního seznamu místo zlomu uvnitř slova „Ben-/da“.

Výše uvedené řešení je výsledkem běhu algoritmu pro případ, kdy nebyla nastavena žádná penalta za zlom v bodě dělení slova. Pokud tuto penaltu nastavíme na hodnotu 60, dáme tím najevo, že preferujeme řešení bez rozdělených slov a výsledek bude takovýto:

Druhého září zmizel herec Benda, mistr Jan Benda, jak se mu říkalo od té doby, co jediným rozběhem vystoupil na jednu z nejvyšších příčlí herecké slávy. Totiž druhého září se nestalo docela nic; posluhovačka, která v devět hodin ráno přišla poklidit Bendův byt, našla zválenou postel a celý ten kančí nepořádek, který obyčejně Bendu obklopoval, jenom pan mistr nebyl doma; ale jelikož na tom nebylo nic neobvyklého, dala byt svým sumárním způsobem do pořádku a šla zase po svém. Dobrá. Ale od té doby nebylo po herci Bendovi ani památky.



Počet řádků, zakončených rozděleným slovem, se zmenšil ze šesti na dva. Změna nastala hned na druhém řádku, kde je nyní poměr stažení 0,75, oproti předchozímu případu, kdy byl pouze 0,05. Celková hodnota *bp* na nejlepší cestě pro všechny řádky kromě posledního se zvýšila z 8,75 na 263,46. Je třeba si ale uvědomit, že 120 bodů z této hodnoty vzešlo z penalt za rozdělení dvou slov.

Tento graf zalámaného odstavce, ukazující obrovské množství možností pro zalomení odstavce, které jsou nalezeny jediným průchodem algoritmu optimum fit, byl uveden ve zprávě [Knuth 80] a stal se hlavním motivem této práce. Právě využití všech těchto řešení, která vznikají jako vedlejší produkt algoritmu optimum fit, je možností, jak obohatit funkce sázecího systému.

6. Stránkový zlom

Poté, co je text rozdělen na řádky, nastává další část úkolu při tvorbě dokumentu: rozložit tyto řádky na stránky. Jinými slovy, nalézt mezi řádky taková místa, v nichž bude aktuální strana končit a další začínat. Stránkový zlom si můžeme zjednodušeně představit jako zlom řádkový ve vertikálním smyslu, postavený na výšku. Opět je zde vstupní materiál, ve kterém jsou možná místa zlomu a my chceme z těchto míst vybrat ta, kde strany zalomíme.

6.1. Textový režim

V textovém režimu použijeme nejspíše vertikální podobu programu `fold`. Pak provedeme zalomení v okamžiku, kdy je na straně požadovaný počet řádků. Pokud máme na vzhled dokumentu vyšší nároky, použijeme pravděpodobně ekvivalent algoritmu `fold-space`. Ten zalomil řádek pouze v mezislovní mezeře, nikoli uvnitř slova. Mezislovní mezera pro něj byla možným bodem zlomu. Ve vertikálním smyslu můžeme za možný bod zlomu definovat například místo mezi odstavci, nebo každé místo mezi řádky kromě toho, které odděluje nadpis kapitoly od jejího textu.

V textovém režimu nemáme příliš velkou flexibilitu, vertikální vzdálenost mezi řádky či odstavci můžeme měnit pouze o celý násobek počtu řádků. Proto je u většiny dokumentů nejvhodnější přebytečné bílé místo vložit na dolní okraj strany a nesnažit se ho rozmístit mezi prvky na stránce, jak to v horizontálním režimu dělá algoritmus `line-buffer`.

6.2. Grafický režim

Při sazbě dokumentů v grafickém režimu se zvyšují možnosti, jak strany zalomit, především proto, že pozice jednotlivých řádků můžeme měnit velmi jemně, podobně jako pozice slov na řádcích. Tomu by měl odpovídat i formát dat, s nímž algoritmy budou pracovat. Opět zde použijeme model box-glue-penalty, poněvadž je dostatečně univerzální i pro popis těch algoritmů, které všech jeho vlastností nevyužijí.

Řádek je reprezentován horizontálním boxem, v němž jsou uloženy všechny jeho elementy, včetně mezer mezi nimi. Má šířku, výšku a hloubku, které jsou odvozeny z rozměrů jejich obsahu. Když skládáme řádky pod sebe a tvoříme stránku, obvykle chceme, aby **vzdálenost účaří** (*baseline skip*) byla konstantní, alespoň mezi řádky v odstavci. Horizontální boxy reprezentující řádky mohou mít ale různé hloubky, součet hloubky aktuálního a výšky následujícího řádku může být dokonce větší, než požadovaná vzdálenost účaří. Tyto rozdíly vyrovná algoritmus vložením vhodných glue mezi řádky. Stejně tak se glue použije ke správnému umístění prvního a posledního řádku na stránce.

Glue může samozřejmě vložit i přímo uživatel, stejně jako může použít penalty pro označení vhodných a nevhodných míst zlomu stránky. Vstupem algoritmu stránkového zlomu pak je **vertikální seznam** složený z boxů, glue a penalt, obdobně jako u zlomu řádkového. Analogický je také úkol: nalézt mezi možnými místy zlomu ta, která vytvoří co možná nejlepší dokument. Jeho krása, respektive nehezkost, je opět definována cenovou funkcí, a my se snažíme nalézt minimum této funkce.

Tím jsme převedli problém na předchozí případ — pro řešení můžeme použít patřičným způsobem zmodifikované algoritmy pro řádkový zlom. Jak first fit, tak best a optimum

fit jsou na takto definovaný problém vhodné. Hlavním rozdílem je několikanásobně větší paměťová náročnost těchto algoritmů, pokud je použijeme pro stránkový zlom. Jeden znak ze řádkového zlomu se nyní mění na jeden řádek, ekvivalentem odstavce je ve vertikálním smyslu kapitola či dokonce celá kniha. Proto je důležité zde pečlivě promyslet použité datové struktury, abychom se vyhnuli nadměrnému kopírování dat v paměti. Většinu úkolů je totiž možné řešit pomocí indexů do pole prvků.

Pokud použijeme tento přístup, musíme odpovídajícím způsobem změnit také část řádkového zlomu. Tam jsme často použili formulaci „zapiš řádek na výstup“, ve skutečnosti ale vždy jde o uložení dat do paměti, neboť nad nimi budou prováděny ještě další operace. K vlastnímu výstupu dat z programu dojde až po zformátování celé stránky, případně stránek.

Jestliže netrváme na optimalizaci vzhledu celé kapitoly či dokumentu a stačí nám optimalizace jedné strany, je nejvhodnější použít algoritmus best fit. Test ve stránkovém zlomu pak voláme vždy, když od řádkového zlomu dostaneme další řádek, glue či jiný materiál. Pouze v případě vysokých nároků na stejnoměrné rozložení textu použijeme ekvivalent přístupu optimum fit, který nalezne nejlepší řešení v rámci delší části dokumentu. Rozumným kompromisem mezi nejvyšší kvalitou a výpočetními nároky může být také optimalizace protilehlých dvoustran knihy. Vzhledem k tomu, že je má čtenář vedle sebe, i malá rozdílnost by byla velmi vidět a je vhodné jí pokud možno zabránit.

6.3. Plovoucí objekty

Praktický rozdíl mezi řádkovým a stránkovým zlomem nespočívá pouze v orientaci lámaného materiálu. Do zlomu stránkového totiž může v danou chvíli vstupovat více než jeden vertikální seznam prvků. Při sazbě vědeckých a technických typů dokumentů se velmi často používají poznámky pod čarou, tabulky či obrázky, které nejsou v textu pevně umístěny (viz. část 2.3.). Tyto prvky jsou při řádkovém zlomu ze základního textového materiálu vyjmuty a zpět se vrací až při zlomu stránkovém, při vlastní kompozici strany. Z textového seznamu vedou odkazy do ostatních vertikálních seznamů a takto propojené elementy je vhodné umístit co nejblíže k sobě, v ideálním případě na stejnou stranu.

U některých typů plovoucích objektů je tento požadavek velmi striktní, například poznámka pod čarou musí začínat na stejné straně, kde je odkazována, byť by její zbytek pokračoval na následující straně. Naopak fotografii, která zabírá dvě třetiny výšky tiskového zrcadla, může sázecí systém odsunout na další stranu či na konec kapitoly. Různé typy plovoucích objektů můžeme také rozdělit na skupiny podle toho, zda požadujeme, aby bylo zachováno jejich pořadí v rámci jednoho typu. U poznámek pod čarou je tento požadavek velmi silný, u tabulek či velkých obrázků již na pořadí může záležet méně. Tato omezení by měl být algoritmus schopen zpracovat, a to nejen v negativním smyslu. Pokud je možné změnou pořadí obrázků dosáhnout lepšího celkového vzhledu dokumentu, program by měl této možnosti využít.

Za základní element ve stránkovém zlomu obvykle považujeme řádek, případně jeden plovoucí objekt. Proto pokud vede odkaz z textu na poznámku pod čarou, stává se z něho po provedení řádkového zlomu odkaz z řádku na plovoucí element. Tyto odkazy tedy vzájemně svazují řádky základního textu a doplňující materiál.

Chceme-li vertikální verzi algoritmů first, best a optimum fit rozšířit o plovoucí objekty, nejsnáze to uděláme tak, že prvky svázané odkazem budeme při testech uvažovat společně. Testy ve zmíněných algoritmech jsou obvykle typu „je možné přidat na stranu ještě jeden řádek“, případně „jaká bude badness, pokud přidáme ještě tento element“. Řádek, který

odkazuje na poznámku pod čarou, můžeme pro potřeby lámacího algoritmu brát jako jeden řádek, který zabere hodně vertikálního místa.

S tímto přístupem vystačíme ale jen u drobných prvků. Půlstránkovou tabulku takto ošetřit nemůžeme, protože by výsledkem bylo půlstránkové volné místo, pokud by se tabulka nevešla na stejnou stránku jako řádek, který se na ni odkazuje. K takovýmto objektům můžeme přiřadit dodatečnou penaltu, která zvýší hodnotu cenové funkce stránky, pokud se plovoucí objekt nedostane na tu stranu dokumentu, odkud je odkazován. Algoritmy best či optimum fit pak k badness, založené na roztažení a stažení vertikálních glue na straně, a k případným explicitním penaltám připočtou patřičné penalty za všechny plovoucí objekty, které musely být odsunuty na další strany. Jinou možností je definovat cenovou funkci, která vyjádří, jak moc jsou ve výsledném dokumentu odkazované věci vzdálené od svých odkazů [Plass 81]. Při tomto přístupu je vhodné zohlednit také fakt, že obrázek umístěný na protilehlé straně dvoustrany v knize způsobí čtenáři menší problémy než obrázek za hranou listu.

Plovoucí objekty mají také různé požadavky na zachování vzájemného pořadí v dokumentu. Můžeme chtít, aby se objevily ve výstupním dokumentu v takové posloupnosti, v jaké byly zadány na vstupu, můžeme také objekty rozdělit do více kategorií a pak zajišťovat zachování pořadí pouze v rámci jedné kategorie. Z tohoto rozhodnutí se pak odvíjí způsob, jak při výpočtu výšky stránky ve stránkovém zlomu započítáváme výšky plovoucích objektů, které se na aktuální straně mohou a nemusí objevit.

Protože počet řádků a ostatních objektů, které mohou být umístěny na jednu stranu, není obvykle příliš velký (50 řádků na stranu formátu A4), nabízí se zde i metoda **hrubé síly** (*brute force*). Můžeme vyzkoušet všechny kombinace počtů řádků textu a plovoucích objektů a vybrat z nich tu nejlepší. Při implementaci se můžeme inspirovat přístupem dynamického programování v algoritmu optimum fit a ve výpočtu uvažovat jenom ta subřešení, která potenciálně mohou vést k výsledku s akceptovatelnou badness [Klein 95].

Z předchozího textu jsme viděli, že základním úkolem algoritmu stránkového zlomu v sázecích systémech je rozhodnutí, které elementy budou umístěny na kterých stranách. Jejich vlastní umístění se provádí až je určena příslušnost textových a plovoucích objektů k jednotlivým stranám. V TEXu se o to stará **výstupní rutina** (*output routine*). Ta zajistí správnou pozici základního textu a podle typů plovoucích objektů i jejich odpovídající rozmístění.

7. Implementace použitá v TEXu

Sázecí systém TEX je typickým dávkovým systémem. Z toho vyplývají i jeho rysy a konkrétní algoritmy v něm použité [Knuth 84], [Olšák 96].

7.1. TEX z pohledu uživatele

TEX čte vstup z textového souboru, typicky s příponou `.tex`. V tomto souboru je uložen jednak vlastní text sázeného dokumentu, jednak řídící sekvence makrojazyka TEXu, které popisují vzhled dokumentu. Tyto řídící příkazy je nutné rozpoznat, nalézt a dosadit jejich argumenty a provést jejich (mnohdy velmi složitou) expanzi. Tuto činnost má na starosti vstupní **analyzátor** (*parser*). Expanze maker se provádí až na úroveň tzv. primitiv, která jsou v systému pevně dána a která jsou jedinými příkazy, které dokáží provést nějakou akci. Ostatní makra jsou pouze nadstavbou nad těmito primitivy. Laický uživatel pak například na začátku nové kapitoly píše `\kapitola` a na jaké primitivní příkazy se toto makro rozexpanduje a jaké vnitřní akce způsobí, ho nemusí příliš zajímat.

Ve vstupním textu můžeme použít i příkaz pro načtení obsahu jiného souboru, takže můžeme využít mnoha maker a balíků maker, které napsali jiní lidé. Protože tato makra by bylo nutné načít a rozexpandovat při každém dávkovém zpracování, podporuje TEX i práci s předkompilovanými formáty, kdy načítá již vnitřní binární reprezentaci rozexpandovaných maker.

Data zpracovaná vstupním analyzátorem jsou posloupností textu a primitivních typografických příkazů, nebo-li čistého typografického materiálu, který má za základ model box-glue-penalty. Práce TEXu je založena na skládání boxů do sebe a vedle sebe. Glue a penalty, stejně jako hodnoty četných parametrů ovlivňují způsob, jakým budou tyto boxy vytvořeny a kde se budou jednotlivé typografické elementy ve výsledném dokumentu nacházet.

Typografický materiál může mít dvě základní podoby: horizontální a vertikální. Toto jsou také dva z režimů práce, ve kterých se TEX může nacházet. Odstavec je tvořen v režimu horizontálním. Prvky horizontálního seznamu, který se láme do řádků za použití optimum fit algoritmu, jsou jednoho z následujících typů:

- box, případně rule, což je termín pro obdélníkovou oblast vyplněnou černě;
- vertikální materiál zadaný například pomocí `\insert` pro plovoucí objekty, `\vadjust` pro vertikální elementy upoutané k řádku a `\mark` pro slovníkové stránkové značky;
- místo dělení — můžeme zadat, jaký bude text před a za rozděleným místem a jaký v případě, že k rozdělení v daném bodě nedojde; použítá se primitivum `\discretionary`;
- speciální značky (`\special`);
- glue, případně příkaz pro opakované výplně (`\leaders`);
- kern — pevné glue;
- penalta;
- značka pro začátek a konec matematického módu.

Lámací algoritmus je oproti našemu popisu (viz. 5.3.) rozšířen o několik funkcí: testuje, zda po sobě nenásledují dva řádky, které by se velmi lišily svým poměrem roztažení, resp. stažení. Pokud ano, automaticky přičítá k cenové funkci v místě zlomu dodatečnou hodnotu `\adjdemerits`. Podobně můžeme například omezit výskyt dvou řádků končících rozděleným slovem nastavením vysoké hodnoty `\doublehyphendemerits`.

Algoritmus pracuje až ve třech průchodech. Nejprve je odstavec zalámán s normálními mezerami a bez dělení slov. Pokud není nalezeno řešení v optimálních mezích, provede se druhý průchod s rozdělenými slovy. A při případném třetím pokusu se navíc zvětšuje roztažitelnost mezislovních mezer o hodnotu `\emergencystretch`; tím povolíme i řádky velmi roztažené.

TEX poskytuje uživateli několik způsobů, jak sázet i do jiné než standardní šířky odstavce `\hsize`. Pomocí maker `\leftskip` a `\rightskip` definujeme glue, které je při lámání přidáno ke každému řádku, `\parshape` dovoluje zadat odsazení a délku každého řádku v odstavci zvlášť a makra `\hangindent` a `\hangafter` slouží ke specifikaci obdélníkových výřezů do tvaru odstavce. Dalšími důležitými parametry jsou `\parindent` a `\parfillskip`, kterými můžeme zadat velikost bílého místa na začátku a na konci odstavce. Počet řádků se můžeme pokusit změnit pomocí `\looseness`.

V TEXu je cenová funkce řádku počítána složitěji než v našem případě, kde jsme pouze sečetli badness b a penaltu p. Zde se chyba, vada řádku označuje termínem **demerits**. Hodnota demerits d je rovna $(l+b)^2+p^2$ pro $0 \leq p < 10000$, $(l+b)^2-p^2$ pro $-10000 \leq p < 0$, a $(l+b)^2$, pokud je p menší než -10000. Proměnná l zde označuje hodnotu `\linepenalty`, která je přidána ke každému zlomu. Takto můžeme jemně regulovat, zda má mít výsledek spíše více či spíše méně řádků. Standardní hodnota je nastavena na nulu. Proměnná p znamená penaltu svázanou s daným koncem řádku a b je badness řádku.

Zlom ovlivňují i hodnoty dalších parametrů. Například `\hyphenpenalty` se přičítá u každého řádku, který končí na rozděleném slově. Hodnota `\finalhyphendemerits` je připočtena, pokud rozděleným slovem končí předposlední řádek odstavce. Vzhledem k tomu, že poslední řádek již většinou nezabírá celou šířku, je rozdělené slovo v předposledním řádku velmi viditelné a může působit rušivě.

Při zlomu se uplatňuje interní proměnná `\prevgraf`, ve které je uchován počet řádků naposledy lámaného odstavce. Využívá se, pokud jsem specifikovali nestandardní tvar odstavce pomocí `\hangindent`/`after` nebo `\parshape` a jeho změnou můžeme dosáhnout korektního zlomu okolo rovnic či dlouhých obdélníkových výřezů v textu.

Výstupem algoritmu řádkového zlomu je vertikální posloupnost hboxů, z nichž každý reprezentuje jeden řádek textu. Mezi těmito hboxy může být navíc materiál, který „vyplul“ z řádků (zadaný makrem `\vadjust`) a bude také zařazen do vertikálního seznamu, a také boxy, které vznikly například jako výsledek sazby tabulek či rovnic (*displayed equations*). Boxy jsou prokládány glue, kterými se zajistí správná vzdálenost účaří. Normálně je mezi řádky vloženo glue, jehož velikost je rovna hodnotě `\baselineskip` bez hloubky předchozího a výšky následujícího boxu. Pouze pokud by se takto boxy k sobě přiblížily na méně než `\lineskiplimit`, je mezi ně vložen `\lineskip`. Před glue jsou do vertikálního seznamu umístěny ještě penalty, kterými se označují některá speciální místa v odstavci. Hodnota `\brokenpenalty` následuje každý řádek končící rozděleným slovem, `\clubpenalty` a `\widowpenalty` jsou přičteny za prvním a před posledním řádkem pro správné ošetření parchantů. Pomocí `\interlinepenalty` můžeme zvýšit či snížit vhodnost stránkového zlomu uvnitř odstavce.

Při změně vertikálního seznamu je opakovaně přepočítávána výška potenciální strany, spolu s hodnotami roztažitelnosti a stažitelnosti. Pokud se navíc jedná o možné místo zlomu, je vypočtena cenová funkce (demerits) pro takto zalomenou stranu. Pokud je nalezen bod, kde je již strana příliš dlouhá a bylo by ji nutno zkrátit o více než její roztažitelnost, je výstupní rutině předán box číslo 255, v němž jsou prvky, které budou obsahem strany. Stejně tak je vyvolána výstupní rutina, pokud narazíme na penaltu, která vynutí zlom, například před začátkem další kapitoly knihy.

TEX podporuje plovoucí objekty díky primitivu `\insert`. Pomocí něho, případně s využitím složitějších maker z formátů, vložíme do vertikálního seznamu pro algoritmus lámání stránky materiál plovoucího objektu. Ten je příslušný jedné ze tříd plovoucích objektů, takže můžeme odlišit poznámky pod čarou od celostránkových obrázků. TEX se snaží na aktuální stranu dostat co nejvíce z plovoucích objektů, pokud se některé z nich nevejdou, pokusí se je vložit na některou z dalších stran. Pro třídu n je podstatný `\box`n, ve kterém je výstupní rutině předán ten materiál dané třídy, který má být vložen do aktuální strany, dále pak `\dimen`n, kterým můžeme shora omezit vertikální místo, které daná třída na straně zabere, a `\skip`n, ve které zadáme dodatečné místo před prvním objektem dané třídy na straně. Parametrem `\count`n můžeme určit poměr, v jakém plovoucí objekt ovlivní výšku strany.

Předchozí popis se týkal **vnějšího** vertikálního módu, kdy TEX vytváří obsah strany. Podobné činnosti se provádějí i v módu **vnitřním**, který je nastartován, pokud použijeme primitivum `\vbox`. Tím se spustí vytváření vertikálního boxu a pravidla jsou při tom podobná, jako tvorby hlavního vertikálního seznamu stránky. Omezení se vztahují na plovoucí objekty a vnitřní vertikální mód nepodléhá automatickému stránkovému zlomu. Místo toho můžeme použít primitivum `\vsplit`, které z vboxu oddělí část zadané výšky. Takto je možné činnost stránkového zlomu simulovat.

Úkolem výstupní rutiny (`\output`) je vlastní sestavení strany. Rutina podle pokynů uživatele nebo s použitím standardních hodnot formátu připojí horní a dolní záhlaví, vysází boxy řádků a ostatního materiálu z `\box`u 255, vloží do strany všechny plovoucí objekty, které na ní mají být, a správně je umístí. Vytvořená strana je nakonec zapsána do výstupního souboru, který má příponu `.dvi` (DeVice Independent).

Vzhledem k tomu, že makra prováděná ve výstupní rutině jsou dostupná uživateli, můžeme zde naprogramovat i jinou činnost než jen sestavení strany a její zapsání do souboru. Ve výstupní rutině se například ošetřuje pomocná informace o křížových odkazech a ostatních pozicích v dokumentu, které jsou známy až poté, co je dokument rozdělen do stran. Rutina dokonce nemusí data do souboru zapsat, ale může s nimi naložit zcela libovolně dle uvážení uživatele. Například při vícesloupcové sazbě je pro každý sloupec textu vytvořena jedna „podstrana“, ale tato data nejsou poslána na výstup, dokud nejsou nastřádány všechny sloupce výsledné fyzické strany. Toto pozdržení výstupu dovoluje i další manipulace s textem, například vyrovnání délek sloupců na poslední straně knihy či kapitoly [Olšák 95].

7.2. Omezení stávající TEXovské implementace

TEX byl svým autorem vytvořen jako velmi specializovaný nástroj pro sazbu několikasvazkového díla The Art of Computer Programming [Knuth 73]. Tomu odpovídají i možnosti, které jsou v něm zahrnuty: propracovaná podpora sazby matematických výrazů, kvalitní řádkový zlom, který umožňuje dosáhnout dobrých výsledků i při sazbě textů s „nezlomitelnými“ matematickými formulemi, podpora plovoucích objektů pro automatické umísťování poznámek pod čarou, tabulek a obrázků. Velkou předností je rozšiřitelnost a flexibilita sázecího systému díky velmi mocnému makrojazyku a desítkám možných parametrů, a samozřejmě také nezávislost na výstupním zařízení a kolekce Computer Modern fontů. To vše přispělo k rozšíření TEX mezi uživatele sázející podobné typy textů jako prof. Knuth, tedy hlavně do akademického prostředí. A protože je to nejen nástroj pro sazbu technických zpráv, ale díky svým makrům velmi obecný programovací systém, je dnes těmito uživateli velmi oblíben i při sazbě jiných dokumentů, ať již hladké sazbě beletrie, tvorbě formulářů nebo technických diagramů [Knuth 88].

Právě potížemi, do kterých se uživatelé často dostávají při sazbě hladkých textů, se budeme nyní zabývat. Tyto dokumenty využijí velmi málo z možností TEXu, neboť se v nich obvykle nevyskytují ani matematické rovnice, ani obrazce a grafy, při jejichž tvorbě bychom ocenili volnost při skládání a kombinování boxů nebo načítání a analýze dat z databází. Uživatelé zde obvykle chtějí dostat s co nejmenší námahou na výstup mnoho stránek textu, který je tvořen odstavci s pevným řádkovým rejstříkem.

Prvním problémem bývá neuspokojivé zalomení strany. V dokumentu, ve kterém není dostatek volnosti ve vertikálních mezerách, se může objevit parchant navzdory tomu, že `\clubpentaly` i `\widowpenalty` byly nastaveny na velmi vysokou hodnotu. Případně TEX stranu zalomí podtečenou či přetečenou.

Důvodem je způsob, jakým TEX přistupuje ke stránkovému zlomu. Předpokládá totiž, že mezi řádky textu najde dostatek dodatečného glue, které bude moci použít k odstranění špatných míst zlomů stran. Na rozdíl od technických textů, kde díky přítomnosti rovnic či grafů tento předpoklad oprávněný, u hladké sazby může systém pracovat pouze s pevných řádkovým rejstříkem, kde glue žádnou roztažitelnost ani stažitelnost mít nemusí. Podívejme se, jak bychom s pomocí TEXu řešili například situaci, kdy ke zlomu dojde za prvním či před posledním (východovým) řádkem odstavce, jako je tomu v následující ukázce z Čapkovy knihy:

. . .

po strop ověšený samými obrazy a oddaně miloval herce Bendu, který k němu choval přátelské opovržení a s jistou blahosklonností mu dovoloval, aby za něj platil; což, mezi námi, nebyla zrovna maličkost. Tragická maska Bendova a zářící obličej doktora Goldberga (který pil jenom vodu), to už patřilo k sobě při všech těch sardanapalských flámech a divokých výtržnostech, které byly druhou stránkou proslulosti velkého mima.

Tedy tomu doktoru Goldbergovi telefonovali z divadla, co prý se děje s Bendou, Odpověděl, že nemá ponětí, ale že se po něm podívá; co neřekl, bylo, že už ho sám po celý týden shání po všech nočních lokálech a výletních hotelech s rostoucím znepokojením; měl tísnivou předtuchu, že se něco Bendovi stalo. To bylo totiž tak: Doktor Gol-

. . .

Toto je reálný případ, se kterým se můžeme setkat velmi často. Protože u hladké sazby známe vzdálenost účaří a výšku strany, můžeme snadno spočítat počet řádků na jednu stranu. Tím jsou dána místa stránkového zlomu mezi řádky. Co se ale stane, pokud stránkový zlom padne na nevhodné místo? V TEXu existují dvě řešení:

Nastavit parametry tak, aby v takovém případě systém zkrátil aktuální stranu o řádek, pak ale mohou mít strany na jedné dvoustraně různý počet řádků. Druhá možnost je do některého z předchozích odstavců vložit příkaz `\looseness`. Ten způsobí, že pokud je to možné, je odstavec prodloužen či zkrácen o zadaný počet řádků. Typicky bychom tedy napsali `\looseness=1`. Pro takovou operaci musíme vybrat odstavec s dostatečně dlouhým východovým řádkem, kde je pravděpodobné, že systém bude schopen splnit náš požadavek bez toho, aby to příliš narušilo vzhled strany.

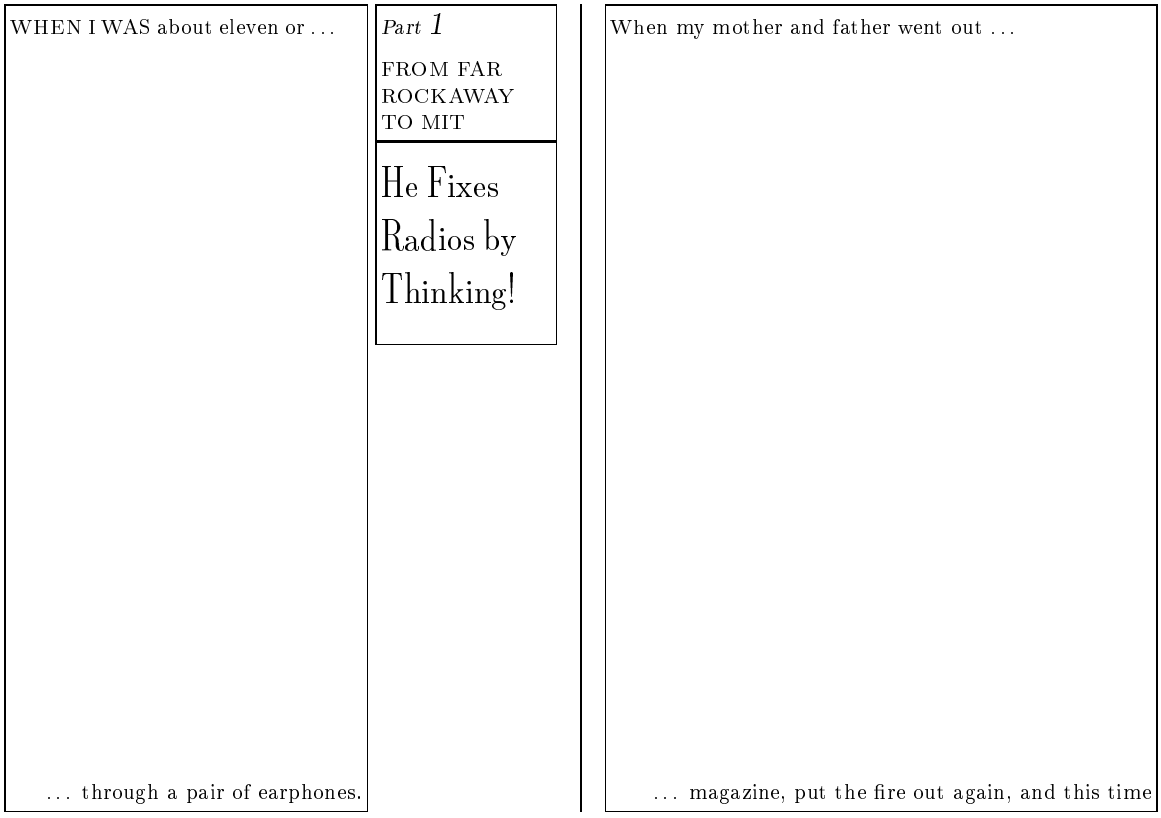
Možnost, kterou předpokládal autor TEXu, změnit řádkování mezi řádky odstavců nebo alespoň vzdálenost mezi nadpisy a textem, nepřipadá při požadavku na pevný řádkový rejstřík v úvahu. Pokud chceme sázet kvalitní knihu, nemůžeme dovolit ani to, aby se změnil počet řádků na straně. Zbývá tedy využít možnost s `\looseness`. Pokud sázíme předem daný text, není problém do textu tímto způsobem manuálně zasáhnout. Pokud ale používáme TEX pro psaní nového dokumentu, kdy se vracíme k předchozím částem a měníme je a přepisujeme a TEX spouštíme opakovaně, abychom viděli, jak bude vypadat výsledný dokument, je toto řešení nevyhovující. Příkaz na prodloužení řádku, který jsme vložili dříve, přestane být po změně textu nutný nebo může mít efekt zcela opačný, začne vhodnému zlomu strany dokonce bránit. Potom si musíme pamatovat, proč jsme které `\looseness` do kterého místa vložili a znovu a znovu rozhodovat, zda je či není nadále potřeba.

Samozřejmě, odpověď profesionálních autorů nebo nekritických zastánců TEXu zní, že sázet se má, až je text úplně hotový a bez chyb, tento argument ale u většiny uživatelů, kteří chtějí se systémem pracovat víceméně interaktivně, neobstojí. Koneckonců i prof. Knuth několikrát opakoval, že autor by měl v případě problémů lámacímu systému pomoci a například změnit pořadí slov v textu. Jeho poznámka se ale týkala více řádkového zlomu než stránkového, kde se v matematických textech obvykle dostatečně pružná glue mezi řádky najdou.

Druhým případem, se kterým se můžeme při sazbě textů s malou či žádnou vertikální flexibilitou setkat, je zlom strany na rozděleném slově. Předpokládejme, že by v ukázce výše byl text na levé straně o řádek delší a zlom by tudíž místo za slovem „stránkou“ padl doprostřed slova „flá-/mech“. Rozdělené slovo na hranici strany není považováno za vhodné, protože ztěžuje situaci čtenáři, který zvláště při obracení strany potřebuje co nejplynulejší přechod. Podobně je tomu i se slovy kratšími než pět písmen, ta by se také na konci strany měla objevovat co nejméně.

K odstranění tohoto typu zlomů TEX uživateli nedává žádný nástroj. Z principu ani nemůže, protože v okamžiku, kdy se láme strana, pracuje již program s posloupností hboxů, u kterých je zcela skryt jejich obsah i (například) informace o tom, zda ten který hbox vzešel ze zalámaného odstavce nebo byl explicitně vytvořen uživatelem. Pokud nám záleží na kvalitním výstupu, musíme i zde sáhnout k ručnímu řešení a buď použít `\looseness` nebo uvnitř odstavce zakázat v dané oblasti řádkový zlom a doufat, že nově nalezené řešení bude lepší.

Jiný zajímavý úkol je sazba do různě širokých sloupců, například v novinách a časopisech, případně sazba na různě široké stránky. V knize Richarda Feynmana Surely You're Joking, Mr. Feynman [Feynm 92] jsou kapitoly řešeny následujícím způsobem:

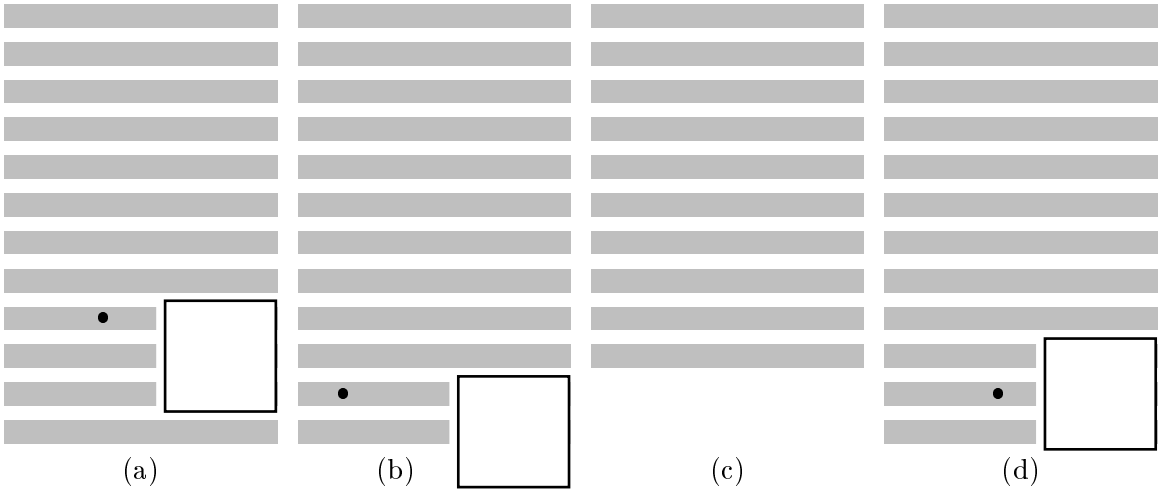


První strana každé kapitoly je rozdělena vertikálně, nadpis kapitoly je nahoře v pravém sloupci, vlastní text pak ve sloupci levém, který zabírá asi dvě třetiny šířky strany. Všechny další strany kapitoly pak pokračují již textem přes celou šířku strany. Jak bychom sázeli takovýto dokument v TEXu? Museli bychom spočítat, nejsnáze jedním testovacím průchodem, za kterým řádkem posledního odstavce na první straně kapitoly dojde ke stránkovému zlomu a pak správně pro tento odstavec nastavit `\hangindent` a `\hangafter`, za ním pak změnit hodnotu `\hsize`. Opět zde jde o ruční zásah do textu dokumentu, který se při případných úpravách textu, kdy dojde k posunu bodů zlomu stran, může stát zbytečnou přítěží.

Další otázka, se kterou se začínající TEXisté často obracejí na své zkušenější kolegy, se týká dodatečných objektů, které jsou umístěny vedle textu. Z výše uvedeného popisu TEXovské implementace je zřejmé, že pravý plovoucí objekt je možno do dokumentu vložit pouze na vertikální úrovni. Pro odstavec, do jehož výřezu má být vložen obrázek, použijeme nejspíše `\hangindent`/`after`, případně `\parhshape`. Tyto ale mají rozsah platnosti omezen pouze na jeden odstavec. Pro vyšší objekty je nutno ošetřit každý zasažený odstavec zvlášť, případně si napsat makro, které toto hlídání provede za nás. Pokud se vydáme cestou ručního zformátování, budeme muset stále kontrolovat, zda se nezměnil text před touto stranou a zda jsou naše vložené příkazy stále ještě platné, podobně jako u výše uvedených parchantů a `\looseness`.

Programování maker na této úrovni vyžaduje již značné znalosti, a proto se často používají k tomu účelu vytvořené styly různých formátů, například LATEXu. To nás ale obvykle omezuje na použitý formát a často se také stává, že takové řešení koliduje s našimi uživatelskými makry. Jak styly, tak formáty jsou totiž vytvořeny pomocí obecného mechanismu maker a proto se příkazy různých autorů mohou často nepříjemně ovlivňovat. Tento problém je vlastně rozšířením předchozího úkolu sazby do různě širokých stran o dynamický výpočet šířky textu, kterou známe až poté, co jsou umístěny plovoucí objekty.

Poznamenejme, že problém plovoucích objektů není uspokojivě dořešen ani v žádném z komerčních produktů. Většina z nich spoléhá na to, že uživatel myší určí místo vloženého obrázku či tabulky a text ho potom obteče. Takové objekty ale nejsou v pravém smyslu slova plovoucí. Jindy je možno definovat plovoucí objekt a dokonce ho svázat s textem a jeho pozice je pak určena dynamicky (obrázek a), na zlomu strany ale dojde téměř vždy k problémům:



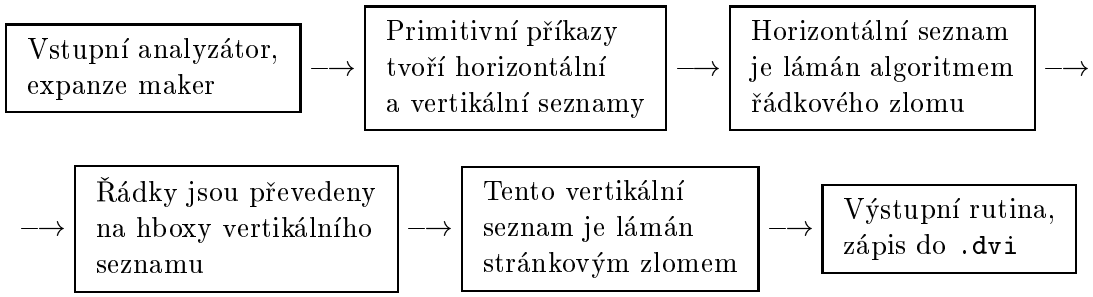
Objekt je buď vysazen i pod hranici stránky (b), nebo je odsunut na další, ovšem s tím, že na předchozí straně zůstane volné místo (c). Obvykle ale spíše chceme, aby byl objekt

vysazen dovnitř strany, byť s tím, že nebude shora zarovnán se řádkem (d), ze kterého je odkazován — toto místo je znázorněno černou tečkou. Anebo můžeme chtít objekt odsunout na další stranu, ovšem s tím, že text na aktuální straně musí plynule pokračovat. Odkaz zde sice vede přes hranici strany, koneckonců jde ale o objekt plovoucí.

Podobně neuspokojivě řeší komerční produkty i ukotvení vůči straně nebo sloupci. Dokud je výsledná pozice vloženého objektu uvnitř strany, je výsledek dobrý. Pokud se ale dostane blíže k místu stránkového zlomu, je nejlepší použít myš a pozici pevně zafixovat.

7.3. Důvody těchto problémů

Případy, které jsme uvedli, mají společný jeden rys: teprve při stránkovém zlomu víme definitivně, jak by měl být vlastně proveden zlom řádkový. Při zlomu stránkovém totiž nalézáme nová omezení a nové podmínky, které nebyly ve chvíli, kdy TEX lámal data z odstavce, známy. Systém se při stránkovém zlomu spoléhá na dostatečnou vertikální flexibilitu dokumentu, kterou ale u hladkých textů nenajdeme, a také předpokládá konstantní šířku textu, kterou povoluje snadno změnit jen lokálně pro jednotlivé odstavce. Jeden sázecí průchod TEXem můžeme znázornit následujícím diagramem:



V tomto schématu jsme nezmínili například tři možné průchody řádkového lámacího algoritmu, který volá algoritmus dělení slov podle vzorů, ani fakt, že výstupní rutina může provádět i jiné akce než jen zápis strany do souboru. Hlavní pozornost totiž zaměříme na krok mezi řádkovým a stránkovým zlomem. Z bohatého stromu řešení problému zalomení odstavce do řádků je vybráno jediné a řádky jsou zformovány do horizontálních boxů, které jsou posléze poskládány pod sebe a odděleny pomocí glue.

TEX v tomto okamžiku tvoří dlouhou vertikální posloupnost, jakoby jedinou stránku. Teprve poté je tento dlouhý seznam rozdělen na výsledné strany, případně sloupce. Na tom nemění nic ani ten fakt, že lámání strany probíhá v čase paralelně se zlomem řádkovým, vždy, když přidáváme do vertikálního seznamu další materiál. Přesun dat od řádkového zlomu ke stránkovému je totiž jednosměrný, stránkový zlom nemá již možnost rozhodnutí řádkového zlomu změnit. Pokud se tedy objeví při kompletaci strany nové omezení či problém, například parchant, který způsobí při nedostatečné vertikální flexibilitě strany přetečený nebo nenaplněný vertikální box strany, musí se tato situace řešit na úrovni vertikální. Řádkový zlom, který by mohl v mnoha případech pomoci, již není k dispozici, nemůžeme se k němu vrátit.

Stránkový zlom je také omezen tím, že je mu skryta informace o původním obsahu horizontálních seznamů, které tvoří odstavce a které byly zalámány do hboxů. Proto je velmi těžké přesně zjistit, jak původně vypadal zalámaný odstavec, či zda hbox vznikl jako výsledek práce řádkového zlomu nebo příkazem uživatele. V TEXu sice existuje makro `\unhbox`, to ale vrátí pouze obsah již zarovnaného boxu, nikoli souvislosti jeho vytvoření.

Styly, která se pokoušejí tato omezení obejít na úrovní maker, nejčastěji spoléhají na makro `\vsplit`, kterým rozdělí vbox a hledají nejlepší řešení. Tím suplují činnost, kterou by mohl mnohem efektivněji dělat stránkový zlom. V další kapitole ukážeme, že vhodnou změnou datových struktur, které jsou interními lámacími algoritmy používány, můžeme schopnosti sázecího systému rozšířit i o funkce, které je dnes nutno řešit na úrovni maker či manuálním zásahem do vstupního textu dokumentu.

8. Rozšíření algoritmů o vzájemnou spolupráci a nové funkce

Hlavní myšlenka, o kterou se celá tato práce opírá, je založena na možnosti vzájemné komunikace algoritmů řádkového a stránkového zlomu. Metoda optimum fit, kterou TEX používá pro nalezení nejlepšího tvaru celého odstavce, je do jisté míry znehodnocena tím, že alternativní řešení jsou po nalezení nejlepšího zalámání zapomenuta. Přitom jsme ukázali, že řádkový zlom nemá v dané chvíli k dispozici všechny informace o tom, za jakých podmínek či do jakého tvaru má být ten který odstavec zformátován. Tato omezení se objeví až při plnění strany a jejím zalamování.

Otázka tedy zní: je možné posečkat s definitivním zalomením odstavce až do chvíle, kdy budou známy okolnosti zlomu stránkového? Ukážeme, že ano, pokud rozšíříme datové struktury v algoritmech používané a pozměníme způsob výpočtu a tok dat uvnitř systému.

8.1. Přístup shora dolů

V TEXu je řídící částí celého programu vstupní analyzátor. Ten svým výstupem iniciuje další činnosti uvnitř systému, data jsou dále transformována řádkovým zlomem a ve formě vertikálního seznamu se dostávají ke zlomu stránkovému. Ten již ale nemá možnost udělat revizi jednou provedených rozhodnutí.

Řešení spočívá v obrácení celého principu sázecího systému a ve stanovení nového řídícího článku. Místo vstupní části jím bude algoritmus lámání a tvorby strany, tedy nejvyšší struktury v dokumentu. Ten pak bude žádat a získávat data od algoritmů, které jsou v procesu zpracování před ním, místo aby je jen „pasivně“ přijímal. To umožní větší flexibilitu při hledání výsledné podoby strany, protože algoritmus stránkového zlomu bude moci zažádat o zalámání jedné strany vícekrát a posléze se rozhodnout, které řešení je z globálního pohledu stránky nejlepší.

```
while (jsou nějaká data na vstupu)
    {
    začni novou stránku;
    vyžádej si vertikální materiál pro naplnění strany;
    while (je nějaká šance na zlepšení výsledku)
        {
        vyžádej si nová data, která změní vzhled strany;
        je-li nové řešení lepší, zapamatuj si ho;
        }
    pošli stránku na výstup;
    }
```

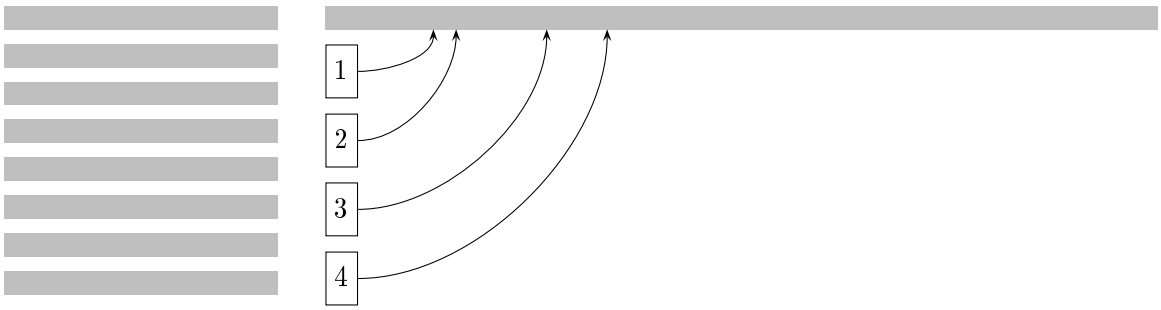
Opakovaně zde voláme část vytvářející hlavní vertikální seznam a testujeme výsledek. Pokud by výsledná strana pro aktuální vertikální seznam obsahovala některý z nežádoucích jevů, například vdovu či sirotka, zažádáme o nové vytvoření vertikálního seznamu ze stejného typografického materiálu. Například můžeme vložit do některého z odstavců stránky pomyslný příkaz `\looseness` a tím dosáhnout změny počtu řádků a odstranění nevhodného stránkového zlomu. Simulujeme zde vlastně chování uživatele za klávesnicí, který vysází dokument a pokud zjistí, že výsledek nesplnil zcela jeho očekávání, spustí formátování dokumentu znovu, s poněkud pozměněnými parametry.

Žádost o nová data neznamená, že se bude znovu číst vstup od uživatele. Ten je po zpracování vstupním analyzátorem převeden na typografický materiál — vertikální a horizontální seznamy s boxy, glue a penaltami. Při opakovaném zpracování strany se tedy bude manipulovat s těmito typografickými daty. Navíc je pravděpodobné, že nový požadavek na zpracování strany se nebude příliš lišit od předchozího, takže při vhodných datových strukturách budeme moci využít minulé výsledky.

V nástinu algoritmu jsme použili volnou formulaci „je šance na zlepšení výsledku“. Pokud prvním průchodem dostaneme podtečenou stránku, protože se algoritmus stránkového zlomu snažil vyhnout parchantu, který byl zakázán nekonečnou penaltou, je oprávněná naděje, že se podaří najít alternativní řešení například tím, že zvětšíme počet řádků v některém z odstavců pomocí ekvivalentu TEXovského `\looseness`. Jestliže dojde ke stránkovému zlomu na rozděleném slově, zažádáme o nalezení nového zalámání posledního odstavce na straně a dělení daného slova nyní nepovolíme. Když nás nové řešení neuspokojí, musíme se znovu rozhodnout, zda je efektivní pokračovat v hledání další varianty, nebo zda jako výsledek použijeme nejlepší z dosud nalezených zalámání. Používáme tedy metodu **backtrackingu** s tím, že nakonec vybereme optimální z nalezených řešení.

8.2. Typ zalámaný odstavec

Protože se zaměřujeme hlavně na odstranění problémů s hladkou sazbou, která je převážně tvořena čistými odstavci, rozšíříme systém právě v oblasti odstavců. Potřebujeme nástroj, kterým bychom se mohli algoritmu řádkového zlomu dotázat několikrát na zalámání toho samého odstavce, vždy s různými parametry. Nechceme tedy ztratit původní obsah horizontálního seznamu. Jako výhodné se zde ukáže být obohacení modelu box-glue-penalty o nový typ: **zalámaný odstavec**. Místo posloupnosti hboxů a meziřádkových glue potom bude výstupem algoritmu řádkového zlomu ten samý horizontální seznam, který do něho vstupoval, k němuž bude připojena informace o tom, jaká místa zlomu byla nalezena a jaké je nejlepší řešení, jako na následujícím obrázku:



Pokud při plnění strany zjistíme, že takto zalámaný odstavec nesplňuje ani po uplatnění meziřádkových penalt naše představy, můžeme se k němu vrátit a zažádat o nové řešení. Pokud změníme jen hodnotu `\looseness`, půjde pouze o nalezení jiného řešení v již jednou vytvořené struktuře míst zlomů v odstavci, jindy to bude nalezení nové množiny řešení nad tím samým horizontálním seznamem.

Při zavádění tohoto nového datového typu je vhodné přiblížit jeho funkční rozhraní již používaným typům. Typ zalámaný odstavec musí poskytovat stránkovému zlomu stejné informace a možnosti, jako TEXovská posloupnost hboxů, glue a penalt. Algoritmus stránkového zlomu například zjišťuje výšku vytvářené strany tak, že sčítá výšky a hloubky hboxů reprezentujících řádky odstavců s glue, které vzešly z meziřádkových mezer. Také nový

datový typ musí poskytovat odpověď na dotazy „jak je vysoký první řádek“, „jaká je roztažitelnost glue mezi předposledním a posledním řádkem“ či „jaké jsou plovoucí objekty, které vzešly ze čtvrtého řádku“. Zároveň by měl zalámaný odstavec pracovat efektivně. Pokud byl odstavec jednou zalámán na deset řádků a stránkový zlom nyní požaduje řádků jedenáct, je odpověď již vypočtena ve struktuře míst zlomů z algoritmu optimum fit a není potřeba znovu spouštět zlom včetně dělení slov a podobně.

Pokud máme být schopni dávat stránkovému zlomu odpovědi na otázky týkající se pomyslných řádkových hboxů bez toho, aniž bychom tyto hboxy vytvářeli, musíme být schopni všechny potřebné informace o těchto boxech získat z bodů zlomu ve výsledné struktuře odstavce. Šířka je k dispozici zcela přirozeně již ze zadání, na jejím základě počítáme místa zlomu, poměr roztažení r i přetečenost či podtečenost boxu. Výpočet výšky a hloubky musíme do optimum fit algoritmu přidat. Jejich hodnota je definována jako maximum z hodnot prvků v boxu obsažených. Algoritmus optimum fit obohatíme o dvě pomocné proměnné v a h, které budou uchovávat výšku a hloubku materiálu v horizontálním seznamu od posledního možného místa zlomu. Pro každé možné místo zlomu x ze vstupního horizontálního seznamu, který vytváří konec potenciálního řádku, a pro jeho začátek a v aktivním seznamu budeme pak schopni určit výšku a hloubku boxu, kterým je řádek ohraničen.

`optimum-fit` rozšířený o výpočet výšky a hloubky řádků:

inicializace seznamu aktivních prvků;
`while` (je nějaký prvek x ve vstupním horizontálním seznamu)
{
`if` (prvek x je možným místem zlomu)
{
`for` (všechny prvky a aktivního seznamu)
{
a->pom_výška = v `if` (`defined` v
&& `not defined` a->pom_výška || a->pom_výška $\leq v$);
a->pom_hloubka = h `if` (`defined` h
&& `not defined` a->pom_hloubka || a->pom_hloubka $\leq h$);
ostatní operace jako v 5.3;
}
`if` (byla nalezena alespoň jedna ceta z a do x)
{
přidej x do aktivního seznamu s nejlepší cestou k předchůdci (a);
x->výška = a->pom_výška;
x->hloubka = a->pom_hloubka;
}
(v, h) = (`undef`, `undef`);
}
v = výška aktuálního prvku v_x `if` (v `not defined` || $v < v_x$);
h = hloubka aktuálního prvku h_x `if` (h `not defined` || $h < h_x$);
}

Proměnné v a h uchovávají výšku a hloubku mezi možnými místy zlomu. Pokud je nalezeno nové možné místo zlomu, je ve všech prvcích aktivního seznamu poznamenáno, jak vysoký a hluboký by byl řádek, kdyby končil v aktuálním místě zlomu. Pokud jsme nalezli nový možný řádek, poznamenáme v jeho koncovém prvku (x), jaké jsou rozměry tohoto řádku.

Uděláme-li v algoritmu tuto změnu, jsme již schopni s jeho výsledky pracovat podobně jako s výsledky původního optimum fit algoritmu. Můžeme pak spustit lámání strany, které vyvolá čtení a zpracování vstupních dat a je také nalezen bod stránkového zlomu, který může padnout mezi některý z „virtuálních“ řádků zalámaného odstavce. Jak v takového situaci rozpoznáme nevhodná místa stránkového zlomu, o nichž jsme mluvili v části 7.2? Hlavním signálem bude neúspěšný pokud o nalezení stránkového zlomu. Parametry, které jsou v TEXu označeny jako `\club` a `\widowpenalty` pro parchanty a `\brokenpenalty` pro řádek ukončený rozděleným slovem, budou k dispozici i při práci s naším typem zalámaný odstavec. Jejich hodnoty jsou vypočteny operativně na základě požadavků algoritmu stránkového zlomu. Pokud se tento dotazuje na řádek ukončený v jistém místě zlomu, je odpovídající penalta jedním z atributů tohoto konce řádku. Pokud penalta, ať již vložená uživatelem či vypočtená podle obsahu odstavce, zabrání vhodnému stránkovému zlomu, je výsledkem strana přetečená nebo podtečená a je nutné provést korigující činnost.

V případě parchantů se jako nejlepší jeví prohledání předchozích odstavců na straně a nalezení takového, který je možné o řádek prodloužit, případně zkrátit — v TEXu parametr `\looseness`. Vzhledem k tomu, že zalámání s alternativním počtem řádků je již v paměti vypočteno, jedná se pouze o prohledání hotových řešení. Je-li odstavců, u nichž je tuto úpravu možné provést, nalezeno více, vybereme z nich ten, který bude mít po změně nejmenší celkovou badness. Novou posloupnost vertikálního materiálu předložíme znovu stránkovému zlomu.

Pokud jde o místa zlomu za řádky končícími rozděleným slovem, opět hledáme alternativní řešení pouze tehdy, pokud díky vysoké penaltě není možno najít úspěšně zalomení strany běžným způsobem. Potom je třeba hledat jiné místo zlomu pro ukončení posledního řádku na stránce. Zavoláme tedy znovu algoritmus řádkového zlomu a určíme, že dané místo zlomu se nesmí ve výsledku objevit. Jelikož pro řádkový zlom používáme algoritmus optimum fit, můžeme při novém lámání využít předchozí výsledek: množina řešení pro předcházející řádky zůstane stejná. Místa zlomu ukončující předposlední řádek na straně vložíme do aktivního seznamu a výpočet začneme od prvního prvku, který může padnout na poslední řádek. Fakt, že jsme změnili podmínky na posledním řádku strany totiž nemůže ovlivnit způsob, jakým jsou zalomeny řádky před ním.

Při tomto přístupu záhy zjistíme, že jako alternativní řešení je často vybráno jiné rozdělené slovo posledního řádku, obvykle dokonce jiné místo dělení stejného slova. Proto je užitečné zavést do algoritmu řádkového zlomu parametr, který zakáže dělení slov v rámci celého jednoho řádku. Je možné, že ani poté není opakovaným průchodem možné najít schůdný stránkový zlom. Poté je vhodné se uchýlit k postupu použitému při odstraňování parchantů, najít na straně odstavec s nastavenou `\looseness`.

Díky otevřenosti datového typu zalámaný odstavec je možné podle potřeby přidávat i další parametry, například penaltu za řádek končící velmi krátkým slovem. Vždy ale platí, že další řešení jsou prohledávána až tehdy, kdy selže první pokus o stránkový zlom. Z časového hlediska nejde tedy oproti TEXu o zpomalení. Opakované hledání jiného řešení je spuštěno jen tehdy, kdy výsledek poskytnutý TEXem by nebyl uspokojivý a uživatel by musel přetečený nebo podtečený vertikální box strany odstraňovat ručně. Paměťová náročnost naopak oproti TEXu vzrostla. Strukturu, kterou TEX uchovává v paměti pouze v průběhu řádkového zlomu odstavce, nemůžeme odstranit dříve, než je zalomená celá strana. Paměť potřebná pro data zalámaných odstavců se tedy násobí počtem odstavců na straně, který je v běžném případě omezen počtem řádků na jednu stranu. Je třeba ale podotknout, že pro velmi krátké odstavce je možností ve výsledku velmi málo, delších odstavců, které mají řešení bohatší, se zase na stranu vejde méně.

8.3. Různě široké strany

Chceme-li vytvořit efektivní nástroj pro sazbu nestejně širokých stran či sloupců, jak je ukázáno na příkladu knihy Richarda Feynmana (část 7.2.), musíme dát algoritmu pro každou stranu dodatečnou informaci o tom, jaké rozměry požadujeme. Začneme s tím, že u každé strany uvedeme výšku a šířku jejího tiskového zrcadla. Parametrem sázecího systému je tedy seznam dvojic, pro každou stranu dvě hodnoty. Takto zadáme rozměry konečného počtu stránek, další pak budou mít hodnoty poslední zadané strany.

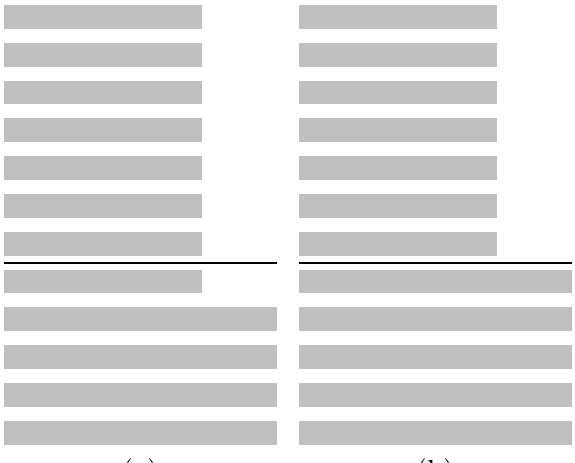
Známe tedy rozměry první strany a v našem algoritmu žádáme vertikální materiál pro její naplnění. Předpokládejme, že na vstupu jsou data z odstavce, která je třeba zalámat do řádků. Musíme tedy říci, jak dlouhé mají řádky být. Pokud vyjdeme z toho, že sázíme víceméně hladký text, můžeme předpokládat, že na první stránku se vejde

$$\text{počet řádků} = \frac{\text{výška první strany}}{\texttt{\textbackslash baselineskip}}$$

řádků. Tyto řádky budou mít požadovanou šířku rovnou šířce první strany, následující se pak budou řídit rozměry druhé strany, atd. Samozřejmě, toto číslo je pouze orientační, uživatel může kdykoli změnit hodnotu parametru `\baselineskip`, může použít jakýkoli typografický příkaz, který změní vertikální pozici textu. Pokud jsme ale postaveni před úkol zalámat první odstavec dokumentu, nemůžeme vyjít z jiných údajů než z těch, které zatím máme k dispozici.

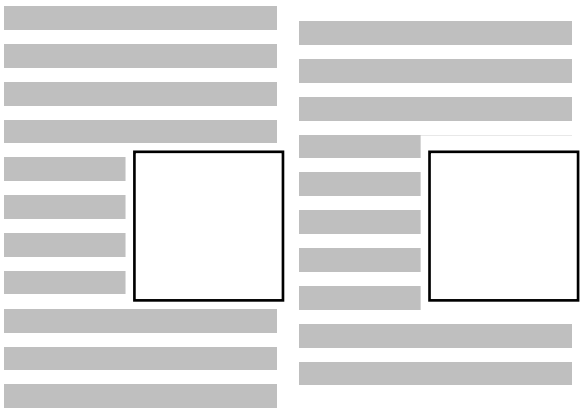
Argumentem prvního spuštení algoritmu optimum fit bude tedy kromě horizontálního seznamu prvního odstavce seznam počtu řádků, které se podle odhadu vejdou na jednotlivé stránky. Odpovědí je síť možných řešení a informace o tom, která cesta touto sítí je nejvhodnější. Pokud řádky z odstavce nenaplnila celou stranu, vypočteme zbývající vertikální místo na ní a zažádáme o načtení a zalámání dalšího odstavce. Vždy pak přepočítáme, do jaké míry je již strana naplněna. Pokud už text přesáhne na další stranu, musíme vypočítat a doladit místo stránkového zlomu.

Může se stát, že kvůli posunům mezi vertikálními elementy vyjde stránkový zlom před nebo za původně zamýšleným počtem řádků, jak to ilustruje obrázek (a). Potom provedeme backtracking a zavoláme lámání posledního odstavce s jiným počtem řádků pro první stranu (b). Stejně jako při odstraňování stránkového zlomu na řádku s rozděleným slovem, i zde využijeme již jednou spočítaného výsledku: množina řešení pro prvních šest řádků na obrázku (b) zůstává stejná. Místa zlomu ukončující tento šestý řádek vložíme do aktivního seznamu a výpočet začneme od prvního prvku, který může padnout na sedmý řádek.



(a) (b)

Zalomení znázorněná na obrázcích (c) a (d) ilustrují další možnost, o kterou můžeme systém rozšířit, pokud budeme schopni provést několik zlomů stejného odstavce po sobě, pokaždé s jinými vstupními parametry. Uživatel může ručně zadat omezení na tvar strany, například tím, že myší umístí do daného místa obrázek, který má být obtečen textem. Tím se po stránkovém zlomu požaduje vytvoření strany, která nemá obdélníkový tvar.



(c) (d)

Výšku vloženého obrázku, omezení vyjádřené v centimetrech či jiných jednotkách fyzického rozměru, musíme převést na počet řádků, které budou tímto obdélníkem ovlivněny. Stejně jako v předchozím případě různě širokých stránek, i zde můžeme vyjít pouze z informací, které máme v průběhu lámání k dispozici. Počet řádků i rozhodnutí, které řádky mají být zkráceny, činíme v okamžiku, kdy o obsahu těchto řádků ještě nic nevíme, protože ještě nebyly zformovány. Prvotní odhad založíme na nastavení globálních parametrů stránkového zlomu, zvláště pak parametru `\baselineskip`. Tak získáme data pro první spuštění řádkového zlomu. Jeho výsledky pak budeme opět korigovat výše uvedeným backtrackingem. Na obrázku (d) jsme kvůli vertikálnímu posunu začátku odstavce museli zvětšit počet řádků, které budou zkráceny.

8.4. Plovoucí objekty

V předchozích odstavcích jsme předpokládali, že použitý stránkový zlom, je v našem návrhu stejný, jak v případě TEXu; potom je stejné i zpracování plovoucích objektů, které svou přítomností ovlivňují vertikální rozměr strany. Jelikož ale máme k dispozici nástroj, jak nechat textem obtéci staticky zadaný výřez, můžeme se pokusit takto zpracovat i plovoucí objekty, jejichž pozice není pevně určena uživatelem a myší, ale je zadána podmínkou ukotvení vůči textu dokumentu. Uvažme případ (c) z předcházejícího obrázku. Na pravém okraji strany je vynecháno obdélníkové místo na dodatečný obrázek či tabulku. Z hlediska stránkového zlomu jde o fakt, který omezuje a mění délky řádků v lámaných odstavcích. Podobným způsobem bychom mohli tato omezení do systému zanést i z objektů plovoucích.

Při řešení úkolů s nevhodnými stránkovými zlomy (viz. 8.2.) jsme použili metodu backtrackingu, jejíž přístup je podobný tomu, jaký by zvolil uživatel, pokud předchozí průchod sázecím systémum nedal uspokojivé výsledky. Při prvním zpracování totiž ještě nejsou známa všechna omezení, která mohou pozměnit tvar dokumentu. V okamžiku, kdy algoritmus zjistí novou skutečnost, přizpůsobí jí svoje parametry a vyvolá nový průchod. Novou skutečností je pro řádkový zlom i fakt, že do textu má být vložena ilustrace, pro kterou je nutné v základním textu dokumentu udělat obdélníkový výřez. Proto i na plovoucí obtékané objekty můžeme použít stejnou metodu.

Kromě plovoucích objektů, které ovlivňují pouze vertikální směr tvořené strany a dokážeme je ošetřit TEXovským přístupem, zavedeme dvě další třídy plovoucích objektů, pro levý a pro pravý okraj strany. Objekty v těchto třídách pak definují tvar levého a pravého okraje strany. Podle těchto okrajů je pak možné vypočítat délky řádků v odstavcích. Plovoucí objekty jsou ale nalezeny až v průběhu zpracování strany, až tehdy zjistíme, jaký tvar mají vlastně okraje strany mít. Potom využijeme faktu, že jsme schopni opakovaně provést zalámání odstavců, pokud se objeví podmínka, který ovlivní jejich tvar.

Tvar strany postupně upřesňujeme — plovoucí objekty, které zabírají celou šířku textu, ovlivňují výšku místa, do kterého sázíme základní text dokumentu, zatímco objekty levého a pravého okraje mění horizontální rozměry tohoto prostoru. Protože stránku takto můžeme dotvářet na více pokusů, je důležité definovat podmínky pro ukončení těchto iterací a výběr nejlepšího řešení. Stejně jako u TEXovských plovoucích objektů se budeme snažit na stranu vložit tolik „postranních“ objektů, kolik se jich na ni vejde. Levý i pravý okraj plníme nezávisle. Přitom ale bereme ohled na případné změny celkové výšky strany. Tím se také může stát, že na stranu umístíme objekt, jehož odkaz posléze padne až na další stranu. Takového odkazy zpět je nutné odstranit, plovoucí objekt musí být umístěn na stranu následující.

Backtracking skončí, pokud již nedochází k žádné změně v tom, jaké objekty jsou na stranu umístěny, a pokud se již nezlepšují hodnoty badness odstavců a vertikálních elementů. Můžeme případně nastavit i statické omezení na maximální počet pokusů o opakované zalámání.

Protože objektů může být na jedné straně více a tím se zvyšuje i potřebné množství průchodů backtrackingem, je vhodné informaci o tom, na jaké straně a v jaké pozici byl ten který umístěn, uložit do pomocného souboru a při příštím průchodu ji využít. Podobně bychom postupovali v systému, kde se doprovodné objekty k textu umisťují ručně. Postupně do textu přidáváme další a další objekty a posuzujeme, zda je pro ně na straně ještě místo. Pokud by pak došlo ke změně textu, museli bychom patřičným způsobem změnit i umístění plovoucích objektů, ovšem zase bychom vyšli z předchozího zalámání.

9. Prototypový systém

Cílem prototypového systému bylo v praxi ověřit teoretické návrhy na zlepšení funkčnosti sázecích programů. Hlavní pozornost byla věnována novému datovému typu — zalámanému odstavci, který byl základem všech úvah v částech 8.2. a 8.3.

9.1. Řešení v jazyku Perl

Při realizaci prototypového systému se jako nejdůležitější ukázalo rozhodování, zda využít již existujícího kódu TEXu a vytvořit odpovídající změnový soubor, případně zasáhnout přímo do zdrojových textů a tak přidat navrhovaná rozšíření, anebo vytvořit novou nezávislou implementaci. Studium zdrojových kódů [Knuth 86] ukázalo, že první cesta bude jen těžko schůdná. TEX je velmi rozsáhlý systém, optimalizovaný k vysokému výkonu. To se odrazilo i na datových strukturách a provázanosti jednotlivých částí programu. Data používaná při výpočtu jsou uložena ve staticky alokované paměti, manipulace s nimi probíhají s pomocí ukazatelů do tohoto dlouhého pole bajtů. Rozšíření o nový, dosti komplexní datový typ a změna řízení a toku dat v celém programu je takovým zásahem, že bylo pro prototypový systém lepší začít znovu a nenést při implementaci s sebou zátěž zpětné kompatibility s původním zdrojovým kódem.

Jakmile bylo zřejmé, že pro demonstraci činnosti navrhovaného systému bude nutné napsat znovu velkou část existujícího programu, bylo potřeba najít prostředí, které by poskytlo vhodné prostředky pro snadné vyjádření již existujících algoritmů, tak pro přehlednou implementaci nových postupů. Volba nakonec padla na jazyk **Perl** [Wall 97] [Wall 96], který má například proti jazyku C několik výhod: je kombinací vysokého i nízkého přístupu k programování. Máme zde k dispozici jak operátory pro manipulace s bity, tak funkce pro analýzu složitých struktur, zpracování textových dat, nezanedbatelné jsou i objektové rysy jazyka a možnost modulárního rozšiřování [Chris 97]. Navíc je přenositelný: pokud je na dané platformě interpret Perlu k dispozici, je velmi pravděpodobné, že standardní skripty v něm napsané budou pracovat bez omezení. Nezanedbatelnou výhodou je také automatická alokace a uvolňování paměti vytvářených objektů.

Protože cílem práce není vytvoření produkčního systému, ale ověření principů na prototypu, byly jeho části vytvořeny s využití modulů a objektové technologie, která je v Perlu dostupná. Tím, že k datům definujeme funkční rozhraní, můžeme snadno sledovat chování algoritmů. Pro účely této práce se toto modulární řešení ukázalo jako velmi výhodné. Samozřejmě, pokud bychom po systému požadovali reálný sázecí výkon na úrovni TEXu, bylo by asi nezbytné oželet elegantní objektové postupy a přejít na manipulaci s bajty.

Prototypový systém se skládá ze čtyř modulů. `BGP.pm`, jehož název je odvozen ze slov box-glue-penalty, definuje několik tříd (package) pro práci s typografickými elementy, jako jsou horizontální a vertikální boxy, seznamy, slova, glue a penalty, odstavce a jejich zalámání. Třídy poskytují do značné míry jednotnou množinu metod, pomocí nichž můžeme manipulovat s jejich datovými složkami. Tento modul tvoří jádro celého systému.

Kromě něho jsou velmi důležité i moduly, které zajišťují vstup a výstup dat: `TFM.pm` poskytuje třídu, která čte a zprostředkovává informace o fontu a znacích v něm, na základě TEX Font Metric souboru. Modul `DVI.pm` slouží pro zápis výstupního souboru ve formátu DeVice Independent a `Hyphen.pm` zpřístupňuje vzory dělení, které se v iniTEXu načítají

příkazem `\patterns`. Při návrhu prototypového systému byl kladen velký důraz na zpětnou kompatibilitu s existujícím programovým vybavením a standardy, které byly okolo sázecího systému TEX vyvinuty.

9.2. Modul BGP.pm

Jasné vymezení objektů a funkcí, které se nad nimi provádějí, bylo jedním z cílů návrhu prototypového systému. Proto i modul `BGP.pm` je pečlivě vytvořen tak, aby respektoval zavedené postupy při odvozovaní a dědění tříd.

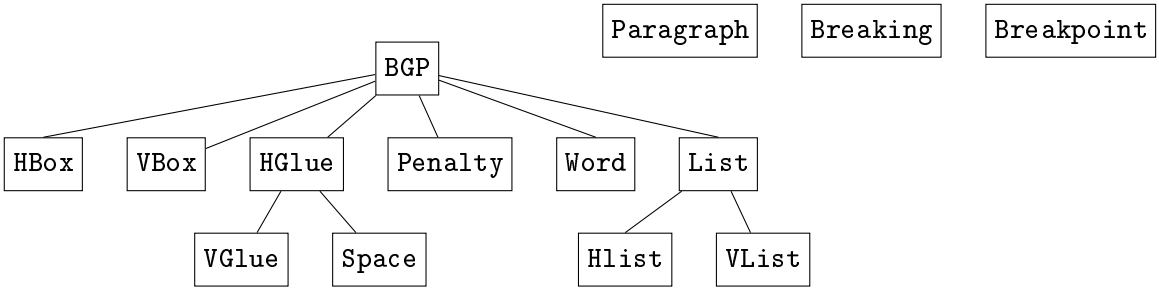
Rodičovskou třídou, z níž se odvíjejí další, je třída (package) `BGP` — box-glue-penalty. Definuje virtuální metody jako `width`, `stretch`, `shrink`, atp., které budou volány pro zděděné třídy, pokud si tyto nenadefinují svoje vlastní. Proto se také nepředpokládá její přímé použití v programech, je vlastně interním, i když velmi důležitým, datovým typem celého modulu `BGP.pm`. Při definici této i dalších tříd bylo hojně využito metody `AUTOLOAD`, která je volána v okamžiku, kdy pro daný objekt není nalezena metoda odpovídajícího jména. To přispělo k přehlednosti zdrojového kódu, protože metody jsou pak uloženy jako prvky asociativního pole na jednom místě.

Z této základní třídy jsou objektově pomocí perlovského mechanismu `@ISA` odvozeny další: `HBox` a `VBox` pro vodorovné a svislé sestavování objektů, `HGlue` a `VGlue` pro glue a třída `Penalty` odrážejí strukturu použitého sázecího modelu. Kromě toho jsou k dispozici i další specializované: `Space` definuje horizontální glue se šířkou a roztažitelností a stažitelností danou aktuálním fontem pro mezislovní mezeru, `Word` je slovo (řetězec znaků) daného fontu.

Toto jsou tedy základní stavební kameny pro typografické elementy dokumentu. Jejich hlavním úkolem je zunifikovat přístup k datovým položkám tak, abychom například při jejich použití příkazem `$object->width()` dostali vždy správnou odpověď bez nutných testů, o jaký objekt se vlastně jedná. Samozřejmě, že v případě potřeby se můžeme na typ explicitně dotázat, například metodou `is_box`. Nad těmito základními typy jsou vystavěny třídy, které již definují vlastní výkonné typografické algoritmy.

Pro vytváření seznamů slouží `HList` a `VList`, odstavec textu je reprezentován třídou `Paragraph`, vytvořenou nad horizontálním seznamem (`HList`). Odstavec je zalámán metodou `break`, která používá algoritmus optimum fit. Návratovou hodnotou této metody je instance třídy `Breaking`. Ta pro daný odstavec a parametry použité při spuštení `Paragraph::break` uchovává informaci o provedeném zalámání. Tím, že pro výsledek řádkového zlomu vytvoříme nový objekt, se zjednodušuje další práce s tímto výsledkem. Pomocí definovaných metod se můžeme dotazovat jednak na globální fakta o zalámaném odstavci (například `lines` vrátí počet řádků), jednak se takto dostaneme na jednotlivé virtuální řádky a elementy odstavce. Při své práci se algoritmus řádkového zlomu opírá o třídu `Breakpoint`, která reprezentuje jedno možné místo zlomu v odstavci.

Toto je struktura tříd modulu `BGP.pm`:



Vstup dat je v prototypovém systému řešen třídou `InParse`: text je načítán po odstavcích, které jsou odděleny jedním nebo více prázdnými řádky. Kromě značky pro nezlomitelnou mezeru (`~`) je tento vstup převeden na posloupnost slov a mezislovních mezer. Toto řešení nechává dostatek prostoru pro další rozšiřování. Jedním z možných směrů by mohlo být použití modulu `Text::TeX`, který vyvíjí Ilya Zakharevich [Zakhar 96]. Modul je ale ještě v raně vývojovém stádiu. V konečné verzi by měl být schopen načítat zdrojový text zapsaný syntaxí TEXu a pomocí heuristických pravidel ho převádět na posloupnost primitivních příkazů. Takový nástroj by pak dovolil naplno využít možností prototypového systému, protože bychom nastavení hodnot parametrů lámacího procesu mohli provést ve vstupním textu. Nyní je nutné všechna dodatečná omezení zadat pomocí parametrů přímo v sázecím systému.

Hlavní cyklus programu opakovaně volá čtení dalšího odstavce ze vstupu. Pro tyto objekty typu `Paragraph` je pak metodou `break` nalezeno jejich zalámání (`Breaking`). Algoritmus potom měří doposud obsazené vertikální místo na straně. Pokud naposledy načtený odstavec již přesáhl přes hranu strany, testuje se, zda je možno zalomení provést bez toho, aniž by výsledkem byl přetečený či podtečený vertikální box. Jestliže se vyskytl některý ze sledovaných jevů, které brání úspěšnému zalomení strany, provede se zpětné hledání alternativního řádkového zalámání (viz. 8.2.).

Při volání algoritmu řádkového zlomu je podstatná velikost vertikálního místa, které zabírají předcházející odstavce, směřované na aktuální stránku. Za pomoci této informace, hodnoty vertikálního glue, můžeme předem odhadnout pozici prvního řádku lámaného odstavce. Pokud jsou v programu definovány pro jednotlivé strany různé šířky nebo jsou požadovány výřezy do základního obdélníkového tvaru strany, určuje předpokládaná pozice prvního řádku a hodnota parametru `\baselineskip` řádky, kterých se změna délky týká. Takto již při prvním volání řádkového zlomu tvarujeme odstavec do žádaných mezí, byť konečné pozice a tím i délky řádků se mohou lišit.

Tvar a výřezy stran jsou ve stávající verzi systému definovány statickým polem údajů. Důvod je ten, že není k dispozici efektivní nástroj, jak tato data načíst ze vstupního textu. Z podobného důvodu také nejsou zpracovávány plovoucí objekty, protože i ty by bylo nutné rozpoznat a korektně zpracovat již při analýze vstupu. Aktuální verze modulu `Text::TeX`, který slibuje být možnou cestou, ještě není natolik vyzrálá, aby mohla být pro tento úkol použita.

Výstup prototypového systému jde jednak do DVI souboru, kam je zapsán zformátovaný dokument, na standardním chybovém výstupu se pak objevují ladící informace a údaje o průběhu sázecího procesu. Množství a strukturu těchto informací je možné nastavit odpovídajícími parametry `$DEBUG`.

9.3. Ostatní použité moduly

Každý z následujících modulů definuje jednu třídu, která zjednodušuje přístup k datům okolo sázecího systému. Uzavření vstupně výstupních funkcí dovnitř metod tříd je velmi výhodné, protože jsme tím odstíněni od specifik operačního systému.

Třída `Font::TFM` je definována v modulu `TFM.pm`. Po jejím vytvoření například příkazem `my $cmr = new Font::TFM "cmr10";` je nalezen a načten do paměti odpovídající soubor TFM. Třída podporuje prohledávání adresářové struktury podle několika kritérií, respektuje nastavené proměnné prostředí i předkompilované `ls-R` seznamy. Font můžeme také načíst v jiné než defaultní velikosti pomocí `new_at`. Tomuto novému objektu pak můžeme klást „dotazy“ ohledně fontu a znaků v něm. Definovány jsou například metody `de-`

`signsize` a `fontsize`, na rozměry znaků se můžeme dotázat pomocí `$cmr->width("A")` či `$cmr->height('x')`. Ke globálním parametrům fontu se dostaneme jednak pomocí metody `param`, které zadáme číslo požadovaného parametru, jednak mají některé z nich přiřazeno odpovídající jméno: `$cmr->slant()` je hodnota sklonu písma, `x_height` či `em_width` definují referenční rozměry fontu.

Třída poskytuje také přístup k informacím o ligaturách a kerningu pro dvojice znaků. Například `$cmr->kern("Wo")` vrátí velikost horizontálního posunu, který je nutné vložit mezi znaky „W“ a „o“. Příkazem `$cmr->lig('fi')` získáme znak, který ve fontu reprezentuje slitek „fi“. Tyto informace jsou ve fontu zapsány pomocí jednoduchého programovacího jazyka a díky metodám třídy `Font::TFM` je obdržíme bez toho, aniž bychom museli formát těchto interních dat znát. Podobně je tomu i s roztažitelnými matematickými znaky, které se skládají z více kousků.

Pro uživatele budou asi nejdůležitější poslední dvě metody, neboť poskytují nejvyšší úroveň přístupu k datům o fontu. Zavoláme-li metodu `expand`, jejímž parametrem je řetězec znaků, provede se v něm expanze ligatur a kerningů a na výstupu dostaneme již informaci o tom, jaké znaky s jakým mezipísmenným prokladem je třeba vysázet, aby byly dodrženy všechny specifikace ve fontu uvedené. Pomocí `word_width` získáme šířku, kterou takto rozexpandované slovo ve výstupním dokumentu zabere.

Tento modul, stejně jako dva následující, obsahuje vestavěnou manuálovou stránku ve formátu POD (Plain Old Documentation), kterou je možné přečíst programem `pod2text` nebo zkonvertovat do podoby stránky pro Unixový systém `man` pomocí `pod2man`. V dokumentaci jsou popsány nejen všechny metody objektu přístupné, ale také uvedeny další informace o chování modulu, ladících výpisech, a podobně.

Modul `DVI.pm` definuje třídu `TeX::DVI`. Ta je určena k zápisu DVI souborů a objekty této třídy mohou používat metody, které jsou ekvivalentem typografických příkazů specifikovaných pro formát DVI. Najdeme zde tedy například `begin_page` a `end_page`, `preamble` a `postamble`, příkazy `push`, `pop`, `font`, `word` či `hskip`.

Tato třída úzce spolupracuje se třídou `Font::TFM`: definice fontu `font_def` předpokládá, že třída, která font reprezentuje, bude mít stejnou strukturu metod jako má `Font::TFM`. Navíc se při zápisu slova na výstup spoléháme na to, že metoda `Font::TFM::expand` vrátí korektně rozexpandované slovo daného fontu. Tato úzká provázanost s předchozím modulem ale neznamená, že není možno použít i jiné metriky fontů. Jedinou podmínkou je, aby byly zpřístupněné třídou, která bude mít stejnou strukturu veřejně přístupných metod.

Posledním modulem je `Hyphen.pm` se třídou `TeX::Hyphen`. Umožňuje dělení slov podle TEXovských vzorů dělení a její použití je následující:

```
use TeX::Hyphen;
my $hyp = new TeX::Hyphen "hyphen.tex";
my $word = "representation";
my @hyphens = $hyp->hyphenate($word);
```

Pomocí `new` vytvoříme v paměti nový objekt a metoda `hyphenate` poté vrátí seznam míst, kde je v předložené slově dělení povoleno. Tato možná místa dělení můžeme zobrazit příkazem `visualize`, který vloží na označená místa znak rozdělovníku. Pokud předpokládáme, že soubor `hyphen.tex` obsahuje anglické vzory dělení, dostali bychom příkazem

```
print $hyp->visualize($word);
```

na výstupu `rep-re-sen-ta-tion`. Dělení pro češtinu či další jazyky dosáhneme tak, že vytvoříme jiný objekt a příkazu `new` při tom zadáme jako parametr soubor s jinými vzory.

Všechny tyto tři externí moduly jsou veřejně dostupné na celosvětově replikovaném archívu perlovského software CPAN (Comprehensive Perl Archive Network), a to v poda-

dresáři `CPAN/authors/id/JANPAZ/`, jejich primární zdroj je na autorově perlovské WWW stránce `http://www.fi.muni.cz/~adelton/perl/`. Zde je také uložen základní modul prototypového systému, `BGP.pm`.

9.4. Praktické výsledky

Prototypový systém byl testován na různých typech vstupních dokumentů. Cílem bylo ověření funkčnosti a správnosti použitých programových postupů.

První pozornost byla směřována na rámcové porovnání rychlosti s programem TEX. Prototypový systém byl asi 30–50× pomalejší než TEX, na ekvivalentních vstupních datech. Tento výsledek není příliš lichotivý, a proto se pokusíme nalézt jeho důvody. Objekty, které jsou v Perlu používány, jsou implementovány jako pole datových položek, případně jako asociativní pole. Například k šířce objektu v modulu `BGP` přistupujeme pomocí metody `width`, a ta provede čtení `$self->{'width'}`. Toto je přístup k asiciativnímu poli pomocí hashovací funkce, která musí zpracovat řetězec `'width'` a pomocí něho najít správnou datovou položku. Vzhledem k tomu, že těchto čtení hodnot objektů se v systému provádí velmi mnoho, zabírá toto vyhledávání velmi mnoho místa. Řešením na úrovní Perlu by mohla být změna asociativních polí na klasická pole, pak bychom v datových položkám přistupovali pomocí indexů do tohoto pole: `$self->[5]`. Takovýto přístup by ale nepřispěl k čitelnosti zdrojového textu, i když částečně bychom tento problém mohli potlačit zavedením odpovídajících jmen indexů.

Druhým důvodem, proč je prototypový systém pomalý, je sama objektová techonologie, která je v něm použita. I když je nesporné, že kód je čistší a čitelnější, je jejím důsledkem velké zpomalení při několikanásobném volání virtuálních metod rodičovských tříd. Toto zpomalení se pak velmi projeví, neboť přístup k datovým položkám objektů je základem sázecích algoritmů. Řešením na úrovni Perlu by bylo přejít na manipulaci s poli bez použití objektů, což by se opět odrazilo především na čitelnosti kódu. Pokud bychom udělali takovýto krok a implementovali systém způsobem podobným TEXu, bylo by pak samozřejmě nejlepší zvolit i jiný programovací jazyk, pravděpodobně C.

Nedostatečný výkon implementace se projevil již na začátku a vedl ke zvažování, zda nezvolit jiné programovací prostředí. Protože ale při tvorbě prototypu nešlo o rychlost reálných výpočtů, ale spíše o správnost algoritmů, byla nakonec dána přednost Perlu. Na několika místech modulu `BGP` jsou tak použity spíše čitelné a objektově správné postupy, i když by jiná řešení byla pravděpodobně rychlejší.

TEX je při sazbě rychlejší také z toho důvodu, že informace například o metrikách fontů či vzorech dělení jsou předkompilovány ve formátu, což je binární obraz části TEXovské paměti. I zde, při načítání `.tfm` souboru či vzorů dělení, byla rychlostní převaha TEXu patrná.

Jako důležitější při posuzování výsledků prototypu se pak ukázalo relativní porovnání efektivnosti systému v případech, kdy byly použity navržené rozšiřující funkce, s případy, kdy byly v systému potlačeny. K opravám stránkových zlomů, končících parchantem, bylo přikročeno průměrně v jedné čtvrtině případů a ve třech čtvrtinách z toho bylo nalezeno alternativní řešení změnou `\looseness` v některém z předchozích odstavců. Četnost případů, kdy bylo nutno opravit stránkový zlom na rozděleném slově, byla velmi zavislá na nastavení penalty za řádkový uvnitř slova. Pro penaltu rovnou nule takto končila polovina stránek, pro hodnotu 80 již pouze 15 procent.

Vstupní text byl také formátován do stran se specifikovanými výřezy. Ukázalo se, že při pevném řádkovém rejstříku je hodnota `\baselineskip` dostatečně kvalitním vodítkem pro

nastavení parametrů iniciálního řádkového zlomu odstavce. Vzhledem k tomu, že byl sázen hladký text, kde výšky a hloubky řádků byly v očekávaném rozmezí, byly všechny průchody sázecím systémem pro strany s výřezy přímočaré, bez nutnosti backtracingu.

Jako vstupní text pro výše uvedené testy pevného řádkového rejstříku sloužily části knihy Karla Čapka Povídky z jedné kapsy [Čapek 64].

Abychom ověřili chování systému při sazbě do neobdélníkových stran i u dokumentů, kde není řádkový rejstřík pevný, byly některé odstavce sázeny i s nastavením `\baselineskip` na hodnotu `12pt plus 5cm`, tedy vlastně ekvivalent TEXovského `fil`, a s hodnotou `\bottomskip` rovnou nule. Podobným způsobem se sázejí například plakáty či programy koncertů, kdy chceme, aby relativně řídký text vyplnil celou plochu. Zde již docházelo k násobným backtracingům, jak se strana postupně zaplňovala dalšími odstavci. Velmi často byly ale při opětovném požadavku na zalámání požadovány stejné délky řádků, jako v některém z již provedených průchodů. Každý odstavec byl na takovéto straně zalámán průměrně dvakrát, dosažené maximum pro jeden odstavec bylo šest pokusů.

Tyto výsledky dávají dobrý předpoklad, že pokud bychom byli schopni dosáhnout rychlejší implementace systému jako celku, neznamenaly by rozšiřující funkce vážné zpomalení. Důvodem nízkého výkonu byla časově náročná vnitřní implementace, nikoli chybně navržené postupy a algoritmy.

10. Závěr

Tato diplomová práce ukázala, že je možné dosáhnout zlepšení výsledků práce a rozšíření funkcí sázecího systému bez velkých dopadů na rychlost jeho práce. Použili jsme metodu backtrackingu, která dovoluje v průběhu výpočtu reagovat na nové podmínky, i když se objeví teprve v průběhu zpracování strany či odstavce.

Pro implementaci prototypového systému byl použit jazyk Perl a zapojeny jeho modulární a objektové možnosti. Díky tomu bylo možné studovat chování navržených postupů a testovat jejich výsledky. Pro praktické uplatnění v reálných systémech počítačové sazby by ale bylo pravděpodobně nezbytné upustit od elegantního řešení a dát přednost výkonnějším postupům a programovacím jazykům.

Hlavní přínos práce spočívá jednak v analýze existujících sázecích algoritmů, jednak v označení míst, která je možné zlepšit. Navrhli jsme teoretická řešení (kapitola 8) a ověřili jejich platnost na prototypovém systému.

Jan Pazdziora
Brno, duben 1997

Literatura

[Chris 97] Tom Christiansen: Stránka dokumentace a FAQ k Perlu, dostupná na URL `http://www.perl.com/perl/info/documentation.html`

[Čapek 64] Karel Čapek: *Povídky z jedné kapsy, Povídky z druhé kapsy*, Státní nakladatelství krásné literatury a umění, Praha, 1964

[Feynm 92] Richard P. Feynman: *Surely You're Joking, Mr. Feynman*, Vintage, 1992, ISBN 0–09–917331–X

[Klein 95] Anne Brüggemann-Klein, Rolf Klein, Stefan Wohlfeil: Pagination reconsidered, Electronic Publishing, vol. 8 (2&3), 139–152, John Wiley & Sons, 1995, CCC 0894–3982/95/020139–14

[Knuth 80] Donald E. Knuth, Michael F. Plass: Breaking Paragraphs Into Lines, Stanford University, CA 94305, 1980, Report No. STAN–CS–80–828

[Knuth 73] Donald E. Knuth: *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, 1973, ISBN 0–201–03803–X

[Knuth 84] Donald E. Knuth: *TEXbook*, Addison-Wesley, 1984, ISBN 0–201–13447–0

[Knuth 86] Donald E. Knuth: *TEX: The Program*, Addison-Wesley, 1986, ISBN 0–201–13437–3

[Knuth 88] Donald E. Knuth: The Errors of TEX, Stanford University, CA 94305, 1988, Report No. STAN–CS–881223

[Olšák 95] Petr Olšák: *Typografický systém TEX*, $\mathcal{CS}$TUG, 1995, ISBN 80–901950–0–8

[Olšák 96] Petr Olšák: *TEXbook naruby*, Public edition, 1996, text je dostupný na URL `http://math.feld.cvut.cz/olsak/tbn.html`

[Plass 81] Michael F. Plass: Optimal Pagination Techniques for Automatic Typesetting Systems, Stanford University, CA 94305, 1981, Report No. STAN–CS–81–870

[Strong 88] Bryan Strong, Jay Hosler: *The UNIX^{TM} word processing book*, John Wiley & Sons, 1988, ISBN 0–471–85795–5

[Wall 96] Larry Wall, Tom Christiansen, Randal L. Schwartz: *Programming Perl*, 2nd Edition, O'Reilly & Associates, 1996, ISBN 1–56592–149–6

[Wall 97] Larry Wall: `perl(1)`: Perl — Practical Extraction and Report Language, manuálová stránka

[Zabala 82] Ignacio A. Zabala: Interacting with graphic objects, Stanford University, 1983, Report No. CSUG83–B60546

[Zakhar 96] Ilya Zakharevich: Dokumentace k modulu `Text::TeX`, modul je zveřejněn na CPANu v adresáři `CPAN/authors/id/ILYAZ/`