Implementation and comparison of grammar compression algorithms

Michalis Kamburelis

14 czerwca 2014

1 Contact

Email: michalis.kambi@gmail.com
WWW: https://github.com/michaliskambi/grammar-compression

2 Overview

The program `grammar_compression` performs compression / decompression using grammar-based algorithms: Sequitur and Sequential. Rytter's algorithm is not implemented.

For testing purposes there is also available algorithm named `none` that simply constructs grammar with one production (non-terminal) that expands to a sequence of terminals corresponding to original text. Sometimes even such algorithm can achieve slight compression because the symbols are packed using as few as possible bits per symbol (see below).

To see usage description, available command-line options etc. just run

```
grammar_compression --help
```

2.1 Text encoding of grammar graph

Compression algorithms based on grammars should have the advantage that they „discover" the structure of our document. Actually all compression algorithms try in some way to discover some document structure, but grammar-based algorithms should do it particularly well. We can dream about grammar-compression algorithm that will be able to reconstruct actual grammar that was used to validate the document. For example if the document is a program source code we can dream about discovering the grammar of used programming language, or we can dream about recovering structure from documents using markup languages (SGML, XML).

That's why I implemented writing the calculated grammar to a text file with a syntax understood by GraphViz. Grammar is recorded as a directed graph.

For example the command

```
grammar_compression --output-graph a.dot input_file
```

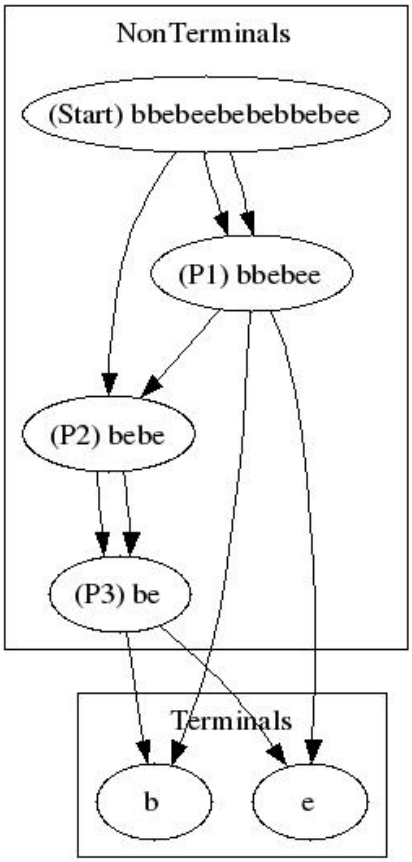
will write to the file `a.dot` the discovered grammar of input_file.

Such graph can be processed and viewed by programs from GraphViz package, e.g. the command

```
dot -Tjpg -oa.jpg a.dot
```

will generate picture `a.jpg` that shows the grammar graph.

Example graph for input text `bbebeebebebbebee`:



Note: Picture `a.jpg` shows for each production in the grammar which symbols (terminals and non-terminals) are used and how many times. The information about in what order the symbols are used in the production is lost. [1] So if you want to see the order of used symbols, you have to simply

[1] There is no way to show it clearly — we would have either to cross the arrows, or duplicate the nodes, which would quickly lead to obfuscated graph pictures. Actually this is a consequence of the fact that I „overuse" here the idea of a „graph" — after all in normal graphs the edges are not ordered in any way, and we are talking here exactly about showing such order.

view the text version of the grammar in file `a.dot`.

Note 2: writing the grammar file for large grammars is relatively slow (mainly because they are just very large files...). So you should not judge the speed of the compression algorithms by looking at the speed of writing the grammar to a text file.

2.2 Binary encoding of grammar (i.e. actual compressed format)

To measure the compression ratio the grammar is encoded in simple binary format. Note that I don't use NMW grammar encoding, since NMW can produce actually slightly larger output file (advantage on NMW is easy on-the-fly decompression, but this is not needed by this project).

Details:

1. I assign indexes for all non-terminals (i.e. productions), starting from 0. Initial grammar symbol always gets index 0. Then all *actually used in text* terminals get assigned further indexes.

2. At the beginning of the file I write the count of productions (this takes 4 bytes). Then I write bit table indicating which chars were used as terminals (this takes 256 / 8 = 32 bytes). This way each compressed file gets 36 bytes header. When decompressing, this header tells me how many productions are present, which terminals were used, and I can figure from this what indexes were assigned to each terminal.

   Of course for very small files 36-byte header can easily make the „compressed" file larger than original file. In a real compression program this is easy to avoid [2] but in our case this is not needed, after all we want to test grammar algorithms so we always want to see some grammar.

3. After the header I write each production as a sequence of indexes for appropriate symbols (non-terminals or terminals). End of each production is marked with index 0. Strictly speaking, 0 is the index of initial grammar production, but we know that it can't occur anywhere on the right side of any production (this would lead to infinite grammar expansion). So I „overuse" index 0 here to separate productions.

4. If you want to actually see the exact indexes written to the file you can compile the program with `DEBUG_BINARY_SAVE` symbol defined.

5. I tried hard to use as few (and as small) indexes to point to particular non-terminals and terminals.

[2] Like this: if the pathological case occurs then write some special value in the first byte of the file and then simply append uncompressed contents of original file. This way overhead of compressed file is at most 1 byte.

Moreover these indexes are recorded using as few bits as possible (with constant amount of bits per symbol). For example if the grammar uses only 1 production (i.e. only the initial production) and 3 terminals, then I'll need only 2 bits for every index. So I'll squish 4 indexes in 1 byte.

This simple coding means that even when the compression algorithm will be poor (for example, special `none` algorithm that always generates grammar with one production that simply expands to whole desired sequence of terminals) we can still get some compression if the file used only a small subset of possible 256 characters.

Writing indexes using constant number of bits is implemented in unit `bitstreams.pas`.

3 Notes about compiling and source code

- You should compile everything using FreePascal, use FPC version suitable for Castle Game Engine — usually it is best to just use latest stable FPC version. Tested under Linux, but the whole code is portable and should compile and work under any Unix and Windows. You can compile using commands

  ```
  make build-debug
  ```

  or

  ```
  make build-release
  ```

- Run `make pasdoc` to generate documentation of units in this project. You will need pasdoc. The generated documentation will be placed in the subdirectory `apidoc/`.

I'm sorry, but the rest of this document is still in Polish. If there is enough interest, I'll translate it to English — so don't hesitate to drop me email if you found this program useful.

4 Szczegóły implementacji

Źródła są opatrzone komentarzami, więc poniżej wymienię tylko bardziej ogólne i znaczące uwagi/problemy/pułapki.

4.1 Sequitur i Sequential

Problemem zasługującym na uwagę jest rekurencyjność wszystkich operacji poprawiających. Dodawanie digramu może spowodować podstawienie produkcji i rozwinięcie produkcji, a podstawienie i rozwinięcie mogą powodować dodawanie digramów. W związku z czym musimy pisać bardzo ostrożnie, bo każda operacja potencjalnie zmieni postać produkcji na której aktualnie pracujemy. Np. prosty kod sprawdzający „bezużyteczność" produkcji i ew. rozwijający je wyglądał w naiwnej wersji tak:

```
  ExpandIfUnderusedProduction(NewProduction.FirstSymbol);
  ExpandIfUnderusedProduction(NewProduction.LastSymbol);
```

W pierwszym podejściu, aby mieć pewność że kod jest poprawny, musiałem go rozwinąć do

```
if ExpandIfUnderusedProduction(NewProduction.FirstSymbol, false) then
  ExpandIfUnderusedProduction(NewProduction.LastSymbol, true);
```

po czym, kiedy udowodniłem sobie że drugi symbol nigdy nie będzie potrzebował rozwinięcia (patrz pod koniec implementacji `CorrectDigraph`), kod został zmieniony do prostego

```
  ExpandIfUnderusedProduction(NewProduction.FirstSymbol);
```

Podobna sytuacja zachodzi pod koniec `Substitute` gdy musimy wywołać `CorrectDigraph` dwa razy (ponieważ wstawienie symbolu tworzy dwa nowe digramy, przed i po wstawionym symbolu). Naiwny kod wygląda tak:

```
  CorrectDigraph(S1);
  CorrectDigraph(S2);
```

ale w praktyce musimy napisać

```
if CorrectDigraph(S1) then
  CorrectDigraph(S2);
```

Co oznacza że jeżeli wstawienie S1 spowodowało rekurencyjne wywołanie `Substitute`, to nie wstawiamy już drugiego digramu. Patrz komentarze pod koniec `Substitute` po uzasadnienie dlaczego tak jest poprawnie (tzn. wbrew pozorom nie naruszamy żadnej właściwości algorytmu, digram S2 tak naprawdę zostanie dodany; ponadto mamy gwarancję że nie będziemy operować na S2 po pierwszym `CorrectDigraph(S1)`, które de facto może sprawić że obiekt S2 już nie będzie istniał).

Konsekwencją dwóch powyższych sytuacji jest np. fakt że `CorrectDigraph` zwraca wartość boolowską (zamiast być zwykłą procedurą która kompletnie „ukrywa" wykonywane przez siebie modyfikacje na gramatyce), oraz że metoda DeleteDigraph nie może zawierać asercji w rodzaju „usuwany digram musi istnieć w tablicy".

5 Testy

5.1 Testy poprawności

Skrypt `mk_test.sh`, wspomagając się programem `mk_test_file.dpr`, wykonuje automatyczne testy. Produkowane są pliki o różnych rozmiarach i używające różnej ilości losowych znaków. Każdy taki plik jest kompresowany (każdym dostępnym algorytmem), następnie dekompresowany i wynik dekompresji jest porównywany z oryginalnym plikiem.

Oczywiście program przechodzi powyższe testy dla wszystkich algorytmów.

5.2 Testy jakości i szybkości kompresji

Pliki testowe:

1. Pliki `test_binary_N` dla N = 10 tysięcy, 100 tysięcy, milion: pliki binarne losowe, każdy wygenerowany przez `mk_test_file 256 N > test_binary_N`. Intuicyjnie: pliki najtrudniej kompresowalne (dla wszystkich algorytmów, nie tylko gramatykowych).

2. Pliki HTML: `kompresja06.html` — z http://www.ii.uni.wroc.pl/~tju/KomprDz06/kompresja06.html, czyli plik HTML prosty, `kompresja_danych_table.html` — z http://www.ii.uni.wroc.pl/~marcinm/dyd/kompresja/, obcięty tylko do tabelki (od `<table>` do `<\table>`), czyli plik HTML z regularną tabelą, `google_result.html` — wynik googla dla `fpc`, czyli plik HTML nieregularny, dużo niezwiązanego tekstu — powinien być trudniejszy do kompresji od poprzednich dwóch.

3. Pliki źródłowe: w Pascalu `seqcompression.pas`, w C `SDL_error.c` (dość nieregularny (jak to w C) plik z SDL).

5.2.1 Testy jakości kompresji

Legenda do kolumn w tabelce poniżej:

1. Nazwa pliku.
2. Rozmiar oryginalny (w bajtach).
3. Stopień kompresji, czyli „rozmiar skompresowany" / „rozmiar oryginalny".

Uwagi:

- Ostatnia kolumna zawiera zawsze 3 wartośći, dla każdego algorytmu: `none` (dla porównania), `sequitur`, `sequential`.
- Dla algorytmu Sequential czas działania był bardzo duży dla większych danych (`test_binary_100000` i `test_binary_100000`) więc nie przeprowadziłem testów.

test_binary_10000	10000	1.13	1.32	1.32
test_binary_100000	100000	1.13	1.87	?
test_binary_1000000	1000000	1.13	1.43	?
kompresja06.html	2458	0.89	0.43	0.41
kompresja_danych_table.html	41676	0.88	0.04	0.04
google_result.html	20893	0.88	0.52	0.49
seqcompression.pas	12076	0.88	0.48	0.44
SDL_error.c	7578	0.88	0.58	0.55

5.2.2 Testy szybkości kompresji

Legenda do kolumn w tabelce poniżej:

1. Nazwa pliku.
2. Szybkość kompresji. Mierzona jako iloraz: rozmiar oryginalny / czas kompresji w sekundach. Czyli „ile bajtów na sekundę kompresujesz". Podana w tysiącach („k").
3. Szybkość dekompresji. Mierzona jak wyżej, czyli „ile bajtów na sekundę dekompresujesz". Uwaga: algorytm dekompresji jest zawsze taki sam, dla wszystkich algorytmów kompresji (bo w pliku zapisana jest tylko gramatyka). Ale być może gramatyki produkowane przez niektóre algorytmy są prostsze do dekompresji ?

Uwagi:

- Jak powyżej, dla testów jakości: podane są 3 czasy dla 3 algorytmów. Algorytm Sequential działał zbyt długo więc w niektórych miejscach ma „?".

- Testy były przeprowadzone dla programu skompilowanego z `PRECISE_MEASURE_TIME` (patrz źródła `grammar_compression.dpr`), każdy test był wykonany 10 razy i wypisane czasy są liczone tak aby być średnią ze wszystkich testów.

test_binary_10000	322 k	138 k	12 k	909 k	769 k	833 k
test_binary_100000	336 k	139 k	?	1030 k	584 k	?
test_binary_1000000	323 k	59 k	?	1071 k	320 k	?
kompresja06.html	351 k	59 k	39 k	614 k	819 k	2458 k
kompresja_danych_table.html	400 k	217 k	356 k	1096 k	8335 k	10419 k
google_result.html	373 k	155 k	10 k	409 k	1899 k	1741 k
seqcompression.pas	389 k	143 k	17 k	1006 k	575 k	1725 k
SDL_error.c	378 k	118 k	19 k	947 k	1263 k	1263 k

5.2.3 Wnioski

- Patrząc na linie dla plików `test_binary_N` widać jasno że algorytmy gramatykowe są dla nich nieopłacalne: stopnie kompresji są gorsze nawet od algorytmu `none`. Algorytm `none` ma stopień kompresji > 1 ponieważ zużywa 9 bitów na 1 znak (ponieważ koduje 256 znaków + znak końca produkcji, a więc potrzebne 9 bitów).

- Algorytm Sequential nie wypadł najlepiej jako ulepszenie Sequitur. Jego stopnie kompresji są tylko minimalnie lepsze od Sequitur, a czas kompresji jest zazwyczaj znacznie gorszy od Sequitur.

- Algorytm Sequitur: dla plików tekstowych osiąga dobrą kompresję, tzn. około dwa razy w porównaniu z algorytmem `none`. Czas kompresji można uznać za dobry — 2–3 dla dobrych plików. Inna sprawa że tradycyjnie używane algorytmy słownikowe radzą sobie dużo lepiej, i z jakością i z szybkością kompresji.

- Na uwagę zasługuje kompresja pliku `kompresja_danych_table.html`: wynik algorytmów gramatykowych jest świetny, stopień kompresji wynosi 0.04. Patrząc na graf wygenerowany przez

  ```
  grammar_compression kompresja_danych_table.html -g a.dot
  ```

  widzimy dużo skojarzeń.

- Niestety, nawet w przypadku `kompresja_danych_table.html`, wygenerowana gramatyka nie przypomina w żaden sposób naszej „oryginalnej" gramatyki, tzn. struktury HTMLa. Marzenia o odkryciu gramatyki HTMLa lub języka programowania sprzed kilku stron okazują się nierealne.

Aby algorytm kompresji gramatykowej działał naprawdę dobrze, należałoby go bardziej dostosować do konkretnych danych. Np. ująć a algorytmie stwierdzenia:

- Ilość białych znaków nie ma znaczenia.
- Znaczniki HTMLa zazwyczaj występują w parach — otwierający i zamykający. W przypadku XML „zazwyczaj" możemy zmienić na „zawsze".