Kompresja gramatykowa i balansowanie: eksperymentalny framework

Maria Cicirko (268386), Nikola Reguła (268382), Aleksandra Rudnik (288831)

22 grudnia 2025

Streszczenie

Ten projekt implementuje niewielki, eksperymentalny framework do kompresji opartej na gramatykach oraz późniejszego balansowania SLP (ang: straight-line program). Projekt zawiera kilka klasycznych metod kompresji, które dla danego napisu konstruują wyprowadzające go SLP. Wyniki poszczególnych metod są następnie porównywane pod względem własności strukturalnych: rozmiaru gramatyki (liczby nieterminali) oraz głębokości drzewa wyprowadzeń. Osobny algorytm balansowania, wprowadzony w niedawnej pracy [Ganardi, Jeż, Lohrey, JACM 2021], przyjmuje jako wejście niezbalansowaną gramatykę i przekształca ją w równoważną gramatykę o logarytmicznej głębokości, przy niewielkim wzroście rozmiaru.

Ideą przewodnią projektu jest skupienie się na kształcie i własnościach gramatyk wytwarzanych przez różne kompresory, a nie na niskopoziomowej złożoności bitowej. W konsekwencji implementacja pomija kwestie takie jak encoding gramatyki na poziomie bitów, czy optymalizacja pod kątem szybkości i użycia pamięci. Zamiast tego celem jest dostarczenie przejrzystych, wysokopoziomowych implementacji w Pythonie, wygodnych porównań oraz wizualizacji drzew wyprowadzeń.

1 Motywacja i zakres

Celem kompresji gramatykowej jest reprezentacja napisu za pomocą gramatyki bezkontekstowej, która wyprowadza jego i tylko jego. Taki rodzaj gramatyki nosi nazwę *SLP* od angielskiego *straight-line program*. Reprezentacja tekstu jako SLP jest rodzajem kompresji bezstratnej. W przypadku tekstów o „niskiej entropii", reprezentacje SLP mogą być znacznie mniejsze niż oryginalny tekst, wykorzystując jego repetytywność (powtarzające się fragmenty).

Z perspektywy teorii informacji, efektywna bezstratna kompresja jest oczywiście niemożliwa dla dowolnych, nieregularnych danych. Zadaniem wszystkich bezstratnych algorytmów kompresji jest zidentyfikowanie i wykorzystanie owej regularnej struktury danych wejściowych.

Różne metody kompresji gramatykowej różnią się od siebie rozmiarem oraz strukturą gramatyki, którą tworzą. Wpływa to nie tylko na stopień kompresji, ale również na trudność operowania na skompresowanej reprezentacji. Najczęściej rozważanymi problemami, przy rozwiązaniu których znaczenie ma np. głębokość drzewa wyprowadzeń, są: dostęp do wybranej pozycji w oryginalnym tekście (random access) oraz wyszukiwaniem wzorca (pattern matching) w oryginalnym tekście.

Celem tego projektu *nie* jest zbudowanie szybkiego ani kompletnego działającego kompresora. Zamiast tego projekt ma stanowić czysty, samowystarczalny framework zawierający:

- implementację kilku klasycznych metod kompresji gramatykowej,
- implementację nowoczesnego algorytmu balansowania SLP, zmniejszającego głębokość gramatyki,

- metody porównywania otrzymanych przez różne metody gramatyk (również po dodatkowym balansowaniu) pod względem rozmiaru i głębokości,

Kluczową decyzją projektową jest niełamanie abstrakcji gramatyki i operowanie na wysokim poziomie. Choć projekt dotyczy "metod kompresji", nigdy nie mierzymy rzeczywistego współczynnika kompresji w bitach. Są ku temu dwa powody:

1. Aby uzyskać rzetelne empiryczne współczynniki, należałoby dobrać starannie duże zbiory danych z rzeczywistego świata (np. powtarzalny tekst języka naturalnego lub sekwencje DNA) oraz zaprojektować eksperymenty, które rozdzielają efekty algorytmiczne od własności danych.

2. Aby mierzyć rozmiar skompresowany w bitach, trzeba zdefiniować i zaimplementować konkretny sposób kodowania gramatyki. Naiwna, tekstowa reprezentacja gramatyki byłaby empirycznie większa nawet od bardzo powtarzalnych napisów nad małym alfabetem w nieskompresowanej formie. Osiągnięcie konkurencyjnej kompresji na poziomie bitów wymagałoby dodatkowej warstwy niskopoziomowego encodingu oraz starannej inżynierii, która bardziej pasowałaby do projektu z innej dziedziny.

Z podobnych powodów projekt nie próbuje mierzyć *czasu* potrzebnego na wspomniany wcześniej dostęp do pozycji w skompresowanym tekście. W praktyce zależałoby to od implementacji dostępu nad zserializowaną reprezentacją gramatyki, przy czym efekty stałych oraz niskopoziomowe optymalizacje odgrywałyby dominującą rolę, większą od użytego w pierwszej fazie algorytmu konstrukcji gramatyki. Dla uproszczenia, dostęp losowy modelujemy jako naiwnie schodzenie po od korzenia do liścia w drzewie wyprowadzeń, a projekt mierzy jedynie *głębokość drzewa* jako abstrakcyjne przybliżenie kosztu takiej operacji.

Kod jest celowo napisany w Pythonie i nie jest zoptymalizowany pod kątem szybkości. Implementacja w innym języku z naciskiem na wydajność oraz zaprojektowanie szybkich struktur danych, przesunęłoby część techniczną projektu w stronę niskopoziomowych detali programistycznych kosztem teorii gramatyk, która jest głównym przedmiotem zainteresowania.

2 Implementacja klasy (RL-)SLP

Główną strukturą danych w projekcie jest klasa `SLP`, która reprezentuje dane SLP. Wbrew swojej nazwie, klasa ta jest tak naprawdę reprezentacją bardziej ogólnego RLSLP (run-length straight-line program), które dopuszcza reguły typu run-length, opisane poniżej. (RL-)SLP można postrzegać jako gramatykę bezkontekstową w następującej postaci: ma skończony zbiór nieterminali, pojedynczy symbol startowy, a każdy nieterminal rozwija się albo do terminala, albo do konkatenacji dwóch nieterminali (lub, w rozszerzeniu run-length, wielu kopii jednego nieterminala). Niskopoziomowo nieterminale są indeksowane dodatnimi liczbami całkowitymi. Reguły są przechowywane w tablicy `rules` par (a, b) o następującym znaczeniu:

- $a = 0$: reguła terminalna, nieterminal (zwany również *preterminalem*) rozwija się do pojedynczego terminala b,
- $a > 0$: reguła binarna, nieterminal rozwija się do konkatenacji nieterminali a oraz b,
- $a < 0$: reguła run, nieterminal rozwija się do $(-a)$ kopii b (reguła długości serii).

Klasa udostępnia metody tworzenia nowych nieterminali i przypisywania reguł terminalnych, binarnych lub run-length, a także automatycznego tworzenia preterminali (jeden nieterminal na każdy unikalny terminal). Klasa śledzi długość rozwinięcia każdego nieterminala i wspiera następujące metody:

- `length(nt)`: długość napisu wyprowadzanego przez `nt`,
- `access(i)`: dostęp losowy do i-tego terminala napisu wyprowadzanego, zrealizowany przez schodzenie po drzewie wyprowadzeń,
- `size()`: liczbę nieterminali (włącznie z preterminalami),
- `depth()`: wysokość drzewa wyprowadzeń zakorzenionego w symbolu startowym.

W projekcie termin *rozmiar* gramatyki zawsze odnosi się do łącznej liczby nieterminali, a *głębokość* odnosi się do głębokości drzewa wyprowadzeń. Są to dwie główne miary porównywane w eksperymentach.

Wszystkie algorytmy kompresji w tym projekcie produkują gramatyki w postaci normalnej Chomsky'ego (CNF): reguły są binarne lub run-length, a terminale pojawiają się wyłącznie poprzez preterminale. Do reprezentacji produkcji z wieloma nieterminalami po prawej stronie używana jest osobna funkcja pomocnicza konstruująca zrównoważone binarne drzewo konkatenacji, aby końcowe SLP pozostało w CNF.

3 Zaimplementowane algorytmy kompresji

Niech n oznacza długość wejściowego napisu. Projekt implementuje trzy rodzaje kompresorów:

1. kompresję w stylu RePair [LM99],
2. kompresję w stylu Sequitur [NMW97],
3. kompresję w stylu recompression, produkującą RLSLP [Jeż15].

Wszystkie przyjmują napis jako wejście i zwracają SLP (lub RLSLP), które go wyprowadza.

3.1 RePair

Pierwszy kompresor to prosty algorytm w stylu RePair [LM99]. Zaczynając od sekwencji preterminali reprezentujących wejściowy napis, działa iteracyjnie:

1. Zlicza wszystkie pary sąsiednich symboli (digramy, nazywane też bigramami) i znajduje najczęściej występującą parę (A, B).
2. Jeśli ta para nie występuje co najmniej dwa razy, algorytm się zatrzymuje.
3. W przeciwnym razie tworzy nowy nieterminal X z regułą $X \to AB$ i zastępuje wszystkie nienakładające się wystąpienia pary (A, B) w sekwencji symbolem X.

Powyższe punkty powtarzamy, aż żaden digram nie występuje co najmniej dwa razy. Jeśli na końcu pozostała sekwencja składa się z jednego nieterminala, staje się on symbolem startowym. W przeciwnym razie na pozostałej sekwencji budujemy zrównoważone drzewo binarne, którego korzeń jest symbolem startowym.

W praktyce wiadomo, że RePair osiąga bardzo silną kompresję na szerokiej gamie wysoce powtarzalnych danych wejściowych. W tym projekcie służy jako główny punkt odniesienia: typowo produkuje on najmniejsze gramatyki *pod względem rozmiaru*, ale bez gwarancji dotyczących głębokości. Dla starannie dobranych wejść RePair potrafi konstruować SLP, których głębokość jest liniowa względem n, co czyni go dobrym źródłem niezbalansowanych przykładów dla algorytmów balansowania.

3.2 Sequitur

Drugi kompresor jest inspirowany algorytmem Sequitur [NMW97]. Operuje na jawnym zbiorze reguł, z jedną regułą specjalną przechowującą sekwencję startową, oraz wszystkimi pozostałymi regułami reprezentującymi produkcje binarne długości 2. W każdej iteracji algorytm:

1. skanuje wszystkie reguły, aby zebrać wystąpienia wszystkich digramów (par sąsiadujących symboli),
2. wybiera digram, który pojawia się co najmniej dwa razy w dowolnym miejscu bieżących reguł,
3. sprawdza, czy istnieje już reguła o dokładnie takiej prawej stronie; jeśli nie, tworzony jest nowy nieterminal,
4. zastępuje wszystkie (nienakładające się) wystąpienia tego digramu we wszystkich regułach (z wyjątkiem jego własnej) odpowiadającym mu nieterminalem.

Powtarza się to, aż żaden digram nie pojawia się co najmniej dwa razy. Na końcu reguła startowa jest konwertowana do postaci normalnej Chomsky'ego: jeśli jej prawa strona składa się z więcej niż dwóch symboli, budujemy drzewo binarne podobnie jak w poprzednim algorytmie. Tak jak w przypadku RePair, otrzymana gramatyka jest czystym SLP bez reguł run-length.

Gramatyki zbudowane algorytmem Sequitur zwykle są tylko nieznacznie większe niż gramatyki RePair i często wyłapują powtarzające się fragmenty w intuicyjny sposób. Koncepcyjnie charakterystyczna różnica polega na tym, że RePair zawsze wybiera pojedynczy *globalnie* najczęstszy digram w bieżącej sekwencji, podczas gdy Sequitur utrzymuje niezmiennik, że żaden digram nie występuje więcej niż raz w całej gramatyce: gdy tylko ten sam digram pojawi się dwa razy (w dowolnej regule), jest wyodrębniany do własnej reguły. Podobnie jak RePair, Sequitur nie kontroluje explicite głębokości wyprowadzeń i może także dawać bardzo głębokie gramatyki na przykładach adwersarialnych.

3.3 Recompression

Trzeci kompresor implementuje algorytm w stylu recompression, w duchu [Jeż15]. Algorytm (podobnie jak RePair) operuje na „aktualnej" sekwencji nieterminali i powtarza dwa kroki kompresji:

1. Kompresuje maksymalne serie jednakowych symboli X^k do reguł run-length $Y \to X^k$. Tutaj reprezentacja jako RLSLP ma kluczowe znaczenie.
2. Wybiera podział bieżących symboli na dwie klasy (0 i 1), a następnie kompresuje wszystkie sąsiednie pary 0-1 do reguł binarnych $Z \to AB$.

Kroki te są wykonywane tak długo, jak sekwencja zawiera więcej niż jeden symbol.

W projekcje algorytm został zaimplementowany w dwóch wariantach:

- wariant losowy, który w każdej iteracji kroku 2 przypisuje symbolom losowe bity,
- wariant deterministyczny, który wyznacza przypisanie bitów zachłannie na podstawie częstości digramów, starając się zmaksymalizować liczbę skompresowalnych par.

Recompression jest w tym projekcie interesujące z dwóch powodów. Po pierwsze, produkuje RLSLP, więc może bezpośrednio wykorzystywać długie serie symboli. Pozwala to zbadać, czy dopuszczenie reguł długości serii daje zauważalne zmniejszenie rozmiaru gramatyki dla pewnych

klas wejść w porównaniu z czystymi SLP produkowanymi przez RePair i Sequitur. Po drugie, recompression konstruuje gramatyki *zbalansowane*, tj. których głębokość wynosi $O(\log n)$ dla napisu wejściowego długości n.

Pozwala to zaobserwować powstające płytkie drzewa wyprowadzeń i zestawić je z potencjalnie bardzo głębokimi gramatykami produkowanymi przez RePair oraz Sequitur.

4 Balansowanie

Poza porównaniem gramatyk wytwarzanych przez kompresory, centralnym elementem projektu jest algorytm balansowania gramatyk, który przyjmuje dowolne SLP jako wejście i przekształca je w równoważne SLP o logarytmicznej głębokości. Algorytm jest inspirowany niedawnym wynikiem, który pokazuje, że każde SLP można zbalansować do głębokości $O(\log n)$, zwiększając jego rozmiar jedynie o stały czynnik [GJL21, Theorem 1.2]. Idea algorytmu jest następująca:

1. **Symetryczna dekompozycja heavy-light.** Dla każdego nieterminala algorytm zlicza liczbę ścieżek wyprowadzeń od symbolu startowego do tego nieterminala. Na jej podstawie, oraz na podstawie długości rozwinięcia, każdy wierzchołek (nieterminal) wyznacza co najwyżej jedno *ciężkie dziecko*: krawędź jest ciężka, jeśli pewne logarytmiczne sygnatury (oparte na liczbie ścieżek i długości) są zgodne. Gramatyka jest dekomponowana na zbiór rozłącznych wierzchołkowo ciężkich ścieżek.

2. **Budowa skrótów wzdłuż ścieżki.** Dla każdej ciężkiej ścieżki algorytm rozważa podłączone do niej "boczne poddrzewa" (lewe i prawe dzieci, które nie leżą na ścieżce). Następnie wywołuje rekurencyjną procedurę pomocniczą, która metodą dziel-i-zwyciężaj buduje nowe nieterminale rozwijające się do konkatenacji coraz większych fragmentów „sufiksów bocznych poddrzew". Implementuje to wariant konstrukcji gramatyki sufiksowej: dla listy symboli tworzy nieterminale, które w zrównoważony sposób wyprowadzają wszystkie sufiksy [GJL21, Proposition 2.3].

3. **Przepisanie reguł wzdłuż ścieżki.** Na koniec, używając nowo utworzonych nieterminali, produkcje wzdłuż ciężkiej ścieżki są przepisywane tak, aby rozwinięcie każdego węzła na ścieżce było wyrażone przez niewielką liczbę konkatenacji z końcem ścieżki oraz bocznymi poddrzewami. Skutecznie spłaszcza to ścieżkę i sprawia, że jej wkład do całkowitej głębokości wyprowadzeń staje się logarytmiczny.

Zastosowanie tej procedury do wszystkich ciężkich ścieżek daje nową gramatykę wyprowadzającą ten sam napis, ale o głębokości $O(\log n)$, gdzie n jest długością wyprowadzanego napisu.

Transformacja działa w miejscu: modyfikuje istniejące SLP przez dodawanie nowych nieterminali i aktualizowanie reguł.

4.1 Balansowanie najdłuższej ścieżki

Oprócz algorytmu opartego na ciężkich ścieżkach, projekt zawiera niewielki wariant eksperymentalny, który najpierw wyznacza pojedynczą najdłuższą ścieżkę w drzewie wyprowadzeń, a następnie stosuje tę samą procedurę balansowania ścieżki tylko do niej. Transformacja ta jest prostsza i nie musi gwarantować logarytmicznej głębokości globalnie, ale jest użyteczna jako porównanie: pokazuje, jak duża część redukcji głębokości jest osiągana już przez aplikację balansowania tylko na najdłuższej ścieżce oraz jaką dodatkową korzyść daje zastosowanie pełnej dekompozycji heavy-light.

5 Dane testowe

Aby badać interakcję między kompresją a balansowaniem, wygodnie jest mieć wejście, na którym RePair (i podobnie Sequitur) naturalnie produkuje bardzo głęboką gramatykę. Projekt zawiera więc parametryczny generator wejść adwersarialnych dla algorytmu RePair.

Dla danej liczby całkowitej n generator najpierw buduje sekwencję "bloków" o rosnącej długości:

$$(1,2),\ (1,2,3),\ (1,2,3,4),\ \ldots,\ (1,2,\ldots,n),$$

przy czym dwa ostatnie bloki mogą się pokrywać. Kluczowa idea polega na tym, że RePair jest zmuszony najpierw skompresować parę $(1,2)$, potem powstały nieterminal wraz z symbolem 3, następnie ten wynik wraz z symbolem 4 itd. W konsekwencji SLP produkowane przez RePair ma rozmiar rzędu n, ale jego głębokość jest także rzędu n, podczas gdy wyprowadzany napis ma długość rzędu n^2. Czyni to instancję jawnie adwersarialną względem głębokości gramatyki.

Generator można skonfigurować flagami dodającymi "szum":

- zamiast zawsze rozszerzać bloki z prawej strony, każdy blok może być lewo- albo prawostronnym rozszerzeniem swojego poprzednika,
- końcowa lista bloków może zostać przetasowana w losowej kolejności.

Opcje te zmieniają powierzchowny wygląd napisu, ale zachowują istotne zachowanie RePair: uruchamiany jest ten sam zagnieżdżony wzorzec kompresji par, prowadząc do bardzo głębokiej gramatyki dobrze nadającej się do testowania procedury balansowania.

Prosty skrypt sterujący uruchamia wszystkie kompresory (RePair, Sequitur, losowy i zachłanny recompression) na różnych małych napisach testowych oraz na instancji konstrukcji adwersarialnej, wypisując dla każdej gramatyki długość napisu, rozmiar i głębokość oraz sprawdzając poprawność poprzez wielokrotne wywołania `access`.

5.1 Możliwe tabele i wykresy

Chociaż obecne repozytorium nie zawiera jeszcze systematycznej ewaluacji na dużych zbiorach danych, naturalne jest uwzględnienie w raporcie niewielkiego porównania na danych syntetycznych. Przykładowo można uruchomić kompresory na wejściu adwersarialnym dla kilku wartości n i zestawić:

- rozmiar i głębokość gramatyki RePair,
- rozmiar i głębokość kompresora inspirowanego Sequitur,
- rozmiar i głębokość algorytmów recompression,
- rozmiar i głębokość tych gramatyk *po* zastosowaniu algorytmu balansowania.

Przykład takiego porównania można zaobserwować w Tablicy 1.

Można też porównać głębokość gramatyki jako funkcję długości wejścia n dla rodziny adwersarialnej na wykresie, uwzględniając wynik zarówno przed, jak i po balansowaniu. Na przykładowym wykresie na Rysunku 1 możemy łatwo zaobserwować przejście od głębokości liniowej do logarytmicznej. Wykres na Rysunku 2 pokazuje zachowanie asymptotyczne rozmiarów produkowanych gramatyk. Wszystkie testy wykonano na próbkach adwersarialnych z obiema losowymi flagami szumu ustawionymi na true.

Metoda	Rozmiar	Głębokość
RePair	595	206
Sequitur	1139	147
RecompRand	2086	27
RecompGreedy	1489	16
Zbalansowany RePair	999	44
Zbalansowany Sequitur	1671	34

Tablica 1: Przykładowe porównanie na adwersarialnej instancji testowej $n = 200$, z obiema losowymi flagami szumu. Długość napisu wynosi 20098.

6 Wizualizacja drzew wyprowadzeń

Aby uzupełnić liczbowe miary rozmiaru i głębokości, projekt zawiera komponent wizualizacyjny, który eksportuje drzewa wyprowadzeń do kodu TikZ. Dla danego SLP wizualizator wykonuje przeszukiwanie w głąb po drzewie wyprowadzeń, aby przypisać współrzędne każdemu węzłowi. Liście (symbole terminalne) są umieszczane wzdłuż osi poziomej zgodnie z ich pozycją w wyprowadzanym napisie, a węzły wewnętrzne są umieszczane na całkowitych głębokościach odpowiadających ich odległości od korzenia.

Wygenerowany rysunek TikZ rozróżnia:

- węzły wewnętrzne (rysowane jako okręgi z identyfikatorem nieterminala),
- liście (rysowane z faktycznym symbolem terminalnym),
- krawędzie lekkie (domyślne krawędzie wyprowadzeń),
- krawędzie ciężkie (np. krawędzie na ciężkiej ścieżce lub na najdłuższej ścieżce), rysowane w innym kolorze,
- węzły i krawędzie wprowadzone przez algorytm balansowania (np. rysowane na zielono).

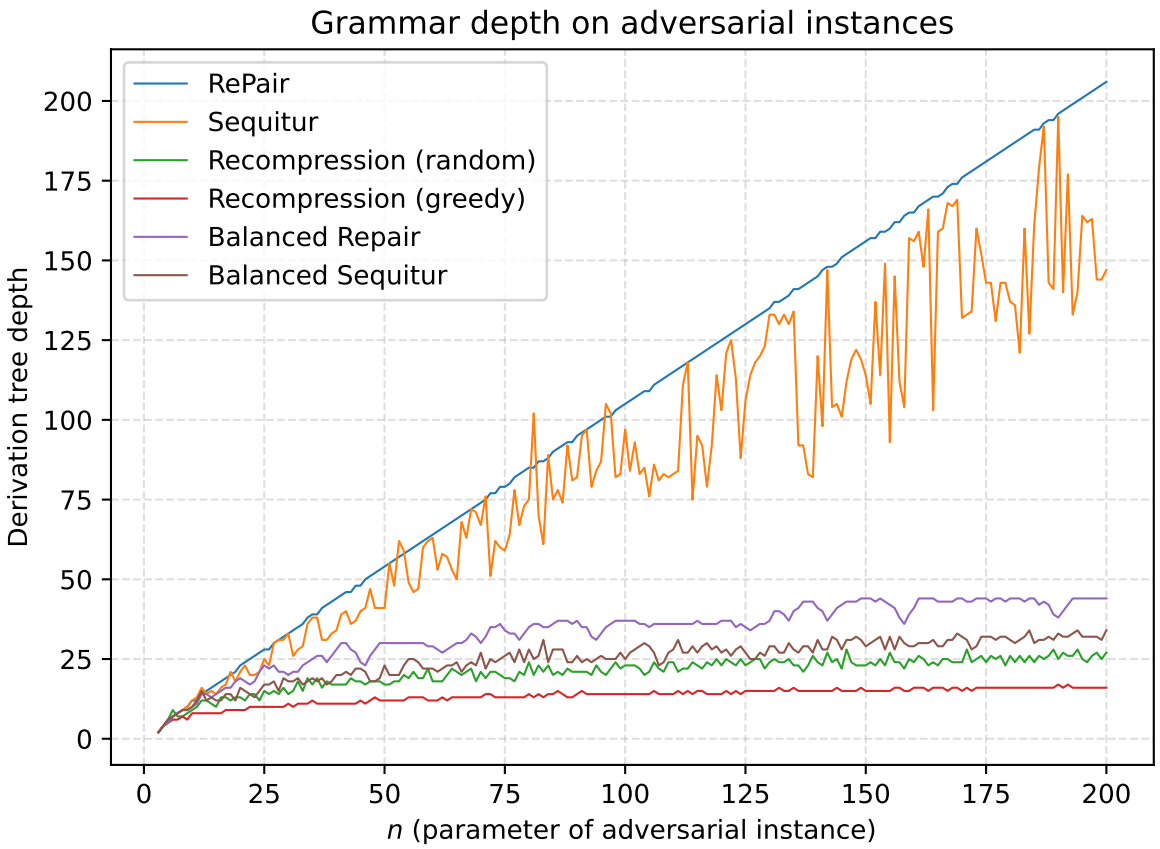
Wizualizator można łączyć z procedurami balansowania. Na przykład można:

1. wygenerować adwersarialny napis wejściowy,
2. skompresować go za pomocą RePair,
3. opcjonalnie wyróżnić ciężkie ścieżki lub najdłuższą ścieżkę,
4. zastosować algorytm balansowania (oznaczając nowo wprowadzone węzły),
5. wyeksportować wynikowe drzewo wyprowadzeń do samodzielnego dokumentu TikZ.

Ułatwia to wizualne porównanie kształtu drzewa wyprowadzeń "przed" i "po" balansowaniu. Raport może odwoływać się do takich rysunków za pomocą standardowych środowisk figure. Przykładową wizualizację można zobaczyć na Rysunku 3

7 Decyzje projektowe i ograniczenia

Ta sekcja podsumowuje najważniejsze wybory dotyczące tego, co projekt robi i czego *nie* robi.



Rysunek 1: Wykres pokazuje wyraźne rozróżnienie między metodami, których głębokości rosną logarytmicznie oraz liniowo względem n.

7.1 Implementacja w Pythonie

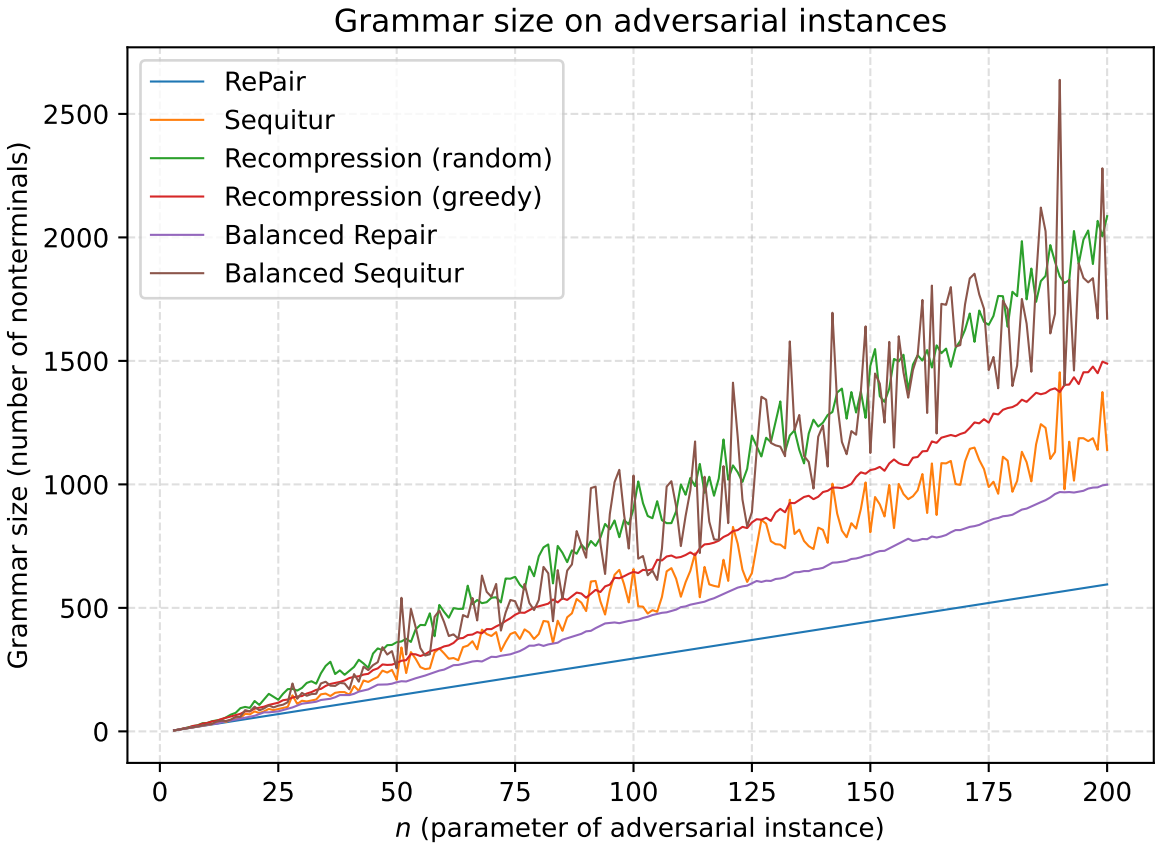
Cały projekt jest zaimplementowany w Pythonie. Ten wybór stawia na czytelność i łatwość eksperymentowania kosztem surowej wydajności. Na przykład:

- struktury danych to głównie zwykłe listy i słowniki Pythona, bez prób minimalizowania stałych czynników;
- kompresory wielokrotnie skanują sekwencje i przeliczają tabele częstości zamiast korzystać z zaawansowanych, przyrostowych struktur danych;
- rekurencja jest używana do konstruowania zrównoważonych drzew oraz do przechodzenia po gramatykach w procedurach balansowania i wizualizacji.

Choć czyni to kod nieodpowiednim dla dużych instancji rzędu gigabajtów, znacznie upraszcza zrozumienie i modyfikowanie algorytmów, co jest głównym celem projektu.

7.2 Brak współczynników kompresji

Chociaż projekt porównuje "metody kompresji", nie mierzy rzeczywistego współczynnika kompresji (stosunku rozmiaru skompresowanego w bitach do oryginalnego rozmiaru w bitach). Zamiast tego wszystkie porównania są wykonywane wyłącznie w kategoriach:



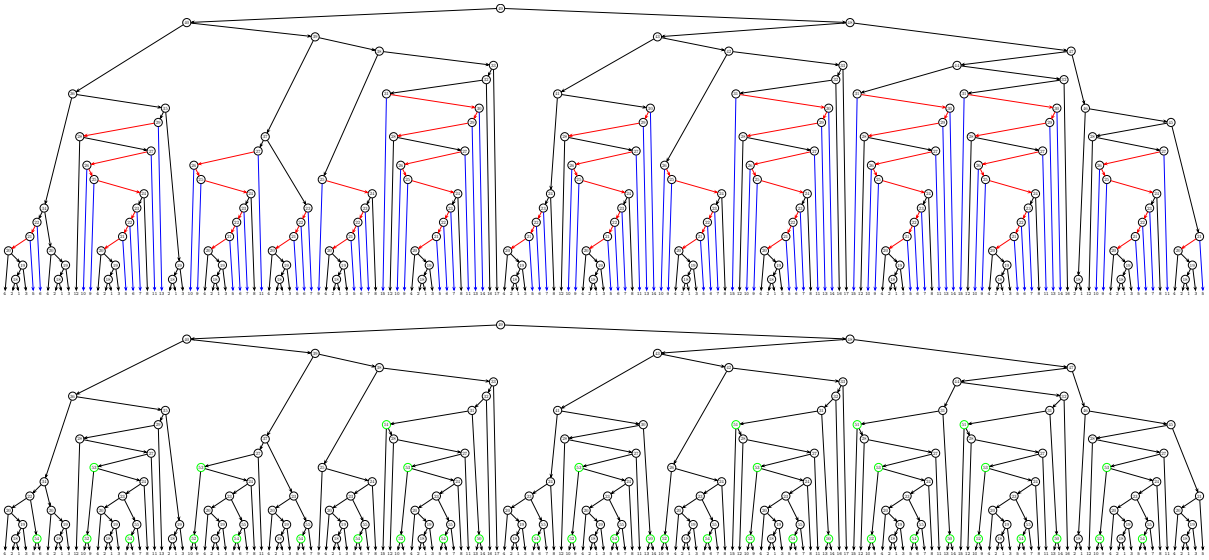
Rysunek 2: Wszystkie metody produkują (RL-)SLP o rozmiarze rosnącym liniowo względem n, ale z różnymi współczynnikami wzrostu.

- rozmiaru gramatyki: liczby nieterminali w (RL-)SLP,
- głębokości gramatyki: wysokości drzewa wyprowadzeń.

Pomiar rozmiarów na poziomie bitów wymagałby decyzji o konkretnym kodowaniu gramatyk: jak przechowywać zbiór reguł, jak kodować identyfikatory nieterminali, jak reprezentować liczności w regułach run-length, itd. Naiwne lub niedopracowane kodowanie zdominowałoby wszelkie korzyści uzyskane dzięki kompresji gramatycznej, szczególnie dla małych alfabetów. Zaprojektowanie i ocena wyrafinowanych kodowań (używających innych technik kompresji bezstratnej, niezwiązanych z gramatykami) istotnie zmieniłoby charakter projektu i dlatego jest świadomie poza zakresem.

7.3 Brak niskopoziomowych benchmarków dostępu losowego

Podobnie projekt nie raportuje rzeczywistych czasów dostępu losowego. Udostępnia funkcję `access(i)`, która naiwnie schodzi po drzewie wyprowadzeń do i-tego symbolu, a głębokość tego drzewa służy jako abstrakcyjne przybliżenie liczby potrzebnych kroków rekurencyjnych. Aby uzyskać realistyczne czasy, należałoby ustalić format serializacji gramatyki, zaimplementować niskopoziomową procedurę dostępu losowego zgodną z tym formatem, a następnie przeprowadzić benchmarki w kontrolowanych warunkach. To ponownie wprowadziłoby dużą ilość inżynierii, której celowo unikamy.



Rysunek 3: Górne drzewo to przykładowe drzewo wyprowadzeń dla napisu adwersarialnego skompresowanego przez RePair, przed balansowaniem. Ciężkie ścieżki są zaznaczone na czerwono, a boczne poddrzewa na niebiesko. Na dole to samo drzewo wyprowadzeń po zastosowaniu balansowania opartego na dekompozycji heavy-light. Można zaobserwować zmniejszoną głębokość i dodatkowe węzły wytworzone przez transformację balansowania (na zielono).

8 Wnioski i obserwacje

Na podstawie przykładowych eksperymentów zwizualizowaych na wykresach na Rysunkach 1 oraz 2 można zaobswerować następujące zachowanie poszczególnych kompresorów na użytych (ograniczonych) danych:

- Recompression w wariancie deterministycznym daje lepsze rezultaty niż w wariancie randomizowanym (pod względem rozmiaru oraz głębokości).
- Recompression w wariancie deterministycznym daje najpłytszą gramatykę. Jest ona jednak RLSLP. Uwzględniając jedynie SLP bez reguł run-length, najpłytszą gramatykę daje Sequitur, po jej późniejszym zbalansowaniu.
- Mimo że RePair daje najmniejszą gramatykę, jest ona również najgłębsza – różne metody mają różne „słabe i mocne strony".
- Pomimo „tradeoffu" pomiędzy głębokością a rozmiarem gramatyki otrzymywanej przez różne algorytmy (stopień kompresji kosztem zbalansowania), niektóre metody są lepsze od innych w obu metrykach. Przykładowo: RePair po balansowaniu daje gramatykę stricte lepszą (pod względem rozmiaru oraz głębokości) od Sequitura bez balansowania. Pokazuje to, że pomimo mocno teoretycznego wydźwięku, balansowanie nie jest metodą niepraktyczną.
- Zgodnie z oczekiwaniami, recompression w obu wariantach oraz RePair/Sequitur po balansowaniu dają gramatyki o wyraźnie mniejszej (asymptotycznie logarytmicznej) głębokości niż RePair oraz Sequitur bez balansowania. Wzrost rozmiaru gramatyki po balansowaniu jest około półtorakrotny (max. dwukrotny).

Literatura

[GJL21] Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. *J. ACM*, 68(4):27:1–27:40, 2021.

[Jeż15] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[LM99] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 296–305, 1999.

[NMW97] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.