Поиск подпоследовательностей в сжатых текстах

Юрий Лифшиц

Петербургское Отделение
Математического Института им. В.А.Стеклова
Российской Академии Наук
E-mail: yura@logic.pdmi.ras.ru

Аннотация

В теоретической информатике хорошо известны эффективные алгоритмы для обработки текстовых строк. В связи с увеличением размера текстов и хранением информации в сжатом виде потребовалось разработать алгоритмы для *сжатых* текстов. Данная работа посвящена поиску подпоследовательностей в сжатом тексте. Иначе говоря, по двум строкам (шаблону и тексту) нужно определить, можно ли вычеркнуть часть букв в тексте так, чтобы получился шаблон.

Мы рассматриваем две постановки. Для случая, когда шаблон задан явно, мы опишем алгоритм проверяющий вложимость шаблона в текст за время, линейное относительно размера сжатого текста. В ситуации, когда обе строки (и шаблон, и текст) представлены в сжатом виде, мы докажем что задача поиска подпоследовательности является NP-трудной и coNP-трудной.

1 Введение

1.1 Мотивация

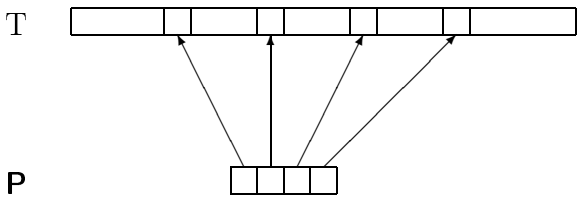
В связи со стремительным ростом объемов информации все большее внимание уделяется алгоритмам обработки сжатых объектов *без разархи-*

вирования. Изучается обработка сжатых изображений, деревьев, схем. В последнее десятилетие активные исследования связаны с операциями над сжатыми текстами. Классическими являются задачи поиска подстрок и подпоследовательностей в текстовых строках. Для поиска подстрок удалось найти полиномиальный алгоритм поиска сжатого шаблона в сжатом тексте [3]. Тем не менее, сложность задачи поиска подпоследовательности оставалась неизвестной.

Отметим, что алгоритмы обработки сжатых текстов не только необходимы для непосредственного применения на практике, но и используются для решения других вычислительных задач. Так, недавно найдены (см. [4]) приложения алгоритмов обработки сжатых текстов для анализа диаграмм последовательностей сообщений (message sequence charts). Прорывом стал алгоритм Пландовского для решения уравнений в словах [8], основанный на сжатом представлении строк.

1.2 Рассматриваемые задачи

В классической постановке задача поиска подпоследовательности (задача вложимости) заключается в следующем. На входе нам даны две строки — текст и шаблон. Нужно определить образуют ли буквы шаблона подпоследовательность в тексте. Мы же рассматриваем задачу, когда текст представлен в сжатом виде.



В работе отдельно рассмотрена постановка, где шаблон задан в явном виде (Раздел 3) и где он представлен в заархивированном виде (Раздел 4). Под сжатым текстом мы понимаем текстовую строку, порожденную *прямолинейной программой*. Эта модель архивации признана наиболее адекватной и удобной для изучения. Доказано, что многие реально используемые алгоритмы (LZ77,LZ78,LZW) легко могут быть сведены к прямолинейным программам. Мы подробнее обсудим модели архивирования в Разделе 2.

1.3 Полученные результаты

Для случая явно заданного шаблона мы построим алгоритм, который определяет вложимость шаблона в текст. Наш алгоритм также выдает количество минимальных подстрок и количество подстрок определенной длины, в которые вкладывается шаблон. Трудоемкость алгоритма равна $mk^2 \log k$, где k — длина шаблона, а m — размер прямолинейной программы, порождающей текст. Таким образом, наш алгоритм линеен относительно размера сжатого текста. Этот факт позволяет сделать важный вывод: использование разработанного алгоритма оправдано для всех текстов, у которых отношение оригинального длины к сжатому размеру больше $k^2 \log k$.

Несмотря на успехи в области поиска подстрок в сжатых текстах, сложность определения вхождения *сжатого* шаблона в сжатый текст как подпоследовательности оставалась совершенно неясной. До недавнего времени не было ни одной разумной оценки сложности этой задачи, она могла оказаться как решаемой за полиномиальное время, так и PSPACE-полной. В данной работе удалось получить нижние оценки, показав, что задача не только NP-трудна, но и (при предположении NP≠coNP) лежит вне класса NP.

Последний результат стал большой неожиданностью, так как задача по своей природе очень напоминает представителей класса NP.

1.4 Предыдущие исследования

Перечислим основные задачи, которые изучаются для сжатых текстов:

Эквивалентность представлений: дано два сжатых текста, требуется определить, совпадают они или нет.

Поиск подстроки в сжатом тексте: дан шаблон и сжатый текст, требуется определить, является ли шаблон подстрокой текста, и при положительном ответе найти первое вхождение.

Поиск сжатой подстроки в сжатом тексте: дан сжатый шаблон и сжатый текст, требуется определить, является ли шаблон подстрокой текста, и при положительном ответе найти первое вхождение.

Принадлежность языку: зафиксирован некоторый язык. Требуется по данной сжатой строке определить, принадлежит она языку или нет.

Вычислительная сложность этих задач рассматривалась в целом ряде

работ [3, 9, 11, 5]. Первые три задачи имеют полиномиальную сложность. Особо выделим работу [3], развивающую идеи [7], где был построен полиномиальный алгоритм для поиска сжатой подстроки в сжатом тексте. Вопрос о принадлежности регулярному языку также решается за полиномиальное время, а задача о принадлежности контекстно-свободному языку уже оказывается PSPACE-полной [5].

Недавно удалось доказать [6], что принадлежность регулярному языку является P-полной (то есть плохо поддается распараллеливанию).

Еще одним важным результатом является эффективное приближенное построение в [12] по данному тексту минимальной прямолинейной программы (ПП), порождающей этот текст. Более формально, построен алгоритм, работающий за время $O(n \cdot \log|\Sigma|)$, который по тексту длины n строит $O(\log n)$-приближение минимальной ПП, порождающей этот текст. Это означает, что поиск адекватного (близкого к максимально возможному) сжатия в модели прямолинейных программ может быть выполнен эффективно.

Кроме того, следует упомянуть работу [1], в которой задачи поиска подстрок обобщены на двумерные тексты. Как оказалось, при этом резко возрастает вычислительная сложность. Поиск явно заданного шаблона в сжатом двумерном тексте является NP-полным, а поиск сжатого шаблона в сжатом двумерном тексте является Σ_2^P-полным.

Как известно, строки являются элементами свободного конечно-порожденного моноида. В работе [5] изучается обработка сжатых элементов различных классов конечно-порожденных моноидов. В зависимости от ограничений на моноид были получены доказательства полноты задачи эквивалентности сжатых представлений для классов P, coNP, PSPACEи EXPSPACE.

В качестве обзоров по обработке сжатых текстов можно порекомендовать [10, 13].

1.5 Организация статьи

В Разделе 2 мы определим понятие прямолинейной программы и обсудим его отношения с другими сжатыми представлениями текстов. Раздел 3 посвящен алгоритму поиска подпоследовательности в сжатом тексте и родственным задачам. В четвертом разделе мы представим нижнюю оценку сложности поиска сжатой подпоследовательности в сжатом тексте. Этот результат получен в два этапа: доказательство NP-трудности

и доказательство coNP-трудности. В Разделе 5 мы подведем итоги и сформулируем дальнейшие направления для исследований.

2 Модели сжатия

Мы рассматриваем конечный алфавит Σ, словом называется любая последовательность его букв, множество всех слов традиционно обозначается как Σ^*.

Центральным понятием нашей работы является сжатие строки (слова). В 1977 году Лемпель и Зив предложили свою модель архивирования [14], которая получила широкое распространение. В девяностые годы для построения алгоритмов стали использовать модель, основанную на грамматиках - *прямолинейные программы* (Straight-Line Programs). Этой последней моделью мы и будем пользоваться. Неформально, прямолинейная программа – это контекстно-свободная грамматика, порождающая только одно слово.

Определение 1. Прямолинейной программой называется контекстно-свободная грамматика $\mathcal{P}$, в которой нетерминальные символы $X_1, \ldots, X_m$ упорядочены (X_m – стартовый символ), и где у каждого нетерминального символа есть только одно правило: $X_i \to a$, где a — терминал, или $X_i \to X_j X_k$ для некоторых $j, k < i$.

Для прямолинейной программы $\mathcal{P}$ мы будем обозначать $eval(\mathcal{P})$ единственное слово, которое описывается $\mathcal{P}$. Таким образом $eval(\mathcal{P}) = eval(X_m)$.

Важно отметить, что прямолинейные программы являются моделью *декомпрессора*, то есть моделируют лишь получение исходного текста из сжатого представления. В данной работе мы совершенно не заботимся о том, как были получены сжатые тексты.

Поясним происхождение термина “прямолинейная программа”. Дело в том, что любой текст, представленный с помощью вышеописанных контекстно-свободных грамматик, может быть порожден с помощью программ, использующих в точности один оператор — оператор присваивания.

Трудность обработки сжатых текстов связана с тем, что соотношение между размером прямолинейной программы и длиной порождаемого текста может быть экспоненциальным. Таким образом, мы не можем себе позволить восстанавливать оригинальный текст. Экспоненциальная

разница может иметь место и для обобщенной версии LZ77-сжатия. Напротив, алгоритм LZ78 допускает лишь квадратичное соотношение между размерами оригинального и сжатого текстов.

3 Алгоритм поиска подпоследовательности

3.1 Решаемые задачи

В этом разделе мы опишем алгоритм для проверки вложимости шаблона P в текст T, представленный прямолинейной программой.

На практике часто требуется найти "достаточно компактную" подпоследовательность. Минимальным окном в тексте T назовем такую подстроку S текста T, которая 1) содержит P, как подпоследовательность, и 2) никакая меньшая подстрока S не обладает свойством 1. Мы будем также называть w-окном подстроку текста T длины w, которая содержит P как подпоследовательность.

Вслед за работой [CGM05] мы рассмотрим следующие пять задач:

1. Определить, является ли шаблон P подпоследовательностью в тексте T?
2. Для данного шаблона P найти количество минимальных окон в тексте T.
3. Определить, вкладывается ли шаблон P в какое-нибудь w-окно текста T?
4. Для данного шаблона P найти количество w-окон в тексте T.
5. Для данного шаблона P найти количество минимальных окон в T, размером не больше w.

В работе [CGM05] был дано описания полиномиального алгоритма для решения этих пяти задач для работы с LZ78-сжатым текстом. Эта модель сжатия допускает лишь квадратичное соотношение между сжатым и исходным текстом. В данной работе мы обобщим этот результат на более сильную модель прямолинейных программ и явно покажем линейность алгоритма относительно размера сжатого текста.

Для решения пяти вышеописанных задач мы определим вспомогательные структуры данных. Далее мы покажем как вычислить элементы этих структур и обсудим итоговый алгоритм.

3.2 Вспомогательные структуры данных

С этого момента мы рассматриваем текст T сжатый ПП $\mathcal{P}$ размера m. Пусть $|P| = k$ и $P_1, \ldots, P_l$ — все различные подстроки шаблона P. Заметим, что $l < k^2$.

Теперь мы определим две основные и три задачно-ориентированные структуры данных.

Левые вложения. Для каждого нетерминала X_i прямолинейной программы $\mathcal{P}$ и подстроки шаблона P_j мы определяем $L_{i,j}$ как длину минимального **префикса** строки $eval(X_i)$, содержащего P_j. Если такого префикса нет, мы устанавливаем $L_{i,j} = \infty$.

Правые вложения. Для каждого нетерминала X_i прямолинейной программы $\mathcal{P}$ и подстроки шаблона P_j мы определяем $R_{i,j}$ как длину минимального **суффикса** строки $eval(X_i)$, содержащего P_j. Если такого префикса нет, мы устанавливаем $R_{i,j} = \infty$.

Нам потребуются также следующие задачно-ориентированные структуры данных:

Минимальные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{P}$ мы определим MW_i как число минимальных окон в строке $eval(X_i)$, содержащих P.

Фиксированные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{P}$ мы определим FW_i как число w-окон в строке $eval(X_i)$, содержащих P.

Ограниченные минимальные окна. Для каждого нетерминала X_i прямолинейной программы $\mathcal{P}$ мы определим BMW_i как число окон, размером не больше w, в строке $eval(X_i)$, которые содержат P.

3.3 Вычисление вспомогательных структур данных

Покажем, как вычислить значения элементов пяти вышеописанных массивов.

Левые вложения. Будем проводить вычисления вдоль конструкции прямолинейной программы. Мы легко можем вычислить значение L для однобуквенных символов и всех возможных P_i. Пусть теперь $X_i \to X_p X_q$.

Если $L_{p,j} \neq \infty$, тогда просто $L_{i,j} = L_{p,j}$. В противном случае, мы должны найти такое разбиение P_j на две части $P_u P_v$, что и $L_{p,u}$, и $L_{q,v}$ — конечны. Нам нужно найти разбиение, максимизирующее первую часть (то есть $|P_u|$). Мы будем делать это бинарным поиском. Если мы нашли такое разбиение, то $L_{i,j} = |eval(X_p)| + L_{q,v}$, иначе $L_{i,j} = \infty$. Таким образом, трудоемкость одного шага — $O(\log k)$, общая трудоемкость — $O(ml \log k) \leq O(mk^2 \log k)$.

Правые вложения. Вычисляем аналогично левым вложениям.

Минимальные окна. Также будем использовать вычисления вдоль конструкции прямолинейной программы, а также данные о левых и правых вложениях. Для подсчета минимальных окон в $X_i \to X_p X_q$ мы должны сложить уже вычисленные значения для X_p и X_q и прибавить некоторое количество *пограничных* окон. Заметим, что для каждого разбиения $P = P_u P_v$ может быть не более одного минимального пограничного окна, в котором P_u вкладывается внутри X_p, P_v внутри X_q. Используя данные о правых и левые вложениях, мы определим, для каких разбиений такие пограничные минимальные окна есть. В этом месте нужно быть аккуратным. Одному минимальному окну может соответствовать несколько последовательных разбиений. Поэтому будем действовать следующим образом. Мы будем рассматривать все разбиения шаблона от "все в X_p" до "все в X_q" и увеличивать счетчик пограничных окон, только когда 1) первая часть разбиения вкладывается в X_p, а вторая в X_q; и 2) получившееся минимальное окно сдвинуто относительно предыдущего успешного вложения. Таким образом, трудоемкость вычисления этого массива равна $O(mk)$.

Фиксированные окна. Схема вычисления такая же, как и для минимальных окон. Здесь мы лишь покажем, как сосчитать пограничные фиксированные окна для $X_i \to X_p X_q$. Основное наблюдение: любое пограничное w-окно, содержащее P, также содержит **минимальное** окно, содержащее P. Также, как и в предыдущем абзаце мы рассматриваем по очереди все разбиения шаблона P. Для каждого разбиения, пользуясь данными о правых и левых вложениях, мы находим минимальное окно, соответствующее этому разбиению. Сравнивая его с предыдущим, мы прибавляем к счетчику количество w-окон, которые содержат новое минимальное окно, но не старое. Трудоемкость вычисления этого массива также равна $O(mk)$

Ограниченные минимальные окна. Используем ту же технику, что и для минимальных окон. Просто при подсчете мы игнорируем слиш-

ком большие пограничные минимальные окна.

3.4 Итоговый алгоритм и его трудоемкость

Наши структуры содержат ответы для всех пяти задач:

1. Шаблон P является подпоследовательностью в тексте T тогда и только тогда, когда $L_{1,m} \neq \infty$ (мы считаем $P_1 = P$),
2. Количество минимальных окон в T, содержащих P, равно MW_m,
3. Шаблон P является подпоследовательностью некоторого w-окна тогда и только тогда, когда $FW_m \neq 0$,
4. Количество w-окон, содержащих P, равно FW_m,
5. Количество окно, размера не больше w, содержащих P, равно BMW_m.

Таким образом, итоговая трудоемкость нашего алгоритма для шаблона длины k и текста, представленного прямолинейной программой размера m, равна $O(mk^2 \log k)$.

Замечание. Легко показать (см. [13]), что для текста, сжатого алгоритмом LZ78, можно построить прямолинейную программу лишь линейно увеличив размер архива. Сама процедура конвертации также линейна. Это значит, что наши задачи могут быть решены за время $O(mk^2 \log k)$, уже где m — размер LZ78-сжатого текста.

Замечание. Для LZ77-сжатия также можно осуществить переход к прямолинейной программе. Трудоемкость и размер получившейся программы оценивается как $O(m \log n)$, где m — размер LZ77-сжатого текста, а n — оригинального. Таким образом, применив сначала переход к прямолинейной программе, а затем наш основной алгоритм, мы получим трудоемкость $O(mk^2 \log k \log n)$ для LZ77-сжатия.

4 Нижние оценки трудности

В этом разделе мы дадим две нижние оценки на вычислительную сложность следующей задачи:

- Поиск сжатой подпоследовательности в сжатом тексте (для краткости — Вложимость). Даны прямолинейные программы, представляющие строку P (шаблон) и строку T (текст). Требуется определить, являются ли буквы шаблона подпоследовательностью в строке T (обозначение: $P \hookrightarrow T$).

В этом разделе представлено два основных результата. Мы начнем со сведения известной NP-полной задачи (Сумма размеров) к задаче Вложимости. Таким образом мы получим NP-трудность последней. Следующим шагом будет сведение Вложимости к Невложимости и наоборот. Немедленным следствием из этой симметричности станет coNP-трудность изучаемой задачи.

4.1 NP-трудность

Напомним формулировку известной NP-полной задачи Сумма размеров (см. [2]):

> Дана двоичная запись натуральных чисел $w_1, \ldots, w_n, t$. Требуется определить, существуют ли $x_1, \ldots, x_n \in \{0, 1\}$ такие, что $\sum_{i=1}^{n} x_i \cdot w_i = t$.

Теорема 1. Сумма размеров *сводится к* Вложимости.

Доказательство. Итак пусть $t, \bar{w} = \langle w_1, \ldots, w_n \rangle$ – входные данные Суммы размеров (будем считать $n > 1$). Мы построим такие прямолинейные программы $\mathcal{F}$ и $\mathcal{G}$, что подмножество $\bar{w}$ с суммой t существует тогда и только тогда, когда $eval(\mathcal{F}) \hookrightarrow eval(\mathcal{G})$.

Введем обозначения $s = w_1 + \cdots + w_n$, $N = 2^n s$. Каждому подмножеству $\bar{w}$ можно поставить в соответствие строчку $x = \overline{x_1 \ldots x_n}$ длины n из нулей и единиц, то есть целое число от 0 до $2^n - 1$. Введем обозначение $x \circ \bar{w} = \sum_{i=1}^{n} x_i w_i$, фактически, $x \circ \bar{w}$ дает сумму подмножества $\bar{w}$, кодируемого числом x.

Конструкция. Наша задача "закодировать" каждый набор входных данных для Суммы размеров через входные данные для Вложимости. Опишем текстовые строки F и G:

$$G = G_0^{5N} \qquad G_0 = G_1 G_2 G_3 G_4$$

$$G_1 = \prod_{x=0}^{2^n-1} (10^s) = (10^s)^{2^n} \qquad G_2 = 0^{2N}$$

$$G_3 = \prod_{x=0}^{2^n-1} (0^{x\circ\bar{w}}10^{s-x\circ\bar{w}}) \qquad G_4 = 0^{t+1}$$

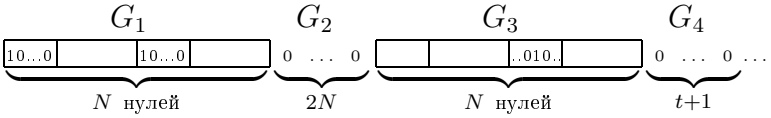
$$F = F_0^{5N-1} \qquad F_0 = 10^{3N+t}10^{N+1}$$

Мы использовали знак $\prod$ для обозначения конкатенации соответствующих слов в порядке изменения индекса от нижнего предела к верхнему.

Этими равенствами мы определили тексты F и G. Ниже мы докажем, что они могут быть порождены прямолинейными программами $\mathcal{F}$ и $\mathcal{G}$. Но сначала убедимся, что вложимость текста F в G равносильна существованию подмножества $\bar{w}$ с суммой t.

Равносильность существования подмножества с заданной суммой и вложимости.

"Существует подмножество $\Rightarrow$ вложимость". Пусть x — код подмножества $\bar{w}$ с суммой t, то есть $x \circ \bar{w} = t$. Рассмотрим начало G, т.е. $G_1G_2G_3G_4G_1\ldots$, и выделим следующее вложение F_0: отметим 1 в x-ом блоке G_1, затем $3N + t$ нулей, далее (в точности потому что $x \circ \bar{w} = t$) стоит 1 и отметив следующие $N + 1$ нулей мы окажемся перед x-ой единицей во втором G_1, таким образом имея $5N$ блоков $G_0 = G_1G_2G_3G_4$ мы сможем вложить $5N - 1$ копий F_0. Что, собственно, и требовалось проверить: $F \hookrightarrow G$.

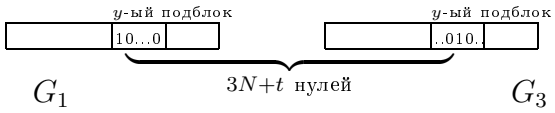


"Вложимость $\Rightarrow$ существует подмножество". Пусть $F \hookrightarrow G$. Предположим также, что ответ в Сумме размеров отрицательный. Выведем из этих двух утверждений противоречие.

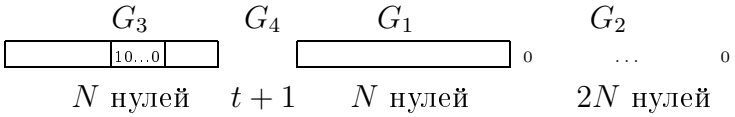
Мы будем искать нули в G, которые не задействованы во вложении F в G. Это вложение представляет собой $5N - 1$ непересекающихся вложений F_0 в G. Слово F_0 содержит две единицы, между которыми расположено ровно $3N + t$ нулей.

Убедимся, что в G нет двух единиц, между которыми расположено в точности столько же нулей, от противного. Рассмотрим два случая: первая единица находится в блоке G_1 (скажем, в подблоке номер y). Тогда, сдвинувшись на $3N+t$ нулей, мы попадем в y-ый подблок G_3. Если

бы в этом подблоке после t нулей стояла единица, то мы бы получили $y \circ \bar{w} = t$ — противоречие с несуществованием подмножества с суммой t.



Второй случай: левая единица лежит в блоке G_3. Тогда, сдвинувшись на $3N + t$ нулей мы попадем внутрь блока G_2, где вообще нет единиц.



Следовательно, при каждом вложении F_0 хотя бы один ноль (между двух единиц) остается незадействованным. Осталось лишь оценить с двух сторон количество нулей в G. По построению, их там $5N \cdot (4N+t+1)$. С другой стороны, каждое слово F_0 содержит $4N+t+1$ нулей и еще хотя бы $5N-1$ ноль не задействован. Таким образом общее количество нулей в G должно быть не меньше, чем $(5N-1) \cdot (4N+t+1) + 5N - 1 = 5N \cdot (4N+t+1) + (N-t-2) > 5N \cdot (4N+t+1)$. Поясним последнее неравенство: $N = s2^n \geq 4s > t+2$. То есть мы получили больше нулей в G, чем там есть по построению, что и дает нам противоречие.

Реализация F и G в виде прямолинейных программ. Заметим, что, за одним исключением, F и G строятся из 0 и 1 только с помощью полиномиального числа конкатенаций и возведения в степень (двоичная запись всех использованных степеней – полиномиального размера). Такие конструкции напрямую реализуются сжатыми словами полиномиального размера. Единственным нетривиальным блоком является G_3. Конструкция сжатой версии G_3 полиномиального размера впервые была предложена Маркусом Лори в работе [5]. Для полноты работы мы приводим эту конструкцию в Приложении А.

Полиномиальность сведения. Для завершения доказательства необходимо убедиться, что построение прямолинейных программ, вычисляющих F_1 и G_1 полиномиально относительно размера входных данных задачи Сумма размеров. Чтобы убедиться в этом, достаточно заметить, что в построении F_1 и G_1 мы использовали лишь конкатенации и

возведение в степени, являющиеся результатами арифметических действий над t и $w_1, \ldots, w_n$. $\square$

Следствие 1. Вложимость ***NP**-трудна.*

4.2 coNP-трудность

Теорема 2. Вложимость *сводится (по Карпу) к* Невложимости. Невложимость *сводится к* Вложимости.

Доказательство. Мы будем доказывать следующее утверждение: существует полиномиальный алгоритм, позволяющий для любых сжатых слов F и G построить сжатые слова F_1 и G_1 (естественно, что их размер может быть лишь полиномиально больше, чем у F и G), такие что

$$F \hookrightarrow G \Leftrightarrow F_1 \not\hookrightarrow G_1. \qquad (*)$$

В случае унарного алфавита обе задачи лежат в **P**. Будем считать, что в алфавите хотя бы 2 буквы. Заметим, что за полиномиальное время можно вычислить последнюю букву F и дописать в конец к G другую букву (получив G') и при этом $F \hookrightarrow G \Leftrightarrow F \hookrightarrow G'$. Так что, не умаляя общности, можно считать, что F и G заканчиваются на разные буквы.

Пусть $F = f_1 \ldots f_k$, $G = g_1 \ldots g_m$. Для каждой буквы алфавита a введем обозначение $X_a = (\Sigma/a)^{m+1}$, где (Σ/a) – конкатенация всех букв алфавита, кроме a, в любом порядке.

Конструкция:

$$F_1 = G = g_1 \ldots g_m$$

$$G_1 = X_{f_1} f_1 \ldots X_{f_k} f_k$$

Проверка свойства $(*)$: Мы можем представить G в следующем виде: $R_1 f_1 \ldots R_l f_l R_{l+1}$, где R_i не содержит буквы f_i. Утверждение $F \hookrightarrow G$ равносильно выполнению равенства $l = k$ для нашего представления.

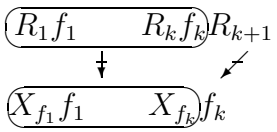
Если $l < k$, то $F_1 \hookrightarrow G_1$, так как для каждого $1 \le i \le l+1$ выполнено $R_i \hookrightarrow X_{f_i}$ и $f_i = f_i$.

$$\begin{array}{ll} R_1 f_1 R_2 X_2 & R_l f_l R_{l+1} \\ \downarrow\ \downarrow & \downarrow\ \downarrow\ \downarrow \\ X_{f_1} f_1 & X_{f_l} f_l X_{f_{l+1}} \ldots \end{array}$$

Если $l = k$, то $R_{l+1} \neq \varnothing$, так как g_m по предположению не равно f_k. По индукции можно убедиться в том, что

$$R_1 f_1 \dots R_i f_i \not\leftarrow X_{f_1} f_1 \dots X_{f_i}.$$

Действительно, при $i = 1$ это следует из факта $f_1 \not\leftarrow X_{f_1}$. Переход $i \to i+1$: если бы вложение существовало бы для $i+1$, то, так как $f_{k+1} \not\leftarrow X_{f_{k+1}}$, то f_{k+1} попадало бы не правее f_k, а значит было бы вложение и для i, что противоречит предположению индукции. Рассмотрим наше утверждение при $i = k$ и в сумме с фактом $R_{k+1} \not\leftarrow f_k$ мы получим $F_1 \not\leftarrow G_1$.



Полиномиальность конструкций: Заметим, что X_a строится за полиномиальное от $\log m$ время. Для построения G_1 остается лишь добавить правила вида $A \to X_a a$ для всего алфавита и использовать в конструкции эти нетерминальные символы вместо соответствующих терминалов. □

Следствие 2. Вложимость ***coNP**-трудна.*

Доказательство. Согласно Теореме 1 Вложимость **NP**-трудна. Следовательно, так как мы используем сведение по Карпу, Невложимость - **coNP**-трудна. Так как Невложимость сводится к Вложимости (Теорема 2), то Вложимость также является **coNP**-трудной. □

5 Выводы и открытые вопросы

В работе рассмотрена задача поиска подпоследовательности в сжатых текстах. Мы рассмотрели две постановки: поиск шаблона, представленного как обычный текст и представленного в сжатом виде.

В первом случае, мы построили алгоритм для поиска подпоследовательности и серии родственных задач. Для двух моделей сжатия (прямолинейные программы, LZ78) наш алгоритм *линеен* относительно размеров сжатого текста. Это значит, что как только сжатый текст будет короче оригинального, сразу представленный алгоритм будет работать

быстрее подхода "распакуй и ищи". Для LZ77 алгоритм всего лишь на логарифмический сомножитель медленнее линейного алгоритма.

Поскольку для рассмотренных задач построен алгоритм, линейный относительно размера текста, то с теоретической точке зрения задача не может быть решена лучше и остается лишь провести практические испытания алгоритма.

Для поиска сжатой подпоследовательности удалось получить нижние оценки на вычислительную сложность. Мы построили доказательство (при предположении NP≠coNP), что эта задача лежит за пределами класса NP. С другой стороны, известна только тривиальная верхняя оценка сложности — легко построить алгоритм из класса PSPACE. Главным открытым вопросом остается сближение этих оценок.

Список литературы

[1] Piotr Berman, Marek Karpinski, Lawrence L. Larmore, Wojciech Plandowski and Wojciech Rytter. *On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts*, Journal of Computer and Systems Science, vol. 65, number 2, pp. 332–350, 2002.

[CGM05] P.Cegielski, I.Guessarian, Yu.Matiyasevich. *Window subsequence problem for compressed texts.* Draft, June 2005.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*, Freeman, 1979.

Русский перевод: М. Гэри и Д. Джонсон *Вычислительные машины и труднорешаемые задачи*, Москва, Мир, 1982.

[3] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski and Wojciech Rytter. *Efficient Algorithms for Lempel-Ziv Encoding (Extended Abstract)*, Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996), Springer-Verlag, LNCS 1097, pp. 392–403, 1996.

[4] Blaise Genest and Anca Muscholl. *Pattern Matching and Membership for Hierarchical Message Sequence Charts*, In Proceedings

of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2002), Springer-Verlag, LNCS 2286, pp. 326–340, 2002.

[5] Markus Lorhey. *Word problems on compressed word*, ICALP 2004, Springer-Verlag, LNCS, 3142, pp. 906–918, 2004.

[6] N. Markey and Ph. Schnoebelen. *A PTIME-complete matching problem for SLP-compressed words*, Information Processing Letters, vol. 90, number 1, pp. 3–6, 2004.

[7] Wojciech Plandowski. *Testing Equivalence of Morphisms on Context-Free Languages*, Second Annual European Symposium on Algorithms (ESA'94), Utrecht (The Netherlands), Springer-Verlag, LNCS 855, pp. 460–470, 1994.

[8] Wojciech Plandowski. *Satisfiability of word equations with constants is in* **PSPACE**, J. ACM, vol. 51, number 3, pp. 483-496, 2004.

[9] Wojciech Plandowski and Wojciech Rytter. *Complexity of Language Recognition Problems for Compressed Words*, Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa, Springer-Verlag, pp. 262–272, 1999.

[10] Wojciech Rytter. *Algorithms on Compressed Strings and Arrays*, Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'99), Springer-Verlag, LNCS 1725, pp. 48–65, 1999.

[11] Wojciech Rytter. *Compressed and fully compressed pattern matching in one and two dimensions*, Proceedings of the IEEE, vol. 88, number 11, pp. 1769–1778, 2000.

[12] Wojciech Rytter. *Application of Lempel-Ziv factorization to the approximation of grammar-based compression*, Theoretical Computer Science, vol. 302, number 1–3, pp. 211–222, 2003.

[13] Wojciech Rytter. *Grammar Compression, LZ-Encodings, and String Algorithms with Implicit Input*, Proceedings of the 31st International Colloquium on Automata, Languages and Programming(ICALP 2004), Springer-Verlag, LNCS 3142, pp. 15–27, 2004.

[14] J. Ziv and A. Lempel. *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory, vol. 23, number 3, pp. 337-343, 1977.

Приложение А

Мы хотим построить прямолинейную программу, вычисляющую

$$G_3 = \prod_{\bar{x}=0}^{2^n-1} (0^{x \circ \bar{w}} 10^{s-x \circ \bar{w}})$$

Вот эта конструкция:

$$S_1 \to 10^{s+w_1}1$$

$$S_{k+1} \to S_k 0^{s-s_k+w_{k+1}} S_k$$

Мы возьмем S_n как стартовый символ и докажем, что он вычисляет в точности G_3, с помощью индукции.

Утверждение: $eval(S_k) = \left(\prod_{\bar{x} \in \{0,1\}^k \setminus \{\bar{1}_k\}} (0^{\bar{x} \cdot \bar{w}_k} 10^{s-\bar{x} \cdot \bar{w}_k}) \right) 0^{s_k}1.$

Для $k = 1$ утверждение сводится к равенству $10^{s+w_1}1 = 0^0 10^{s-0} 0^{s_1} 1$. Для $k+1 \le n$ мы получим следующую цепочку равенств:

$$eval(S_{k+1}) = \left(\prod_{\bar{x} \in \{0,1\}^{k+1} \setminus \{\bar{1}_{k+1}\}} (0^{\bar{x} \cdot \bar{w}_{k+1}} 10^{s-\bar{x} \cdot \bar{w}_{k+1}}) \right) 0^{s_{k+1}}1 =$$

$$\underbrace{\left(\prod_{\bar{x} \in \{0,1\}^k} (0^{\bar{x} \cdot \bar{w}_k} 10^{s-\bar{x} \cdot \bar{w}_k}) \right)}_{eval(S_k) 0^{s-s_k}} \underbrace{\left(\prod_{\bar{x} \in \{0,1\}^k \setminus \{\bar{1}_k\}} (0^{\bar{x} \cdot \bar{w}_k + w_{k+1}} 10^{s-\bar{x} \cdot \bar{w}_k - w_{k+1}}) \right) 0^{w_{k+1}} 0^{s_k} 1}_{0^{w_{k+1}} eval(S_k)} =$$

$$eval(S_k) 0^{s-s_k+w_{k+1}} eval(S_k) = eval(S_{k+1}),$$

что доказывает утверждение.

Осталось заметить, что при $k = n$

$$eval(S_n) = \prod_{\bar{x} \in \{0,1\}^n} \left(0^{\bar{x} \cdot \bar{w}} 10^{s-\bar{x} \cdot \bar{w}}\right) = G_3.$$