1. Введение

В этом отчёте производится сравнение двух алгоритмов поиска частых наборов бинарных данных - AD и DepthProject. На вход подается булева матрица $L = (a_{ij})$ размера $m \times n$, в которой элемент a_{ij} равен 1, если в транзакции с номером i значение порядка равно 1. Подается минимальная поддержка s - целое число в диапазоне от 1 до m.

2. Алгоритм AD

2.1. Описание

Алгоритм AD строит частые наборы пошагово. Введём обозначения: R(L) - множество всех столбцов матрицы L, $S_1(L)$ - множество столбцов из R(L) с не менее чем s единицами, $e_1(R)$ и $e_2(R)$ - столбцы соответственно с наименьшим и наибольшим номерами в $R \subseteq R(L)$. Для набора $Q_t = j_1, ..., j_t$ обозначим R_t - множество столбцов, номера которых больше j_t, G_t - множество столбцов h из R_t, которые в объединении с Q_t дают совместимый набор.

Сначала строится первый частый набор $Q = e_1(S_1(L))$. Затем для каждого $Q = \{j_1, ..., j_r\} \neq \{e_2(S_1(L))\}$ строится следующий за ним набор ΔQ по следующим правилам:

1) $G_r \neq \varnothing$: $\Delta Q = Q \cup \{e_1(G_r)\}$;

2) $G_r = \varnothing$:

a) $r = 1$: $\Delta Q = \{e_1(S_r)\}$;

b) $r > 1$ и $G_{r-1} \neq \varnothing$: $\Delta Q = Q_{r-1} \cup e_1(G_{r-1})$;

c) $r > 1$ и $G_{r-1} = \varnothing$: если $r = 2$, то $\Delta Q = \{e_1(R_r)\}$, иначе $\Delta Q = Q_{r-2} \cup \{e_1(R_{r-1})\}$;

Таким образом, алгоритм обходит дерево решений, вершины которого - частые наборы. Максимальные наборы находятся среди висячих вершин. В терминах алгоритма это значит, что если $G_t = \varnothing$, то Q_t является либо максимальным частым набором, либо подмножеством ранее найденного максимального частого набора. Проверку на максимальность можно проводить как по ранее найденным наборам (тогда сложность алгоритма зависит от выхода задачи), так и по базе данных (модификация, предложенная Н. А. Драгуновым). Второй способ значительно быстрее.

Частый наборы выводятся в алфавитном порядке.

2.2. Реализация

Сначала строятся первый частый набор $Q[1] = \{e_1(R(L))\}$ и последний частый набор $L = \{e_2(R(L)\}$. Затем последовательно строятся наборы $Q[2], Q[3], ...$ из одной ветви дерева решений вплоть до первого максимального набора. После его нахождения начинает работать основной цикл, выход из которого осуществляется если очередной найденный набор равен L.

Используется процедура Обновить(T, Q), которая:

1) Зануляет строки матрицы T, дающие в пересечении со столбцами из Q нули.

2) Удаляет столбцы с номерами из Q.

Таким образом, первый совместимый столбец модифицированной матрицы есть первое частое расширение набора Q.

Ниже представлен псевдокод алгоритма AD:

Алгоритм 1 DepthProject

Процедура AD(БазаДанных : T, Поддержка : s)
  $Q = \varnothing$
  $L = $ ПоследнийЧастыйСтолбец(T, s)
  $t = $ ПервыйЧастыйСтолбец(T, s)
  До тех пока $t \neq \varnothing$ **выполнять**
    $Q = Q \cup \{e_1(G_r)\}$ ▷ $r = |Q|$
    Q – частый набор
    $t = \{e_1(G_r)\}$
  Конец цикла
  $T' = T$
  До тех пока $Q \neq L$ **выполнять**
    $T' = $ Обновить(T', Q)
    $t = $ ПервыйЧастыйСтолбец(T', s)
    Если $t \neq \varnothing$ **тогда**
      $Q = Q \cup \{t\}$
    иначе
      Q – частый набор
      Проверка Q на максимальность
      Если $|Q| = 1$ **тогда**
        $Q = $ ПервыйЧастыйСтолбец(T', Q_1, s)
      иначе
        Если $|Q| = 2$ **тогда**
          $Q = $ ПервыйЧастыйСтолбец(T', Q_1, s)
        иначе
          $p = $ ПредпоследнийЭлемент(Q)
          $T' = $ Обновить(T, Q_{r-2})
          $t = $ ПервыйСовместимыйСтолбец(T', p, s)
          $Q = Q \cup \{t\}$
        Конец условия
      Конец условия
    Конец условия
  Конец цикла
Конец процедуры

3. Алгоритм DepthProject

3.1. Пару слов о дереве решений

В DepthProject производится рекурсивный обход дерева решения. Введём некоторые обозначения. Пусть имеется дерево решений $Д_L$:

1) Каждый узел соответствует частому набору. Корень дерева соответствует пустому набору элементов и обозначается $Null$.

2) Узел $P = \{i_1, ..., i_k\}$ имеет предка $Q = \{i_1, ..., i_{k-1}\}$.

Обозначим за $E(P)$ множество частых расширений узла P. Обозначим за $F(P)$ множество частых расширений родительского узла Q, номера которых превышают последний элемент в P. Справедливы вложения: $E(P) \subseteq F(P) \subset E(Q)$.

Алгоритм обходит это дерево решений и выводит наборы в алфавитном порядке. Древовидная структура позволяет на каждом шаге сужать область поиска частых расширений, рассматривая так называемых кандидатов. Преимущество такого подхода в том, что кандидаты ищутся за константное время по предыдущему узлу.

3.2. Описание

Вызывается процедура DepthFirst, на вход которой подаются узел N, база данных T и маска B. При первом запуске N = Null, T - вся база данных (т.е. вся бинарная матрица), маска $B = (1, 1, ..., 1)$ - двоичный вектор длины m. В каноническом изложении процедура выглядит следующим образом:

Алгоритм 2 DepthProject

Процедура DepthFirst(Узел : N, БазаДанных : T, Маска : B)
  $C = F(P)$ ▷ Множество кандидатов на частое расширение
  $E = E(P)$ ▷ Множество частых расширений $E = \{i_1, ..., i_{|E|}\}$
  Положить $N \cup i_r$ в узлы ДР
  $B' =$ ПостроитьМаску(N, B, T)
  Если Условие проекции **тогда**
    $T' =$ Проекция(T, E, N, B')
  иначе
    $T' = T$
  Конец условия
  Цикл <r := 1 до |E|> **выполнять**
    $DepthFirst(N \cup \{i_r\}, T', B')$
  Конец цикла
Конец процедуры

$F(P)$ - это номера столбцов из $E(Q)$, которые имеют номер, больший последнего номера в узле N. Здесь Q - ближайший предок вершины P. Маска B имеет 1 в $i-$й позиции тогда и только тогда, когда все столбцы текущего узла имеют единицу в $i-$й позиции. Проекция позволяет снизить время работы алгоритма. Вообще говоря,

это аналог функции "Обновить"в реализации алгоритма AD с той лишь разницей, что строки не зануляются, а удаляются. Уменьшая на каждом шаге базу данных T мы сужаем область поиска частых расширений $E(P)$. Также важно построить хороший метод поиска частых расширений.

3.3. Реализация

В текущей реализации DepthProject используется древовидная структура. Первым шагом находится множество кандидатов F(P) - эта операция производится быстро и сокращает дальнейший поиск частых расширений E(P). Маска готовится перед вызовом этих функций. Здесь маска - это логическое умножение столбцов текущего узла. Для её нахождения не обязательно перемножать все столбцы, достаточно на каждом шаге умножать текущую маску на последней столбец в узле. Итак, алгоритм примет вид:

Алгоритм 3 DepthProject (реализованный)

Процедура DepthFirst(Узел : N, Маска : B)
  $B =$ ПостроитьМаску(N, B, T)
  $C = E(N)$
  $E = E(N, T, B, C)$
  Цикл <r := 1 до |E|> **выполнять**
    $N \cup \{i_r\} \rightarrow$ множество частых наборов
    $DepthFirst(N \cup \{i_r\}, T', B')$
  Конец цикла
  Если тогда $E = \varnothing$
    Если тогда $N -$ максимальный набор
      $N \rightarrow$ множество максимальных наборов
    Конец условия
  Конец условия
Конец процедуры

В этой реализации нет проекции, ведь можно обойтись без неё и даже получить выигрыш в скорости. В самом деле, зная маску и последний элемент в текущем узле мы легко найдём следующий совместимый столбец по исходной базе данных и потратим столько же времени, сколько занимает одна проекция. Возможно, есть способ сделать проекцию быстрее поиска с маской, но преимущество в скорости должно быть достаточно большим, поскольку сам факт рекурсивной передачи столь большого массива данных занимает определённое время. На практике одна лишь передача базы данных без последующей проекции увеличивает время работы алгоритма в два раза. Так или иначе, ни на количество шагов, ни на сложность шага это не влияет. Проекция - програмистское ухищрение авторов алгоритма, которое позволило им получить прирост в скорости. В моей реализации преимущество в скорости даёт маска и знание последнего элемента текущего узла.

4. Сравнение двух алгоритмов

При реализации алгоритмов использовалась одна и та же модель бинарной матрицы. В обоих алгоритмах маска - это бинарный вектор bitset (C++ 11 версии и выше).

На вход подаётся бинарная матрица, m - кол-во строк (транзакций), n - кол-во столбцов (элементов). Минимальная поддержка s = 0.1.

	n = 10	**n = 20**	**n = 30**
m = 10	183, 5 0.73 2.56	49557, 10 49 52	1029918, 10 1064 4250
m = 100	169, 76 0.89 1.52	1560, 785 3.39 6.52	9165, 4962 16.1 17.3
m = 1000	172, 117 1.34 1.87	1350, 1136 6.2 8.6	4518, 4053 23.3 33.9

В первой строке - кол-во частых наборов и кол-во макс. частых наборов. Во второй строке время работы AD в миллисекундах, в третьей - время работы DepthProject. Результаты были усреднены по 20 запускам.

Ниже приведена аналогичная таблица для минимальной поддержки s = 0.3.

	n = 10	**n = 20**	**n = 30**
m = 10	64, 10 0.184 0.398	573, 30 1.62 1.73	1225, 43 2.6 11.2
m = 100	16, 10 0.084 0.2	50, 34 0.383 0.216	110, 86 0.924 1.26
m = 1000	10, 10 0.154 0.222	20, 20 0.533 0.693	30, 30 0.817 1.043

Алгоритм AD немного быстрее, особенно в тех моментах, когда количество всех частых наборов много больше максимальных частых наборов. Нетрудно догадаться, что в этом случае дерево решений имеет большую глубину. Следовательно, рекурсивный вызов процедуры DepthProject имеет большую вложенность, а передача объектов в процедуру нагружает процессор. В этом и заключается преимущество последовательного подхода.

Изобразим на графике зависимость времени работы двух алгоритмов от количества столбцов при фиксированном количестве строк.

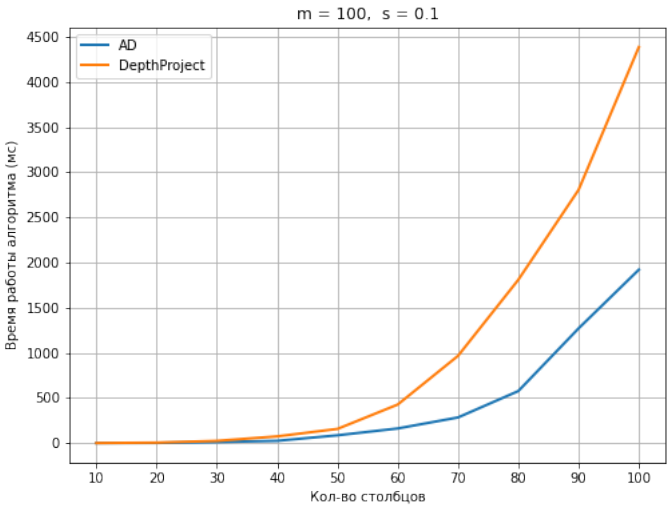

Рис. 1: Зависимость времени работы от кол-ва столбцов

Как видим, скорости счёта двух алгоритмов отличаются в константное число раз. Изобразим на графике отношение времени работы алгоритмов:

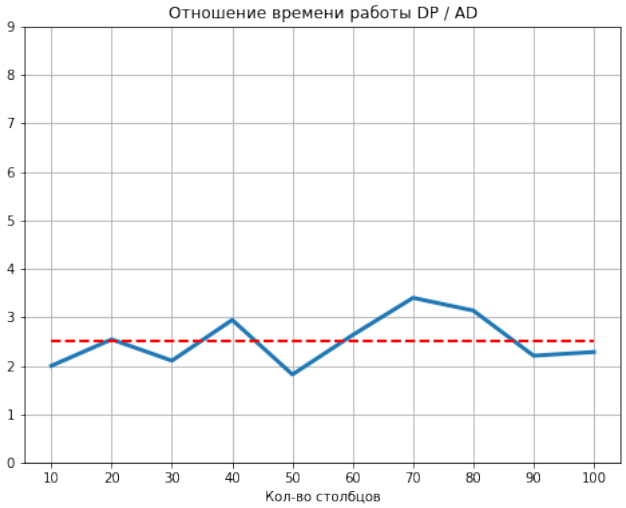

Рис. 2: Отношение времени работы DepthProject и AD

В среднем скорость счёта отличается в 2.5 раза. Это подтверждает тот факт, что алгоритмы совершают одинаковое число шагов. Сложность шага в DepthProject чуть больше, но это связано исключительно с реализацией.

Теперь изобразим на графике зависимость времени работы двух алгоритмов от количества строк при фиксированном количестве столбцов. Положим $n = 20$, $s = 0.1$:

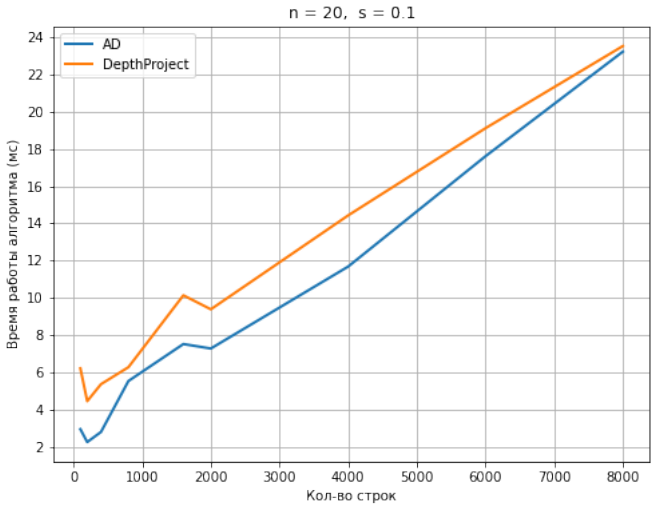

Рис. 3: Зависимость времени работы от кол-ва строк

Время работы двух алгоритмов асимптотически совпадает. Ниже представлен график отношения скоростей счёта:

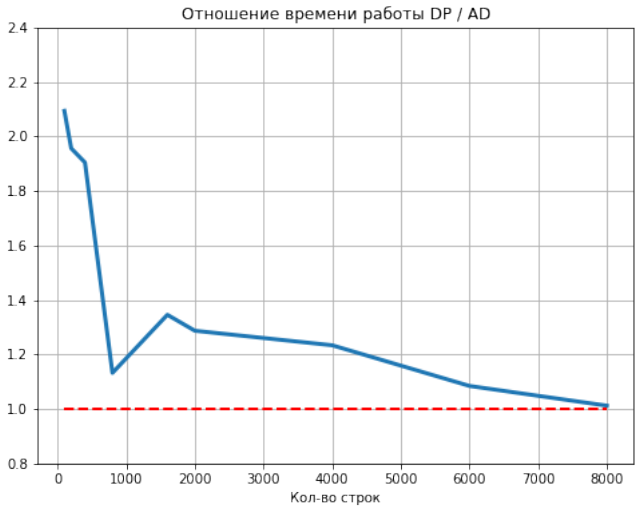

Рис. 4: Отношение времени работы DepthProject и AD

5. Вывод

Алгоритмы AD и DepthProject совершают одинаковое число шагов, имеют схожую сложность шага и выводят наборы в алфавитном порядке. Основное отличие заключается в том, что AD выполняется последовательно и использует меньше вычислительных ресурсов, в то время как DepthProject работает рекурсивно и строит полное дерево решений.

6. Дополнение

6.1. Удаление нечастых столбцов

Пусть дана бинарная матрица $T \in B^{m \times n}$, $B = \{0, 1\}$ и минимальная поддержка $s \in [0, 1]$. Мы можем удалить все столбцы матрицы T, имеющие меньше $m \cdot s$ единиц, ведь они не войдут ни в один частый набор. Назовём такие столбцы нечастыми.

Обозначим среднее количество нечастых столбцов за Y. Пусть матрица T является случайной и берётся из равномерного распределения. Ясно, что при малых s нечастых столбцов почти не будет, а при $s \approx 0.5$ и $s > 0.5$ такие столбцы почти наверное найдутся. Посчитаем, при каких s, m, n среднее количество таких столбцов отлично от нуля.

Элементы матрицы T имеют распределение Бернулли с $p = 0.5$. Следовательно, распределение единиц в отдельно взятом столбце представляет собой биномиальное распределение. Обозначим кол-во единиц в столбце за k, тогда

$$P(k = i) = C_m^i \cdot (\tfrac{1}{2})^i \cdot (\tfrac{1}{2})^{m-i} = \tfrac{1}{2^m} \cdot C_m^i.$$

Тогда несложно посчитать вероятность того, что столбец нечастый:

$$P(k < s) = \sum_{i=0}^{s-1} P(k = i) = \tfrac{1}{2^m} \cdot \sum_{i=0}^{s-1} C_m^i$$

Чтобы найти среднее количество нечастых столбцов умножим полученную вероятность на n. Окончательно получим:

$$Y = \tfrac{n}{2^m} \cdot \sum_{i=0}^{s-1} C_m^i$$

Можно показать, что $Y > 1$ начиная с $s > 0.4 \cdot m$ (примерно). Итак, если минимальная поддержка $s > 0.4$, то есть смысл перед поиском частых наборов удалить нечастые столбцы.