



# Содержание

|   |   |    |
|---|---|----|
| 1 | Введение                                  | 3  |
| 2 | Алгоритм Фафьяни и Кратча                 | 6  |
| 3 | Структура данных <code>supersets[]</code> | 7  |
| 4 | Линейная память                           | 8  |
| 5 | Алгоритм ван Беверна                      | 11 |
| 6 | Тестирование и сравнение алгоритмов       | 15 |
| 7 | Заключение                                | 17 |

## Аннотация

Линейные алгоритмы кернелизации для задачи о вершинном покрытии  $d$ -гиперграфа были предложены ван Беверном в 2014 году и Фафьяни и Кратчем в 2015 году. Обоим алгоритмам нужна структура данных, способная обращаться к данным, ассоциированным с кортежем чисел, за константное время. Ван Беверн экспериментировал с разными структурами данных. Хеш-таблицы имеют константный доступ к данным в среднем, но линейный в худшем случае. Сбалансированные деревья поиска обеспечивают линейную память, но неконстантное обращение. Префиксные деревья предоставляют доступ к памяти за константное время, но объём памяти, необходимый для построения такого дерева, сверхлинейный. В этой работе описан способ получить одновременно линейное время работы и линейную память. Оптимизация памяти применяется к алгоритму Фафьяни и Кратча и к алгоритму ван Беверна. Для оценки эффективности оптимизации проводятся вычислительные эксперименты.

## 1 Введение

Некоторые труднорешаемые задачи в частных и практически значимых случаях решаются быстро. В попытке обобщения таких “хороших” частных случаев для задачи вводятся *параметры*, относительно которых можно анализировать сложность задачи. Например, параметром может быть размер ответа задачи. В графовой задаче параметром может быть число компонент сильной связности или ограничение на максимальную степень вершины.

Приведём формальное определение [8].

**Определение 1.1.** *Параметризованная задача (язык задачи) — это множество  $L \subseteq \{0, 1\}^* \times \mathbb{N}$ . Здесь пара  $(X, k) \in \{0, 1\}^* \times \mathbb{N}$  состоит из входа задачи  $X$  и параметра  $k$ .*

**Определение 1.2.** *Параметризованная задача  $L \subseteq \{0, 1\}^* \times \mathbb{N}$  называется решаемой полиномиально с параметрозависимым множителем (принадлежит классу  $FPT$ ), если существует алгоритм  $A$ , определяющий принадлежность пары  $(X, k)$  языку задачи  $L$  за время  $f(k)|X|^c$ . Здесь  $f$  — произвольная вычислимая функция, а  $c$  — константа. Алгоритм  $A$  называется полиномиальным с параметрозависимым множителем.*

Одним из подходов к построению полиномиального с параметрозависимым множителем алгоритма является “сжатие” входа задачи  $X$  до размера, не зависящего от его исходного размера, а только лишь от  $k$ . Тогда, при запуске любого экспоненциального алгоритма решения задачи на “сжатом” входе  $X'$ , мы получим, что время работы экспоненциального алгоритма от размера исходного входа зависеть не будет.

Такое “сжатие” называется *редукцией данных* или, если размер “сжатого” входа зависит только от параметра  $k$ , *кERNELИЗАЦИЕЙ*.

**Определение 1.3.** *Кернелизация* параметрической задачи  $L \in \{0, 1\}^* \times \mathbb{N}$  — это полиномиальный алгоритм, входом которого является  $(X, k)$ , а выходом —  $(X', k')$ . При этом выполняются условия

1.  $(X, k) \in L \Leftrightarrow (X', k') \in L$ ,
2.  $|X'| \leq f(k)$  для некоторой вычислимой функции  $f$ ,
3.  $k' \leq g(k)$  для некоторой вычислимой функции  $g$ .

Результат  $(X', k')$  кернелизации называется *ядром*, а  $f(k)$  — его *размером*.

Поскольку редукция данных потенциально применяется к большим входам, важна полиномиальная или лучше линейная трудоёмкость кернелизации.

В этой работе предлагается новая кернелизация для следующей задачи.

**Задача о вершинном покрытии  $d$ -гиперграфа.**

*Дано:* гиперграф  $G = (V, E)$  такой, что  $\forall e \in E : e \subseteq V, |e| \leq d$ .

*Требуется:* найти *вершинное покрытие*  $S \subseteq V$  минимального размера, т.е. такое множество  $S \subseteq V$ , что  $\forall e \in E : e \cap S \neq \emptyset$  и  $|S| \rightarrow \min$ .

В дальнейшем для краткости будем называть гиперграф графом, а гиперрёбра — рёбрами.

Заметим, что ограничение на размер максимального ребра в гиперграфе  $d$  является частью задачи. В дальнейшем мы будем считать  $d$  константой. Решение задачи с таким ограничением представляет интерес и для достаточно маленьких  $d$ . Например, задача о кластеризации графа сводится к задаче о вершинном покрытии гиперграфа при  $d = 3$  [3], а задача о линейках Голомба — при  $d = 4$  [5]. Последняя задача имеет приложение при максимизации использованных радиочастот с минимизацией интерференций.

Параметризуем задачу, взяв в качестве параметра  $k$  — ограничение на размер вершинного покрытия  $S$ .

**Параметризованная задача о вершинном покрытии  $d$ -гиперграфа.**

*Дано:* гиперграф  $G = (V, E)$  такой, что  $\forall e \in E : e \subseteq V, |e| \leq d$ .

*Параметр:*  $k$ .

*Требуется:* определить, существует ли вершинное покрытие  $S \subseteq V$  размера не более  $k$ , т.е. такое множество  $S \subseteq V$ , что  $\forall e \in E : e \cap S \neq \emptyset$  и  $|S| \leq k$ .

Такая задача всё еще остается NP-трудной, однако для неё существует алгоритм полиномиальный с параметрозависимым множителем, работающий за время  $O(d^k |E|)$  [10].

В данной работе рассматриваются алгоритмы кернелизации для задачи о вершинном покрытии гиперграфа, с помощью которых можно ускорить известные полиномиальные с параметрозависимым множителем алгоритмы. Входом алгоритма кернелизации будет являться гиперграф  $G$ , а выходом — гиперграф  $G'$ .

В настоящее время существует несколько алгоритмов кернелизации для параметризованной задачи о вершинном покрытии. В 2003 году Нидермейер и Росманит [9] предложили алгоритм кернелизации задачи для  $d = 3$  с ядром размера  $O(k^3)$ . Для произвольного  $d$  первый алгоритм был предложен в 2006 году Флюмом и Гроэ [2]. В ядре кернелизации  $O(k^d)$  рёбер, что является асимптотически наилучшим ядром: в 2010 году Делл и ван Мелькебеек показали, что нельзя за полиномиальное время построить ядро  $O((k+1)^{d-\epsilon})$ , если не схлопывается полиномиальная иерархия [7]. Время работы алгоритма Флюма и Гроэ сверхлинейное, поэтому интерес в данной работе будут составлять более поздние алгоритмы. Алгоритмы ван Беверна 2014 года [4] и Фафьяни и Кратча 2015 года [6] получают ядра размера  $O(k^d)$  и работают за линейное время, однако требуют сверхлинейного объёма памяти. В следующих частях будут предложены модификации обоих алгоритмов, сокращающие объём требуемой памяти до линейного.

В дальнейшем будем пользоваться обозначениями:  $G = (V, E)$  — исходный граф,  $G' = (V', E')$  — результат кернелизации,  $n = |V|$ ,  $m = |E|$ . Ограничение на размеры рёбер —  $d$ . Вершины будем считать пронумерованными от 1 до  $n$ , а размером входа при подсчёте времени работы будем считать  $O(n + dm)$ .

## 2 Алгоритм Фафьяни и Кратча

В 2015 году Фафьяни и Кратч предложили алгоритм кернелизации (Алгоритм 1), работающий за линейное время [6]. Этот алгоритм сокращает число рёбер до  $(k + 1)^d$ .

Алгоритм 1 по очереди обрабатывает рёбра исходного графа  $G$  и решает, взять ли очередное ребро  $e$  в конечный граф  $G'$ . Для этого он рассматривает каждое подмножество  $s$  ребра  $e$  и проверяет, сколько рёбер, уже взятых в конечный граф  $G' = (V, E')$ , являются надмножествами  $e$ . Это число надмножеств хранится в структуре данных  $supersets[]$ . Если  $supersets[s]$  уже имеет предельное допустимое значение  $(k + 1)^{d-|s|}$  для некоторого  $s \subseteq e$ , то ребро  $e$  в граф можно не брать. Доказательство этого факта выходит за рамки данной работы, его можно найти в оригинальной работе Кратча и Фафьяни [6]. Если ребро  $e$  всё же было взято в конечный граф, то алгоритм увеличивает на 1 значения  $supersets[s]$  для всех подмножеств  $s$  ребра  $e$ , поскольку  $e$  теперь является новым надмножеством для  $s$ . Заметим, что после этого сохранилось соотношение  $supersets[s] \leq (k + 1)^{d-|s|}$  для любого  $s$ .

---

### Алгоритм 1: Алгоритм Фафьяни и Кратча

---

**Вход:**  $d$ -гиперграф  $G = (V, E)$ , натуральное число  $k$ .

**Выход:**  $d$ -гиперграф  $G' = (V, E') : |E'| \leq (k + 1)^d$ .

```
1  $L \leftarrow \emptyset$ ;  
2 foreach  $s \subseteq e, e \in E$  do  
3    $supersets[s] \leftarrow 0$ ;  
4  $E' \leftarrow \emptyset$ ;  
5 foreach  $e \in E$  do  
6    $take \leftarrow True$ ;  
7   foreach  $s \subseteq e$  do  
8     if  $supersets[s] \geq (k + 1)^{d-|s|}$  then  
9       // ребро  $e$  не попадёт в граф  $G'$   
9        $take \leftarrow False$ ;  
10  if  $take$  then  
11     $E' \leftarrow E' \cup \{e\}$ ;  
12    foreach  $s \subseteq e$  do  
13       $supersets[s] \leftarrow supersets[s] + 1$ ;  
14 return  $G' = (V, E')$ ;
```

---

Получить оценку размер выходного графа несложно. Любое ребро  $e \in E'$  является надмножеством для  $\emptyset$ . Для любого множе-

ства  $s$  сохраняется соотношение  $\text{supersets}[s] \leq (k+1)^{d-|s|}$ . Поэтому число рёбер  $|E'| = \text{supersets}[\emptyset] \leq (k+1)^d$ .

### 3 Структура данных `supersets[]`

Как было видно в предыдущем разделе, алгоритм 1 использует структуру данных `supersets[s]`. В ней хранится число взятых в граф  $G'$  рёбер-надмножеств для каждого возможного подрёбра. Фафьяни и Кратч в качестве такой структуры данных используют префиксное дерево, предложенное ван Беве́рном в его алгоритме [4]. Сначала рассмотрим такую структуру данных и покажем, что она потребляет сверхлинейный объём памяти.

Префиксное дерево строится следующим образом. Пусть  $S$  — множество всех возможных подмножеств всех рёбер  $e \in E$ , то есть

$$s \in S \iff \exists e \in E \mid s \subseteq e.$$

Для каждого множества  $s \in S$  создадим свою вершину  $\text{node}_s$  в дереве и выделим для неё блок памяти. Блок состоит из счётчика  $\text{counter}_s$ , который будет хранить значение  $\text{supersets}[s]$ , и массива указателей `nextset_s[]` размера  $n$ . Элемент массива `nextset_s[x]` указывает на блок памяти для вершины дерева  $\text{node}_{s'}$ , где  $s' = s \cup \{x\}$ , тогда и только тогда, когда  $s' \in S$  и  $\forall y \in s : x > y$ . Иначе `nextset_s[x]` не указывает никуда.

Мы требуем, чтобы номер вершины  $x$  был больше, чем у любой другой в  $s$ , чтобы путь от корневой вершины  $\text{node}_\emptyset$ , до любой другой вершины  $\text{node}_s$  определялся единственным образом. Чтобы найти блок памяти, соответствующий  $\text{node}_s$  для  $s = \{x_1, x_2, \dots, x_{|s|}\}$ , отсортируем  $x_1, x_2, \dots, x_{|s|}$  по возрастанию. Теперь можно считать, что  $x_1 \leq x_2 \leq \dots \leq x_{|s|}$ . Будем переходить поочередно по вершинам дерева, соответствующим множествам  $s_0 = \emptyset$ ,  $s_1 = s_0 \cup \{x_1\}$ ,  $\dots$ ,  $s_{i+1} = s_i \cup \{x_{i+1}\}$ ,  $\dots$ ,  $s_{|s|} = s_{|s|-1} \cup \{x_{|s|}\}$ . Пусть  $p_0$  — указатель на блок памяти, соответствующий корневой вершине. Имея указатель  $p_i$  на память для множества  $s_i$  можем получить  $p_{i+1} = \text{nextset}_{s_i}[x_{i+1}]$ . Чередой таких переходов получим нужный  $p_{|s|}$ .

Так как для каждого множества  $s$ , нужно сделать  $|s| \leq d$  переходов, каждый из которых делается за константное время, то доступ к  $\text{counter}_s$  происходит за время  $O(d)$  для любого множества, где  $d$  — константа. Значит доступ к `supersets[s]` — константный по времени.

Теперь, проанализируем размер памяти, необходимый для работы такой структуры данных. Число вершин в дереве — это хотя

бы число различных множеств в  $S$ . Каждое множество требует наличия в дереве еще  $d$  множеств, чтобы построить путь от корня до него. Эти множества —  $\emptyset, \{x_1\}, \{x_1, x_2\}, \dots, s \setminus \{x_{|s|}\}$ . Но все эти множества мы уже посчитали, так как они входят в множество  $S$ . Поэтому число вершин в дереве ровно  $|S|$ . Для каждой из этих вершин мы храним массив из  $n$  указателей и один счетчик. Поэтому необходимый размер памяти  $\Theta(|S|n)$ . Так как  $|S| \leq 2^d m$ , то асимптотика потребляемой памяти  $O(2^d mn) = O(mn)$ . Поскольку  $S$  содержит по крайней мере все рёбра графа,  $S \geq m$ . Это значит, что необходимо иметь память размера  $\Omega(mn)$ .

Проблема в огромном числе неиспользуемых указателей, которые никуда не указывают. Можно их убрать и хранить только полезные указатели, но это будет стоить добавочного времени работы. В следующей части будет показано, как сохранить константное время обращения к  $supersets[s]$  и добиться резервирования лишь линейной памяти.

## 4 Линейная память

Опишем метод сокращения объема требуемой памяти до линейной, сохраняя линейное время работы, для алгоритма Фафьяни-Кратча (Алгоритм 2). Метод будет основан на предварительной сортировке запросов к структуре данных  $supersets[]$ .

Введём несколько обозначений. Нужно уметь адресоваться к рёбрам, поэтому для каждого ребра  $e \in E$  введём идентификатор ребра  $e\_id$ . Это уникальное для каждого ребра целое число в отрезке  $[1, m]$ . Множество  $s \subseteq V : |s| \leq d$  в памяти компьютера представляем следующим способом.

**Определение 4.1.** Код множества  $s$  — это кортеж  $code(s)$  размера  $d$  такой, что первые  $|s|$  его значений — номера вершин из  $s$  в возрастающем порядке. Последние  $d - |s|$  его значений — нули.

Такой способ представления множеств позволяет нам хранить каждое множество в виде  $d$  чисел. Также, при таком представлении, можно сравнивать два множества лексикографически. Для этого просто представим эти коды множеств строками длины  $d$  с символами из алфавита  $\{0, 1, 2, \dots, n\}$  и сравним их как строки.

Заметим, что Алгоритм 1 всегда обращается к  $supersets[s]$  для подмножества  $s$  некоторого ребра  $e$  только, когда обрабатывает это ребро.



**Определение 4.2.** *Запрос* к структуре данных  $supersets[]$  — это пара  $(code(s), e\_id)$ , где  $e \in E, s \subseteq e$ . Будем называть  $code(s)$  *аргументом запроса*.

Запросы будем сравнивать между собой по аргументу запроса. То есть

$$(code(s1), e1\_id) < (code(s2), e2\_id) \Leftrightarrow code(s1) < code(s2).$$

Улучшенный алгоритм Фафьяни и Кратча (Алгоритм 2) работает следующим образом.

Пусть  $L$  — список всех возможных запросов к структуре данных  $supersets[]$ . То есть  $L = \{(code(s), e\_id) : e \in E, s \subseteq e\}$ . Этот список имеет линейный от входа размер  $O(2^d m)$ . Найти его можно за линейное время. Для этого совершим проход по всем рёбрам  $e \in E$ , для каждого из них найдем все подрёбра  $s \subseteq e$  и запрос  $(code(s), e\_id)$  положим в  $L$  (строка 3 Алгоритма 2).

Так как  $code(s)$  сравнимы между собой, то список  $L$  можно отсортировать и получить список  $L'$  (строка 4). Сортируем по аргументу запроса, который является кортежем длины  $d$  из чисел в отрезке  $[0, n]$ . Значит это можно сделать за время  $O(dn + 2^d dm)$  цифровой сортировкой [1].

Заметим, что при сортировке мы не различали между собой запросы  $(code(s), e1\_id)$  и  $(code(s), e2\_id)$ , так как сортировали только по аргументу запроса. Все запросы с одинаковым аргументом будут располагаться в  $L'$  непрерывным отрезком. Назовём такой отрезок *классом эквивалентности*.

Легко разбить  $L'$  на классы эквивалентности за линейное время и выдать каждому элементу в  $L'$  уникальный идентификатор его класса (строки 5-13). Для этого выдадим первому элементу в  $L'$  идентификатор 1, что будет означать первый класс эквивалентности. Теперь пробежим по всем остальным элементам в  $L'$  по порядку и будем делать следующее. Пусть, мы сейчас рассматриваем элемент  $(code(s_{cur}), e_{cur\_id})$ , а предыдущим элементом был  $(code(s_{prev}), e_{prev\_id})$ . Если  $code(s_{cur}) = code(s_{prev})$ , то текущий элемент в том же классе эквивалентности, что и предыдущий. Выдадим ему такой же идентификатор. Иначе выдадим ему идентификатор на 1 больше, чем у предыдущего элемента. Это будет означать новый класс эквивалентности.

---

## Алгоритм 2: Модифицированный алгоритм Фабьяни и Кратча

---

**Вход:**  $d$ -гиперграф  $G = (V, E)$ , натуральное число  $k$ .

**Выход:**  $d$ -гиперграф  $G' = (V, E') : |E'| \leq (k + 1)^d$ .

```

1  $L \leftarrow \emptyset$ ;
2 foreach  $s \subseteq e, e \in E$  do
3    $\lfloor$  put ( $code(s), e\_id$ ) in  $L$ ;
4  $L' \leftarrow$  ЦифроваяСортировка( $L$ );
5  $classes\_number \leftarrow 0$ ;
6  $prev\_key \leftarrow code(\emptyset)$ ;
7  $sizes \leftarrow \emptyset$ ;
8 foreach  $(key, e\_id) \in L'$  do
9   if это первая итерация  $\vee key \neq prev\_key$  then
10      $classes\_number \leftarrow classes\_number + 1$ ;
11      $sizes[classes\_number] \leftarrow$  число ненулевых
12       значений в  $key$ ;
12   положить  $classes\_number$  в  $subset\_list[e\_id]$ ;
13    $prev\_key \leftarrow key$ ;
14 foreach  $s\_id \in [1 \dots classes\_number]$  do
15    $\lfloor$   $supersets[s] \leftarrow 0$ ;
16  $E' \leftarrow \emptyset$ ;
17 foreach  $e \in E$  do
18    $take \leftarrow True$ ;
19   foreach  $s\_id \in subset\_list[e\_id]$  do
20     if  $supersets[s\_id] \geq (k + 1)^{d - sizes[s\_id]}$  then
21       // ребро  $e$  не попадёт в граф  $G'$ 
22        $take \leftarrow False$ ;
22   if  $take$  then
23      $E' = E' \cup \{e\}$ ;
24     foreach  $s\_id \in subset\_list[e\_id]$  do
25        $\lfloor$   $supersets[s\_id] \leftarrow supersets[s\_id] + 1$ ;
26 return  $G' = (V, E')$ ;

```

---

Каждый класс эквивалентности обозначает уникальное множество  $s \in S$ . Номер класса, соответствующего множеству  $s$ , назовём  $s\_id$ . Это число в отрезке  $[1, |S|]$ . Заметим, что мы перешли от множеств  $s$  к кодам  $code(s)$  и наконец к числам  $s\_id$ , так что теперь обращаться к структуре данных  $supersets[]$  можно по целочислен-

ному индексу, а значит в качестве *supersets*[] можно использовать обычный массив.

Чтобы при обработке очередного ребра  $e$  знать индексы  $s\_id$  всех его подрёбер  $s$ , будем хранить их в списке *subset\_list*[ $e\_id$ ][]. Обработывая ребро  $e$ , алгоритм пробегает по всем подмножествам  $e$  в некотором порядке. Теперь этот порядок будет определен порядком элементов в *subset\_list*[ $e\_id$ ][], потому что алгоритм будет пробегать по этому списку, для всех  $i \in [1 \dots 2^{|e|}]$  обращаясь к *supersets*[*subset\_list*[ $e\_id$ ][ $i$ ]] (строка 19).

Все, что осталось сделать, — это заполнить для каждого ребра  $e \in E$  список *subset\_list*[ $e\_id$ ][]. Для каждого возможного запроса  $(code(s), e\_id) \in L'$  положим в *subset\_list*[ $e\_id$ ][] номер класса эквивалентности  $s\_id$ , в котором лежит эта пара (строка 12).

Подготовка структуры данных *supersets*[] закончена, и можно запускать алгоритм кернелизации. Остаётся одна маленькая деталь: для каждого класса эквивалентности нужно знать размеры  $|s|$  его представителей  $s$ , чтобы поддерживать ограничение из строки 8 в Алгоритме 1. Можно хранить эти значения для каждого класса напрямую или запоминать произвольного представителя для каждого класса. В Алгоритме 2 для этого используется массив *sizes*[].

Время работы состоит из времени сортировки  $O(dn + 2^d dm)$  и времени работы алгоритма Фафьяни и Кратча  $O(2^d m)$ . Итоговое время работы  $O(dn + 2^d dm)$ .

Массивы *supersets*[] и *sizes*[] имеют размер  $O(2^d m)$ , а список  $L$  —  $O(2^d dm)$ . Ещё  $O(n + 2^d m)$  памяти потребуется для сортировки списка  $L$ . Итоговый размер выделяемой памяти:  $O(n + 2^d dm)$ .

Таким образом, показано:

**Теорема 4.1.** Для задачи о вершинном покрытии  $d$ -гиперграфа на  $n$  вершинах с  $m$  рёбрами может быть построена кернелизация

1. с линейной трудоёмкостью  $O(dn + 2^d dm)$ ,
2. с линейной памятью  $O(n + 2^d dm)$ ,
3. с не более чем  $(k + 1)^d$  рёбрами в ядре.

## 5 Алгоритм ван Беверна

Алгоритм ван Беверна (Алгоритм 3) работает за линейное время и сокращает число рёбер до  $d!d^{d+1}(k + 1)^d$ . Такая оценка на размер ядра хуже, чем у алгоритма Фафьяни и Кратча, однако

последний не всегда может сократить вход, когда этоо делает алгоритм ван Беверна. Подробнее об этом в разделе 6.

Для понимания работы алгоритма введём понятие *подсолнуха*.

**Определение 5.1.** *Подсолнухом* с центром  $s$  в гиперграфе  $G = (V, E)$  называется множество  $F \subseteq E$  рёбер такое, что  $s \subseteq e$  для каждого ребра  $e \in F$ , и каждая пара рёбер  $e_1, e_2 \in F$  пересекается ровно по множеству  $s$ . Рёбра  $e \in F$  называются *лепестками*. Размер подсолнуха —  $|F|$ .

Можно показать, что для подсолнуха размера  $k + 2$ , любой его лепесток можно выкинуть из графа, и ответ на параметризованную задачу о вершинном покрытии  $d$ -гиперграфа не изменится. На этом факте и основан Алгоритм 3.

---

### Алгоритм 3: Алгоритм ван Беверна

---

**Вход:**  $d$ -гиперграф  $G = (V, E)$ , натуральное число  $k$ .

**Выход:**  $d$ -гиперграф  $G' = (V, E') : |E'| \leq d!d^{d+1}(k + 1)^d$ .

```

1 foreach  $s \subseteq e, e \in E$  do
2    $petals[s] \leftarrow 0$ ;
3   foreach  $v \in e$  do
4      $used[s][v] \leftarrow False$ ;
5  $E' \leftarrow \emptyset$ ;
6 foreach  $e \in E$  do
7    $take \leftarrow True$ ;
8   foreach  $s \subseteq e$  do
9     if  $petals[s] > k$  then
10       $//$  ребро  $e$  не попадёт в граф  $G'$ 
11       $take \leftarrow False$ ;
12   if  $take$  then
13      $E' \leftarrow E' \cup \{e\}$ ;
14     foreach  $s \subseteq e$  do
15       if  $\forall v \in e \setminus s : used[s][v] = False$  then
16          $petals[s] \leftarrow petals[s] + 1$ ;
17         foreach  $v \in e \setminus s$  do
18            $used[s][v] \leftarrow True$ ;
19 return  $G' = (V, E')$ ;

```

---

Алгоритм 3 просматривает все рёбра графа  $G = (V, E)$  по очереди и для каждого из них решает, брать ли его в кернелизованный граф  $G' = (V, E')$ . По ходу работы алгоритм строит подсолнух  $F_s$

для каждого множества  $s \subseteq e : e \in E$ . При взятии ребра  $e$  в  $E'$ , он выясняет для каждого множества  $s \subseteq e$ , можно ли добавить  $e$  в подсолнух  $F_s$ . Если в подсолнухе  $F_s$  и так есть хотя бы  $k + 1$  лепесток, то ребро  $e$  можно не брать в граф  $G'$  (строки 9-10 Алгоритма 3). Если каждый из подсолнухов  $F_s$ ,  $s \subseteq e$  еще недостаточно велик, то можно в каждый из них попробовать добавить новый лепесток  $e$ . Алгоритм смотрит, не пересечёт ли множество  $e \setminus s$  какое-нибудь другое ребро  $e' \in F_s$  (строка 14). Для этого он хранит в структуре данных  $used[s][v]$  булево значение для каждой вершины  $v$ , истинное тогда и только тогда, когда  $v \in \bigcup_{e' \in F_s} e' \setminus s$ . Если для всех вершин  $v \in e \setminus s$  выполняется  $used[s][v] = False$ , то  $e$  не пересекает других лепестков подсолнуха не по множеству  $s$ . Тогда оно само может стать лепестком. Для этого алгоритм помечает  $True$  нужные значения в  $used[s][v]$  и увеличивает на 1 счётчик лепестков  $petals[s]$ .

Доказательство оценки на размер ядра можно прочитать в работе ван Беверна [4].

Память, требуемая алгоритмом для его работы, сверхлинейна из-за структур данных  $petals[]$  и  $used[][]$ . Первая из них — это префиксное дерево, а вторая — префиксное дерево, у которого в вершинах, соответствующих рёбрам графа, хранится массив из  $n$  булевых значений. Улучшим потребление памяти и этого алгоритма тоже.

Заметим, что структура данных  $petals[]$  так же как и  $supersets[]$  хранит счётчик для некоторых подрёбер  $s \subseteq e : e \in E$ . Значит, все суждения относительно структуры данных  $supersets[]$  применимы и к структуре данных  $petals[]$ . Запросы к ней можно отсортировать и сделать  $petals[]$  массивом. Для  $used[][]$  применим похожую методику сортировки запросов.

**Определение 5.2.** *Запрос* к структуре данных  $used[][]$  — это тройка  $(code(s), v, e\_id)$ , где  $e \in E, s \subseteq e, v \in e \setminus s$ . Будем называть  $code(s)$  и  $v$  *первым* и *вторым аргументами запроса* соответственно.

Сравнивать такие запросы будем по первому аргументу запроса, а затем по второму, то есть

$$\begin{aligned} & (code(s1), v1, e1\_id) < (code(s2), v2, e2\_id) \Leftrightarrow \\ & \Leftrightarrow code(s1) < code(s2) \vee (code(s1) = code(s2) \wedge v1 < v2). \end{aligned}$$

Заметим, что к таким запросам снова применима цифровая сортировка, так как можно считать  $code(s) \cup \{v\}$  одним кортежем длины  $d + 1$  с элементами из  $[0, n]$ .

Улучшенный алгоритм ван Беверна (Алгоритм 4) работает следующим образом.

---

**Алгоритм 4:** Модифицированный алгоритм ван Беверна

---

**Вход:**  $d$ -гиперграф  $G = (V, E)$ , натуральное число  $k$ .  
**Выход:**  $d$ -гиперграф  $G' = (V, E') : |E'| \leq d!d^{d+1}(k+1)^d$ .

- 1  $L \leftarrow \emptyset$ ;
- 2 **foreach**  $e \in E, s \subseteq e, v \in e \setminus s$  **do**
- 3      $\lfloor$  put ( $code(s), v, e\_id$ ) in  $L$ ;
- 4  $L' \leftarrow$  ЦифроваяСортировка( $L$ );
- 5  $classes\_number \leftarrow 0$ ;
- 6  $subclasses\_number \leftarrow 0$ ;
- 7 **foreach**  $(key1, key2, e\_id) \in L'$  **do**
- 8     Пересчитать  $classes\_number$  и  $subclasses\_number$  сравнив  $key1$  и  $key2$  в текущем элементе списка и в предыдущем;
- 9     **if**  $subset\_list[e\_id] = \emptyset \vee subset\_list[e\_id][size(subset\_list[e\_id]) - 1] \neq classes\_number$  **then**
- 10          $\lfloor$  положить  $classes\_number$  в  $subset\_list[e\_id]$ ;
- 11          $\lfloor$  положить  $\emptyset$  в  $vertex\_list[e\_id]$ ;
- 12         положить  $subclasses\_number$  в  $vertex\_list[e\_id][size(vertex\_list[e\_id]) - 1]$ ;
- 13 Занулить все значения  $petals[]$  и  $used[][]$ ;
- 14 Запустить алгоритм ван Беверна;
- 15 **return**  $G' = (V, E')$ ;

---

Положим в список  $L$  все запросы к структуре данных  $used[][]$  из Алгоритма 3, отсортируем и разобьем на классы эквивалентности с одинаковым первым аргументом запроса. Заметим, что таким образом мы уже для каждого ребра  $e$  нашли все  $s\_id$  для дальнейшей индексации в массивах  $petals[]$  и  $used[]$ . Положим  $s\_id$  в массив  $subset\_list[e\_id][][]$  для каждого ребра  $e$ , присутствующего в запросах из класса эквивалентности с номером  $s\_id$ .

Каждый класс эквивалентности теперь можно разбить на *подклассы эквивалентности*, такие непрерывные отрезки, для которых совпадает второй аргумент запроса. Выдадим каждому подклассу класса с номером  $s\_id$  свой уникальный идентификатор  $v\_id$ . После строки 8 Алгоритма 4 значения  $classes\_number$  и  $subclasses\_number$  будут равны  $s\_id$  и  $v\_id$  соответственно. Те-

перь можно обращаться в  $used[s\_id][v\_id]$  по индексу  $v\_id$ .

Рассмотрим подкласс эквивалентности  $C$  с номером  $v\_id$  в классе эквивалентности с номером  $s\_id$ . Для каждого ребра  $e$  в запросе  $(code(s), v, e\_id) \in C$  в массив  $vertex\_list[e\_id][s\_id']$  положим  $v\_id$ . Здесь  $s\_id'$  — это номер класса эквивалентности  $s\_id$  в списке  $subset\_list[e\_id]$ . То есть мы не храним для ребра  $e$  лишних списков номеров вершин, а только непустые. Теперь нужные значения  $v\_id$  хранятся в  $vertex\_list[e\_id][s\_id']$  для того, чтобы при обработке ребра  $e$  и его подребра  $s$ , были известны значения  $v\_id$ , по которым будут происходить обращения в массив  $used[e\_id]$ .

Время работы Алгоритма 4 складывается из времени сортировки  $O(dn + 2^d dm)$  и времени работы алгоритма ван Беверна  $O(2^d dm)$ . Итоговое время работы  $O(dn + 2^d dm)$ .

Массив  $petals$  имеет размер  $O(2^d m)$ , а список  $L$  —  $O(2^d dm)$ . Для сортировки  $L$  потребуется  $O(n + 2^d m)$  дополнительной памяти. Сумма размеров массивов  $used[s\_id]$  по всем  $s\_id$  не превышает числа запросов к структуре данных  $used$ , а значит имеет значение  $O(2^d dm)$ . Итоговый размер выделяемой памяти:  $O(n + 2^d dm)$ .

## 6 Тестирование и сравнение алгоритмов

Оба Алгоритма 2 и 4 были реализованы на языке C++, скомпилированы при помощи GNU C++ 5.4.0 с уровнем оптимизации -O2 и протестированы на компьютере с процессором Intel Core i7-4600U с тактовой частотой 2.10GHz и 64-битной операционной системой Ubuntu 16.04.

Тестирование проходило на гиперграфах, порождаемых задачей о подлинейках Голомба. Это важная подзадача в построении линейек Голомба: множеств  $R \subseteq \mathbb{N}$  таких, что разность любых двух чисел в нём уникальна. То есть для любой четверки чисел  $a, b, c, d \in R$  выполняется  $|a - b| = |c - d| \Rightarrow \{a, b\} = \{c, d\}$ . Задача о подлинейках Голомба — это задача поиска наибольшей по размеру линейки Голомба  $R \subseteq M$  для некоторого  $M \subseteq \mathbb{N}$ . Эта задача сводится к задаче о вершинном покрытии 4-гиперграфа [5]. Для этого построим гиперграф конфликтов  $G = (M, E)$ , где множество рёбер  $E$  будет содержать все четверки  $\{a, b, c, d\} : |b - a| = |c - d|$  и тройки  $\{a, b, c\} : |a - b| = |b - c|$ . Ответ к задаче о вершинном покрытии гиперграфа для  $G$  — это множество всех чисел, которые не войдут в ответ к задаче о подлинейках Голомба  $R$ . В наших тестах

$M = \{1, \dots, n\}$  для различных  $n \in [100, 300]$ . Параметр  $k$  выбран равным нижней оценке на размер ответа. Для её нахождения, мы ищем максимальное по включению множество попарно непересекающихся рёбер. Ясно, что размер такого множества не превосходит размер вершинного покрытия.

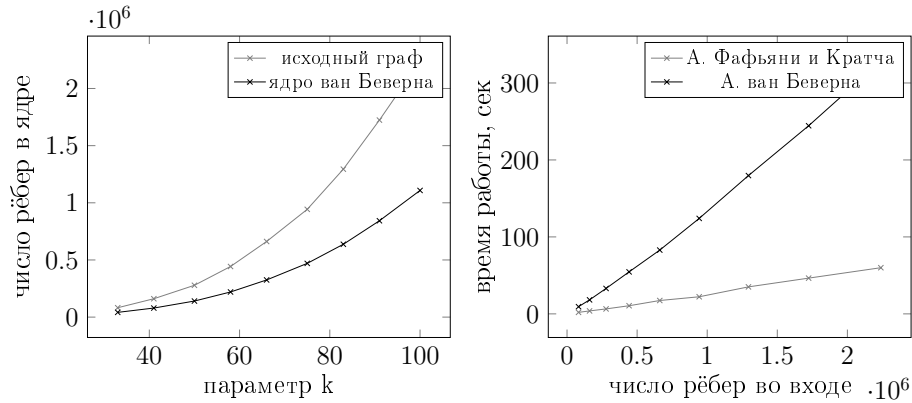


Рис. 1: Сравнение работы алгоритмов ван Беверна и Фэфьяни и Кратча.

В ходе тестирования было замечено, что Алгоритм 2 (основанный на алгоритме Фэфьяни-Кратча) не всегда сжимает гиперграф, когда алгоритм ван Беверна это делает, хотя верхняя оценка на число рёбер в ядре для алгоритма ван Беверна больше. Более того, для задачи о подлинейках Голомба, алгоритм Фэфьяни и Кратча не сокращал число рёбер вовсе, и размер ядра совпадает с размером исходного графа. Параметр  $k$  в тестах, порождённых задачей о подлинейках Голомба, настолько велик, что оценка на число рёбер в ядре  $(k + 1)^d$  для алгоритма Фэфьяни и Кратча превышает исходное число рёбер, и сокращения графа не происходит. Алгоритм ван Беверна имеет худшую оценку на размер ядра, но всё-таки сокращает число рёбер приблизительно в 2 раза.

Для более эффективного сжатия входного графа, можно использовать оба алгоритма: модифицированный Фэфьяни-Кратча и ван Беверна в этом порядке. Тогда, более тяжеловесный по времени и памяти алгоритм ван Беверна будет работать на уже сжатом графе, и, возможно, сократит его сильнее.



## 7 Заключение

В данной работе приведены два алгоритма кернелизации для задачи о вершинном покрытии гиперграфа. Оба алгоритма работают за линейное время, повторяя предыдущие результаты, однако достигают линейности по памяти.

Реализованы программы на языке C++. Было произведено сравнение алгоритма Фафьяни и Кратча с алгоритмом ван Беверна и обнаружена относительно низкая эффективность первого на примере задачи о линейках Голomba. Предложена идея о комбинировании алгоритмов.

Остаётся открытым вопрос о меньшем потреблении памяти при кернелизации для задачи о вершинном покрытии гиперграфа. Предпосылки к этому есть: Фафьяни и Кратч предложили алгоритм кернелизации, работающий за время  $O(m^{d+2})$ , однако требующий лишь логарифмическую память.

## Список литературы

- [1] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. *Алгоритмы: построение и анализ*. Москва. МЦНМО. 2000.
- [2] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006
- [3] F. Hüffner, C. Komusiewicz, H. Moser, R. Niedermeier. Fixed-Parameter Algorithms for Cluster Vertex Deletion. *Theory of Computing Systems* 47(1): 196-217, 2010.
- [4] R. van Bevern. Towards Optimal and Expressive Kernelization for  $d$ -Hitting Set. *Algorithmica* 70(1):129-147, 2014. arXiv:1112.2310
- [5] M. Sorge, H. Moser, R. Niedermeier, and M. Weller. Exploiting a hypergraph model for finding Golomb rulers. *Acta Informatica* 51(7):449-471, 2014.
- [6] S. Fafianie and S. Kratsch. A shortcut to (sun)flowers: Kernels in logarithmic space or linear time. In *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS'15)*, volume 9235 in LNCS, pp. 299-310, Springer, 2015.
- [7] H. Dell and D. van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *Journal of the ACM*, 61(4): 23:1-23:27

- [8] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015
- [9] R. Niedermeier and P. Rossmanith. An efficient fixed-parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms* 1(1), 2003, pages 89–102
- [10] R. Niedermeier. *Invitation to Fixed Parameter Algorithms*. Oxford University Press, 2006