

# Конспект по вычислительной геометрии (Ковалёв)

Volkov M., Mukhutdinov D., Belyy A.

25 февраля 2016 г.

## Содержание

<b>1</b>	<b>Как пользоваться этим документом</b>	<b>4</b>
<b>2</b>	<b>Tickets</b>	<b>5</b>
<b>3</b>	<b>3 1: Skip quadtree</b>	<b>AVBELYY 6</b>
3.1	3 Сжатое квадрдерево . . . . .	6
3.2	3 Randomized skip quadtree . . . . .	6
<b>4</b>	<b>3 2: Пересечение прямоугольника с множеством пр-уг/отр</b>	<b>FLYINGLEAFE 9</b>
4.1	3 Пересечение прямоугольника запроса $q$ с множеством прямоугольников $P$ . . . . .	9
4.1.1	3 Нахождение множества точек, попадающих в прямоугольник запроса. . . . .	9
4.1.2	3 Нахождение множества отрезков, пересекающих прямоугольник запроса (но не лежащих концом внутри) . . . . .	12
4.1.3	3 Нахождение всех прямоугольников, в которые попадает точка. . . . .	16
4.2	3 Пересечение прямоугольника с множеством непересекающихся отрезков. . . . .	17
4.2.1	3 Segment tree . . . . .	17
<b>5</b>	<b>1 3: Пересечение отрезков и поворот</b>	<b>VOLHOVM 19</b>
<b>6</b>	<b>3 4: Локализация в многоугольнике</b>	<b>AVBELYY 21</b>
6.1	3 Выпуклый многоугольник . . . . .	21
6.2	3 Невыпуклый многоугольник . . . . .	21
<b>7</b>	<b>3 5: Статические выпуклые оболочки в <math>\mathbb{R}</math></b>	<b>FLYINGLEAFE 22</b>
7.1	3 Джарвис (заворачивание подарка) . . . . .	22
7.2	3 Грэм . . . . .	22
7.3	3 Эндрю . . . . .	22
7.4	3 Чен . . . . .	23
7.5	3 QuickHull . . . . .	23
7.6	3 Оболочка многоугольника . . . . .	24
7.7	3 Оболочка полилинии . . . . .	24
<b>8</b>	<b>3 6: Динамическая выпуклая оболочка</b>	<b>VOLHOVM 26</b>
8.1	3 Задача объединения двух верхних $CH$ . . . . .	26
8.2	3 Итеративный алгоритм . . . . .	27
<b>9</b>	<b>1 7: Трехмерные выпуклые оболочки (CHN)</b>	<b>FLYINGLEAFE 28</b>
<b>10</b>	<b>3 8: Триангуляция (существование и ушная триангуляция)</b>	<b>AVBELYY 30</b>
10.1	3 Существование . . . . .	30
10.2	3 Ушная триангуляция . . . . .	30
10.2.1	Алгоритм (ушная триангуляция) . . . . .	31

<b>11 3 9: Триангуляция с заматающей прямой</b>	<b>AVBELYY 32</b>
11.1 3 Идея и основные определения . . . . .	32
11.2 3 Алгоритм 1 (разбиение на монотонные части)	32
11.2.1 Split–вершина . . . . .	33
11.2.2 Merge–вершина . . . . .	33
11.2.3 Start–вершина . . . . .	33
11.2.4 End–вершина . . . . .	33
11.2.5 Regular–вершина . . . . .	33
11.3 3 Оценка времени работы алгоритма разбиения . . . . .	34
11.4 3 Алгоритм 2 (триангуляция монотонного многоугольника)	34
11.5 3 Оценка времени работы алгоритма триангуляции . . . . .	34
<b>12 2 10: Полуплоскости и выпуклые оболочки</b>	<b>AVBELYY 36</b>
12.1 3 Построение выпуклой оболочки . . . . .	36
12.1.1 Сведение к двойственным задачам . . . . .	36
12.1.2 Алгоритм . . . . .	36
12.2 2 Разрезание прямоугольника . . . . .	37
12.2.1 IntersectConvexPolygonAndLine(P, l) . . . . .	37
<b>13 2 11: Пересечение множества отрезков</b>	<b>AVBELYY 38</b>
13.1 Идея . . . . .	38
13.2 Подход 1. Близость вдоль оси Y . . . . .	38
13.3 Подход 2. Близость вдоль оси X . . . . .	38
13.4 Алгоритм . . . . .	38
13.4.1 Начало отрезка . . . . .	39
13.4.2 Конец отрезка . . . . .	39
13.4.3 Пересечение отрезков . . . . .	39
13.5 Структуры данных . . . . .	39
13.6 Время работы алгоритма . . . . .	40
<b>14 3 12: PSLG и DCEL: построение PSLG множества прямых</b>	<b>FLYINGLEAFE 41</b>
14.1 3 Определение . . . . .	41
14.2 3 Построение PSGL по множеству прямых . . . . .	41
<b>15 2 13: PSLG overlaying</b>	<b>AVBELYY 44</b>
15.1 Объединение вершин и полуребер . . . . .	44
15.2 Обновление фейсов . . . . .	44
<b>16 3 14: Локализация в PSLG</b>	<b>FLYINGLEAFE 46</b>
16.1 3 Метод полос . . . . .	46
16.1.1 3 Персистентные деревья . . . . .	46
16.1.2 3 Очень классные персистентные деревья!	46
16.2 3 Киркпатрик . . . . .	47
<b>17 3 15: Трапецидная карта</b>	<b>FLYINGLEAFE 50</b>
17.1 3 Определение и основные свойства . . . . .	50
17.2 3 Алгоритм построения карты и поисковой структуры . . . . .	51
17.2.1 Алгоритм . . . . .	51
17.2.2 Асимптотика . . . . .	53
<b>18 3 16: Вращающиеся калиперы</b>	<b>VOLHOVM 55</b>
<b>19 2 17: Сумма Минковского</b>	<b>VOLHOVM 57</b>
<b>20 3 18: Вероятностный алгоритм мин.охв.окружности</b>	<b>FLYINGLEAFE 61</b>
<b>21 2 19: Граф видимости и планирование движения</b>	<b>VOLHOVM 63</b>
21.1 2 Точечный объект . . . . .	63
21.1.1 3 Граф видимости . . . . .	63
21.1.2 3 Трапецидная карта, наивное решение . . . . .	64
21.1.3 2 Решение с помощью триангуляции . . . . .	64
21.1.4 2 Слепой жук: точечный объект, нет знания карты . . . . .	65

21.2	<b>3</b> Неточечный объект . . . . .	65
21.2.1	<b>3</b> Задача для круга . . . . .	65
21.2.2	<b>3</b> Задача для полигона без вращения . . . . .	66
21.2.3	<b>3</b> Задача для полигона с вращением . . . . .	66
<b>22</b>	<b>3</b> <b>20:</b> Триангуляция Делоне	<b>VOLNOVM</b> <b>67</b>
22.1	<b>3</b> Оптимальная триангуляция . . . . .	67
22.2	<b>3</b> Триангуляция Делоне . . . . .	68
22.3	<b>3</b> Использование триангуляции Делоне . . . . .	69
<b>23</b>	<b>2</b> <b>21:</b> Делоне: алгоритм + корректность	<b>VOLNOVM</b> <b>70</b>
23.1	<b>2</b> Изменение статической триангуляции . . . . .	70
23.2	<b>2</b> Алгоритм . . . . .	70
23.3	<b>2</b> Асимптотика . . . . .	71
23.4	<b>2</b> Удаление . . . . .	74
<b>24</b>	<b>2</b> <b>22:</b> Диаграмма Вороного	<b>VOLNOVM</b> <b>75</b>
24.1	<b>3</b> Определения и свойства . . . . .	75
24.2	<b>3</b> Двойственность Делоне . . . . .	76
24.2.1	<b>2</b> Построение из триангуляции диаграмму . . . . .	77
24.2.2	<b>2</b> Построение из диаграммы триангуляции . . . . .	77
24.3	<b>2</b> Алгоритм построения и асимптотика . . . . .	78
24.4	<b>2</b> Высшие порядки . . . . .	78
24.4.1	<b>2</b> Алгоритм с удалением точек . . . . .	78
24.4.2	<b>3</b> Алгоритм с пересечениями . . . . .	78
24.5	<b>3</b> Диаграмма минус первого порядка . . . . .	79
<b>25</b>	<b>Бонусные задачи и нетронутые темы</b>	<b>80</b>
25.1	Из множества прямых произвольных восстановить DCEL . . . . .	80
25.2	Поиск касательной точки и многоугольника . . . . .	80
25.3	Масштабирование дорог . . . . .	80
25.4	Звездные многоугольники . . . . .	81
<b>26</b>	<b>Источники</b>	<b>81</b>

# 1 Как пользоваться этим документом

1. Писать билетики параллельно
2. Все формулы оформлять в латехе.
  - <http://www.math.uiuc.edu/~hildebr/tex/course/intro2.html>
  - <https://en.wikibooks.org/wiki/LaTeX/Mathematics>
  - [https://en.wikibooks.org/wiki/LaTeX/Advanced\\_Mathematics](https://en.wikibooks.org/wiki/LaTeX/Advanced_Mathematics)
  - Рекомендуется смотреть конспект Сугака тут.
3. Можно пользоваться (`org-toggle-pretty-entities`) чтобы latex отображался юникодом в emacs'e.
4. **TODO** обозначают степень написанности материала. Положительное значение интерпретируется как процент. X – метка для "тут нифига не ясно и проблемы".
5. Смотреть превью формул с помощью `C-c C-x C-l`.
6. Экспортить с помощью `C-c C-e l` p. Если не работает – сначала в ‘tex‘, а потом уже экспортировать ‘pdflatex’ом. Экспорт игнорирует ошибки, поэтому сделать это руками и пофиксить их – не так плохо.
7. Если что-то не собирается, писать @volhovm.
8. Синтаксис теха смотреть хз где, но есть <http://detexify.kirelabs.org/classify.html> для непонятных символов.
9. Синтаксис орга – это тут:
  - <http://orgmode.org/manual/Emphasis-and-monospace.html>
  - <http://orgmode.org/manual/LaTeX-and-PDF-export.html#LaTeX-and-PDF-export>
  - <http://orgmode.org/tmp/worg/org-tutorials/org-latex-export.html>
  - <http://orgmode.org/manual/Embedded-LaTeX.html#Embedded-LaTeX>
  - Остальное (списки, хедеры, блабла) очевидно.

## 2 Tickets

- 
- 4 1 Принадлежность точки выпуклому и невыпуклому многоугольникам
- 21 1 Триангуляция Делоне.  
Алгоритм и доказательство его корректности.
- 5 2 Статические выпуклые оболочки на плоскости.  
Джарвис, Грэм, Эндрю, Чен, QuickHull. Оболочка многоугольника, оболочка полилинии.
- 20 2 Триангуляция Делоне.  
- существование;  
- приводимость любой триангуляции флипами к ТД;  
- эквивалентность критерия Делоне для треугольников критерию для ребер.
- 8 3 Триангуляция многоугольника.  
Существование, ушная триангуляция.
- 17 3 Сумма Минковского (определение, вычисление)
- 12 4 ППЛГ и РСДС (PSLG и DCEL): определение, построение РСДС множества прямых
- 15 4 Трапецидная карта.
- 10 5 Пересечение полуплоскостей, связь с выпуклыми оболочками
- 13 5 Пересечение многоугольников (PSLG overlaying)
- 7 6 Выпуклая оболочка в n-мерном пространстве. Quick-hull и вероятностный алгоритм.
- 11 6 Пересечение множества отрезков.
- 9 7 Триангуляция многоугольника заметающей прямой
- 14 7 Локализация в ППЛГ.  
- методом полос (персистентные деревья);  
- Киркпатрик.
- 6 8 Динамическая выпуклая оболочка (достаточно  $\log^2$  на добавление/удаление)
- 19 8 Граф видимости и планирование движения.  
- построение графа видимости заметающим лучом;  
- сокращение графа видимости;  
- построение навигационного графа на трапецидной карте;  
- планирование маршрута невыпуклого тела с вращением (без суммы Минковского).
- 3 9 Пересечение отрезков и поворот: определение, свойства, вычисление
- 22 9 Диаграмма Вороного.  
- определение и свойства;  
- диаграмма Вороного высших порядков, построение;  
- связь с подразбиением Делоне (ближайший и дальнейший);  
- алгоритм построения ДВ.
- 1 10 Skip quadtree: определение, время работы
- 18 10 Минимальная охватывающая окружность множества точек. Вероятностный алгоритм.
- 2 11 Пересечение прямоугольника с множеством прямоугольников и непересекающихся отрезков:  
- range tree + fractional cascading;  
- interval tree;  
- segment tree;  
- priority search tree;  
- k-d tree. van Kreveld, de Berg, Overmars, Cheong
- 16 11 Диаметр множества точек (вращающиеся калиперы)
-

### 3 3 1: Skip quadtree

AVBELYY

#### 3.1 3 Сжатое квадродерево

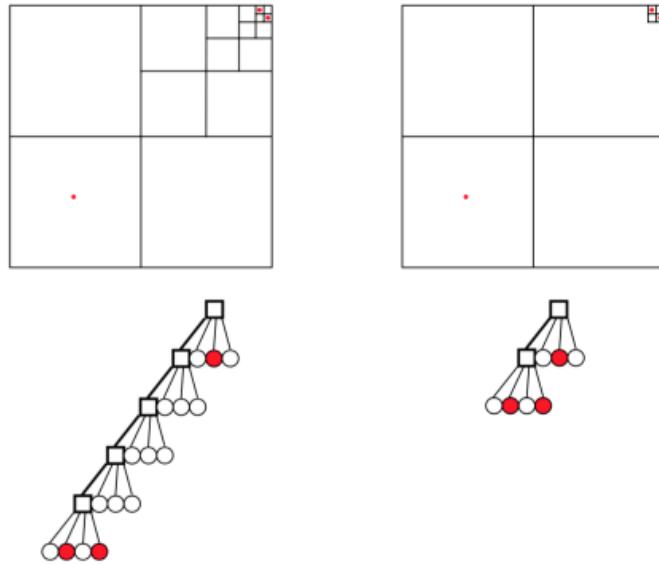


Рис. 1: Обычное и сжатое квадродеревья, содержащие по 3 точки.

**Определение** (Квадродерево). Дерево, каждая внутренняя вершина которого содержит 4 ребенка.

**Определение** (Интересный квадрат). Квадрат, в котором содержится хотя бы одна точка из  $P$ .

**Определение** (Сжатое квадродерево). Квадродерево, внутренним вершинам которого соответствуют только интересные квадраты.

**Построение сжатого квадродерева** происходит по обычному квадродереву следующим образом: у внутренней вершины заводим по 4 указателя для 4 четвертей. Если в четверти 2 и более точки  $P$  - указатель ссылается на наибольший интересный квадрат этих точек, если одна - ссылается на неё саму, если 0 - указатель NULL.

**Время работы операций в сжатом квадродереве** –  $O(n)$  на локализацию, вставку и удаление.

#### 3.2 3 Randomized skip quadtree

**Определение** (Randomized skip quadtree). Последовательность сжатых квадродеревьев над подмножествами точек  $P : P_0, P_1, \dots, P_k$ , где  $P_0 = P$  – исходное множество точек,  $P_i \in P_{i-1}$  и каждый элемент  $P_{i-1}$  входит в  $P_i$  с вероятностью  $p \in (0, 1)$ . Skip quadtree – это последовательность уровней  $Q_i$ , где  $Q_i$  – сжатое квадродерево над точками  $P_i$ .

**Время работы операций в skip quadtree** –  $O(\log n)$ . Сначала опишем, как проходят операции, а потом докажем их время работы.

- **Общая операция подъем**

По вершине с уровня  $i$  нужно получить эту же вершину на уровне  $i - 1$ . За  $O(1)$ . Как сделать? Проще всего как в skip list: "пропустить" ссылками на вершину уровня выше каждую внутреннюю вершину каждого квадродерева.

- **Локализация**

Локализуемся на уровне  $k$ , далее сделаем подъем, окажемся в квадродереве уровня ниже и локализуемся в нем, но уже не от корня, а с того квадрата, который нашли на предыдущем уровне. Повторим это  $k$  раз. В результате локализуемся на нулевом уровне.

- **Вставка**

Локализуемся на всех уровнях, запоминая ссылки. Сделаем вставку в квадродерево нулевого уровня, далее с вероятностью  $p$  сделаем вставку на 1 уровне и так далее до первого недобавления. Количество уровней при этом увеличится максимум на 1 (с вероятностью  $p^k$ ).

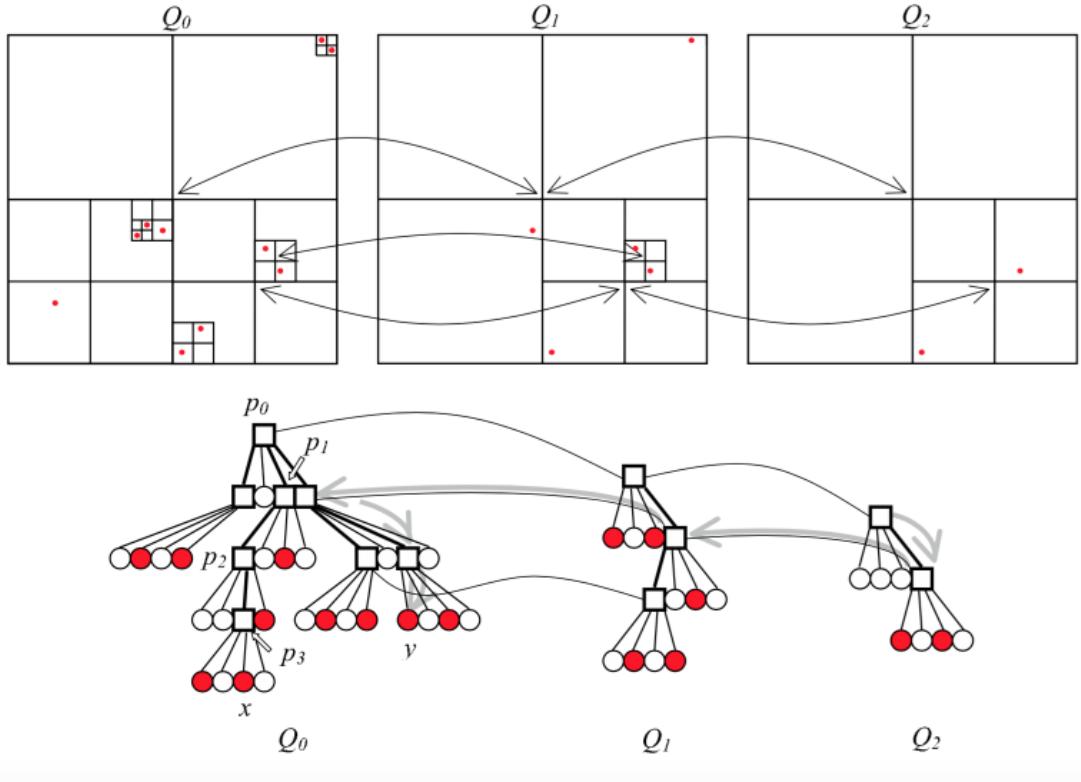


Рис. 2: Skip quadtree, состоящее из трёх уровней –  $Q_0$ ,  $Q_1$  и  $Q_2$ .

- Удаление

Локализуемся на всех уровнях, удалим квадрат везде и обновим ссылки. Если уровень стал пустым – удалим его.

**Лемма 3.1** (О количестве шагов на одном уровне). *На каждом уровне в среднем совершается  $O(1)$  шагов для поиска точки  $x$ .*

*Доказательство.* Пусть в  $Q_i$  (то есть на  $i$ -ом уровне) поиск точки  $x$ , начинающийся с корня, проходит по квадратам  $p_0, p_1, \dots, p_m$ . Пусть случайная величина  $j$  – количество шагов поиска в  $Q_i$ , тогда  $p_{m-j}$  – последний квадрат из  $p_0, p_1, \dots, p_m$ , являющийся интересным в  $Q_{i+1}$ .

Оценим вероятность того, что делается  $j$  шагов. Забьём на случай  $j = 0$ , так как он не важен при расчёте мат. ожидания. На пути  $p_{m-j+1}, \dots, p_m$  будет хотя бы  $j + 1$  непустых четвертинок. У первого квадрата на этом пути есть хотя бы 2 непустые четвертинки, одна из них – следующий квадрат на пути, в котором тоже хотя бы 2 непустые четвертинки, и так далее. В последнем квадрате просто хотя бы 2 непустые четвертинки. Чтобы  $p_{m-j}$  был последним из  $p_0, p_1, \dots, p_m$  интересным квадратом в  $Q_{i+1}$  необходимо, чтобы среди этих как минимум  $j + 1$  непустых четвертинок только одна (вероятность этого назовём  $Pr_1$ ) или ноль (вероятность этого назовём  $Pr_0$ ) были непустыми в  $Q_{i+1}$ . Иначе, если будет хотя бы пара непустых четвертинок, то их наименьший общий предок в дереве будет интересным квадратом и будет находиться глубже  $p_{m-j}$ . Таким образом, искомая вероятность не превосходит  $Pr_0 + Pr_1$ .

Пусть  $q = 1 - p$ .

$Pr_0 \leq q^{(j+1)}$ , потому что это в сущности вероятность того, что ни одна точка из как минимум  $j + 1$  непустых четвертинок не попала на уровень выше.

$Pr_1 \leq (j + 1) \cdot pq^j$ , потому что это в сущности вероятность того, что ровно одна точка из как минимум  $j + 1$  непустых четвертинок попала на уровень выше.

$$E(j) = \sum_{j=1}^m j \cdot Pr(j) \leq \sum_{j=1}^m j \cdot (q^{(j+1)} + (j + 1) \cdot pq^j) \leq \sum_{j=1}^{\infty} j \cdot (q^{(j+1)} + (j + 1) \cdot pq^j)$$

Это почти геометрическая прогрессия, только на полином домножили, определяется всё экспоненциальным множителем, так что это  $O(1)$ . Можно совсем строго оценить, но и так понятно, что ряд сходится, а сойтись он может только к константе.  $\square$

**Лемма 3.2** (О количестве уровней). *Математическое ожидание количества уровней составляет  $O(\log n)$ .*

*Доказательство.*

- Для оценки мат. ожидания посчитаем вероятность того, что количество уровней  $h$  равно  $k$ .

$$p(h = k) = 1 - p(h > k) - p(h < k).$$

$p(h < k) = (1 - p^k)^n$ , потому что вероятность того, что точка дойдёт до уровня  $k$ , равна  $p^k$ .

$p(h > k) = (1 - (1 - p^{k+1})^n)$ , потому что вероятность того, что точка не дойдёт до уровня  $k + 1$ , равна  $1 - p^{k+1}$ .

$$p(h = k) = 1 - p(h > k) - p(h < k) = 1 - (1 - (1 - p^{k+1})^n) - (1 - p^k)^n = (1 - p^{k+1})^n - (1 - p^k)^n \leq 1 - (1 - p^k)^n \leq np^k$$

- Теперь посчитаем мат. ожидание количества уровней:

$$E(h) = \sum_{k=1}^{\infty} k \cdot p(h = k) = p(1) \cdot 1 + \dots + p(\log_{1/p} n) \cdot \log_{1/p} n + \sum_{k=\log_{1/p} n+1}^{\infty} k \cdot p(k)$$

- Оценим первую сумму:

$$p(1) \cdot 1 + \dots + p(\log_{1/p} n) \cdot \log_{1/p} n \leq p(1) \cdot \log_{1/p} n + \dots + p(\log_{1/p} n) \cdot \log_{1/p} n = O(\log(n))$$

, поскольку сумма этих вероятностей не превосходит единицы.

- Оценим вторую сумму:

$$\sum_{k=\log_{1/p} n+1}^{\infty} k \cdot p(k) \leq \sum_{k=\log_{1/p} n}^{\infty} k \cdot np^k = n \cdot \sum_{k=\log_{1/p} n}^{\infty} k \cdot p^k$$

- Рассмотрим эту сумму:

$$\begin{aligned} \sum_{k=\log_{1/p} n}^{\infty} k \cdot p^k &= p^{\log_{1/p} n} \cdot \sum_{k=0}^{\infty} (k + \log_{1/p} n) \cdot p^k \\ &= p^{\log_{1/p} n} \cdot \left( \sum_{k=0}^{\infty} kp^k + \log_{1/p} n \cdot \sum_{k=0}^{\infty} p^k \right) \\ &= p^{\log_{1/p} n} \cdot (O(1) + \log_{1/p} n \cdot O(1)) \\ &= 1/n \cdot O(\log n) \end{aligned}$$

- Суммируя всё вышесказанное, получаем, что  $O(\log(n))$ .

□

**Теорема 3.1** (О времени работы операций в skip quadtree).

*Локализация, вставка и удаление работают в среднем за  $O(\log n)$ .*

*Доказательство.* Следует из двух предыдущих лемм. Будем подниматься через  $O(\log n)$  уровней до нулевого, делая на каждом уровне  $O(1)$  шагов. □

## 4 3 2: Пересечение прямоугольника с множеством пр-уг/отр FLYINGLEAFE

### 4.1 3 Пересечение прямоугольника запроса $q$ с множеством прямоугольников $P$

Задача: существует множество прямоугольников  $P$ . Стороны прямоугольников параллельны осям координат. Задается прямоугольник запроса  $q$ , такой же. Нужно быстро определить множество прямоугольников, пересекающихся с  $q$ .

Формально: нужно найти множество  $S = \{p \in P : p \cap q \neq \emptyset\}$

Разобьем это множество на три множества-подзадачи:

$$\begin{aligned} S &= A \cup B \cup C \\ A &= \{p \in P : \exists i : p_i \in \text{corners}(p), p_i \in q\} \\ B &= \{p \in P : \exists i : s_i \in \text{sides}(p), s_i \cap q \neq \emptyset\} \\ C &= \{p \in P : \exists i : p_i \in \text{corners}(q), p_i \in p\} \end{aligned}$$

Иначе говоря,  $A$  – это случай, когда  $p$  целиком лежит в  $q$ ,  $B$  – когда  $p$  и  $q$  пересекаются,  $C$  – когда  $q$  целиком лежит в  $p$ . Разберем все три случая отдельно.

#### 4.1.1 3 Нахождение множества точек, попадающих в прямоугольник запроса.

Для начала быстро рассмотрим одномерный случай. Быстро выдавать множество точек, попадающих в отрезок, можно с помощью сбалансированного дерева поиска. (**Примечание:** можно и с помощью отсортированного массива и бинпоиска, но в такую структуру данных нельзя эффективно вставить новую точку).

Как это делать – очевидно: идем вглубь дерева, пока не встретим узел, разделяющий концы отрезка. После этого ищем каждый конец отрезка в отдельности и добавляем к ответу все поддеревья, лежащие справа (для левого конца) и слева (для правого конца) от пути.

Такое дерево можно построить за  $O(n \log n)$ , она будет занимать  $O(n)$  памяти, запрос будет обработан за  $O(\log n + k)$ , где  $k$  – величина ответа. Как расширить эту структуру на двумерный случай и добиться сопоставимых результатов?

##### 1. Способ 1. Range trees

Рассмотрим обработку двумерного запроса как обработку 2 одномерных запросов по отдельности: по  $x$ -координате и по  $y$ -координате. То есть, сначала мы отсеиваем все точки, попадающие в запрос по  $x$ , а потом из них выбираем точки, попадающие в этот запрос по  $y$ .

Мы делаем это, на самом деле, очень просто: строим для всего множества точек бинарное дерево поиска по  $x$ , а в каждом узле этого дерева дополнительно строим дерево поиска по  $y$  для соответствующего поддерева.

**Лемма 4.1.** Такая структура данных, несмотря на кажущуюся громоздкость, занимает  $O(n \log n)$  памяти

*Доказательство.* Рассмотрим некоторую точку  $p$ . В дереве первого уровня путь от корня до нее занимает  $O(\log n)$  узлов. Значит, она содержится в каждом дереве 2-го уровня, встретившемся на пути, но не встречается больше ни в каких деревьях 2-го уровня. Таким образом, количество копий каждой точки во всей структуре данных оценивается в  $O(\log n)$ . Всего точек  $n$ , значит, структура занимает  $O(n \log n)$  памяти.  $\square$

**Лемма 4.2.** Такую структуру данных можно построить за  $O(n \log n)$

*Доказательство.* Если строить каждое дерево второго уровня вступую за  $O(n \log n)$ , то это, конечно, будет долго. Однако, если мы сначала отсортируем список вершин по  $y$ , то деревья второго уровня можно будет строить за  $O(n)$  снизу вверх. Таким образом, каждый узел основного дерева будет строиться за  $O(m)$ , где  $m$  – количество точек в поддереве узла. Значит, узел строится за такое время, сколько памяти занимает, а вся структура занимает  $O(n \log n)$  памяти. Поэтому за столько же по времени произойдет построение дерева.  $\square$

**Лемма 4.3.** Двумерный запрос в таком дереве займет  $O(\log^2 n + k)$  времени.

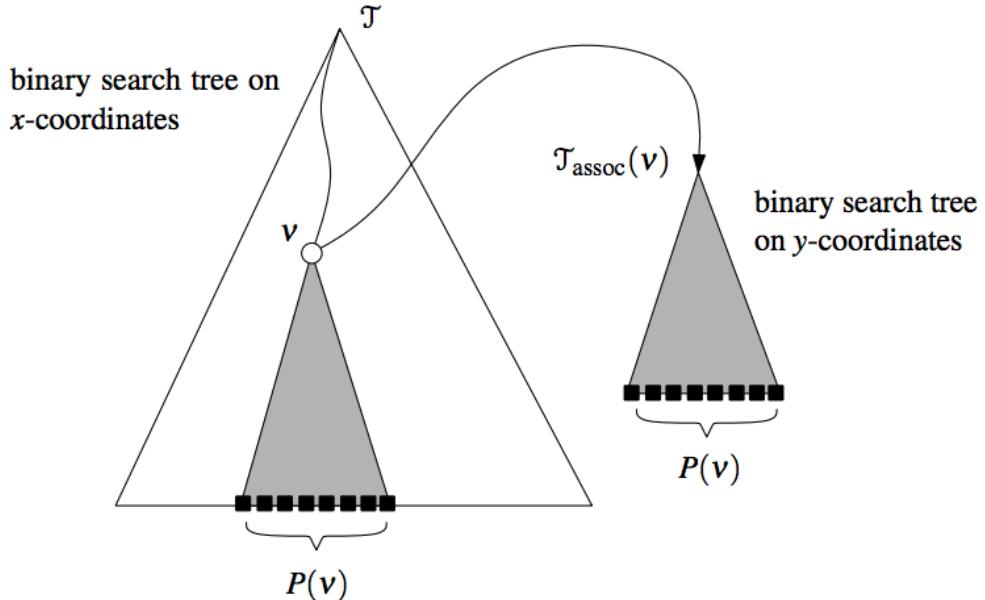


Рис. 3: Иллюстрация к Range-tree

*Доказательство.* Запрос в дереве 1 уровня пройдет за  $O(\log n)$ . При этом во время выполнения запроса вызовутся запросы по  $y$  для всех деревьев поиска 2 уровня, встретившихся по пути - таких  $O(\log n)$ . В дереве поиска 2 уровня ситуация аналогична одномерной + нужно время на вывод результата. Итого:  $O(\log^2 n + k)$   $\square$

**Замечание** Легко видеть, что Range tree легко обобщается на высшие размерности - достаточно просто понапихать деревьев низших уровней. В таком случае на построение и память уйдет  $O(n \log^d n)$ , а на запрос -  $O(\log^d + k)$ . Доказательство этих фактов оставим читателю (это легко, мне лень, Ковалев не спросит).

#### Fractional cascading

Время запроса  $O(n \log^2 n + k)$  - казалось бы, не так уж плохо, но на серьезных  $n$  этот квадрат у логарифма даст серьезный прирост во времени. Можно ли от него избавиться? Оказывается, да! Для этого существует техника, называемая fractional cascading.

**Идея:** Мы делаем  $O(\log n)$  запросов по деревьям второго уровня для одного и того же рендука по  $y$  - координате. Может быть, мы можем как-то использовать результаты запросов в старших поддеревьях для младших?

Проиллюстрируем идею fractional cascading на более простом примере. Предположим, у нас есть 2 массива объектов -  $A_1, A_2$  - отсортированных по ключу, причем  $A_2 \subset A_1$ . Приходит запрос - выдать все объекты из обоих массивов с ключом, лежащим в отрезке  $(l, r)$ . Можно вступую сделать бинпоиск на обоих массивах. А можно делать бинпоиск только в большем массиве  $A_1$ , а во втором массиве выдать ответ сразу, воспользовавшись ссылками, ведущими из каждого объекта в  $A_1$  в первый объект в  $A_2$ , больший или равный данному. (см. иллюстрацию)

Так вот! Теперь заметим, что каждое дерево 2-го уровня в Range-tree содержится в своем предке. А давайте тогда не будем делать деревья 2 уровня, а вместо них сделаем вот такие каскадные массивы. Причем в каждом элементе такого массива есть 2 ссылки - на upper-bound в левом и правом сыне.

Такая структура данных называется layered range-tree. Она занимает столько же памяти, сколько и обычный range-tree (очевидно). Докажем пару других фактов.

**Лемма 4.4.** *Layered range-tree строится за  $O(n \log n)$*

*Доказательство.* Корневой массив строим, просто отсортировав точки по  $y$  за  $O(n \log n)$ . Покажем способ построить из массива длины  $n$  два дочерних массива с каскадными ссылками за  $O(n)$ .

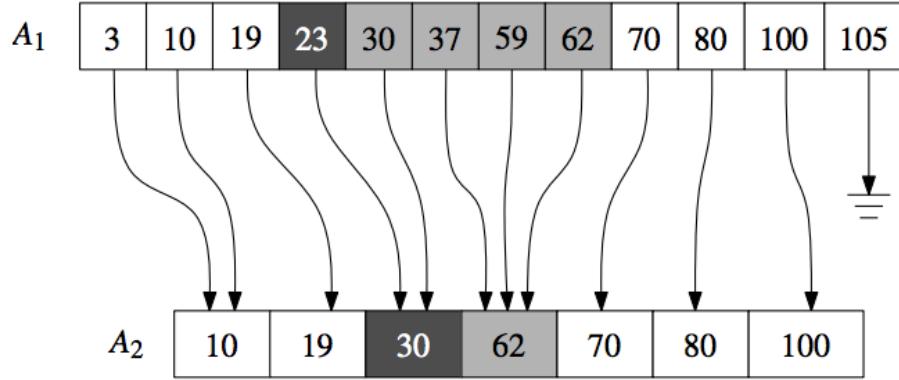


Рис. 4: Иллюстрация к Fractional Cascading

Разделить отсортированный массив по некоему  $x$  на два можно за  $O(n)$  легко - идем по элементам по порядку и в зависимости от их  $x$  координаты кладем их в конец левого или правого массивов. Проставить ссылки же можно с помощью трех указателей.

Ставим указатели  $i, j, k$  в начало корневого массива, левого и правого сына соответственно. Перебираем  $A[i]$ , смотрим в  $A_l[j]$  и увеличиваем  $j$  до тех пор, пока  $A_l[j] < A[i]$ . Аналогично делаем с  $A_r$ . Проставляем в  $A[i]$  ссылки на  $A_l[j]$  и  $A_r[k]$  и увеличиваем  $i$ . Эта процедура, очевидно, занимает линейное время.  $\square$

**Лемма 4.5.** Запрос в layered range-tree занимает  $O(\log n + k)$  времени

**Доказательство.** Запрос по 1 уровню занимает  $O(\log n)$ , на выходе мы получаем  $O(\log n)$  каскадных массивчиков, в которых нужно провести запрос по  $y$ . Но мы за  $O(\log n)$  делаем этот запрос только в корне, и за этот же  $O(\log n)$  спускаемся вниз по каскадным ссылкам, зацепляя все найденные массивы. Итого -  $O(\log n + k)$   $\square$

**Замечание** Fractional cascading, вообще говоря, снижает время запроса по range-tree в d-мерном пространстве до  $O(\log^{d-1} n + k)$  (доказывать не буду, я заебался уже)

## 2. Способ 2. k-d trees

Идея: давайте поиск по  $y$  - координате будем делать не после того, как завершили поиск по  $x$  - координате, а как бы вперемешку. Строим "слоеное"дерево: сначала поделим множество точек примерно пополам по  $x$ , потом по  $y$ , потом снова по  $x$  и так далее.

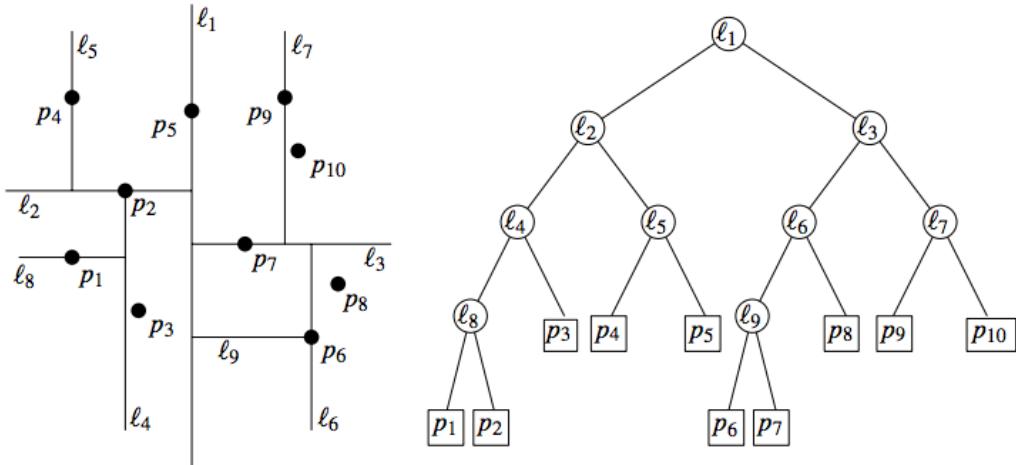


Рис. 5: K-d tree

**Лемма 4.6.** K-d tree построится за  $O(n \log n)$  и будет занимать  $O(n)$  места.

*Доказательство.* Дерево строится рекурсивно, множество, по которому строятся правое и левое поддерево, примерно одинаковы - для этого ищется медиана. Таким образом, глубина дерева -  $O(\log n)$ . Медиану можно найти за  $O(n)$  (сложно, но можно, проще посортировать все заранее), разделить множество по медиане - тоже за  $O(n)$ . Итого, каждый вызов без учета рекурсии занимает  $O(n)$ , всего вызовов  $O(\log n)$ , итого  $O(n \log n)$

Что касается места: k-d tree имеет вид обычного сбалансированного бинарного дерева с  $n$  листьями, в котором каждый узел занимает  $O(1)$  памяти. Значит, всего памяти  $O(n)$   $\square$

Пусть нам теперь нам прилетел прямоугольный запрос. Рекурсивно спускаемся с ним по дереву, "нарезая" его медианами. Если в запрос какая-то область попала целиком - возвращаем все поддерево.

```

Function QueryKDT( $v, R$ )
Data: Корень дерева  $v$ , и прямоугольник  $R$ 
Result: Множество точек, попавших в прямоугольник
if  $v$  – лист then
| if  $point(v) \in R$  then
| | report  $point(v)$ 
| end
else
| if область узла  $v$  полностью содержитя в  $R$  then
| | report-all( $v$ )
| else
| | if область  $v_l$  пересекается с  $R$  then
| | | QueryKDT( $v_l, R$ )
| | end
| | if область  $v_r$  пересекается с  $R$  then
| | | QueryKDT( $v_r, R$ )
| | end
| end
end

```

**Алгоритм 1:** Алгоритм построения PST

**Лемма 4.7.** Запрос в k-d tree занимает  $O(\sqrt{n} + k)$  времени.

*Доказательство.* Без учета рекурсивных вызовов и вывода ответа, QueryKDT выполняется за  $O(1)$ . С выводом ответа все понятно, но сколько будет рекурсивных вызовов? Их будет столько, сколько на пути попадется областей, пересекаемых границей прямоугольника. Границы прямоугольника могут быть сколь угодно длинными. Поэтому верхней оценкой на количество областей, пересекаемый 1 гранью многоугольника, можно считать количество областей, пересекаемых случайной вертикальной прямой (с горизонтальной гранью – симметрично).

Обозначим количество таких областей в k-d tree для  $n$  точек как  $Q(n)$ , пересекающую прямую –  $l$ . Пусть разделяющая прямая в корне – вертикальная. Тогда прямая  $l$  пересечет область в корне и ровно одного из сыновей - 2 пересечения. Но в сыне разделяющая прямая будет горизонтальной! То есть, прямая точно пересечет обоих внуков. А вот во внуках ситуация уже будет аналогична корню. В каждом внуке примерно  $n/4$  точек. Теперь мы можем записать рекуррентную формулу:

$$Q(n) = \begin{cases} O(1), & \text{if } n = 1 \\ 2 + 2Q(n/4), & \text{otherwise} \end{cases}$$

Тут надо сделать какой-то шмяк-шмяк и сказать, что эта рекурсия имеет решение  $Q(n) = O(\sqrt{n})$ . На самом деле, так и есть.  $\square$

Итого: k-d tree отвечает на запросы явно помедленнее range tree (даже не layered), однако занимает меньше памяти и явно проще в исполнении.

#### 4.1.2 3 Нахождение множества отрезков, пересекающих прямоугольник запроса (но не лежащих концом внутри)

Пусть есть множество отрезков  $S$ , таких, что каждый отрезок параллелен либо оси  $x$ , либо оси  $y$ . Дан прямоугольник запроса  $q$ . Нужно найти все отрезки, пересекающие  $q$ . Утверждается, что не существует

отрезков, лежащих в нем концом. (типа, все такие отрезки мы можем найти, рассмотрев только их концы в предыдущей задаче).

Пусть найден такой отрезок  $s$ . Тогда, если он горизонтальный, он пересекает обе вертикальные грани  $q$ , а если вертикальный - обе горизонтальные. Значит, достаточно найти отрезки, что пересекают левую или нижнюю грани прямоугольника.

Следовательно, задача свелась к нахождению множества горизонтальных отрезков, пересекающих с данным вертикальным отрезком (с горизонтальным случай симметричный).

Сначала решим более простую задачу: найдем все горизонтальные отрезки, пересекающие вертикальную прямую с координатой  $q_x$ .

### 1. Interval tree

Эта задача, на самом деле, одномерная,  $y$  - координаты не важны. Давайте попробуем построить эдакое бинарное дерево на отрезках. Найдем медиану  $x_{mid}$  всех середин отрезков и разобьем отрезки на 3 множества,  $I_{mid}, I_l, I_r$  - отрезки, пересекающие медиану, и лежащие целиком слева и справа соответственно. Для  $I_l$  и  $I_r$  рекурсивно построим поддеревья. А вот что делать с  $I_{mid}$ ?

У  $I_{mid}$  есть одно хорошее свойство - все отрезки оттуда пересекают  $x_{mid}$ . Это значит, что если  $q_x > x_{mid}$ , то  $q_x \in [l, r] \in I_{mid}$  iff  $q_x \leq r$ . Слева ситуация симметричная.

Тогда, чтобы искать все отрезки из  $I_{mid}$ , входящие в ответ, за  $O(k_\nu)$  (где  $k_\nu$  - это количество таких отрезков), достаточно хранить  $I_{mid}$  в виде двух отсортированных списков. Первый список отсортирован по убыванию координат правого конца отрезка, левый - по возрастанию координат левого конца. Тогда, если  $q_x \geq x_{mid}$ , мы идем по правому списку до тех пор, пока координата очередного конца не станет меньше  $q_x$ . В обратном случае поступаем симметрично с левым списком.

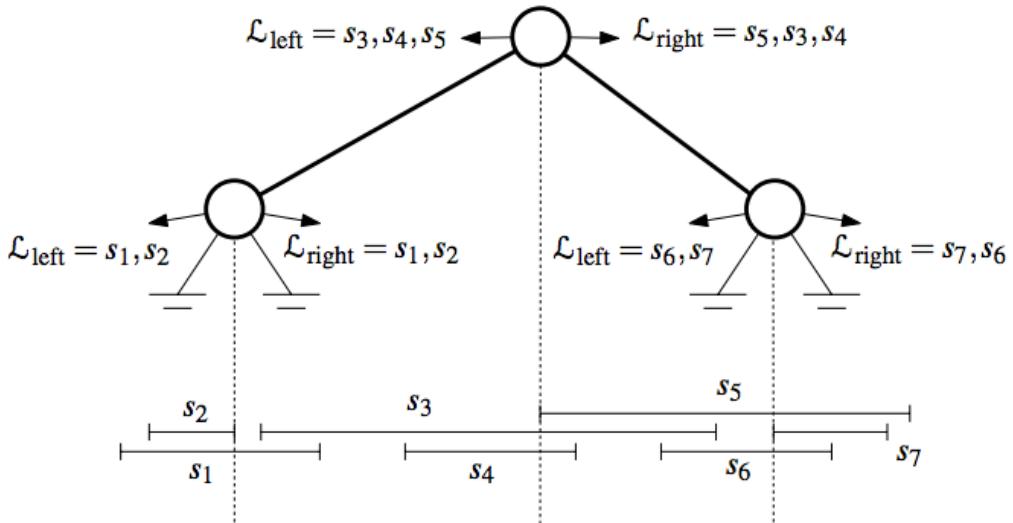


Рис. 6: Устройство interval tree

**Лемма 4.8.** *Interval tree имеет глубину  $O(\log n)$  и размер  $O(n)$*

*Доказательство.* Глубина: медиана на то и медиана, чтобы делить отрезки примерно пополам, поэтому дерево выходит сбалансированным. Размер: каждый отрезок входит только в один из  $I_{mid}$ , и в этом  $I_{mid}$  он хранится дважды - в левом и правом списках. Итого  $O(n)$  места.  $\square$

**Лемма 4.9.** *Interval tree строится за  $O(n \log n)$*

*Доказательство.* Определим процедуру  $\text{buildTree}(I)$  так:

- Если  $I = \emptyset$ , возвращаем пустой лист, иначе
- Найдем медиану всех отрезков в  $I$  (за  $O(n)$ )
- Разобьем  $I$  на  $I_{mid}, I_l, I_r$  по медиане (спон-спон-фигакс за  $O(n)$ )
- Посортируем 2 раза  $I_{mid}$  по координатам концов и сформируем списки  $L_l, L_r$  (за  $O(n_{mid} \log n_{mid})$ )

(e)  $\text{buildTree}(I_l)$ ,  $\text{buildTree}(I_r)$

Всего без учета рекурсивных вызовов -  $O(n + n_{mid} \log n_{mid})$

Рекурсивных вызовов будет  $O(\log n)$ , поэтому с их учетом  $O(n)$  составляющая даст  $O(n \log n)$ . Суммарное время на сортировку всех списков также  $O(n \log n)$ , так как их суммарная длина  $O(n)$ .  $\square$

**Лемма 4.10.** Запрос в interval tree занимает  $O(\log n + k)$  времени.

*Доказательство.* В каждом узле мы тратим  $O(k_\nu)$  на вывод всех подходящих отрезков в нем и делаем рекурсивный вызов. Так как рекурсивных вызовов будет  $O(\log n)$ , суммарное время работы  $O(\log n) + \sum O(k_\nu) = O(\log n + k)$   $\square$

Мы научились находить все горизонтальные отрезки, пересекающие прямую запроса. Но нужно нам отрезок, то есть нам надо как-то включить  $y$  - координаты в игру. Внезапно мы можем это сделать достаточно просто - давайте в нашем interval tree  $I_{mid}$  хранить не как два отсортированных списка, а как range tree! В этом range tree мы будем делать запросы по бесконечному с одной стороны прямоугольнику:  $(-\infty, q_x] \times [q_y, q'_y]$ , если  $q_x < x_{mid}$ , и  $[q_x, \infty) \times [q_y, q'_y]$  иначе.

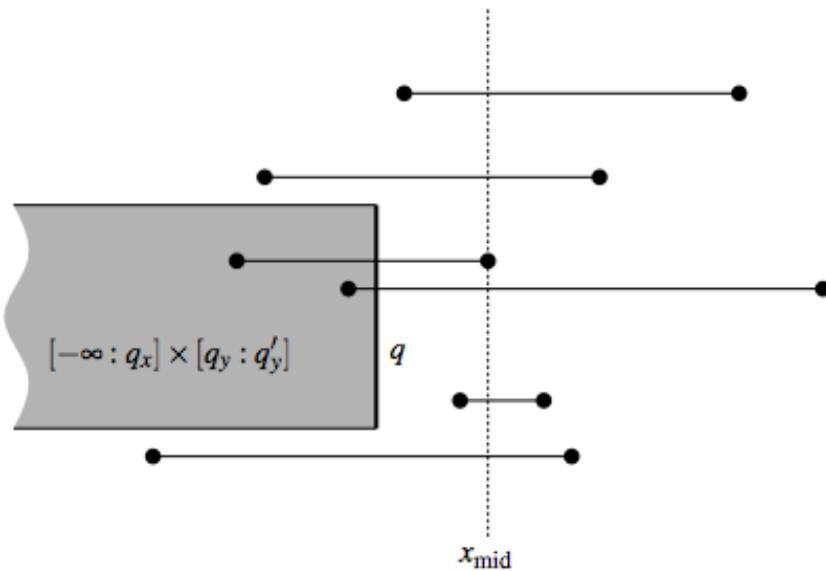


Рис. 7: Вид запроса к range tree

**Лемма 4.11.** Interval tree с range tree вместо списков будет занимать  $O(n \log n)$  памяти, построится за  $O(n \log n)$ , и будет отвечать на запрос за  $O(\log^2 n + k)$

*Доказательство.* Память: каждый  $I_{mid}$  занимает  $O(n_{mid} \log n_{mid})$  памяти,  $\sum n_{mid} = n$ , значит,  $\sum |I_{mid}| = O(n \log n)$ .

Время построения: range tree строится за  $O(n \log n)$ , как и 2 отсортированных списка, поэтому предыдущая оценка работает.

Время запроса: запрос в range tree (с fractional cascading) занимает  $O(\log n + k)$ , всего таких range tree  $O(\log n)$ , поэтому суммарное время запроса  $O(\log^2 n + k)$   $\square$

## 2. Priority search trees

Использовать range trees как подструктуру для interval trees - на самом деле громоздко и overkill. Сделаем структуру, которая специально заточена на обработку запросов в виде бесконечных стаканов.

Priority search tree - это такой heap, очень похожий на декартку. Точки в нем упорядочены по приоритету –  $x$  - координате. Однако оно не является корректным деревом бинпоиска по  $y$  - координате. Декартка нам не подходит, потому что в случае неравномерного разброса точек декартка получится очень несбалансированной. А в priority search tree мы при построении сыновей

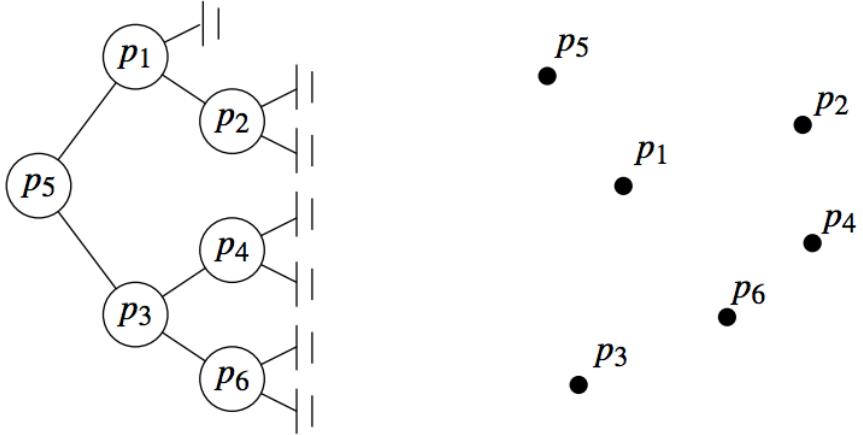


Рис. 8: Priority-search tree

для текущего узла мы делим поддерево по  $y$  - координате, поэтому дерево всегда получается сбалансированным.

Приведем алгоритм построения priority-search tree.

**Function** BuildPST( $P$ )

<b>Data:</b> Набор точек $\{p_i\}$ , отсортированный по $x$ - координате <b>Result:</b> Корректное PST на точках $V_p \leftarrow p_1;$ $y_{mid} \leftarrow$ медиана $\{p_2, \dots, p_n\};$ $P_l \leftarrow \{p \in P \setminus \{p_1\} \mid p_y < y_{mid}\};$ $P_r \leftarrow \{p \in P \setminus \{p_1\} \mid p_y \geq y_{mid}\};$ $V_l \leftarrow \text{BuildPST}(P_l);$ $V_r \leftarrow \text{BuildPST}(P_r);$ return $V;$
---

**Алгоритм 2:** Алгоритм построения PST

**Лемма 4.12.** Этот алгоритм работает за  $O(n \log n)$

*Доказательство.* Сортировка по иксу работает  $O(n \log n)$ . Шаг алгоритма без учета рекурсивных вызовов работает за  $O(n)$ , а глубина рекурсии  $O(\log n)$ . Итого  $O(n \log n)$   $\square$

**Замечание:** Если отсортировать массив не по  $x$ , а по  $y$ , то можно будет построить PST как декартку снизу вверх за  $O(n)$ .

Очевидно, что PST занимает  $O(n)$  памяти.

В таком дереве мы можем выдать все точки, лежащие левее некоего  $q_x$ , за  $O(k)$ . Действительно - просто спускаемся вглубь и выдаем все подходящие точки, пока они подходят. Если не подходит какая-то точка - не подходит и все ее поддерево.

Зато теперь мы понимаем, как отвечать на "стаканные" запросы. Приведем алгоритм:

```

Function QueryPST( $v, (-\infty, q_x] \times [q_y, q'_y]$ )
Data: Корень PST и область запроса ('стакан')
Result: Набор всех точек из дерева, удовлетворяющих запросу
Идем вниз дерева, ищем  $q_y$  и  $q'_y$ . Обозначим как  $v_{split}$  узел, на котором пути поиска разделяются;
for  $u$  – узел на пути поиска  $q_y$  или  $q'_y$  do
| if  $u_p \in (-\infty, q_x] \times [q_y, q'_y]$  then
| | report  $u_p$ 
| end
| end
| for  $u$  – узел на пути поиска  $q_y$  в левом поддереве  $v_{split}$  do
| | if путь поиска поворачивает влево then
| | | report-all( $u_r, q_x$ )
| | end
| end
| for  $u$  – узел на пути поиска  $q'_y$  в правом поддереве  $v_{split}$  do
| | if путь поиска поворачивает вправо then
| | | report-all( $u_l, q_x$ )
| | end
| end
end

```

**Алгоритм 3:** Алгоритм запроса в PST

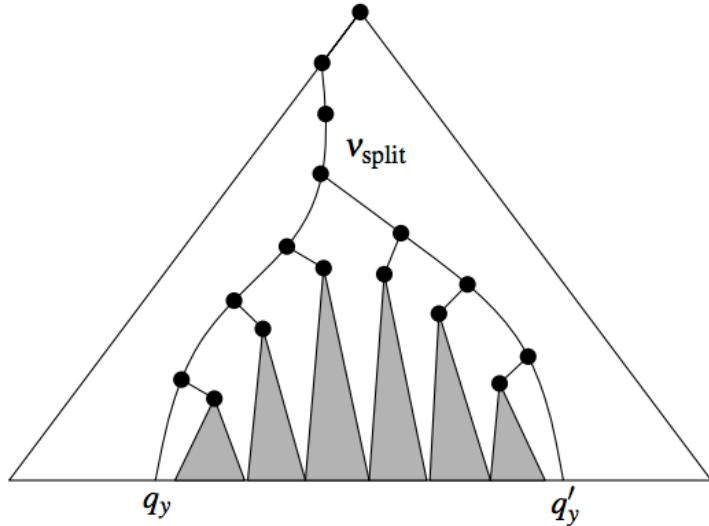


Рис. 9: Иллюстрация к алгоритму запроса в PST

**Лемма 4.13.** Запрос в PST работает за  $O(\log n + k)$

*Доказательство.* Длина пути поиска в дереве –  $O(\log n)$ , а кроме спуска по этому пути мы только выводим вершины, попавшие в ответ. Итого  $O(\log n + k)$   $\square$

**Лемма 4.14.** Interval tree с двумя PST вместо списков будет занимать  $O(n)$  памяти, построится за  $O(n \log n)$ , и будет отвечать на запрос за  $O(\log^2 n + k)$

*Доказательство.* PST занимает столько же места ( $O(n)$ ) и строится такое же время, как списки ( $O(n \log n)$ ), поэтому оценки на память и препроцессинг такие же. Время ответа на запрос такое же, как у layered range tree, поэтому и оценка времени ответа аналогичная ( $O(\log^2 n + k)$ )  $\square$

#### 4.1.3 3 Нахождение всех прямоугольников, в которые попадает точка.

Каждый axis-aligned прямоугольник можно задать 2 отрезками - его проекциями на оси  $x$  и  $y$ . Задача проверки попадания точки в прямоугольник сводится к проверке того, что проекции точки попадают

в соответствующие проекции прямоугольника.

Двухуровневый interval tree не подойдет, так как при каждом запросе нам нужно будет выделить из  $I_{mid}$  подмножество подходящих отрезков, а выделить из interval tree второго уровня это множество не представляется возможным.

Мы возьмем другую структуру данных для локализации по  $x$  - проекции: segment tree из следующего подраздела. В каждом узле которого мы будем хранить, конечно, не просто список отрезков, а interval tree на этих отрезках (но уже по  $y$  - проекции). Так как interval tree занимает  $O(n)$  памяти, оценка на память для всего segment tree не изменится. Запрос будет занимать  $O(\log^2 n + k)$ : на пути поиска попадется  $O(\log n)$  interval-деревьев, в каждом из них запрос будет работать за  $O(\log n + k)$ .

Правда, препроцессинг будет работать за  $O(n \log^2 n)$ : нам нужно будет построить кучу interval-деревьев для множества отрезков суммарного размера  $O(n \log n)$ , а так как каждое из деревьев строится за  $O(m \log m)$ , суммарное время построения будет  $O(n \log n \cdot \log(n \log n)l) = O(n \log n \cdot (\log n + \log \log n)) = O(n \log^2 n)$

## 4.2 3 Пересечение прямоугольника с множеством непересекающихся отрезков.

Если мы хотим пересекать прямоугольник с рандомно повернутыми отрезками, идея interval tree не работает. Сделаем другую структуру, она будет опираться не на факт axis-aligned'ности отрезков, а на факт их непересекаемости.

### 4.2.1 3 Segment tree

Это не то дерево отрезков, к которому мы привыкли.

Посмотрим на одномерный случай (та же задача, что решалась с пом. interval tree). Пусть  $I = \{[x_1 : x'_1], [x_2 : x'_2], \dots, [x_n : x'_n]\}$  – множество отрезков в  $\mathbb{R}$ . Возьмем концы этих отрезков и отсортируем их, получим точки  $p_1, p_2, \dots, p_m$ . Назовем множеством элементарных интервалов  $E = \{(-\infty : p_1), [p_1 : p_1], (p_1 : p_2), \dots, (p_m : +\infty)\}$ .

Построим на множестве  $E$  сбалансированное дерево поиска. Листьями этого дерева являются сами элементарные интервалы, а внутренними узлами – их объединения. Обозначим интервал, сопоставленный узлу  $v$  как  $Int(v)$ .

Если мы для каждого узла – элементарного интервала – будем хранить список отрезков из  $I$ , в которые он входит, мы сможем легко найти все отрезки, содержащие точку запроса  $q_x$ . Но это супернезэффективно по памяти, если у нас есть куча перекрывающихся отрезков (будет  $O(n^2)$ ). Но это можно исправить! В уже построенное дерево на элементарных интервалах будем вставлять очередной отрезок сверху вниз, оставляя его в тех узлах, которые он полностью покрывает.

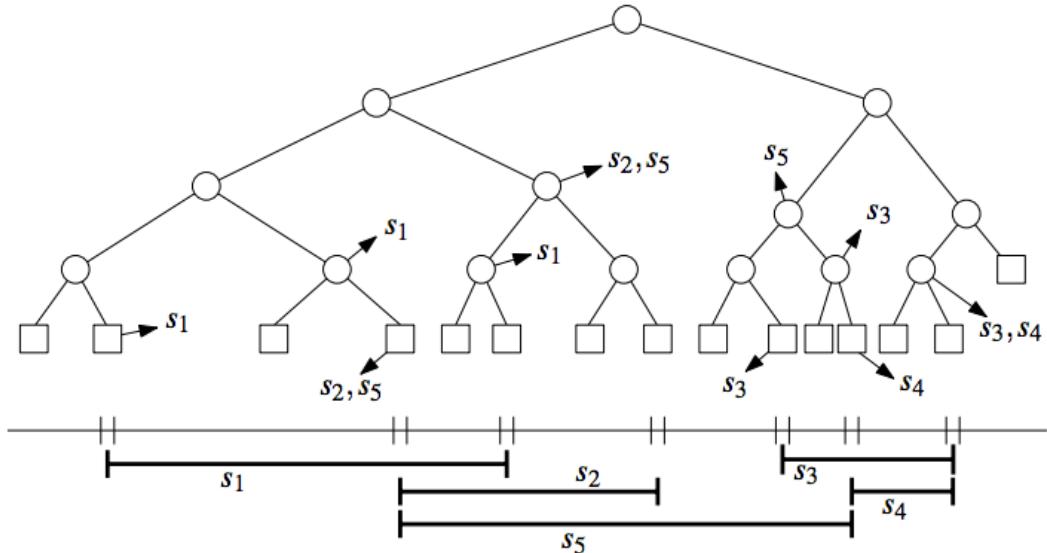


Рис. 10: Структура Segment tree

Обозначим множество отрезков из  $I$ , хранящихся в узле  $v$ , как  $I(v)$

```
Function InsertSegment( $v, [x, x']$ )
  Data: Корень дерева и вставляемый отрезок
  Result: Дерево, в которое вставлен отрезок
  if  $Int(v) \subset [x, x']$  then
    |  $I(v) = I(v) \cup \{[x, x']\}$ 
  else
    | if  $Int(v_l) \cap [x, x'] \neq \emptyset$  then
      | | InsertSegment( $v_l, [x, Int(v_l)_r]$ )
    end
    | if  $Int(v_r) \cap [x, x'] \neq \emptyset$  then
      | | InsertSegment( $v_r, [Int(v_r)_l, x']$ )
    end
  end
```

**Алгоритм 4:** Вставка отрезка в дерево

**Лемма 4.15.** 1. Такое дерево строится за  $O(n \log n)$

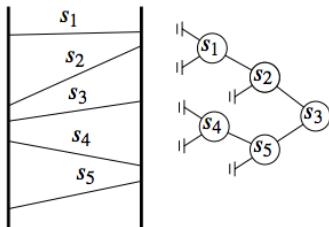
2. Оно занимается  $O(n \log n)$  памяти

*Доказательство.* 1. **Время построения:** Само дерево строится, как любое дерево бинпоиска  $O(n \log n)$ . Вставка отрезка – по алгоритму выше – занимает  $O(\log n)$ , всего отрезков  $O(n)$ , итого  $O(n \log n)$ .

2. **Память:** Посмотрим, в каких из узлов дерева может содержаться некий отрезок  $[x, x']$ . Заметим, что на одной и той же глубине в дереве отрезок может содержаться не более, чем в 2 узлах – по построению алгоритма. Следовательно, дерево содержит  $O(\log n)$  копий каждого отрезка. Итого  $O(n \log n)$  памяти.

□

### Возвращаемся к двумерному случаю



Пусть мы хотим пересечь множество непересекающихся отрезков с вертикальным отрезком  $q_x \times [q_y, q'_y]$ . Построим segment tree для проекций отрезков на ось  $x$  – сможем находить все отрезки, пересекающие вертикальную прямую  $q_x$ . Заметим, что так как отрезки непересекающиеся, то в пределах одного элементарного интервала они не меняют своего порядка по  $y$ . Значит, мы можем хранить  $I(v)$  для каждого узла как дерево поиска по отрезкам, в котором сможем делать запросы по точке пересечения (как статус Бентли-Оттмана)

Так как запрос в дереве поиска выполняется за  $O(\log n)$  без учета вывода ответа, и таких деревьев поиска надо проверить  $O(\log n)$ , время ответа на запрос в segment tree составляет  $O(\log^2 n + k)$

Рис. 11: Структура данных 2-го уровня: дерево поиска

## 5 1 3: Пересечение отрезков и поворот

VOLHOVM

Рассмотрим задачу проверить пересечение отрезков.

Вот есть у нас  $S_1 = (p_{11}, p_{12}), S_2 = (p_{21}, p_{22})$ .

В общем случае с Евклидовым пространством возникают какие-то проблемы, поэтому рассмотрим следующее определение Аффинного пространства:

$A$  – аффинное пространство, если  $A$  – такой набор точек, что:

1. В пространстве существует хотя бы одна точка.
2.  $A, B, \leftrightarrow v = \overrightarrow{AB}$ , причем  $B = A + v$ .
3. Точка + вектор = точка.
4. ... и еще 40 аксиом векторного пространства

Аффинное пространство отличается от стандартного евклидового тем, что в нем все точки равноправны, то есть ноль не зафиксирован. Типа у нас в этом пространстве есть точки, а векторы строятся из них.

Рассмотрим гиперплоскость в  $n$ -мерном аффинном пространстве. Она, очевидно, задается  $n - 1$  вектором, или как минимум  $n$  точками.

Возьмем другую случайную точку  $B$  и посмотрим, как меняются координаты при переходе из системы координат, связанной с  $A$  в систему, связанную с  $B$  (очевидно, что такой набор векторов может задавать базис, если он ЛНЗ).

Рассмотрим точку  $X$  в базисах из векторов  $\{\overrightarrow{AP_i}\}_i$  и  $\{\overrightarrow{BP_i}\}_i$ . Тут точки  $P_i$  задают гиперплоскость, то есть принадлежат ей и не линейно зависимы друг относительно друга в ней.

$$X = X_A^1 \overrightarrow{AP_1} + X_A^2 \overrightarrow{AP_2} + \cdots + X_A^n \overrightarrow{AP_n} = X_B^1 \overrightarrow{BP_1} + X_B^2 \overrightarrow{BP_2} + \cdots + X_B^n \overrightarrow{BP_n}$$

Для каждого вектора  $\overrightarrow{AP_i}$  выразим его в базисе векторов  $\overrightarrow{BP_i}$ .

$$\begin{aligned} \overrightarrow{AP_1} &= \alpha_1^1 \overrightarrow{BP_1} + \cdots + \alpha_1^n \overrightarrow{BP_n} \\ &\dots \\ \overrightarrow{AP_n} &= \alpha_n^1 \overrightarrow{BP_1} + \cdots + \alpha_n^n \overrightarrow{BP_n} \end{aligned}$$

Подставим выраженные  $AP_i$  в первое уравнение.

$$\begin{aligned} X &= X_A^1 \left( \sum \alpha_1^i \overrightarrow{BP_i} \right) + X_A^2 \left( \sum \alpha_2^i \overrightarrow{BP_i} \right) + \cdots + X_A^n \left( \sum \alpha_n^i \overrightarrow{BP_i} \right) \\ &= \overrightarrow{BP_1} \left( \sum \alpha_1^i X_A^i \right) + \overrightarrow{BP_2} \left( \sum \alpha_2^i X_A^i \right) + \cdots + \overrightarrow{BP_n} \left( \sum \alpha_n^i X_A^i \right) \end{aligned}$$

Сопоставив это с  $X$ , выраженным через  $\{\overrightarrow{BP_i}\}_i$ , получим следующую зависимость:

$$(X_B^1, X_B^2, \dots, X_B^n) = (X_A^1, X_A^2, \dots, X_A^n) \times \begin{pmatrix} \alpha_1^1 & \cdots & \alpha_1^n \\ \vdots & \ddots & \vdots \\ \alpha_n^1 & \cdots & \alpha_n^n \end{pmatrix} + (\overrightarrow{BA}^1, \dots, \overrightarrow{BA}^n)$$

Последнее – вектор перехода из точки  $B$  в  $A$ . Пусть дана точка  $O$ , которая воспринимается как ноль координат. Пусть также дана точка  $O'$ , которая выражается через  $O$ . Тогда матрица  $A$  записывается следующим образом:

$$A = \begin{pmatrix} P_1 - O' \\ P_2 - O' \\ \dots \\ P_n - O' \end{pmatrix}$$

Тут  $P_i$  и  $O'$  – это точки, координаты которых записаны относительно базиса  $O\{e_1, \dots, e_n\}$ .

Заметим, что мы можем разбить все пространство на три класса согласно того, какой знак перехода из  $O$  в  $O'$ .  $A$  – матрица перехода от  $O$  к  $O'$ ,

Ориентация – свойство точки относительно базиса  $O\{e_1, \dots, e_n\}$  и гиперплоскости, заданной точками  $\{P_i\}_{i=1}^n$ .

Знак детерминанта матрицы  $A$  действительно зависит от положения точки относительно гиперплоскости. Это принимается как что-то очевидное.

Заметим, что если у точки  $C$  ориентация относительно 0 совпадает с ориентацией точки  $D$  относительно нуля, тогда весь отрезок  $CD$  имеет ту же ориентацию – это следует из выпуклости полупространства.

Если точки  $D$  и  $C$  находятся по разные стороны гиперплоскости (имеют разную ориентацию относительно нуля), то любая кривая их связывающая, пересекает гиперплоскость. Покажем, что знак детерминанта матрицы  $A$  действительно зависит от положения точки относительно гиперплоскости.

Известный факт из линейной алгебры:

$$|\alpha_j^i| = \begin{vmatrix} P_1 - A & \vec{P}_1 & 1 \\ P_2 - A & \vec{P}_2 & 1 \\ \vdots & \vdots & \vdots \\ P_n - A & \vec{P}_n & 1 \\ & A & 1 \end{vmatrix} = \begin{vmatrix} \vec{P}_1 & 1 & \vec{P}_1 - \vec{A} & 0 \\ \vec{P}_2 & 1 & \vec{P}_2 - \vec{A} & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \vec{P}_n & 1 & \vec{P}_n - \vec{A} & 0 \\ & A & A & 1 \end{vmatrix}$$

Возьмем  $A, B$ , рассмотрим множество точек  $\{\vec{A}t + \vec{B}(1-t)\}$  и линейную комбинацию определителей:

$$|\alpha_j^i| = \begin{vmatrix} \vec{P}_1 & 1 & \vec{P}_1 & 1 & P_1 & 1 \\ \vec{P}_2 & 1 & \vec{P}_2 & 1 & P_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vec{P}_n & 1 & \vec{P}_n & 1 & P_n & 1 \\ At + B(1-t) & (1*t + (1-t)*1) & At + B(1-t) & 1 & A & 1 \end{vmatrix} = t \times \begin{vmatrix} \vec{P}_1 & 1 \\ \vec{P}_2 & 1 \\ \vdots & \vdots \\ \vec{P}_n & 1 \\ At + B(1-t) & 1 \end{vmatrix} + (1-t) \times \begin{vmatrix} P_1 & 1 \\ P_2 & 1 \\ \vdots & \vdots \\ P_n & 1 \\ A & 1 \end{vmatrix}$$

Такая комбинация имеет один знак, если детерминанты в последней сумме одного знака.

Доказательство перехода границы гиперплоскости при разных знаках такое: функция, созданная из предыдущей формулы, будет непрерывна, потому что определители близких точек близки. Тогда поскольку на концах пути точки разные, то по теореме о среднем найдется пересечение нуля.

Еще рассмотрим объем как  $V = \int_W dx_1 dx_2 \dots dx_n$ , при смене координат получим  $V' = \int_W dx'_1 dx'_2 \dots dx'_n$ . Выразим один через другой:

$$V' = \int_W \left| \frac{\partial(x'_1, x'_2, \dots, x'_n)}{\partial(x'_1, x'_2, \dots, x'_n)} \right| dx_1 dx_2 \dots dx_n$$

Это короче внутри Якоби, его можно вынести вне интеграла и заметить, что  $\frac{\partial x'_j}{\partial x_i} = |\alpha_i^j|$ .

## 6 З 4: Локализация в многоугольнике

AVBELYY

Задача: есть многоугольник  $P$ . Задается точка  $q$  в пространстве. Нужно определить, находится ли эта точка внутри или снаружи многоугольника.

### 6.1 З Выпуклый многоугольник

Время работы –  $O(\log n)$ .

Пусть задан выпуклый многоугольник с вершинами, упорядоченными против часовой стрелки и произвольная начальная точка  $p_0$ . Тогда:

- Если  $q$  лежит левее грани  $p_0p_1$  или правее грани  $p_0p_{n-1}$ , точка находится снаружи многоугольника.
- Иначе бинпоиском найдем ребро  $p_ip_{i+1}$  такое, что  $\text{turn}(p_0, p_i, q)$  и  $\text{turn}(p_0, p_{i+1}, q)$  имеют разный знак.
- Проверим поворот  $\text{turn}(p_i, p_{i+1}, q)$ .
  - Если он левый – точка находится внутри.
  - Если правый – снаружи.

### 6.2 З Невыпуклый многоугольник

Время работы –  $O(n)$ .

- Пустим луч из точки вдоль оси  $X$ , посчитаем количество пересечений с границей многоугольника.  
**Замечание:** если луч пройдет по границе многоугольника, то какие-то пересечения учтутся дважды. Условимся, что при пересечении с горизонтальным ребром мы на него забиваем, а с негоризонтальным по точке  $Q$  – учитываем, если  $Q$  – верхняя точка для ребра.
- Тогда, если количество пересечений чётно, точка находится внутри.
- Если нечётно – снаружи.

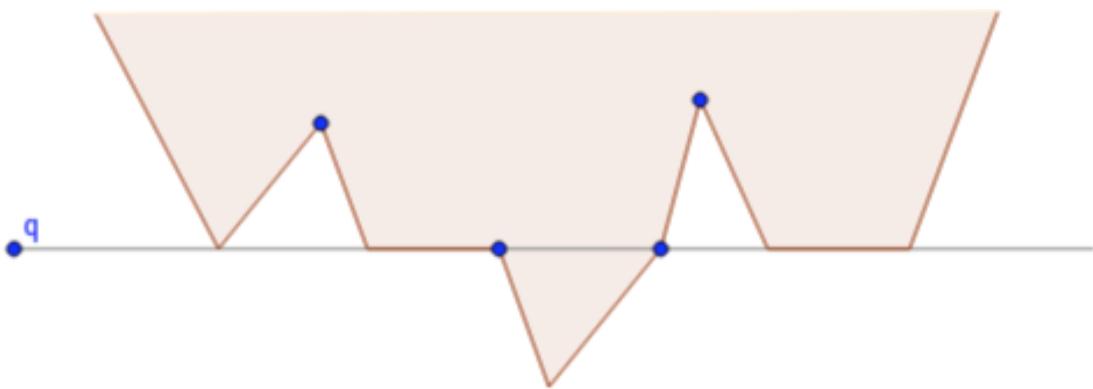


Рис. 12: Все случаи пересечения луча из точки  $q$  с границей невыпуклого  $P$ .

В задачах с целочисленными координатами точек можно пускать луч под небольшим наклоном, так, чтобы он наверняка не проходил через точки с целочисленными координатами. Тогда считать пересечения будет намного проще.

## 7 3 5: Статические выпуклые оболочки в $\mathbb{R}$

FLYINGLEAFE

### 7.1 3 Джарвис (заворачивание подарка)

1. Берем самую нижнюю левую точку  $p_0$ .
  2. За  $O(n)$  перебираем все точки, берем минимальную точку по углу относительно  $p_0$ .
- Пояснение:** Пусть мы хотим сравнить по этому параметру точки  $p_i$  и  $p_j$ . Тогда  $p_i < p_j \Leftrightarrow \text{turn}(p_0, p_i, p_j) < 0$ .
3. Добавляем выбранную точку в оболочку, проделываем то же самое с ней и т. д.

Общее время работы, очевидно,  $O(n^2)$

#### Доказательство корректности

Пусть после завершения Джарвиса осталась точка  $P$ , не лежащая внутри полученной оболочки. Это значит, что она лежит справа от некоторого ребра  $AB$  (считаем, что ребра оболочки упорядочены против часовой стрелки, так что все внутренние точки лежат слева от них).

Но тогда  $P$  меньше по повороту относительно  $A$  чем  $B$ . Значит, мы должны были выбрать ее, а не  $B$ , для построения очередного ребра оболочки, когда мы рассматривали точку  $A$ . **Противоречие.** Следовательно, такой точки  $P$  не существует.

### 7.2 3 Грэм

Возьмем самую левую нижнюю точку  $p$ . Отсортируем все остальные точки по повороту, который они образуют с этой каким-нибудь нормальным алгоритмом (за  $O(n \log n)$ ). Если все три точки лежат на одной прямой, то меньшей считается та точка, которая ближе к  $p$ .

Положим в стек точку  $p$  и первую точку из отсортированного списка остальных. Далее идем по всем точкам из списка и делаем следующее:

1. Обозначим рассматриваемую точку как  $a$ , а последние 2 точки, лежащие на стеке - как  $b$  и  $c$ .
2. Если  $\text{turn}(c, b, a) \geq 0$  (правый), то скидываем со стека точку  $b$  и возвращаемся к пункту 1
3. Иначе кладем  $a$  на стек и рассматриваем следующую вершину по списку.

В конце в стеке будут лежать вершины выпуклой оболочки.

#### Корректность

Докажем корректность алгоритма по индукции.

- **База** На третьем шаге алгоритм, очевидно, построит корректную выпуклую оболочку для первых 3 точек (просто потому, что невыпуклую построить нельзя))) )
- **Переход** Пусть на  $k - 1$  шаге построена корректная выпуклая оболочка для первых  $k - 1$  точек. Докажем, что на  $k$ -ом шаге будет построена корректная выпуклая оболочка для  $k$  точек.

1. В силу отсортированности точек по повороту, точки  $p_1..p_{k-1}$  лежат слева от ребра  $p_k p_0$  (возможно,  $p_{k-1}$  лежит на ребре)
2. На шаге 2 алгоритма из прошлой оболочки будут выброшены все вершины, видные из  $p_k$ , то есть, ни с каким из оставшихся в оболочке ребер  $p_k$  не будет образовывать правый поворот.
3. Следовательно, все ребра новой оболочки будут образовывать со всеми остальными вершинами левый (или нулевой) поворот, что нам и нужно.

#### Асимптотика

Сортировка точек за  $O(n \log n)$ . Проход по точкам за  $O(n)$ , так как каждая точка может 1 раз быть добавлена в стек и 1 раз из него удалена, всего точек  $n$ . Итого  $O(n \log n)$ .

### 7.3 3 Эндрю

Эндрю - это почти в точности Грэм.

1. Возьмем самую левую и самую правую точки -  $p_0$  и  $p_n$
2. Разделим все множество точек на "верхние" и "нижние выше прямой  $p_0 p_n$  и ниже ее, соответственно.

3. Для "верхних" и "нижних" точек построим верхнюю и нижнюю оболочки соответственно. Строить будет Грэмом, но представляя, что точка  $p_0$  лежит в  $\infty$  и  $-\infty$  соответственно. Тогда мы можем сказать, что обычная сортировка точек по координате  $x$  эквивалентна сортировке по повороту относительно бесконечно удаленной точки. Значит, отсортируем на самом деле точки каждой из половин по  $x$ -координате и запустим Грэма.
4. Объединим верхнюю и нижнюю оболочки.

### Корректность

Грэм корректен, а значит, верхняя и нижняя оболочки будут корректны. Тогда и вся оболочка корректна.

### Асимптотика

Ровно такая же как у Грэма. Но на практике Эндрю чуть быстрее лишь потому, что сортировка идет по  $x$ -координате, а не по повороту, и это быстрее.

## 7.4 3 Чен

Чен - это продукт классической методики улучшения каких-то алгоритмов: возьмем 2 известных алгоритма - один просто хороший, а другой - обладающий неким нужным свойством. Разобьем задачу на подзадачи, подзадачи решим одним алгоритмом, а объединим решения другим. Останется подобрать константу посерединке.

Так и здесь - Чен объединяет просто хороший алгоритм Грэма с output-sensitive алгоритмом Джарвиса, получая хороший output-sensitive алгоритм с временем работы  $O(n \log k)$ , где  $k$  - количество вершин выпуклой оболочки.

### Алгоритм

Разобьем все точки на произвольные группы по  $m$  (или меньше) штук в каждой. Тогда всего групп будет  $r = \frac{n}{m}$

1. Для каждой группы в отдельности найдем ее выпуклую оболочку Грэмом за  $O(m \log m)$ . Значит, всего на этот шаг уйдет  $O(r) \cdot O(m \log m) = O(\frac{n}{m}) \cdot O(m \log m) = O(n \log m)$  времени.
  2. Теперь запустим на всех точках Джарвиса. Однако заметим, что среди точек, входящих в одну группу, мы можем выбрать самую левую по повороту бинпоиском - так как для группы построена выпуклая оболочка. (Бинпоиск - это вот эта прекрасная тема с вложенными выпуклыми оболочками, например)
- Значит, на одном шаге Джарвисанам нужно перебрать все группы, среди которых подходящую точку мы ищем за  $O(\log m)$ . Итого -  $O(r \log m) = O(\frac{n}{m} \log m)$ . Всю выпуклую оболочку мы найдем за  $O(\frac{kn}{m} \log m)$ .

Сложив асимптотики двух шагов, видим, что полное время работы -  $O(n(1 + \frac{k}{m}) \log m)$ . Из этого получится желанная асимптотика  $O(n \log k)$ , если мы с самого начала выберем  $m = k$ . Но как нам это сделать?

Давайте просто перебирать  $m$ , начиная с маленького. Если вдруг во время выполнения на  $m + 1$  шаге Джарвис еще не построил выпуклую оболочку, значит,  $m < k$  и нам надо взять его побольше.

Но как перебрать  $m$  достаточно быстро, и при этом не переборщить на последнем шаге? Давайте возьмем начальный  $m = 2$  и на каждом шаге перебора будем возводить его в квадрат. Иными словами,  $m = 2^{2^t}$ , и  $t$  перебирается от 0 до  $\lceil \log \log k \rceil$

Докажем, что такой перебор не замедлит общее время работы:

$$\sum_{t=0}^{\lceil \log \log k \rceil} O\left(n \log(2^{2^t})\right) = O(n) \sum_{t=0}^{\lceil \log \log k \rceil} O(2^t) = O(n \cdot 2^{1 + \lceil \log \log k \rceil}) = O(n \log k)$$

Итак, мы получили алгоритм с гарантированным временем работы  $O(n \log k)$ .

## 7.5 3 QuickHull

Как QuickSort, только QuickHull.

1. Возьмем крайние по иксу точки (они точно войдут в оболочку), обозначим их как  $p_0$  и  $p_1$
2. Разобьем множество на точки, лежащие ниже и выше прямой  $p_0p_1$  (посвопаем 2 указателями, как в квиксорте)
3. Для верхнего множества найдем самую удаленную от  $p_0p_1$  точку -  $q_1$

4. Выкинем все точки, лежащие внутри треугольника  $p_0p_1q_1$
5. Разделим оставшиеся точки на  $S_1$  - лежащие выше  $p_0q_1$ , и  $S_2$  - лежащие выше  $q_1p_1$ .
6. Рекурсивно повторим пункт 3 для  $S_1$  и  $S_2$ .
7. Повторим пункт 3 для нижнего множества.
8. Объединим верхнюю и нижнюю оболочки

Утверждается, что для случайного набора точек этот алгоритм отработает за  $O(n \log n)$ . Понятно, что в худшем случае алгоритм отработает за  $O(n^2)$  - мы можем построить такой выпуклый многоугольник, что на шаге 4 никогда ничего не будет выкинуто, а на шаге 5 в  $S_1$  будут входить все оставшиеся точки.

Докажем, что для случайно разбросанных точек алгоритм отработает за  $O(n \log n)$

### **WARNING: ЭТО ГОВНО Я ПРИДУМЫВАЛ САМ (почти) НА САМОМ ДЕЛЕ ДОКАЗАТЕЛЬСТВО НЕ НУЖНО УРА**

Пусть время, необходимое для нахождения оболочки над некой прямой и множеством точек  $S$  есть  $T(S)$ . Тогда  $T(S) = O(|S|) + T(S_1 \in S) + T(S_2 \in S)$ , где  $S_1$  и  $S_2$  из пункта 5.

За  $O(|S|)$  мы находим самую удаленную от прямой точку  $q_1$ . Заметим, что вообще все рассматриваемые точки находятся в прямоугольнике, ограниченном прямой  $p_0p_1$  снизу, и вершиной  $q_1$  сверху. Заметим также, что треугольник  $p_0q_1p_1$  занимает половину площади этого прямоугольника. Это значит, что при равномерном распределении точек внутрь треугольника попадет примерно половина всех точек. А значит, количество рассматриваемых точек на следующем шаге рекурсии будет меньше в 2 раза. Значит, всего шагов рекурсии будет  $O(\log n)$ , что в итоге дает оценку  $O(n \log n)$ .

## 7.6 3 Оболочка многоугольника

Дан многоугольник без самопересечений. Хотим найти для него выпуклую оболочку за линию.

Сделаем обход Грэма по многоугольнику, начиная с самой левой точки. Это займет  $O(n)$  времени. Докажем, что в итоге выпуклая оболочка получится корректной.

Не всегда является правдой то, что стек после  $k$ -ой итерации представляет собой список вершин корректной выпуклой оболочки для  $k - 1$  вершин – очередная вершина многоугольника может повернуть сильно назад, и после исключения криво повернутых вершин оболочка может перестать их заключать. Однако является правдой то, что построенная на  $k$ -ом шаге оболочка является корректной выпуклой оболочкой для всех вершин, лежащих справа от ребра  $p_0p_k$ . Так как в многоугольнике нет самопересечений, это ребро пройдет через любую точку нечетное количество раз, а значит, в конечном итоге каждая точка войдет в выпуклую оболочку.

## 7.7 3 Оболочка полилинии

Дана последовательность точек, образующая полилинию без самопересечений. Нужно за  $O(n)$  построить ее выпуклую оболочку.

Для этого заведем дек, в котором будут поддерживаться 2 инварианта:

1. Вершины в деке – это вершины корректной выпуклой оболочки для всех рассмотренных вершин.
2. Последняя рассмотренная вершина лежит спереди и сзади дека (если она принадлежит текущей оболочке).

Дек инициализируется первыми тремя вершинами полилинии (очевидно, корректно)

Когда мы добавляем очередную вершину, мы можем обнаружить ее в одной из четырех позиций (см. рис); не можем обнаружить ее "спрятанной" от текущей головной вершины позади всей оболочки из-за несамопересекаемости полилинии. В позиции 4 (внутри текущей оболочки) ничего делать не надо, в позициях 1 или 2 – попаем вершины с соответствующей стороны дека, пока не получится корректный поворот; в случае 3 попаем по 1 вершине с обоих сторон дека. Во всех случаях в конце кладем эту вершину с обоих концов дека. Алгоритм корректен, так как на каждом шагу выполняются оба инварианта. Работает за линию, так как добавить и выбросить каждую вершину в дек мы можем только 1 раз.

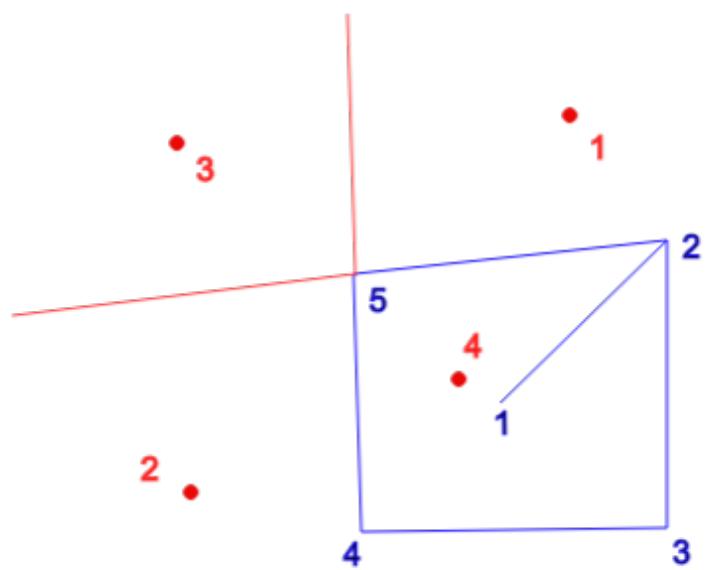


Рис. 13: Добавление новой вершины в оболочку. Состояние дека: 5, 4, 3, 2, 5

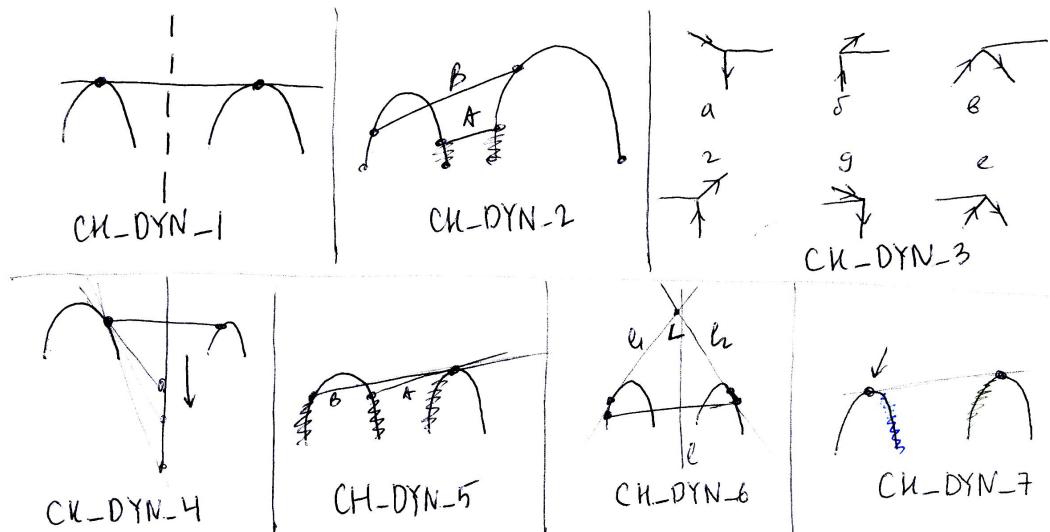


Рис. 14: Иллюстрации к динамической выпуклой оболочке

### 8.1 З Задача объединения двух верхних CH

Начнем с подзадачи: пусть у нас есть две каких-то верхних оболочки в  $\mathbb{R}^2$ , разделенных по иксу (CH\_DYN\_1). Мы хотим объединить эти верхних оболочки, проводя касательную сверху. Как такую касательную построить? (inb4 такая существует, потому что "палка сверху падает на холмики"). Как искать такую касательную за логарифм?

Очевидно, что касательная не проходит по экстремальным точкам (нарисуем большой холмик и рядом маленький).

Как добиться асимптотики  $O(\log n)$ ? Предположим, что есть пара точек на холмах. Будем типа пользоваться некоторым подобием бинпоиска на двух холмах сразу – держать четыре границы одновременно. Ну, два массива – это два множества точек для двух оболочек, отсортированных по иксу (См. CH\_DYN\_2).

(CH\_DYN\_3) описывает классификацию всех попаданий касательной к кускам выпуклой оболочки для левой и правой кучи. Эта классификация важна, так как по ней мы будем определять текущее состояние бинпоиска. Как эти состояния отличать, понятно – считаем повороты касательной с ребром, куда она попала. Случай с двумя точками по одну сторону классифицируются поворотом. Проверка на два случая делается за  $2 \times 2 = 4$  поворота.

Рассмотрим случай  $A$  в CH\_DYN\_2. Случай  $A$  распознается так: это случай слева а), а справа г). Рассмотрим прямую  $l$  и какую-то касательную к левой куче. Утверждается, что если мы будем поворачивать касательную вокруг точки касания, поворачивать вниз, то пересечение касательной и  $l$  как точки, будет опускаться вниз (CH\_DYN\_4). Из этого следует, что можно отрезать нижние куски выпуклых оболочек.

Рассмотрим остальные случаи, например  $B$  в CH\_DYN\_2. В этом случае мы можем откинуть нижнюю часть правой оболочки. Симметричный случай тоже очевиден.

Случай с двумя касательными (случаи в), е) в диаграмме) тоже распознается однозначно и есть ответом бинпоиска.

Пусть на правом холме у нас касательная, а на левом точка из случая а) – случай  $A$  в CH\_DYN\_5. Тогда на левом холме мы можем откусить нижний кусок, а на правом – левый нижний от касательной. Симметрично тоже.  $B$  тоже так решается, то есть можно слева откусить нижний, а справа нижний левее точки касания.

Теперь рассмотрим самый нетривиальный случай (CH\_DYN\_6): пусть слева б), а справа д). Рассмотрим пересечение прямых  $l_1$  и  $l_2$ . Прямые проведем через текущие вершины и следующие выше. Проверим точку  $L$  пересечения  $l_2$  и  $l_1$ . Тогда если прямая  $L$  лежит полностью в интервале между холмами, то можем выкинуть и у левого и у правого нижние куски. Если точка  $L$  лежит в левом холме (левее самой правой точки левого холма), то мы выкидываем весь нижний кусок только левого холма вместе с этой точкой. Аналогично с правым холмом.

Теперь мы умеем решать задачу найти касательную двух верхних полуоболочек.

Задача поиска всех четырех касательных для двух выпуклых множеств сводится к этой: разобьем на несколько подмножеств (верхние и нижние) и решим алгоритмом выше.

В реализации алгоритма удобно хранить две оболочки skip-листами и вместо бинпоиска просто спускаться на нижний уровень и продолжать алгоритм на нем. Вот мы идем по какому-то уровню, выбираем вершину. Пусть мы определили, что нам необходимо отрезать какую-то часть оболочки, к примеру, левую – просто пойдем вправо по текущему уровню, пометив "отрезанную" вершину флагом. Спуск на нижний уровень будет происходить, если нужно пойти в какую-то сторону, а та вершина уже "отрезана".

## 8.2 3 Итеративный алгоритм

Теперь мы хотим честного итеративного построения: есть некоторая структура, в которой мы храним верхнюю оболочку, и мы хотим ее быстро изменять (добавлять или удалять точки).

Для начала вспомним, как мерджить skip-листы. Лист мы держим сверху за вершину самого высокого уровня, на каждом уровне мы можем распознать первую и терминальную вершины.

- Сплит: дали нам вершинку, мы нашли ее в самом нижнем уровне. Запускаемся для левой стороны: удаляем вершину, обрезаем. Идем влево, пока не можем подняться наверх, поднимаемся, делаем вершинку терминальной на этом уровне, и так до верхнего уровня. Аналогично для правой стороны помечаем вершину первой, идем вправо, поднимаемся если можем, и так до самого высокого уровня.
- Мердж делается так же, про доказательство асимптотики думать не надо (бернуллевость не испортится).

Пусть есть оболочка, являющаяся общей частью двух оболочек подмножеств точек (`CH_DYN_7`). Есть также указатель на точку, по которой нужно разделиться. Причем у нас есть синяя и красная (карандашом) части. Тогда мы можем разделить нашу оболочку на две за  $2 * \log n$  на объединение двух skipлистов.

Общая структура для хранения оболочки итеративно наивно представляется так: дерево, в котором листья – наши точки, а другие узлы – это верхняя оболочка сыновей. Это  $O(n \log\{n\})$  памяти. Такая структура имеет два недостатка – памяти много и неочевидно, как делать удаление. Добавление реализуется прокидыванием вершины вниз и перестраиванием все оболочки вверх во время просеивания. Если дерево нужно балансировать, то во время поворотов нужно будет перестраивать узлы.

Более удобная структура выглядит следующим образом: в самом верхнем узле будет храниться честная выпуклая оболочка всех точек. Не верхнем, будем хранить только ту часть выпуклой оболочки, которая не является общей с родителем. На `CH_DYN_7` "не общие части" как раз обозначены синим и серым цветом. Продавливание точки вниз становится существенно понятнее и проще: разбиваем текущую выпуклую оболочку (сначала корневую), объединяем за  $\log n$  с детьми. Определяем, куда кидать точку – влево или вправо. Ту часть, в которую не нужно добавлять, не трогаем. Так проходим вниз и добавляем вершинку. Заметим, что теперь уже не нужно хранить ничего в листах, так как два соседних листа однозначно определяются оболочкой в их родителе. Дальше строим оболочку и просеиваем вверх. При просеивании вверх берем двух детей, объединяем, отдаем родителю оболочку, себе оставляем только те части, которые не входят в парента. Удаление происходит аналогично.

Итого конечный алгоритм поддерживает оболочку с удалением и добавлением за  $\log^2 n$ .

## 9 1 7: Трехмерные выпуклые оболочки (CHN)

FLYINGLEAFE

Чтобы понять, как будет работать рандомизированный алгоритм поиска выпуклой оболочки в 3d, изменим Quickhull в 2d следующим образом.

Начнем с построения случайного треугольника на трех случайных точках. Потом перебираем все точки в случайном порядке, определяем грани текущей выпуклой оболочки, видимые из выбранной. Очевидно, что все такие грани будут рядом друг с другом, поэтому у видимого промежутка оболочки есть 2 грани ("горизонт видимости"). В плоском случае эти две точки совпадают с 2 точками касания прямых, проведенных из выбранной точки. Искать касательные к выпуклому многоугольнику мы умеем за  $O(\log n)$ , поэтому в плоском случае этот алгоритм очень прост и работает всегда за  $O(n \log n)$ .

А вот в 3d все сложнее: граней много, ребер много, непонятно, как эффективно определять горизонт видимости – трюк с касательными не прокатит.

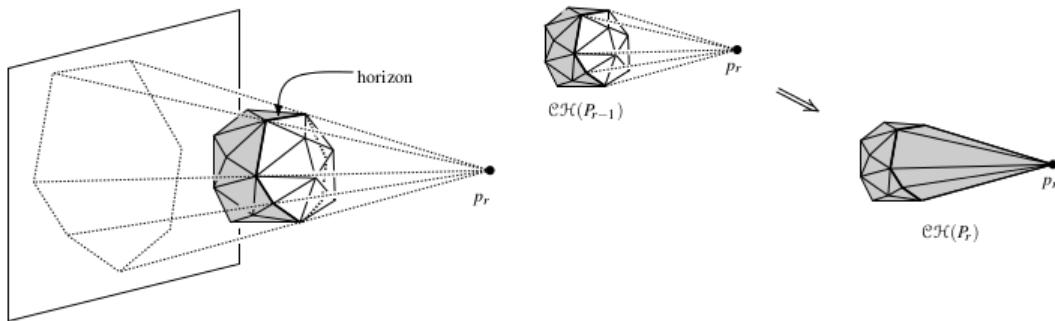


Рис. 15: Процесс добавления очередной вершины к трехмерной оболочке.

Для начала дадим более-менее строгое определение **видимости** грани из точки.

**Определение.** Грань  $f$  выпуклого многогранника  $P$  **видима из точки**  $p$ , если  $P$  и  $p$  лежат по разные стороны плоскости  $h_f$ , содержащей  $f$ .

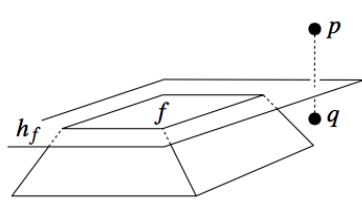


Рис. 16: Грань  $f$  видима из  $p$ , но не из  $q$ .

Начинается алгоритм в 3d аналогично – строим случайный тетраэдр, случайно перемешиваем остальные точки. Чтобы добавить очередную точку, нам нужно провести из этой точки треугольные фейсы к ребрам горизонта видимости, и удалить из оболочки грани, бывшие видимыми. Из определения видимости очевидно, что каждое ребро горизонта видимости является опорным для какой-либо касательной плоскости для  $p$  и  $P$ , поэтому полученный после такой операции многогранник останется выпуклым.

Но как найти видимые грани, не перебирая их все? Для этого будем поддерживать вспомогательную структуру, которую назовем **граф конфликтов**.

**Определение.** Граф конфликтов – это двудольный граф, в левой доле которого содержатся вершины, а в правой – текущие грани выпуклой оболочки. Между вершиной  $p$  и гранью  $f$  проведено ребро тогда и только тогда, когда  $f$  видна из  $p$ .

Понятно, что по графу конфликтов можно за  $O(k)$  выдать множество граней, видимых из данной точки (где  $k$  – количество таких граней). Научимся его поддерживать.

Инициализируется граф просто перебором точек для изначального тетраэдра (за  $O(n)$ ).

При добавлении очередной точки нужно удалить все грани, видимые из нее. Все эти грани мы переберем за  $O(k)$  обходом в графе конфликтов, и удалим все ребра в этом графе, ведущие в них. **Замечание:** если из точки не видима ни одна грань, это в точности значит, что точка лежит внутри оболочки.

После этого нам надо добавить новые треугольные грани, опирающиеся на ребра горизонта видимости. Как соответствующим образом обновить граф конфликтов?

Если новая треугольная грань лежит в одной плоскости со своей соседней старой гранью, то нам нужно ее с этой гранью слить в одну. Это можно сделать

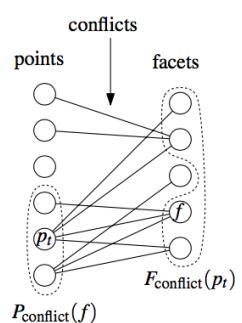


Рис. 17: Граф конфликтов

просто обновлением формы старой грани. Множество вершин, видящих старую грань, очевидно, не изменится.

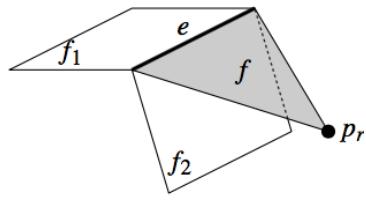


Рис. 18: Граф конфликтов

В ином же случае заметим, что любая точка, видящая новую грань  $f$ , должна видеть либо грань  $f_1$ , либо грань  $f_2$ , где  $f_1$  и  $f_2$  – это грани, формирующие опорное для  $f$  ребро горизонта видимости. Значит, чтобы найти множество точек, видящих  $f$ , нам нужно только перебрать точки, видящие либо  $f_1$ , либо  $f_2$ , что явно лучше, чем перебирать все.

#### Детали реализации

Выпуклую оболочку мы будем хранить в DCEL, ровно так же, как и плоский планарный граф, потому что любой выпуклый многоугольник (даже любой гомеоморфный выпуклому) можно планарно разложить на плоскости. Разница только в том, что координаты в вершинах будут трехмерными. Из этого же и следует, что по памяти структура для его хранения будет линейна относительно количества вершин (как и любой DCEL).

**Асимптотика**  $O(n \log n)$ , мамой клянусь! Ковалев сказал, что доказывать не нужно.

## 10 3 8: Триангуляция (существование и ушная триангуляция) AVBELYY

### 10.1 3 Существование

**Определение** (Триангуляция). *Разбиение многоугольника  $P$  на множество треугольников, чьи внутренние области попарно не пересекаются, а объединение дает многоугольник  $P$ .*

**Определение** (простой многоугольник). *Многоугольник без самопересечений.*

**Теорема 10.1** (О существовании триангуляции многоугольника).

*У любого простого многоугольника  $P$  с  $n$  вершинами всегда существует триангуляция, причем количество треугольников в ней равно  $n - 2$ .*

*Доказательство.* Докажем оба утверждения по индукции.

- **База**  $n = 3$  – у треугольника есть триангуляция, состоящая из 1 треугольника)).
- **Переход** Пусть для простых многоугольников с  $k < n$  вершинами существует триангуляция. Покажем, что и для  $n$ -угольников триангуляция существует.
  - Возьмем самую левую вершину многоугольника  $P$ , назовём её  $v$ . Тогда диагональю будет либо ребро:
    - \* между её соседями слева и справа;
    - \* между  $v$  и вершиной  $p$ , наиболее удаленной от ребра между соседями и находящейся по одну сторону с  $v$ .
  - Диагональ поделит  $P$  на многоугольники  $P_1$  и  $P_2$  ( $|P_1| + |P_2| = n - 2$ ) меньшего размера, у которых по предположению существует триангуляция.
  - В триангуляции  $P_1$  и  $P_2$ , также по предположению, будет соответственно  $|P_1| - 2$  и  $|P_2| - 2$  треугольника. Тогда в триангуляции  $P$  будет  $(|P_1| - 2) + (|P_2| - 2) = n - 2$  треугольника.

□

### 10.2 3 Ушная триангуляция

**Определение** (Ухо). *Вершина многоугольника  $v_i$  называется ухом, если диагональ  $v_{i-1}v_{i+1}$  лежит строго во внутренней области многоугольника.*

**Теорема 10.2** (О существовании двух ушей в многоугольнике).

*У любого простого многоугольника  $P$  с  $n$  вершинами всегда существует два не пересекающихся междуд собой уха.*

*Доказательство.* По индукции.

- **База**  $n = 4$  – всё ясно.
- **Переход** Возьмём произвольную вершину  $v$ . Рассмотрим два случая:
  - $v$  – ухо. Отрежем его, получим  $n - 1$ -угольник, в котором, по предположению индукции, есть два непересекающихся уха. Они также являются ушами исходного  $n$ -угольника, поэтому теорема верна.
  - $v$  – не ухо. Значит, треугольник  $(\text{prev}(v); v; \text{next}(v))$  содержит вершины  $P$ . Как и в теореме о существовании триангуляции, выберем наиболее близнюю к  $v$  вершину, поделим  $P$  на  $P_1$  и  $P_2$  по диагонали, у  $P_1$  и  $P_2$  по индукции есть по два уха – и вновь всё хорошо.

□

### 10.2.1 Алгоритм (ушная триангуляция)

Положим вершины  $n$ -угольника в циклический двусвязный список. Пройдемся по списку, проверяя вершины на "уховость" пока не упремся в самую первую вершину во второй раз. Если текущая вершина  $p_i$  – ухо, то сделаем следующее:

- Добавим в ответ треугольник  $(p_{i-1}, p_i, p_{i+1})$ ;
- Удалим  $p_i$  из списка;
- Проверим на "уховость" следующую за  $p_i$  вершину.

Если  $p_i$  оказалась не ухом, перейдем к вершине  $p_{i+1}$ .

**Корректность** алгоритма следует из существования ушей в любом  $n$ -угольнике, доказанного выше.

**Время работы** проверки на уховость –  $O(n)$ . Всего проверяется  $n - 3$  вершины. Значит, время работы всего алгоритма –  $O(n^2)$ .

# 11 3 9: Триангуляция с заматающей прямой

AVBELYY

## 11.1 3 Идея и основные определения

Заметим, что триангуляция выпуклого многоугольника делается очень просто за  $O(n)$ . Однако побить произвольный  $P$  на выпуклые куски сложно. Придумаем что-то похожее, на что побить и что триангулировать будет несложно и недолго.

**Определение** (Монотонный многоугольник). *Многоугольник  $P$  называется монотонным относительно прямой  $\ell$ , если любая  $\ell'$ , ортогональная  $\ell$ , пересекает стороны  $P$  не более двух раз (то есть либо не пересекает  $P$  совсем, либо пересекает по точке или отрезку).*

**Определение** ( $Y$ -монотонный многоугольник). *Многоугольник, монотонный относительно оси  $Y$ .*

**Определение** (start, end, split, merge и regular-вершины). *Пусть  $\phi$  – внутренний угол при вершине в многоугольнике. Тогда назовем вершину:*

- *Start* – если два ее соседа лежат ниже ее самой и  $\phi < \pi$ .
- *Split* – если два ее соседа лежат ниже ее самой и  $\phi > \pi$ .
- *End* – если два ее соседа лежат выше ее самой и  $\phi < \pi$ .
- *Merge* – если два ее соседа лежат выше ее самой и  $\phi > \pi$ .
- *Regular* – если один сосед лежит выше, а другой ниже ее самой.

**Лемма 11.1** (Достаточное условие у-монотонности). *Если в многоугольнике нет split- и merge-вершин, то он у-монотонен.*

**Доказательство.** Докажем контрапозицию этого утверждения: в любом у-немонотонном многоугольнике  $P$  имеется либо split-, либо merge-вершина.

Поскольку  $P$  немонотонен, есть горизонтальная прямая  $\ell$ , которая пересекает его больше двух раз. Возьмем ее так, чтобы первое пересечение проходило по отрезку  $[p; q]$ . Пройдемся наверх от вершины  $q$ , пока не упремся в точку  $r$  на прямой. Рассмотрим 2 случая:

- (a)  $r \neq p$ . Видно по рисунку, что самая высокая вершина на пути  $[q; r]$  будет split-вершиной.
- (b)  $r = p$ . Пройдемся в другую сторону (на этот раз вниз) от  $p$ , снова пересечем прямую в точке  $r'$ .  $r' \neq p$ , ведь это означало бы, что прямая пересекает многоугольник всего дважды. Значит, путь  $[q; r']$  проходит под прямой  $\ell$  и самая нижняя вершина на этом пути будет merge-вершиной.

□

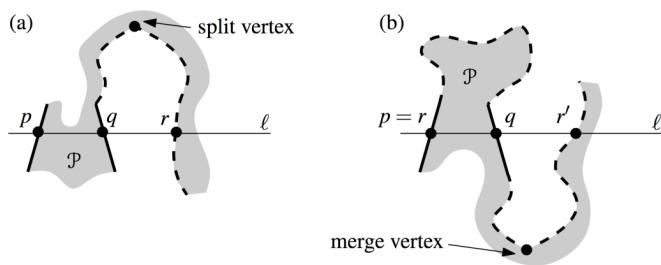


Рис. 19: Два случая в лемме о достаточном условии.

## 11.2 3 Алгоритм 1 (разбиение на монотонные части)

В условиях предыдущей леммы становится понятен алгоритм разбиения многоугольника на монотонные куски: нужно избавиться от split- и merge-вершин, проводя из них диагонали вниз и вверх соответственно.

**Алгоритм:** пройдем заматающей прямой по всем вершинам многоугольника сверху вниз. Останавливаясь на split- и merge-вершинах, будем проводить диагонали из них. Пусть  $v_i$  – текущая вершина,  $e_j$  и  $e_k$  – ближайшее слева и справа ребро от  $v_i$  соответственно. Чтобы быстро получать левое ребро  $e_j$  для вершины, заведем бинарное дерево  $T$  на ребрах (типа упорядочим их слева направо). Ниже рассмотрим действия, предпринимаемые алгоритмом в разных типах вершин:

### 11.2.1 Split-вершина

Мы хотим провести такую диагональ, которая точно будет лежать в  $P$  и не пересекать других ребер. Куда ее провести? Утверждается, что всегда подойдет **самая низкая вершина между  $e_j$  и  $e_k$  повыше 1**. Заведем у ребер  $P$  дополнительное поле  $helper$ , куда будем записывать по ходу движения прямой вершины, видящее ребро слева от себя. Тогда в split-вершине просто посмотрим на  $helper(e_j)$  – в нем и будет нужная нам вершина. Если  $helper(e_j) == \text{NULL}$ , возьмем верхний конец  $e_j$ . Проведем диагональ  $[v_i; helper(e_j)]$  и пойдем дальше.

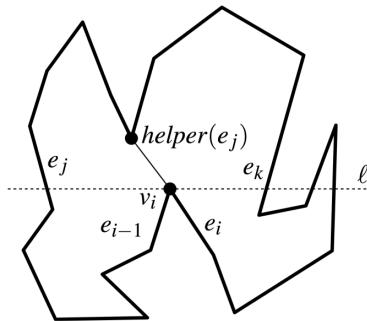


Рис. 20: Заметающая прямая  $\ell$  встретила split-вершину.

### 11.2.2 Merge-вершина

Merge-вершину мы встречаем раньше, чем ее диагональную пару, поэтому сразу диагональ из нее не можем. Но можем записать ее в  $helper(e_j)$  и пойти дальше, а в следующей по ходу прямой вершине проверить, является ли  $helper(e_j)$  merge-вершиной. Если является, немедленно проведем диагональ из нее. Если для merge-вершины не нашлось диагональной пары, соединим ее с нижним концом  $e_j$ .

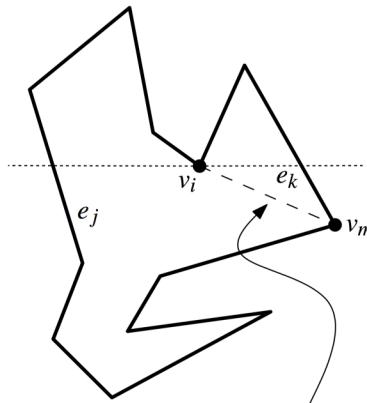


Рис. 21: Диагональ из merge-вершины проводится из вершины пониже.

### 11.2.3 Start-вершина

Вставим  $e_i$  в  $T$ . Установим  $helper(e_i)$  в  $v_i$ .

### 11.2.4 End-вершина

Если в  $helper(e_{i-1})$  merge-вершина, то соединим ее диагональю с текущей. Удалим  $e_{i-1}$  из  $T$ .

### 11.2.5 Regular-вершина

1. **P расположен справа от  $v_i$ .** Проделаем действия как для end-вершины ребра  $e_{i-1}$  и start-вершины ребра  $e_i$ .
2. **P расположен слева от  $v_i$ .** Проведем ребро в  $helper(e_j)$ , если  $helper(e_j)$  – merge-вершина. Установим  $helper(e_j)$  в  $v_i$ .

### 11.3 3 Оценка времени работы алгоритма разбиения

Оценим время работы такого алгоритма. События заметающей прямой – вершины многоугольника, их ровно  $n$  штук. Для каждой вершины выполняется константное количество манипуляций над деревом ребер  $T$  суммарной сложностью  $O(\log n)$  и, возможно, добавление диагоналей в  $P$  (это можно сделать за  $O(1)$ , если хранить многоугольник в двусвязном списке). Итоговая сложность –  $O(n \log n)$ , а занимаемая память –  $O(n)$ .

### 11.4 3 Алгоритм 2 (триангуляция монотонного многоугольника)

**Алгоритм:** пойдем сверху вниз по ребрам границы, добавляя диагонали пока это возможно.

(1) Заведем дополнительно стек для вершин  $S$ , которые мы уже прошли, но которым все еще может понадобиться диагональ. Когда мы встречаем вершину, мы проводим максимально возможное количество диагоналей из нее в вершины стека.

Заметим, что вершины в  $S$  имеют определенную форму: они образуют выпуклую воронку, все вершины которой представляют собой последовательную цепь вершин  $P$  и имеют внутренний угол  $> \pi$ .

(2) Переберем вершины  $v_i$ ,  $i=1..n-1$ . Посмотрим, какие ребра мы можем добавить к вершинам стека из текущей вершины  $v_i$ . Разберем два случая:

- (a) Текущая вершина лежит напротив вершин  $S$ .

Соединим диагоналями текущую вершину и все вершины стека. Кроме последней. Между последней и текущей, по инварианту, уже есть ребро (см. рис). Добавим вершину стека в стек обратно, потому что для нее еще не составлен треугольник, и сохраним инвариант воронки.

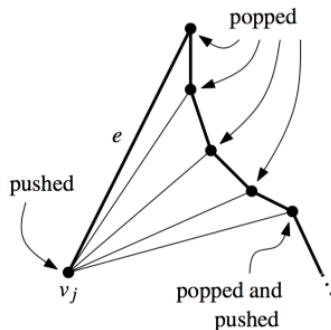


Рис. 22: Случай (a):  $v_i$  по разные стороны  $P$ .

- (b) Текущая вершина лежит на одной стороне с вершинами  $S$ .

Вытащим и выкинем из стека первую вершину. С остальными проделаем следующее: вытащим и проверим, можем ли провести диагональ из нее в  $v_i$ . Если да - продолжим снимать со стека вершины, если нет - откатим стек до той вершины, до которой еще могли провести диагональ и остановимся. В конце положим еще  $v_i$ . Инвариант "выпуклой воронки" вновь сохраняется (см. рис).

(3) Для последней вершины  $v_n$  проведем диагонали ко всем вершинам стека, кроме первой и последней. После этого монотонная часть будет триангулирована.

### 11.5 3 Оценка времени работы алгоритма триангуляции

- Шаг (1) занимает константное время.
- Цикл на шаге (2) исполняется  $O(n)$  раз и каждая итерация может занять линейное время. Но заметим, что за итерацию в  $S$  положат не более  $2x$  вершин, то есть суммарно  $O(n)$  push'ей и не больше pop'ов. Итого  $O(n)$ .
- Шаг (3) также занимает линейное время.

Значит, справедливо следующее:

**Лемма 11.2.** Алгоритм триангуляции монотонной части многоугольника работает за линейное время от числа вершин монотонной части.

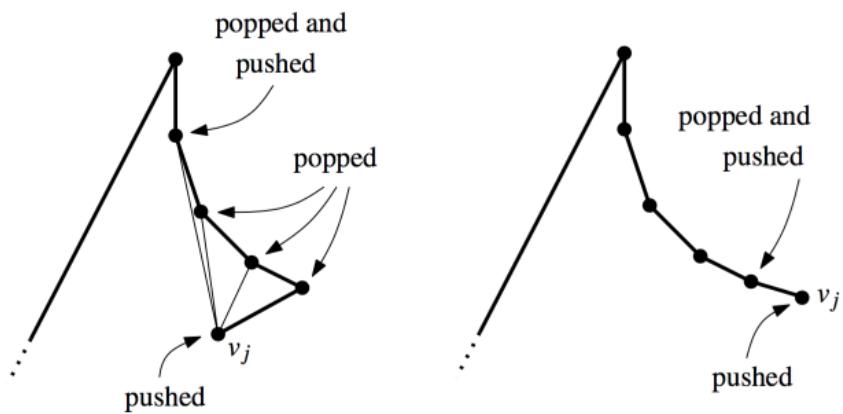


Рис. 23: Случай (б):  $v_i$  по одну сторону Р.

### Теорема 11.1.

Алгоритм монотонной триангуляции работает за  $O(n \log n)$  времени и использует  $O(n)$  памяти.

*Доказательство.* Алгоритм 1 работает за  $O(n \log n)$  как доказано выше, алгоритм 2 работает за линейное от размера монотонной части время. Значит, суммарное время для всех частей Р – линейное от числа вершин Р.

Алгоритм 1 использовал двусвязный с вершин и двоичное дерево ребер, алгоритм 2 использовал стек вершин. Каждое ребро и вершина встречались в соответствующих структурах не более 1 раза. Итого –  $O(n)$  памяти.  $\square$

## 12 2 10: Полуплоскости и выпуклые оболочки

AVBELYY

Задача: найти фигуру, образованную пересечением  $n$  полуплоскостей  $(l_1, l_2, \dots, l_n)$ , или сообщить, что оно пусто.

Эту задачу можно решать разными способами, рассмотрим два из них.

### 12.1 3 Построение выпуклой оболочки

#### 12.1.1 Сведение к двойственным задачам

**Определение** (Двойственные преобразования). Пусть точка  $p = (a, b)$ , прямая  $l = (c, d)$ .

Тогда:

- $p^*$  – двойственная прямая к  $p$  [ $p^* = ax - b$ ]
- $l^*$  – двойственная точка к  $l$  [ $l^* = (c, -d)$ ].

**Определение.**  $UH(P)$  – верхняя выпуклая оболочка точек  $P$ .

**Определение.**  $LE(L)$  – граница фигуры, образованной пересечением полуплоскостей  $L$ , смотрящих вниз.

Заметим, что найти фигуру – это то же самое, что найти ее границу, поэтому поиск  $LE(L)$  – то же самое, что исходная задача.

**Теорема 12.1.**

Задача поиска  $LE(L)$  – двойственная к задаче поиска  $UH(L^*)$ .

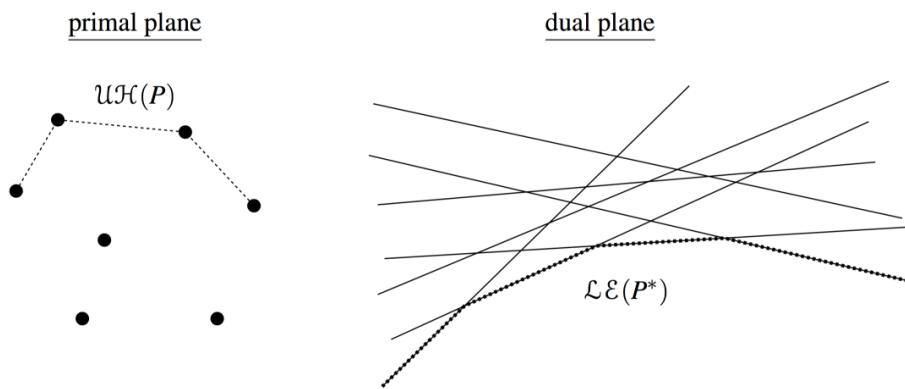


Рис. 24: Пересечение полуплоскостей эквивалентно нахождению выпуклой оболочки.

**Доказательство.** Точка  $p$  принадлежит  $UH(P) \Leftrightarrow \exists$  не-вертикальная прямая  $l$ , проходящая через  $p$  так, что все остальные точки  $P$  лежат ниже  $l$ .

То же самое в двойственном пространстве:  $\exists$  точка  $l^*$ , лежащая на прямой  $p^*$  так, что точка  $l^*$  лежит ниже всех остальных прямых из  $P^*$   $\Leftrightarrow l^* \in LE(P^*)$ .

Точки в  $P$  возрастают в  $UH(P)$  по  $x$ -координате. Прямые  $P^*$  убывают в  $LE(P^*)$  по углу наклона. Поскольку угол наклона прямой  $p^*$  совпадает с  $x$ -координатой точки  $p$ , порядок  $UH(P)$  совпадает с порядком  $LE(P^*)$ .

Следовательно,  $UH(P) = LE(P^*)$ , или  $UH(L^*) = LE(L)$ . □

**Замечание:** как можно догадаться, пересечение полуплоскостей, смотрящих вверх – это задача, двойственная к поиску нижней выпуклой оболочки. Это нетрудно доказать, но мы оставим это в качестве упражнения читателю.

#### 12.1.2 Алгоритм

Для начала заметим, что получившаяся фигура будет выпуклой, поскольку:

- полуплоскость выпукла;
- пересечение выпуклых фигур выпукло.

Учитывая вышедоказанное, алгоритм получается такой:

1. Поделим плоскости на "смотрящие" вниз и вверх.
2. Найдем пересечение плоскостей, смотрящих вниз.
3. Найдем пересечение плоскостей, смотрящих вверх.
4. Объединим фигуры, полученные на 2 и 3 шаге.

### Теорема 12.2.

Время работы алгоритма –  $O(n \log n)$ .

*Доказательство.* Деление плоскостей по углу занимает  $O(n)$  времени. Пересечение плоскостей, смотрящих вниз/вверх имеет ту же асимптотику, что и поиск выпуклой оболочки, то есть  $O(n \log n)$ . Объединить выпуклые фигуры также можно за  $O(n \log n)$  (есть какой-то алгоритм).  $\square$

## 12.2 2 Разрезание прямоугольника

Пусть мы знаем, что, если фигура в пересечении существует и ограничена, то всегда принадлежит какому-то прямоугольнику  $D_0$ . Тогда задачу можно решить следующим способом:

```
Function Cut-Polygon( $D_0, \{l_i\}$ )
  Data: Ограничивающий прямоугольник  $D_0$ 
        Прямые, задающие полуплоскости  $\{l_i\}$ 
  Result: Фигура  $D_n \subset D_0$ , образованная пересечением полуплоскостей
  Перебираем  $i$  с 1 до  $n$ ;
  if  $D_{i-1} \cap l_i = \emptyset$  then
    |  $D_i \leftarrow D_{i-1}$ ;
  else
    |  $D_i \leftarrow \text{IntersectConvexPolygonAndLine}(D_{i-1}, l_i);$ 
  end
```

### Алгоритм 5: Алгоритм разрезания многоугольника

Тогда, если пересечение ограничено, то мы получим его в  $D_n$ . Иначе какая-то из сторон  $D_n$  будет представлять из себя подотрезок стороны  $D_0$ . Проверка сторон  $D_n$  осуществляется за  $O(n)$  (так как в  $D_n$  не более  $n$  сторон), время работы `IntersectConvexPolygonAndLine` –  $O(\log n)$ , итоговое время –  $O(n \log n)$ .

#### 12.2.1 IntersectConvexPolygonAndLine(P, l)

Алгоритм такой:

1. Найдем точки пересечения прямой  $l$  и выпуклого многоугольника  $P$  за  $O(\log n)$
2. Посмотрим, куда смотрит полуплоскость  $h_P$ , задаваемая прямой  $l$ , и сколько точек получилось в пересечении
  - Точек в пересечении 0 или 1,  $h_P$  смотрит "на"  $P$ : return  $P$
  - Точек в пересечении 0 или 1,  $h_P$  смотрит "против"  $P$ : return  $\emptyset$
  - Точек в пересечении 2. Разрежем  $P$  на 2 части, возьмем ту, на которую смотрит  $h_P$  и склеим ее со стороной  $[p; q]$ . Операцию разрезания и склеивания можно эффективно реализовать за  $O(\log(n))$  например с помощью декартова дерева.

## 13 2 11: Пересечение множества отрезков

AVBELYY

Задача: найти все точки, в которых пересекаются либо касаются точкой начала или конца какие-то два и более отрезков из множества  $P$ .

Понятно, как это сделать за  $O(n^2)$ , однако хочется придумать output-sensitive алгоритм.

### 13.1 Идея

Не будем проверять на пересечение отрезки, которые находятся далеко друг от друга. Что значит "далеко"? Рассмотрим ниже два разных подхода к определению "дальности смещение которых и позволит нам создать эффективный output-sensitive алгоритм.

### 13.2 Подход 1. Близость вдоль оси Y

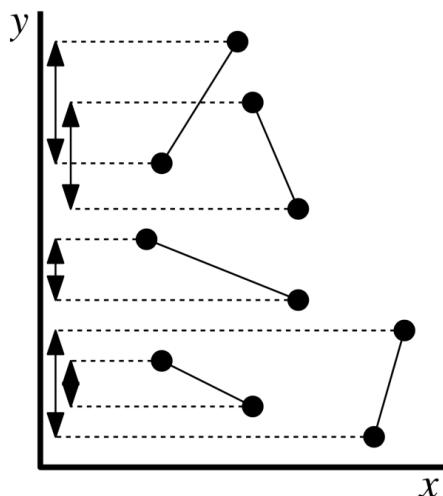


Рис. 25: Иллюстрация к первому подходу.

Понятно, что прямые, чьи вертикальные проекции на ось Y не пересекаются, не пересекаются. Тогда алгоритм, учитывающий это, звучит так: пройдемся заметающей прямой сверху вниз. При прохождении через начало отрезка добавляем отрезок в "статус список текущих отрезков и проверяем его на пересечение с другими уже добавленными отрезками. При достижении конца отрезка удаляем его из статуса.

Что плохо: может быть много отрезков, пересекающих одну и ту же горизонтальную прямую (например, ось X). Они могут не пересекаться между собой, но все равно алгоритм отработает за квадрат в этом случае. Значит, он не output-sensitive.

### 13.3 Подход 2. Близость вдоль оси X

Чтобы учитывать близость и по оси X, научим нашу заметающую прямую ходить также вдоль оси X. Для этого отрезки "статуса" будем хранить в том порядке, в котором их встретит заметающая прямая (например, слева направо). Проверять на пересечение будем только соседние отрезки. Как определять соседство? Первоначально - по близости x-координат точек начала отрезков, но в дальнейшем отрезки могут становиться соседями с другими отрезками.

### 13.4 Алгоритм

Все готово для того, чтобы окончательно сформулировать алгоритм. Сперва сделаем это, не учитывая некоторые вырожденные случаи, а именно:

- Пересечение более двух отрезков в одной точке
- Наличие горизонтальных отрезков
- Пересечение отрезков в более чем одной точке

Сначала докажем корректность наших идей. А именно то, что для любого пересечения двух отрезков существует событие, в котором обнаружится их пересечение.

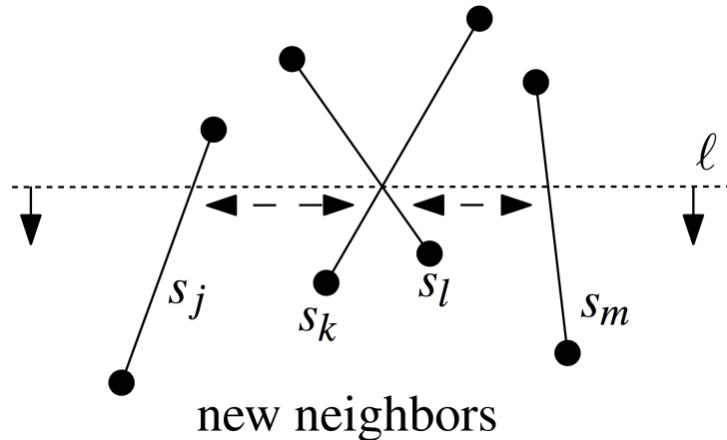


Рис. 26: Иллюстрация ко второму подходу.

**Лемма 13.1.** Пусть  $s_i$  и  $s_j$  - два негоризонтальных отрезка, имеющих единственную точку пересечения  $p$ , которая лежит во внутренности каждого отрезка. Тогда во множестве  $Q$  существует событие, лежащее до  $p$ , во время которого  $s_i$  и  $s_j$  становятся соседями и проверяются на пересечение.

**Доказательство.** Рассмотрим заметающую прямую чуть выше  $p$ . В этот момент отрезки  $s_i$  и  $s_j$  являются соседними в статусе. С другой стороны, в самом начале работы алгоритма заметающая прямая не пересекала ничего и статус был пуст, то есть в этот момент  $s_i$  и  $s_j$  не были соседями. Значит, где-то между началом и "чуть выше  $p$ " отрезки стали соседями и были проверены на пересечение.  $\square$

Теперь, когда мы убедились, что все работает, вновь последовательно изложим наш подход к решению:

- Пройтись по всем отрезкам горизонтальной прямой сверху вниз
- Останавливаться на точках начала, конца и пересечения отрезков
- Поддерживать список пересекающих прямую отрезков (статус)

Опишем действия, которые необходимо выполнить в точках разного типа.

#### 13.4.1 Начало отрезка

Проверить пересечение отрезка и его соседей ниже заметающей прямой, если оно есть - добавить соответствующее событие в  $Q$ .

#### 13.4.2 Конец отрезка

Удалить отрезок из  $Q$ , также проверить его соседей и добавить новое событие, если нужно.

#### 13.4.3 Пересечение отрезков

Во время пересечения отрезки меняют взаимное направление, а значит, могут поменять своих соседей. Нужно пересмотреть пересекающиеся отрезки и проверить их на пересечение с новыми соседями, опять же, добавляя события при необходимости.

### 13.5 Структуры данных

Введем порядок на точках из событий следующим образом:  $p < q$  если выполняется либо  $p.y > q.y$ , либо  $p.x == q.x$  и  $p.x < q.x$  (как бы пройдемся прямой сверху вниз слева направо). Во время работы алгоритма нам понадобится добавлять новые события и проверять наличие существующего события в очереди. Чтобы делать это быстро, используем для хранения  $Q$  любое сбалансированное дерево поиска.

Для элементов статуса также введем порядок ( $=x$ -координата пересечения  $l$ ). Здесь нам понадобится быстро добавлять элемент во множество, а также находить соседей для уже добавленных. Опять-таки воспользуемся BST для хранения  $T$ .

## 13.6 Время работы алгоритма

**Теорема 13.1.**

*Время работы алгоритма составляет  $O((n + k) \log n)$ .*

*Доказательство.* Время инициализации очереди составляет  $O(n \log n)$ . Количество событий в очереди  $O(n + k)$ , а на каждое событие случается константное количество операций добавления и удаления в  $Q$  и  $T$  сложностью  $O(\log\{n\})$ . Итого –  $O((n + k) \log n)$ .  $\square$

## 14 3 12: PSLG и DCEL: построение PSLG множества прямых FLYINGLEAFE

### 14.1 3 Определение

**Определение.** Planar straight line graph (ПЛЛГ – планарный прямолинейный граф) – плоская укладка планарного графа, в которой все ребра представлены отрезками прямой.

**Определение.** Face (грань) PSLG – это максимальное связное подмножество плоскости, не содержащее точек ребер или вершин PSLG

**Определение.** Doubly-connected edge list (РСДС – реберный список с двойными связями) – структура данных для представления PSLG. Состоит из записей трех типов: вершина, фейс и полуребро.

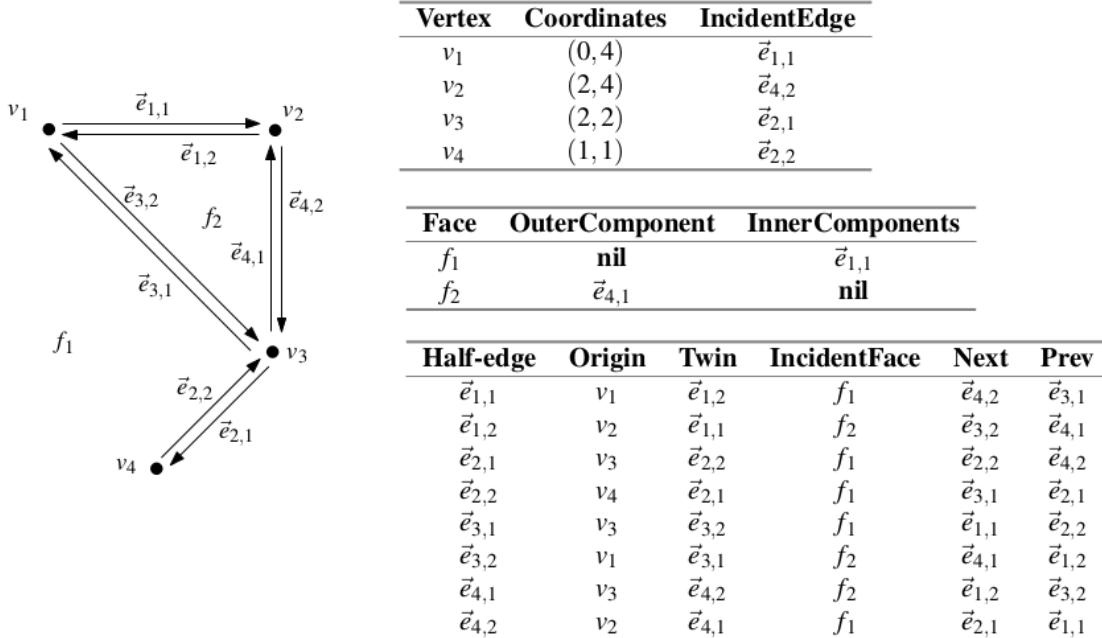


Рис. 27: Наглядная иллюстрация структуры DCEL

**Определение.** Вершина (в смысле DCEL) – это структурка данных, представляющая собой вершину PSLG в DCEL. Хранит в себе координаты точки ( $v.x$  и  $v.y$ ) и указатель на инцидентное (исходящее из этой вершины) полуребро  $v.incEdge$

**Определение.** Фейс (в смысле DCEL) – это структурка данных, представляющая собой фейс PSLG в DCEL. Хранит в себе указатель на какое-либо из своих внутренних полуребер  $face.edge$ , а так же список указателей на внешние полуребра 'дырок' (фейсов, лежащих целиком внутри данного и не связанных с остальными), если таковые имеются ( $face.holes$ ).

**Определение.** Полуребро (в смысле DCEL) – это структурка данных, представляющая собой направленное ребро PSLG в DCEL. Полу – потому что для каждого неориентированного ребра в PSLG мы храним 2 разнонаправленных полуребра. Полуребра ориентированы так, чтобы каждый фейс обходился по ним против часовой стрелки.

Полуребро содержит следующие поля:

1. Указатель на следующее полуребро  $e.next$
2. Указатель на предыдущее полуребро  $e.prev$
3. Указатель на ребро "близнец" (полуребро соседнего фейса, направленное в другую сторону и соответствующее тому же ребру)  $e.twin$
4. Указатель на вершину, из которой исходит ребро  $e.origin$
5. Указатель на фейс, которому инцидентно ребро  $e.face$

### 14.2 3 Построение PSLG по множеству прямых

Дано множество прямых на плоскости. Стоит задача восстановить из этих прямых DCEL для разбиения плоскости, которое они образуют.

Для начала докажем несколько фактов о таком разбиении.

### Теорема 14.1.

Пусть  $L$  – множество прямых, а  $A(L)$  – разбиение плоскости, задаваемое им.

1. Количество вершин в  $A(L)$  не больше, чем  $n(n - 1)/2$
2. Количество ребер в  $A(L)$  не больше, чем  $n^2$
3. Количество фейсов в  $A(L)$  не больше, чем  $n^2/2 + n/2 + 1$

Равенство выполняется тогда и только тогда, когда в  $L$  нет параллельных прямых и когда не больше 2 прямых пересекаются в 1 точке.

*Доказательство.* С точками просто – каждая прямая пересекается с не более чем  $n - 1$  другими прямыми, причем каждое пересечение учитывается 2 раза, итого  $n(n - 1)/2$ . Если какие-то точки пересечения совпадают, или какие-то прямые параллельны, это количество уменьшается.

С ребрами не сложнее – на каждой прямой не более  $n - 1$  точек пересечения, поэтому каждая прямая делится на не более чем  $n$  ребер: итого  $n^2$ .

Чтобы оценить количество фейсов, будем добавлять прямые по одной и наблюдать, сколько фейсов добавилось. Изначально был 1 фейс на всю плоскость. Пусть на шаге  $k$  мы добавили новую прямую, на которой образовалось не более  $k$  ребер. Каждое ребро поделило какой-то фейс на 2. Значит, добавилось  $k$  новых фейсов. Итого:

$$1 + \sum_{k=1}^n k = n^2/2 + n/2 + 1$$

□

Таким образом, результирующий DCEL будет занимать квадрат памяти, поэтому оптимальным будет алгоритм, работающий за квадрат. Придумаем такой!

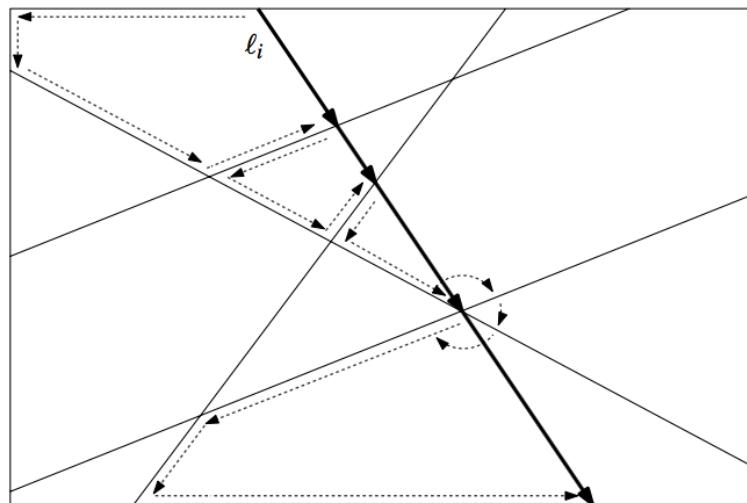


Рис. 29: Обход DCEL вдоль прямой.

Для начала заключим всю "интересную" область (ту, в которой есть пересечения) в большой прямоугольник (для этого за  $O(n^2)$  найдем все пересечения и определим крайние из них). Это мы сделаем для того, чтобы не париться о бесконечности а также о локализации первого пересечения очередной прямой.

Пусть мы добавляем очередную прямую. Найдем самое левое (верхнее, если прямая вертикальна) пересечение этой прямой с уже имеющимся DCEL – это всегда будет пересечение с внешним прямоугольником, поэтому это мы сделаем за  $O(n)$ .

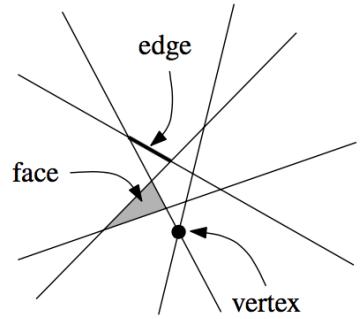


Рис. 28: Прямые формируют DCEL

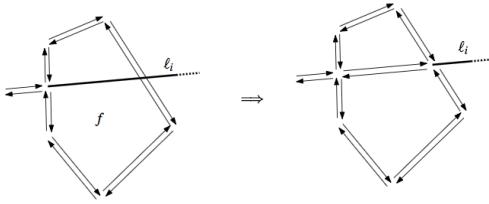


Рис. 30: Разбиение фейса прямой.

Потом мы обходим полуребра фейса, в который мы попали, пока мы не дойдем до второго полуребра, которое мы пересекли. По *twin*-ссылке переходим в следующий фейс. Если мы попали точно в ребро, обойдем инцидентные ему ребра, пока не найдем нужный фейс.

Каждый встреченный фейс мы делим на 2. Вместо пересеченных полуребер и самого фейса вставляем по два новых (как показано на иллюстрации).

### Теорема 14.2.

Этот алгоритм работает за  $O(n^2)$

*Доказательство.* Вообще, алгоритм работает за  $O(\sum_{i=1}^k m_i)$ , где  $m_i$  – количество ребер в  $i$ -ом по счету пересеченном фейсе. Всего (полу)ребер вообще –  $O(n^2)$ , но это плохая верхняя оценка, так как в итоге на все линии получается  $O(n^3)$ .

Докажем, что количество ребер в зоне, затрагиваемой вновь добавленной прямой  $l$ , составляет  $O(n)$ . Н. у. о. считаем, что прямая горизонтальна. Будем называть ребро **левым** по отношению к фейсу, если оно ограничивает его слева, аналогично **правым** – если справа. Оценим количество левых ребер, оценка на правые ребра будет симметричной.

1. База. Для DCEL из одной прямой есть 1 левое ребро – оценка  $O(n)$  работает.
2. Переход. Рассмотрим самую правую прямую, пересекающую  $l$ . Обозначим ее как  $l_1$ . Уберем ее – по индукционному предположению, количество ребер, обходимых вдоль  $l$  без  $l_1$ , составляет  $O(n)$ . Докажем, что ее добавление прибавит  $O(1)$  новых левых ребер.

Обозначим за  $v$  первое пересечение  $l_1$  с чем-нибудь сверху от  $l$ ,  $w$  – снизу. Заметим, что  $vw$  является новым левым ребром, и обе эти точки делят старые левые ребра на 2. Итого, количество левых ребер увеличилось на 3. Если же через точку пересечения  $l$  и  $l_1$  проходит еще одна прямая, то  $vw$  будет само поделено на 2 ребра, а также разделит ребро, ранее проходившее через точку пересечения, на 2. Итого – 5 новых левых ребер, больше быть не может, так как нас не интересуют новые ребра выше  $v$  или ниже  $w$  – они не войдут в зону обхода для  $l$ .

Для правых ребер доказательство симметрично. Итого при добавлении новой прямой придется обойти  $O(n)$  ребер, а значит, весь алгоритм работает за  $O(n^2)$ .  $\square$

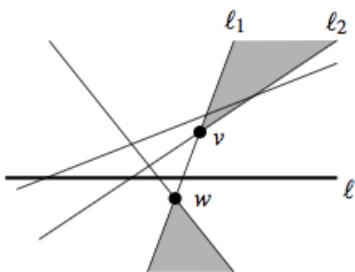
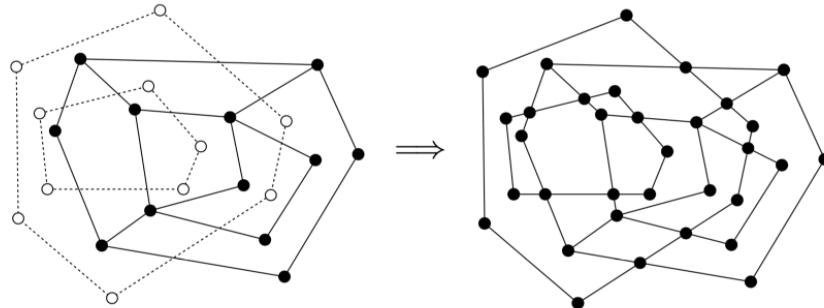


Рис. 31: Иллюстрация к доказательству асимптотики

## 15 2 13: PSLG overlaying

AVBELYY

Задача: заданы два планарных графа  $S_1$  и  $S_2$  в виде DCEL. Посчитать фигуру, полученную объединением этих двух графов.



Вершины и полуребра объединения  $O(S_1, S_2)$  мы получим, решив задачу поиска всех точек пересечения алгоритмом заметающей прямой. Фейсы построим позже, пройдясь по циклам полуребер  $O(S_1, S_2)$  и поддерживая дополнительную структуру данных в sweep line.

Сразу сформулируем теорему об итоговом времени работы и потреблении памяти алгоритма.

**Теорема 15.1.**

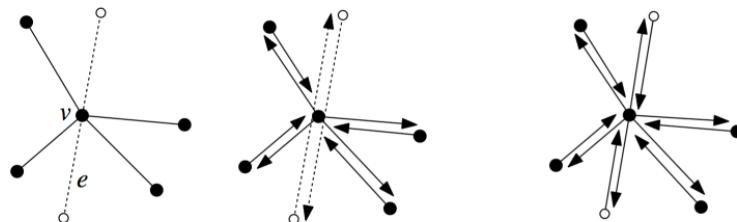
Алгоритм поиска фигуры пересечения двух планарных графов работает за  $O((n + k) \log n)$  времени и потребляет  $O(n + k)$  памяти, где  $n$  - общее количество вершин в  $S_1$  и  $S_2$ ,  $k$  - количество новых вершин.

### 15.1 Объединение вершин и полуребер

Пройдемся алгоритмом заметающей прямой по отрезкам из объединения  $S_1.halfEdge \cup S_2.halfEdge$ . Будем поддерживать event и status queue, как и раньше, а также DCEL, в который будем вставлять новые точки и ребра.

Событие заметающей прямой для нас является интересным, только если в нем участвуют полуребра разных планарных графов.

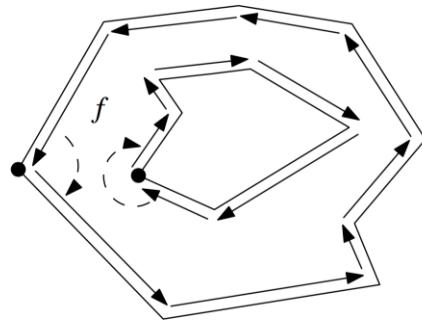
Какие бывают случаи пересечения и как их обрабатывать – понятно. Вот, например, поясняющая картинка для случая пересечения полуребра  $S_1$  и вершины  $S_2$ .



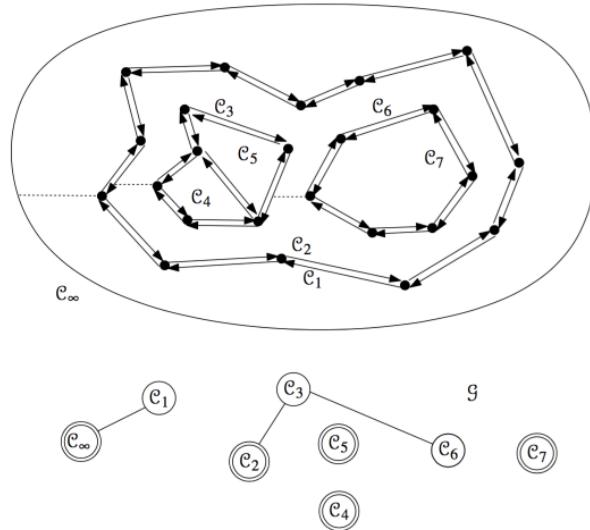
**Лемма 15.1.** Построение вершинной и полуреберной части DCEL для  $O(S_1, S_2)$  можно сделать за  $O((n + k) \log n)$ .

### 15.2 Обновление фейсов

Пройдемся по циклам и определим, какой фейс они ограничивают. Как понять, идем мы внутри или снаружи фейса? Есть один трюк: найдем самую левую (если несколько, возьмем самую нижнюю) вершинку цикла и посмотрим на угол при ней. Если он  $< pi$ , цикл внутри. Иначе – снаружи фейса.

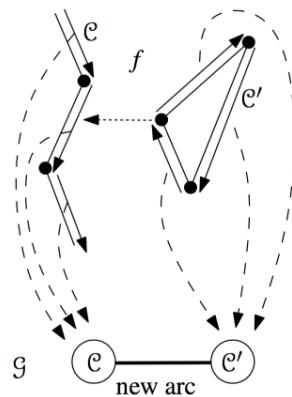


Построим специальный граф  $G$ , который поможет нам понять, какие циклы ограничивают один и тот же фейс. Вершинами  $G$  будут циклы  $O(S1, S2)$ , а ребро между вершинами проведем тогда и только тогда, когда один из циклов ограничивает дырку "извне а у другого есть полуребро, которое является ближайшим слева для самой левой вершины первого цикла (см. рисунок).



Тогда, как подсказывает нам интуиция, компоненты связности вершин в  $G$  соответствуют одному и тому же фейсу в  $O(S1, S2)$  (доказывать это не будем, помашем просто руками).

**Построение  $G$**  осуществляется во время работы заметающей прямой. Сначала создадим вершины  $G$ , соответствующие циклам. Далее, когда мы обрабатываем событие заметающей прямой, соответствующее самой левой точке  $v$  какого-то внешнего цикла, посмотрим на ближайшее слева полуребро  $e$  (оно лежит в статусе). Если таковое существует, проведем ребро между циклами для точки  $e$  и полуребра  $v$  в  $G$ .



Построенный граф  $G$  позволяет нам обновить структуру фейсов для DCEL, а так же выставить IncidentFace для полуребер.

**Лемма 15.2.** *Обновление структуры фейсов DCEL для  $O(S1, S2)$  можно сделать за линейное время от числа вершин.*

## 16 3 14: Локализация в PSLG

FLYINGLEAFE

Весь билет супер-подробно в одной статье: <http://www.link.cs.cmu.edu/15859-f07/papers/point-location.pdf> (скорее всего, не нужно)

У нас есть PSLG (представленная как DCEL). Идут запросы в виде точек. Нужно уметь быстро определять, в какой фейс (ребро?) попала точка.

### 16.1 3 Метод полос

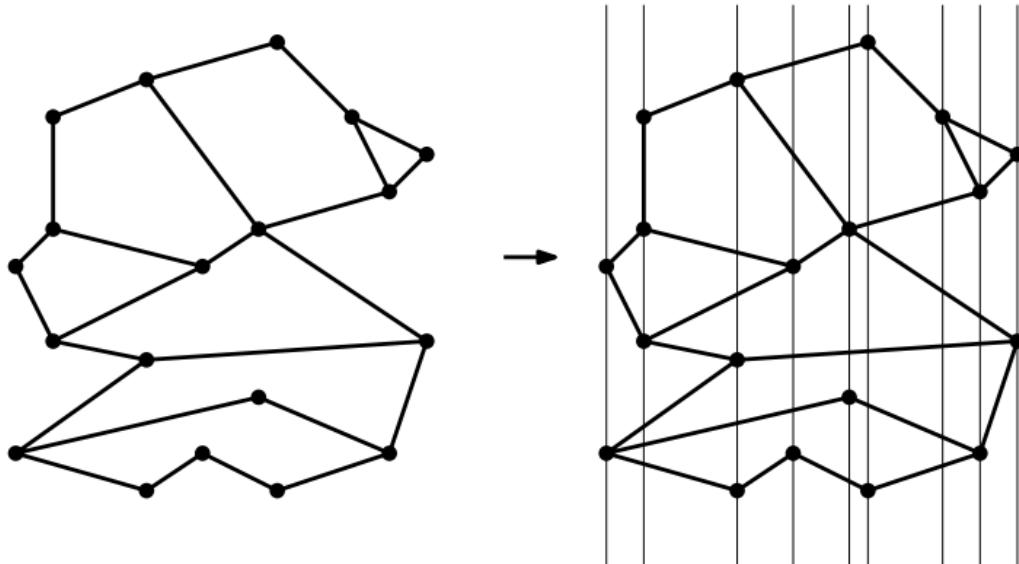


Рис. 32: Разбиение PSLG на slabs

Идея старая: давайте локализоваться сначала по  $x$ , потом по  $y$ . Как? Проведем через каждую вершину PSLG прямую, разбив ее на полоски (slabs), как на рисунке.

Отсортируем теперь эти полоски по  $x$  - координате левой границы. Теперь бинпоиском мы легко можем находить полоску, в которую попала наша точка. Заметим, что по построению ребра могут пересекаться только на границах полосок, а значит, в рамках 1 полоски все ребра вертикально упорядочены. Давайте в каждой полоске построим на на ребрах дерево поиска по  $y$  и будем радостно локализовать точку между ними за  $O(\log n)$ . Ура!

Не так быстро. Такая структура данных в худшем случае занимает  $O(n^2)$  памяти, что очень плохо. Однако, ее можно улучшить!

#### 3 Персистентные деревья

Давайте рассмотрим  $x$  - координату как **время**. Двигаясь вправо по  $x$ , мы двигаемся во времени. Пусть у нас есть дерево бинпоиска, изначально пустое. Когда мы встречаем начало отрезка, мы добавляем в него в дерево с текущей  $y$  - координатой. Когда мы встречаем конец отрезка, удаляем его из дерева. (**NB:** если отрезок(ки) лежит(ат) на вертикальной прямой, события начала/конца сортируются по  $y$ , а если один отрезок заканчивается, а другой начинается в одной и той же точке, событие начала идет раньше (ну как в Бентли-Оттмане, понятно))

Так как отрезков  $n$ , то всего событий (а значит, операций изменения дерева) всего  $2n$ . Вернемся к первоначальной задаче. Применяя к slabs метафору времени, один slab - это отрезок времени, когда ничего не происходило. То есть, на один slab приходится одна версия персистентного дерева (а не отдельное дерево бинпоиска, как раньше). С точки зрения операции поиска ничего не изменилось, а вот потребление памяти уменьшилось до  $O(n \log n)$  - так как в персистентном дереве а-ля Хаскель при операции добавления/удаления прибавляется/освобождается  $O(\log n)$  памяти (узлы на пути от корня до вставленной/удаленной вершины +  $O(1)$  на перебалансировку), а всего таких операций  $O(n)$

#### 16.1.2 3 Очень классные персистентные деревья!

Можно сделать персистентные деревья, которые занимают  $O(n)$  памяти, невероятно! Как именно? Используем тот факт, что нам не нужна **полная** персистентность (возможность менять все ревизии),

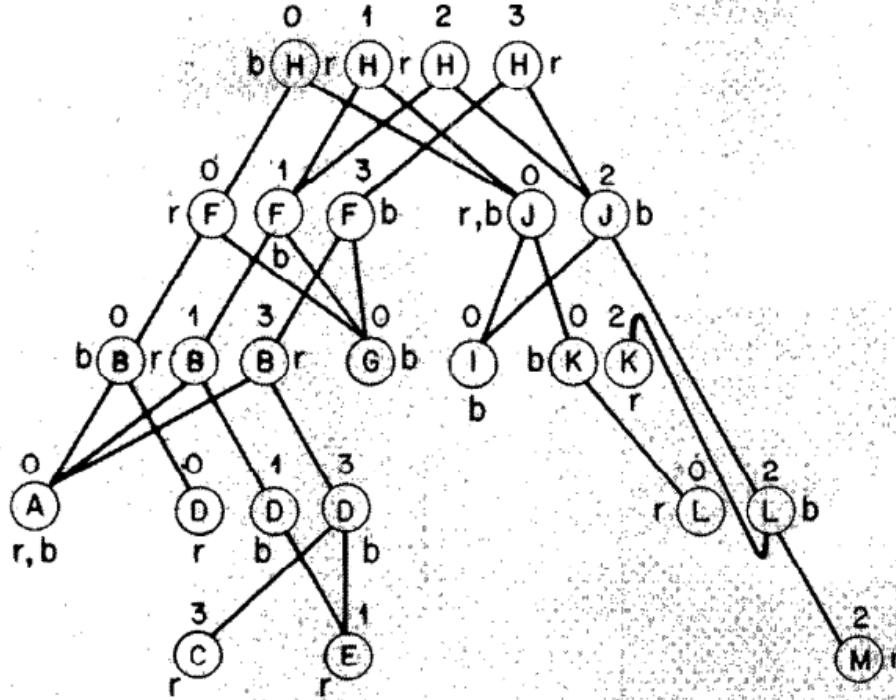


Рис. 33: Персистентное дерево с копированием путей

нам достаточно только **частичной** (можем менять и получать новые ревизии только из последней, но делать запросы можем по всем). Чтобы понять идею, попробуем сначала сделать что-нибудь попроще, но такое же модное - частично персистентный список, например.

Давайте в узле списка хранить не один указатель на следующий элемент, а 2 - next и next2. Дополнительно мы будем хранить номер **первой ревизии списка**, начиная с которой используется указатель next2. Так же мы будем поддерживать таблицу (хэшмап или массив)  $revision \rightarrow root$ .

Пусть мы хотим вставить очередной элемент в такой список между элементами  $i$  и  $i + 1$  – создать новую ревизию под номером  $k$ . Мы начинаем идти от корня, соответствующего ревизии  $k - 1$  до элемента  $i$ . Всякий раз мы выбираем соответствующий самой свежей ревизии указатель из двух (это всегда будет next2, если он не null). Пусть мы дошли до  $i$ -го узла. Если его указатель next2 пуст, мы создаем новую вершину, указатель next которой мы подвешиваем на  $i + 1$ , а указатель  $i.next2$  подвешиваем к новой вершине. В противном случае нам придется скопировать  $i$  и всех его предков до тех пор, пока мы не встретим предка со свободным указателем next2.

Такую же тактику применим в деревьях: добавим в каждый узел дерева по дополнительному указателю next2 и номер ревизии + флагок направления: влево или вправо смотрит next2. Балансировочную информацию (размер поддерева или там цвет вершины) мы будем ненадежно перезаписывать – мы все равно не собираемся менять старые ревизии.

**Лемма 16.1.** Частично персистентное дерево с limited node copying занимает  $O(n)$  памяти

**Доказательство.** В худшем случае, конечно же, нам придется копировать  $O(\log n)$  узлов. Но мы с амортизируем эту оценку. Заметим, что копируем мы только те узлы, в которых уже занят указатель next2. Давайте будем платить 2 монетки за обновление указателя next2 и ревизии в узле: одну за саму операцию, а другую отложим в узел про запас. Таким образом, в каждом заполненном узле будет лежать запасенная монетка. Когда нам нужно будет скопировать этот узел, мы потратим только уже отложенные монетки.

Таким образом, амортизированная оценка для дополнительной памяти на операцию изменения в дереве –  $O(1)$ , а так как операций изменения  $O(n)$ , то и памяти всего требуется  $O(n)$   $\square$

## 16.2 3 Киркпатрик

Алгоритм Киркпатрика работает с PSLG, представляющими собой триангуляцию (вообще говоря, за  $O(n \log n)$  заметающей прямой можно триангулировать любой PSLG, так что все ок)

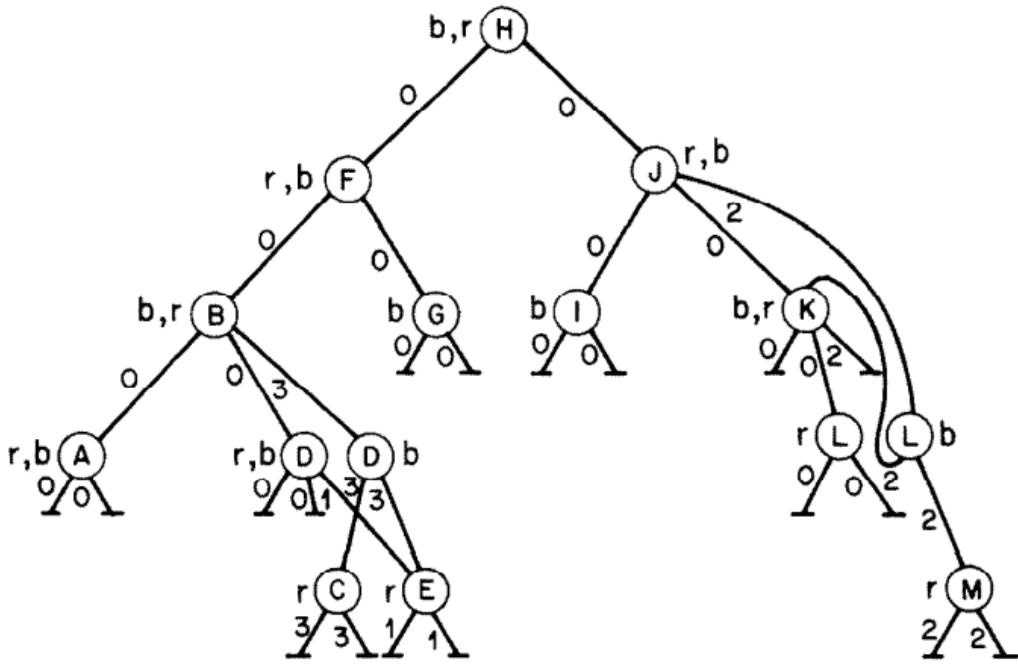


Рис. 34: Частично персистентное дерево с limited node copying

Идея: давайте сделаем последовательность триангуляций, где первая триангуляция совсем простая, а каждая следующая – посложнее, но ненамного. Каждый треугольник в простой триангуляции знает, какие треугольники в более сложной он пересекает. Локализовавшись в простой триангуляции, мы сможем спускаясь все ниже и ниже уровнем дойти до искомой триангуляции и локализоваться в ней.

Вопрос только в том, как строить эти уровни так, чтобы их было не слишком много и чтобы переходить от уровня к уровню было не слишком сложно.

#### Алгоритм

Пусть  $S_1$  – данная триангуляция с  $n$  вершинами (Мы предполагаем, что ее внешняя грань представляет собой треугольник, если это не так, охватывающий треугольник мы можем построить сами). Построим последовательность  $S_1, S_2, \dots, S_{h(n)}$  упрощающихся триангуляций, переходя от  $S_{i-1}$  к  $S_i$  таким образом:

1. Удалим из  $S_{i-1}$  некоторое множество попарно несмежных вершин, лежащих не на границе, вместе с инцидентными ребрами. Какие именно вершины удалять – в этом тонкость.
2. Триангулируем оставшиеся многоугольники.

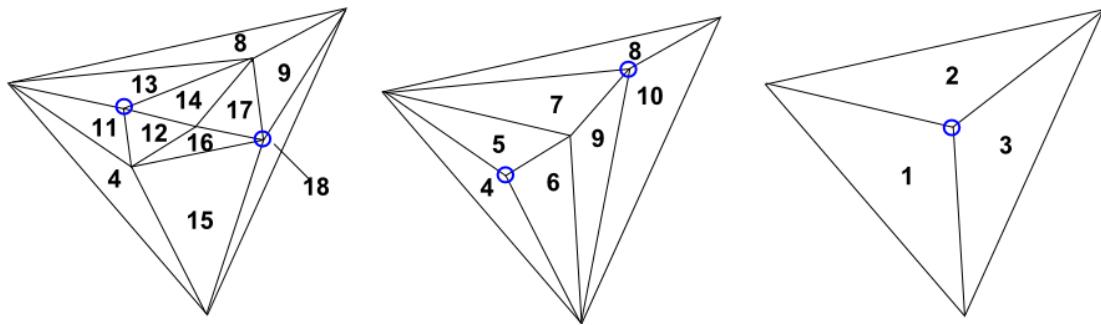


Рис. 35: Последовательность упрощающихся триангуляций.

Поисковая структура представляет из себя ациклический ориентированный граф, состоящий из 'слоев'. Каждый слой соответствует некоему  $S_k$ . Узел из слоя  $S_i$  имеет ссылки на те треугольники в  $S_{i+1}$ , которые она пересекает.

Но как правильно выбирать удаляемые вершины? Пусть мы умеем удалять вершины так, что для любого  $i > 1$  выполняются следующие свойства:

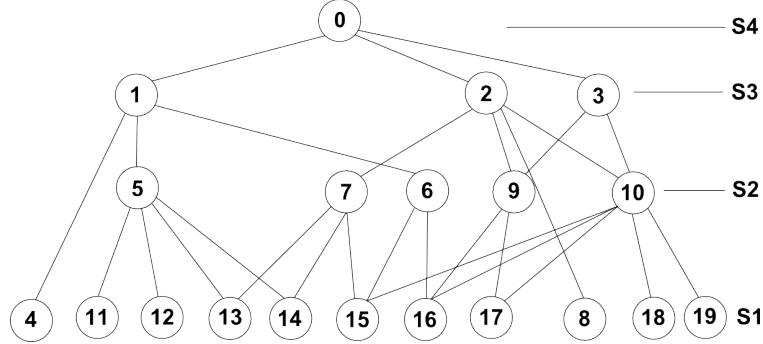


Рис. 36: Поисковая структура данных для алгоритма Киркпатрика.

1.  $|S_i| = a_i |S_{i-1}|$ , где  $a_i \leq a < 1$
2. Каждый треугольник в  $S_i$  пересекается не более чем с  $H$  треугольниками в  $S_{i-1}$  и наоборот.

Первое свойство означает, что количество треугольников всякий раз уменьшается не менее чем в  $a$  раз, что влечет логарифмическую оценку на количество слоев:  $h(n) = O(\log n)$ . Второе же свойство означает  $O(1)$  на перебор всех треугольников более богатого слоя, пересекающихся с текущим. Кроме того, это ограничивает используемый объем памяти до  $O(n)$ :

$$\sum_{i=1}^{h(n)} |S_i| \leq |S_1| \cdot (1 + a + a^2 + \dots + a^{h(n)-1}) \leq |S_1| \cdot \frac{1}{1-a} = O(n);$$

так как число граней ( $|S_1|$ ) линейно по отношению к числу вершин, а  $\frac{1}{1-a}$  – это константа. Нужно еще учесть, что в каждом узле содержится сколько-то указателей, но их не больше, чем  $H$  – константное число, поэтому оценка не портится.

Как поддержать такие свойства? Докажем следующую теорему:

### Теорема 16.1.

*Если при построении очередного слоя триангуляции удалять несмежные вершины со степенью меньше некоторого  $K$ , то эти свойства будут выполняться.*

**Доказательство. Какие-то сложные доказательства** Число вершин –  $n$ , значит, по формуле Эйлера, с учетом того, что все грани имеют по 3 ребра, верно  $E = 3n - 6$ . Так как каждое ребро инцидентно ровно 2 вершинам, и к степени каждой из них оно добавит единичку, то сумма степеней всех вершин  $\leq 6n$ . Это значит, что количество вершин, степень которых не меньше 12, не может быть больше  $\frac{n}{2}$ , а значит, количество вершин со степенью меньше 12 не меньше, чем  $\frac{n}{2}$ .

Пусть мы хотим выбрать для удаления все несмежные вершины со степенью меньше  $K = 12$ . Обозначим число выбранных вершин как  $v$ . Поскольку мы выбираем несмежные вершины, каждая выбранная вершина исключает из дальнейшего рассмотрения максимум 11 вершин, смежных с ней. Таким образом, в худшем случае мы сможем выбрать одну вершину из 12 подходящих (то есть имеющих степень меньше  $K$ ). Кроме того, мы не можем выбрать 3 граничные вершины. Поэтому можем оценить  $v$  снизу так:  $v \geq \lfloor \frac{1}{12} (\frac{N}{2} - 3) \rfloor$ . Получается, коэффициент уменьшения  $a \approx 1 - \frac{1}{24} < 1$ , что подтверждает первое свойство.

Второе свойство выводится тоже просто: при удалении вершины степени меньше  $K$  и всех инцидентных ей ребер получится многоугольник с числом ребер меньше  $K$ , который будет триангулирован на менее чем  $K - 2$  треугольника. Таким образом, любой новый треугольник (из триангуляции этого многоугольника) будет пересекать не более  $K$  старых, и наоборот.  $\square$

### Теорема 16.2.

*Алгоритм Киркпатрика требует  $O(\log n)$  времени на запрос,  $O(n)$  памяти и  $O(n \log n)$  времени на препроцессинг.*

**Доказательство.** Оценки на память и время запроса мы доказали выше. Чтобы доказать оценку на препроцессинг, нужно доказать, что построение очередного слоя работает за  $O(n)$ : так как всего слоев  $O(\log n)$ , это повлечет за собой искомую оценку.

Выбрать все вершины для удаления мы можем, очевидно, за  $O(n)$ . После удаления вершин останется  $v$  многоугольников, которые нужно триангулировать. Однако заметим, что все эти треугольники – ‘звездные’, поэтому каждый из них можно триангулировать за  $O(m)$ , где  $m$  – число вершин в многоугольнике. Суммарное число вершин в многоугольниках –  $O(n)$ , поэтому они все будут триангулированы за  $O(n)$ .  $\square$

## 17 3 15: Трапецидная карта

FLYINGLEAFE

### 17.1 3 Определение и основные свойства

Трапецидная карта - это структура данных, позволяющая локализовать точку в PSLG (вообще - просто в куче непересекающихся отрезков) за  $O(\log n)$  и занимающая  $O(n)$  памяти.

Что собой представляет трапецидная карта? Из каждой вершины PSLG выпустим вверх и вниз вертикальные лучи до тех пор, пока не упремся в какое-нибудь ребро. Чтобы лучи не получились бесконечными, окружим все PSLG большим прямоугольником  $R$ . Получится трапецидная карта.

Здесь и далее мы допускаем, что в PSLG не встречается вершин, лежащих строго вертикально друг над другом. Далее мы покажем, как избавиться от этого допущения.

**Лемма 17.1.** Любой face трапецидной карты ограничен 1 отрезком сверху, одним снизу и 1 или 2мя вертикальными - по бокам. (То есть, face трапецидной карты - это трапеция или прямоугольник)

**Доказательство.** Пусть фейс ограничен сверху более чем 1 невертикальным отрезком. Но отрезки изначально не пересекаются, значит, они встречаются в вершине. Тогда из вершины должен был по построению опущен луч, разделивший бы рассматриваемый фейс. Значит, и сверху и снизу есть только 1 ограничивающий отрезок. Ну а с вертикальными сторонами (лучами) все и так ясно.  $\square$

**Лемма 17.2.** Трапецидная карта на  $n$  отрезках содержит максимум  $6n + 4$  вершины и  $3n + 1$  трапецид.

**Доказательство.** Вершины карты - это вершины отрезков ( $2n$ ) плюс вершины охватывающего прямоугольника  $R$  (4), плюс вершины, получившиеся в результате пересечения вертикальных лучей с отрезками ( $2 \cdot 2n = 4n$ , так как есть 2 луча из каждой вершины, каждый из которых оборвется на первом пересечении с отрезком). Итого  $6n$ .

Теперь подсчитаем количество трапецидов (фейсов). Заметим, что каждому трапециду  $\Delta$  можно сопоставить точку  $leftp(\Delta)$ , определяющую его левую границу. Эта точка может быть либо левым нижним углом  $R$ , либо концом какого-либо отрезка. Левый нижний угол  $R$  относится только к одному трапециду. Правый конец какого-либо отрезка может быть  $leftp(\Delta)$  только для одного трапецида, а левый конец - максимум для двух. (Чтобы понять, почему это так, посмотрите на иллюстрацию со всеми возможными расположениями  $leftp(\Delta)$ . Итого  $3n + 1$  трапецид).  $\square$

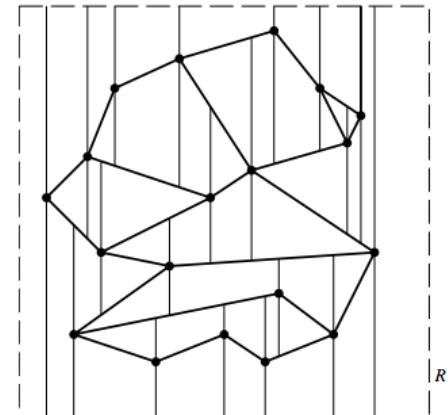


Рис. 37: Трапецидная карта.

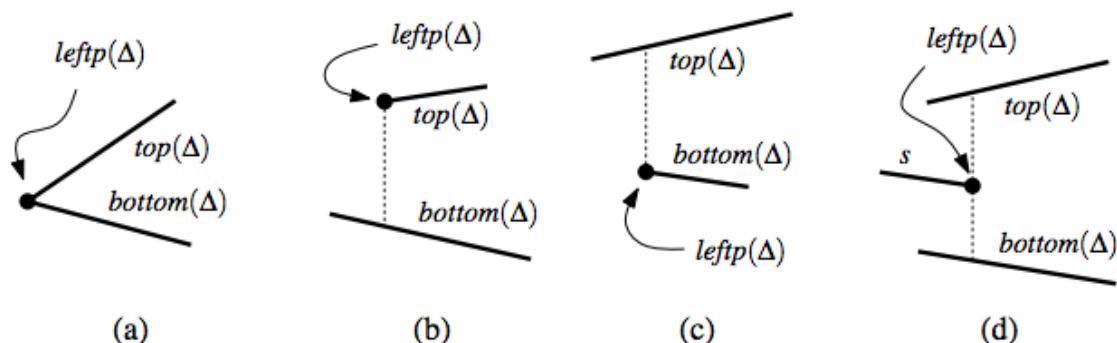


Рис. 38: Всевозможные конфигурации левой границы трапецида

В структуре данных для трапецида  $\Delta$  будем хранить следующие поля:

1. Указатели на вершины, определяющие левую и правую границы трапецида:  $leftp(\Delta)$ ,  $rightp(\Delta)$
2. Указатели на ребра, являющиеся верхней и нижней границами трапецида:  $top(\Delta)$ ,  $bottom(\Delta)$
3. Указатели на всех соседей:  $prev(\Delta)$ ,  $next(\Delta)$ ,  $up(\Delta)$ ,  $down(\Delta)$

Вся карта - это просто список таких трапецидлов.

Для эффективной локализации точки в карте мы будем строить специальную структуру, которая представляет из себя 'почти дерево' – ациклический ориентированный граф, где у каждой вершины 2 исходящих ребра, кроме листьев, которые являются трапецидловами.

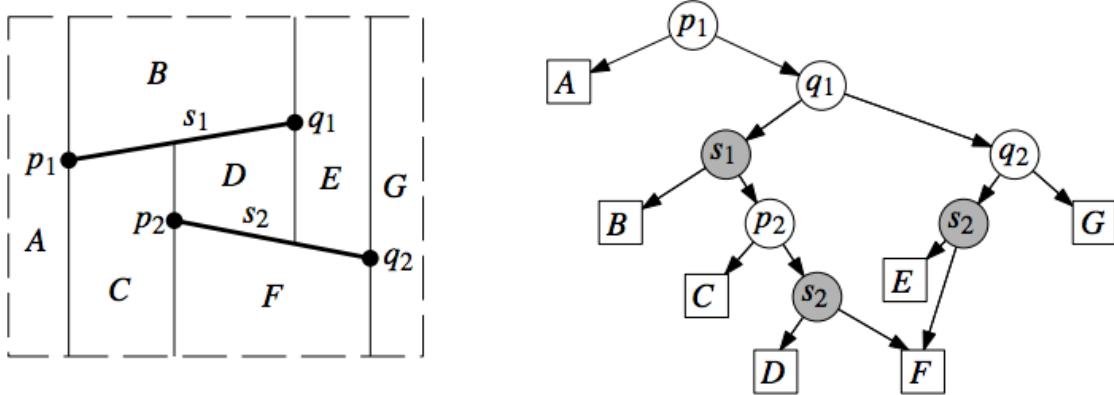


Рис. 39: Поисковая структура для трапецидной карты

Внутренние узлы графа могут быть 'вершинами' или 'ребрами'. Узел-'вершина' разделяет множество трапецидлов по  $x$ , 'ребро' – по  $y$ . Уже понятно, как локализовать точку в такой структуре: топаем, начиная от корня, в ту сторону, с какой стороны точка запроса лежит от соответствующей вершины или ребра. Такой запрос отработает за  $O(h)$ , где  $h$  - глубина графа.

Однако, глубина этого графа может быть самой разной, вплоть до  $O(n)$ , в зависимости от того, в каком порядке его строить. В следующем подразделе мы с этим разберемся.

## 17.2 3 Алгоритм построения карты и поисковой структуры

Нам нужно как-то минимизировать глубину поисковой структуры для трапецидной карты. Для этого мы помахаем руками и докажем, что для случайного набора отрезков в случайному порядке глубина так и так выйдет небольшой.

### 17.2.1 Алгоритм

Будем строить карту и поисковой граф инкрементально: с самого начала карта состоит из одного трапецидла –  $R$ , а граф состоит из одного листа. Будем добавлять отрезки по одному, предварительно рандомно их перемешав.

Чтобы добавить отрезок, надо предварительно определить, какие трапецидлы он пересек. Делаем это так:

```

Function IntersectSegment( $T, D, s$ )
  Data: Трапецидная карта  $T$ , поисковая структура для нее  $D$ , отрезок  $s$ 
  Result: Список трапецидлов, пересекаемых отрезком
   $p, q \leftarrow$  левый и правый концы  $s$ , соответственно;
   $\Delta_0 \leftarrow$  трапецид, в котором мы локализовали  $p$  (с помощью  $D$ );
   $j \leftarrow 0$ ;
  while  $q$  находится справа от  $rightp(\Delta_j)$  do
    if  $s$  выше  $rightp(\Delta_j)$  then
      |  $\Delta_{j+1} \leftarrow$  правый верхний сосед  $\Delta_j$ 
    else
      |  $\Delta_{j+1} \leftarrow$  правый нижний сосед  $\Delta_j$ 
    end
     $j \leftarrow j + 1$ ;
  end
  return  $\Delta_0, \dots, \Delta_j$ 
```

**Алгоритм 6:** Нахождение трапецидлов, пересекающихся с данным отрезком

**Лемма 17.3.** Этот алгоритм работает за  $O(h + k)$ , где  $h$  – глубина  $D$ ,  $k$  – размер ответа.

**Доказательство.** Очевидно: мы локализуем  $p$  за  $O(h)$  и переходим к каждому следующему трапецидлу за  $O(1)$ .  $\square$

После того, как мы нашли трапециды, их нужно удалить и на их место поставить новые, как в  $T$ , так и в  $D$ .

Сначала рассмотрим простой случай: отрезок полностью содержится в 1 трапециде  $\Delta$ . Тогда нам нужно сделать из этого трапецида 4, приставить в них и в отрезке соответствующие указатели на соседей, обновить в соседях указатели, которые раньше ссылались на  $\Delta$ , и заменить в  $D$  лист, соответствующий  $\Delta$  на поддерево, как показано на рисунке. Это делается, очевидно, за  $O(1)$

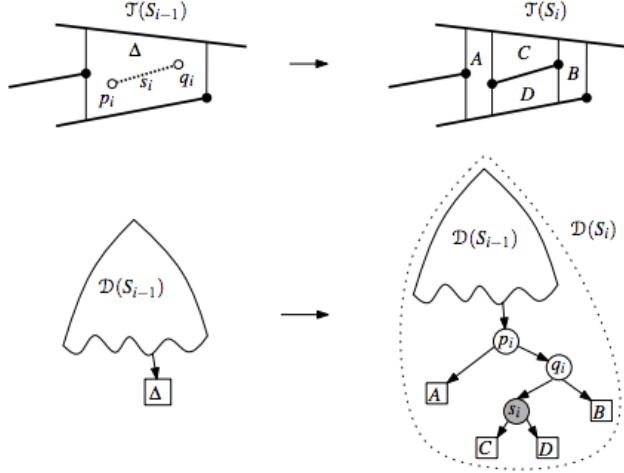


Рис. 40: Обновление трапецидной карты и поискового графа в простом случае

В случае пересечения с несколькими трапецидами, на самом деле, все примерно так же, надо только по-разному обработать 3 случая:

1. левый конец отрезка лежит в трапециде
2. правый конец отрезка лежит в трапециде
3. отрезок полностью пересекает трапецид

Кроме того, так как отрезок  $s$  обрубит некоторые вертикальные лучи, придется померджить какие-то трапециды вдоль него. Смотрим картинку.

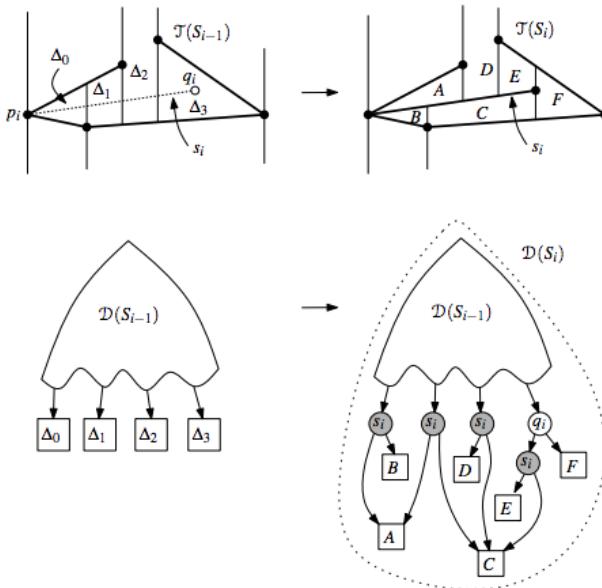


Рис. 41: Обновление трапецидной карты и поискового графа в сложном случае

Чтобы правильно мерджить трапециды вдоль отрезка в один, определим, с какой стороны от  $s$  лежит  $rightp(\Delta_0)$ . Н. у. о. она лежит сверху, поэтому снизу остается трапецид, который продолжится вдоль отрезка. Расставим корректно все ссылки, за исключением того, что у нижнего трапецида оставим  $rightp(\Delta_{low}) = \text{null}$ . Запомним его. При обработке следующего трапецида будем знать, что снизу

лежит  $\Delta_{low}$ , и расставим ссылки соответствующе. Будем повторять так, пока не встретим  $rightp(\Delta_j)$ , лежащую снизу от  $s$  – она определит правую границу  $\Delta_{low}$ . С трапецидами, тянувшимися сверху, поступаем аналогично.

В  $D$  же листы, соответствующие пересеченным трапецидам, заменим на 'поддеревья': если  $\Delta_j$  разбился на 2 трапецида  $A$  и  $B$ , то заменим его лист реберным узлом, относящимся к  $s$  и ссылающимся на листы  $A$  и  $B$ . Если на три –  $A, B, C$  – то заменим на деревце из вершинного и реберного узлов (см. картинку).

Все это делается за  $O(k)$ . Итого, процедура локализации нового отрезка и его вставки займет  $O(h + k)$  времени.

Так мы построим корректную карту и граф поиска, потому что на каждом шаге они корректны для текущего подмножества. Замечательно, осталось доказать, что это будет не слишком долго.

### 17.2.2 Асимптотика

В худшем случае все это работает очень плохо: при добавлении каждого отрезка глубина  $D$  может увеличиться максимум на 3 (в случае попадания отрезка целиком в трапецид), поэтому worst-case оценкой на глубину  $D$  является  $3n$ , что влечет  $O(n)$  на запрос и  $O(n^2)$  на построение. Отвратительно. Но если включить теорвер, то выяснится, что в большинстве случаев все гораздо лучше.

Давайте зафиксируем множество отрезков  $S$  и некую точку запроса  $q$ . Обозначим за  $S_1, \dots, S_n$  все префиксы списка  $S$ . Обозначим за  $X_i$  количество вершин на пути поиска  $q$ , созданных на итерации  $i$  алгоритма построения трапецидной карты.  $X_1, \dots, X_n$  – это случайные величины, и матожидание длины пути через них можно выразить так:

$$E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i]$$

Мы знаем, что при добавлении отрезка глубина  $D$  увеличивается не более, чем на 3, следовательно,  $X_i \leq 3$ . Значит, если мы обозначим за  $P_i$  вероятность встретить на пути поиска вершину, созданную на  $i$ -ой итерации, будет верно следующее:

$$E[X_i] \leq 3P_i$$

Нам осталось как-то оценить  $P_i$ . Подумаем, в каком случае вершина, созданная на  $i$ -ой итерации, попадет в путь поиска точки  $q$ . Рассмотрим частичную трапецидную карту  $T(S_{i-1})$ . Пусть точка  $q$  в ней локализуется в трапециде  $\Delta_q(S_{i-1})$ . В каком случае на  $i$ -ой итерации путь поиска точки  $q$  изменится? В том и только в том, если трапецид  $\Delta_q(S_{i-1})$  будет подразбит на этой итерации, и точка попадет в новый трапецид  $\Delta_q(S_i)$ , поменьше.

Так какова вероятность того, что трапецид  $\Delta_q(S_i)$  был создан именно на  $i$ -ой итерации? Применим анализ 'задом-наперед': пусть дана трапецидная карта, построенная на множестве отрезков  $S_i$  и удаляется случайный отрезок  $s_i$ . Какова вероятность того, что  $\Delta_q(S_i)$  при этом исчезнет?

Он может исчезнуть в 4 случаях:

1.  $s_i = top(\Delta_q(S_i))$
2.  $s_i = bottom(\Delta_q(S_i))$
3.  $leftp(\Delta_q(S_i))$  - конец  $s_i$
4.  $rightp(\Delta_q(S_i))$  - конец  $s_i$

Значит, существуют 4 отрезка, удаление которых приведет к исчезновению  $\Delta_q(S_i)$ , а значит, вероятность его исчезновения ( $\$ = P_i \$$ ) –  $\frac{4}{i}$ .

Таким образом, если ожидаемую длину пути поиска обозначить за  $L_q$ , верно следующее:

$$L_q \leq \sum_{i=1}^n 3P_i = \sum_{i=1}^n \frac{12}{i} \approx 12 \cdot \ln n = O(\log n)$$

Последний переход основан на том, что суммы гармонического ряда сверху и снизу ограничены логарифмом (известный факт из матана, вы что, его не ботали?).

Ура! Мы доказали, что ожидаемая глубина графа поиска, а значит, и время запроса –  $\$O(\log n)\$!$   
**Память**

Осталось доказать, что эта структура данных весит не очень много. Сама трапецидная карта  $T$  занимает  $O(n)$  памяти, потому что трапецидов  $O(n)$ . А вот с графиком поиска  $D$  все чуть сложнее. Если нам не повезет, и на каждой итерации мы будем отрезком пересекать все имеющиеся трапециды, то мы на этой итерации будем создавать  $O(n)$  узлов, что выльется в  $O(n^2)$  по памяти. Но это худший случай, интересна ожидаемая оценка.

Обозначим количество вершин в  $D$ , добавленных на  $i$ -ой итерации построения, как  $k_i$ . Тогда ожидаемый объем памяти для хранения  $D$  составит  $O(n) + \sum_{i=1}^n E[k_i]$ .

Для нахождения  $E[k_i]$  опять применим мышление задом-наперед. Введем следующую функцию:

$$\$ \delta(\Delta, s) = \begin{cases} 1, & \text{if } \Delta \text{ disappears when } s \{ \text{ is removed} \} \\ 0, & \text{otherwise} \end{cases} \$$$

Известно, что для любого  $\Delta$   $\delta(\Delta, s) = 1$  не более, чем для 4 отрезков  $s$  (соседних с этим трапецидом с одной из 4 сторон). Значит,

$$\sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq 4|T(S_i)| = O(i)$$

Теперь запишем формулу для  $E[k_i]$ , зная, что вероятность удаления любого отрезка -  $\frac{1}{i}$ :

$$E[k_i] = \sum_{s \in S_i} \frac{1}{i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) = \frac{1}{i} \cdot O(i) = O(1)$$

Таким образом, в среднем за 1 итерацию мы увеличим потребление памяти на  $O(1)$ , а значит, всего структура займет  $O(n)$  памяти.

Здесь же мы можем дать ожидаемую оценку на время построения. Мы знаем, что добавление одного отрезка в карту происходит за  $O(h + k)$ . Но теперь мы знаем, что в большинстве случаев  $O(h) \approx O(\log n)$ , а  $O(k) \approx O(1)$ , а значит, в среднем добавление нового отрезка произойдет за  $O(\log n)$ , что дает  $O(n \log n)$  на построение всей структуры данных.

### Замечание

В начале мы сделали допущение (довольно нереалистичное), что в PSLG нет вершин, имеющих одинаковую  $x$ -координату. От него можно избавиться, представив, что мы всю плоскость немножко скосили на эпсилон. Смоделировать такое поведение можно просто упорядочив точки не просто по  $x$ , а лексикографически: точка  $(x, y_1)$  считается правее  $(x, y_2)$ , если  $y_1 > y_2$ . Из за этого частенько будут возникать вырожденные трапециды, ну и ладно, зато ничего не будет ломаться.

## 18 3 16: Вращающиеся калиперы

VOLHOVM

Вращающиеся калиперы – это несложный паттерн проектирования различных алгоритмом, требующих последовательного обхождения выпуклых многоугольников в  $\mathbb{R}^2$ . Рассмотрим применение метода сразу на практической задаче.

Пусть дано некоторое множество точек. Определим его диаметр как максимальное расстояние между какими-либо двумя точками. Покажем, как можно найти диаметр этого множества.

**Лемма 18.1.** *Диаметр множества лежит на выпуклой оболочке этого множества*

**Доказательство.** Очевидно от противного: пусть мы нашли диаметр множества  $P - ab$ , причем, не теряя общности,  $b \notin CH(P)$ . Тогда утверждается, что можно пропустить луч  $ab$  и посмотреть, в какой точке он пересечет выпуклую оболочку. Легко показать, что как минимум одна точка, формирующая ребро выпуклой оболочки, пересеченное лучем, имеет дистанцию до  $a$  больше  $dist(a, b)$ .  $\square$

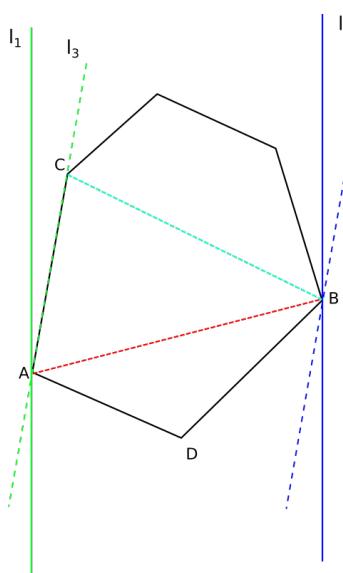


Рис. 42: Поиск диаметра множества точек с помощью вращающихся калиперов

Алгоритм поиска таков: для начала найдем минимальную и максимальную точку среди точек выпуклой оболочки (лексикографически)  $A$  и  $B$ . Мысленно создадим две вертикальные параллельные прямые, проходящих через соответствующие точки –  $l_1$ ,  $l_2$ . Будем заворачивать калиперы по часовой стрелке, поэтому условимся, что  $l_1$  смотрит вверх, а  $l_2$  вниз. Добавим текущие точки, на которых стоят параллельные прямые, в список ребер ответа ( $\langle A, B \rangle$ ). Сравним угол между  $l_1$  и  $AC$  с углом между  $l_2$  и  $BD$ . Выберем меньший и перейдем к следующей точке относительно выбранной, повернув при этом соответствующую прямую на величину угла, так, чтобы прямая теперь совпадала с  $AC$ . Будем всегда поддерживать параллельность калиперов (прямых), поэтому вторую прямую тоже мысленно повернем. Получим две новые прямые  $l_3$ ,  $l_4$ . Перейдем в начало процесса, добавив  $\langle C, B \rangle$  в список ребер ответа. Когда алгоритм придет в начальное положение, на выходе получим список ребер, среди которых будет исскомое. Переберем их за линию и найдем максимальное (можно делать это in-place во время алгоритма, формируя один большой fold).

В реализации углы сравниваются поворотами, и функция хранит только одно доминирующее ребро, на котором калипер "лежит" полностью, а второй калипер параллелен первому и соответствует некоторой точке. Исключение составляет первый шаг, на котором калиперы могут лежать исключительно на точках – можно добавить фиктивное ребро с координатами  $(x, y+1)$  относительно минимальной или максимальной точки.

**Лемма 18.2.** *Алгоритм поиска диаметра работает корректно, среди найденных ребер найдется диаметр.*

**Доказательство.** Докажем от противного. Пусть среди всех пар точек, формирующих ребра, нет нужной. Отметим, что метод выдает все ребра, для которых верно, что существует пара параллельных прямых, не пересекающих многоугольник, проходящих через эти точки. Очевидно, что если для двух точек такие прямые не построить, то и диаметр на них лежать не может – легко показать, что взяв соседнюю, мы увеличим расстояние между ними (следует из непараллельности, надо аккуратно посмотреть углы).

Отсюда диаметр лежит в классе пар точек, на которых прямые строятся. Поскольку алгоритм просматривает все такие (несложно показать), то диаметр будет лежать среди ребер ответа.  $\square$

Кроме уже рассмотренной, метод вращающихся калиперов может быть использован для решения следующих задач:

1. Поиск расстояния между двумя многоугольниками. Абсолютно аналогично проводим алгоритм для поиска диаметра (в инициализации у более левого/нижнего берем минимальную точку, а у другого максимальную), но добавляем в список ответа пару  $\langle$  доминирующее ребро, вершина  $\rangle$ .
2. Поиск двух общих касательных у выпуклых многоугольников. В инициализации у обоих полигонов берем минимальные точки, калиперы сонаправлены. В момент, когда калиперы меняются местами (один становится выше другого), мы проходим точку касания.
3. Поиск суммы Минковского двух объектов.

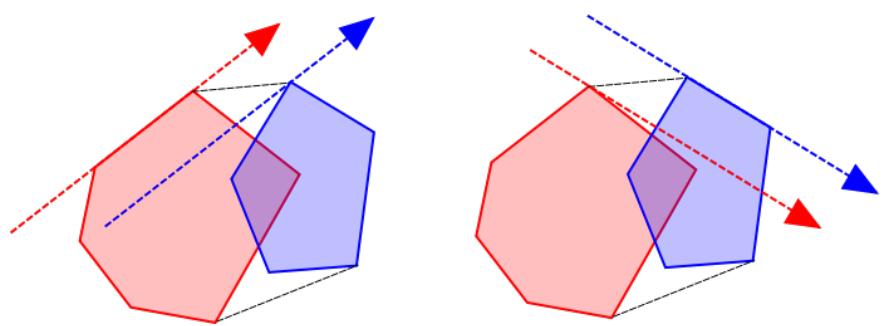


Рис. 43: Момент смены положения калиперов в поиске общих касательных

Для решения проблемы планирования движения для неточечного объекта (скажем, выпуклого полигона) используется подход расширения препятствий. Более формально: пусть  $A$  – агент с выделенной точкой внутри, а  $C$  – препятствие. Пусть  $A(x, y)$  – это многоугольник, полученный параллельным переносом агента так, чтобы его центр находился в  $(x, y)$ . Тогда расширенное препятствие формализуется так:

$$\{(x, y) : A(x, y) \cap C \neq \emptyset\}$$

**Определение** (Сумма Минковского). *Сумма Минковского двух многоугольников  $A$  и  $B$  есть*

$$A \oplus B \equiv \{p + q : p \in A, q \in B\}$$

, где сумма точек в привычном понимании по координатам.

**Лемма 19.1.** *Пусть  $A$  – агент (выпуклый многоугольник),  $P$  – препятствие. Тогда раздвинтое препятствие для  $P$  есть  $P \oplus (-A(0, 0))$*

*Доказательство.* Покажем, что  $A(x, y) \cap P \neq \emptyset \Leftrightarrow (x, y) \in P \oplus (-A(0, 0))$ .

Пусть  $A(x, y) \cap P \neq \emptyset$ , рассмотрим точку  $q$  в пересечении. По определению пересечения,  $q \in A(x, y)$ , а значит  $(q_x - x, q_y - y) \in A(0, 0)$ , что эквивалентно  $(x - q_x, y - q_y) \in -A(0, 0)$ . Также из пересечения следует  $q \in P$ , а значит следствие вправо доказано.

Обратно: пусть  $(x, y) \in P \oplus (-A(0, 0))$ . Тогда существуют такие точки  $(r_x, r_y) \in A(0, 0)$ ,  $(p_x, p_y) \in P$ , что  $(x, y) = (p_x - r_x, p_y - r_y)$ . Тогда по определению есть пересечение.  $\square$

**Теорема 19.1** (Свойства суммы Минковского выпуклых многоугольников).

*Пусть  $P, Q$  – выпуклые полигоны, имеющие  $n$  и  $m$  ребер соответственно. Тогда многоугольник  $P \oplus Q$  выпуклый и имеет максимум  $n + m$  ребер.*

*Доказательство.* Выпуклость доказывается напрямую из определения. Для любого отрезка  $seg \in P \oplus Q$ , для каждой точки  $s = (x, y)$ , которая ему принадлежит, верно, что  $s = p + q$ , где  $p \in P, q \in Q$ . Найдем точки  $p_1, q_1 : p_1 + q_1 = seg.start$ , и  $p_2, q_2 : p_2 + q_2 = seg.end$  (они найдутся по определению). Заметим, что сумма Минковского двух сегментов есть сегмент, а значит из того, что  $[p_1, p_2] \in P$  и  $[q_1, q_2] \in Q$  следует  $[seg.start, seg.end] = seg \in P \oplus Q$ .

Для доказательства линейности суммы рассмотрим произвольное ребро  $e \in P \oplus Q$ . У него есть внешняя нормаль  $\vec{n}$ , и из выпуклости  $P$  следует, что ребро  $e$  экстремально в направлении  $\vec{n}$ . Значит, оно должно быть сгенерировано какими-то точками из  $P$  и  $Q$ , которые тоже экстремальны в этом же направлении. Более того, как минимум один многоугольник из  $P, Q$  должен иметь ребро  $e'$ , которое экстремально в этом направлении (потому что получить ребро нельзя из двух точек). Установим соответствие  $e \Leftrightarrow e'$ , и такое соответствие для каждого ребра единственно в силу единственности нормали. Итого, имеем максимум  $n + m$  ребер в сумме Минковского (ровно, если у двух многоугольников нету параллельных ребер).  $\square$

**Определение** (Псевдодиск). *Будем говорить, что пара планарных объектов (в частности полигонов)  $(A, B)$  называется парой псевдодисков, если  $A$   $B$  связано и  $B$   $A$  связано.*

**Определение** (Набор псевдодисков). *Будем называть набором псевдодисков такое множество планарных объектов  $\{P_i\}$ , что каждая пара элементов в нем является псевдодисками.*

К примеру, любые два прямоугольника со сторонами, параллельными осям, являются псевдодисками.

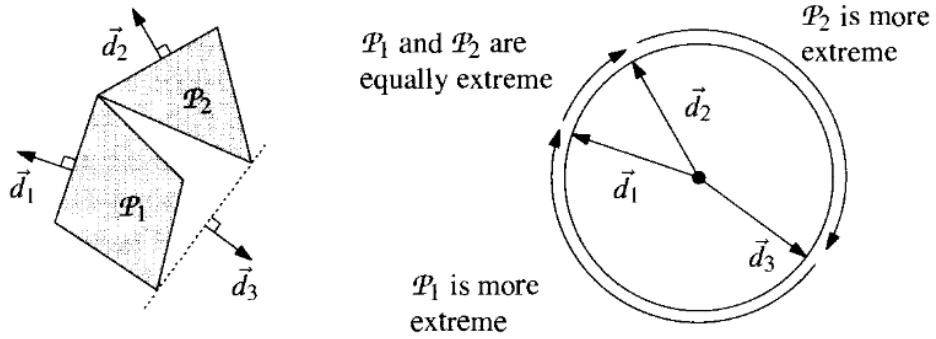
**Замечание.** Границы двух псевдодисков могут пересекаться максимум в двух точках.

Немного расширим также понятие экстремальности точки или ребра, перенеся его на многоугольник в общем.

**Определение.** *Будем говорить, что многоугольник  $A$  более экстремален, чем многоугольник  $B$  в направлении  $\vec{n}$ , если для движущейся прямой, перпендикулярной  $\vec{n}$ , движущейся в направлении  $\vec{n}$ , последняя точка, которую она пересекает при движении, принадлежит  $A$ .*

*Если в последний момент, когда прямая что-то пересекает, она пересекает оба многоугольника сразу, то будем говорить что в этом направлении экстремальность многоугольников равна.*

Поскольку экстремальность характеризуется вектором, то будем говорить, что  $P$  более экстремален  $Q$  в направлениях от  $\vec{n}_1$  до  $\vec{n}_2$ , если это верно для любого направления от  $angle(\vec{n}_1)$  до  $angle(\vec{n}_2)$  (против часовой стрелки).



**Замечание.**

Пусть  $P_1$  и  $P_2$  – выпуклые многоугольники с непересекающимися внутренностями, причем  $P$  более экстремален чем  $Q$  в направлениях  $\vec{n}_1$  и  $\vec{n}_2$ . Тогда  $P$  экстремальнее  $Q$  во всех направлениях либо от  $\vec{n}_1$  до  $\vec{n}_2$ , либо от  $\vec{n}_2$  до  $\vec{n}_1$ .

Другими словами, сектор угла, соответствующий экстремальности многоугольника, единственен и непрерывен.

### Теорема 19.2.

Пусть  $P$  и  $Q$  – два непересекающихся выпуклых многоугольника, а  $A$  – другой выпуклый полигон. Тогда суммы Минковского  $P \oplus A$  и  $Q \oplus A$  являются псевдодисками.

**Доказательство.** Обозначим  $CP := P \oplus A$ ,  $CQ := Q \oplus A$ . Докажем из определения псевдодисков, что  $CP \cap CQ$  связно (достаточно только этого, из симметричности).

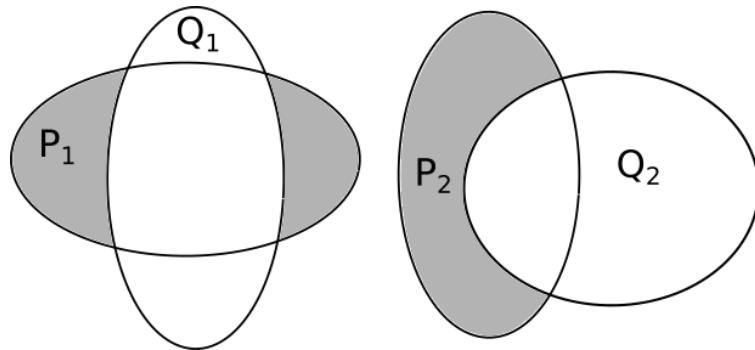


Рис. 44: Первая пара не являются псевдодисками, вторая – является

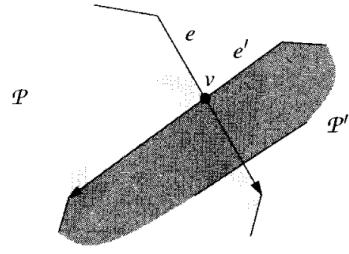
По выше доказанной теореме  $CP$  и  $CQ$  – выпуклые полигоны. Докажем от противного: пускай  $CP \cap CQ$  несвязно. Заметим, что для выпуклых многоугольников есть только один возможный вариант пересекаться и не быть псевдодисками – это иметь 4 точки пересечения на границе (см. иллюстрацию).

Заметим, что по выпуклости  $CP$  должен иметь две точки на границе выпуклой оболочки  $CP \cup CQ$ . Также он должен иметь два направления  $\vec{n}_1, \vec{n}_2$ , в которых он более экстремален чем  $CP$ . По замечанию выше полигон не может иметь два разрывных промежутка экстремальности, значит компонента связности одна. Противоречие с предположением.  $\square$

### Теорема 19.3.

Пусть  $S$  – набор псевдодисков, имеющих суммарно  $n$  ребер. Тогда объединение элементов из  $S$  имеет  $O(n)$  ребер.

**Доказательство.** Поскольку объединение полигонов – полигон, то мы можем доказать теорему, используя стандартный метод амортизационного анализа: раздадим каждой вершине из набора по 2 монеты и покажем, что для каждой вершины объединения за нее кто-то заплатит. Таким образом мы ограничим количество вершин величиной  $2n$ .



В объединении будет 2 вида вершин – вершины, которые принадлежат границе какого-то псевдодиска и вершины, образованные пересечением двух псевдодисков.

За каждую вершину первого типа заплатим одной монетой из нее самой же.

Вторые вершины из объединения делятся на две категории. Рассмотрим точку  $v$ , являющуюся пересечением ребер  $e_1 \in S_1$  и  $e_2 \in S_2$ . Ребро  $e_1$  может пересечь псевдодиск  $S_2$  либо единожды, либо два раза (в этом случае выйдет, что весь полигон  $P_2$  проходит через это ребро и заканчивается внутри полигона  $P_1$ ).

В первом случае заплатим за вершину пересечения точкой ребра  $e_1$ , лежащей внутри  $P_2$ .

Во втором случае есть вторая точка на границе  $P_2$ , в которой граница  $P_2$  пересекается с  $e_1$ . Очевидно, что ребро  $e_2$  имеет точку внутри  $P_1$ , иначе у нас полигоны пересекаются в 4x точках, что портит свойство псевдодиска. Тогда заплатим за пересечение  $v$  точкой на конце  $e_2$ , лежащей внутри  $P_1$ . Если так оказалось, что внутри  $P_1$  лежит всего одна точка из  $P_2$ , то у нас на ней две монетки – мы можем покрыть оплату обоих пересечений  $e_1$  с  $P_2$ .  $\square$

Покажем два алгоритма для получения суммы Минковского двух полигонов.

```
Data: Два выпуклых полигона  $P, Q$ 
Результат:  $P \oplus Q$ 
points ← new vector of points;
for  $p_1 \leftarrow P$  do
    for  $p_2 \leftarrow Q$  do
        | points.insert( $p_1 + p_2$ );
    end
end
return ConvexHull(points);
```

**Алгоритм 7:** Наивный алгоритм нахождения суммы Минковского двух многоугольников

```
Data: Выпуклый полигон  $P$  с точками  $p_1, \dots, p_n$ 
выпуклый полигон  $Q$  с точками  $q_1, \dots, q_m$ 
Результат:  $P \oplus Q$ 
 $i \leftarrow 1; j \leftarrow 1;$ 
 $p_{n+1} \leftarrow p_1;$ 
 $q_{m+1} \leftarrow q_1;$ 
repeat
    Добавить  $p_i + q_j$  в сумму  $P \oplus Q$ ;
     $bool_1 \leftarrow angle(p_i p_{i+1}) < angle(q_j q_{j+1})$ ;
     $bool_2 \leftarrow angle(p_i p_{i+1}) > angle(q_j q_{j+1})$ ;
    if  $\neg bool_1$  and  $\neg bool_2$  then
        |  $i++$ ;  $j++$ ;
    end
    if  $bool_1$  and  $\neg bool_2$  then
        |  $i++$ ;
    end
    if  $\neg bool_1$  and  $bool_2$  then
        |  $j++$ ;
    end
until  $i = n + 1$  and  $j = m + 1$ ;
```

**Алгоритм 8:** Поиск суммы Минковского методом вращающихся калиперов

Первый алгоритм работает достаточно плохо на больших множествах (нужно перебирать все пары точек), а второй за линейное время, так как используется метод калиперов.

**Теорема 19.4.**

*Сумма Минковского двух выпуклых тел с  $n$  и  $m$  вершинами соответственно, считается за  $O(n+m)$  времени.*

*Доказательство.* Алгоритм подсчета калиперами представлен выше.  $\square$

**Теорема 19.5.**

*Пусть  $P$  – невыпуклый многоугольник с  $n$  вершинами, а  $Q$  – выпуклый с  $m$ . Тогда количество вершин в  $P \oplus Q$  есть  $O(nm)$ .*

*Доказательство.* Заметим, что следующее равенство верно:

$$S_1 \oplus (S_2 \cup S_3) = (S_1 \oplus S_2) \cup (S_1 \oplus S_3)$$

Возьмем триангуляцию  $P$ , для каждого  $P_i$  посчитаем его сумму и объединим. Более формально:

$$P \oplus Q = \bigcup_{i=1}^{n-2} P_i \oplus Q$$

Исходя из оценки сложности суммы двух выпуклых полигонов, каждое объединение  $P_i \oplus Q$  будет иметь максимум  $m+3$  ребер. Поскольку треугольники триангуляции не пересекаются, то  $P_i \oplus Q$  есть набор псевдодисков. Объединение псевдодисков линейно по вершинам/ребрам, значит общая сложность  $P \oplus Q$  есть  $O(nm)$ .  $\square$

**Теорема 19.6.**

*Пусть  $P$  и  $Q$  – невыпуклые многоугольники с  $n$  и  $m$  вершинами соответственно. Тогда сложность  $P \oplus Q$  есть  $O(n^2m^2)$ .*

*Доказательство.* Найдем триангуляции данных многоугольников  $\{P_i\}$  и  $\{Q_i\}$ . Сложность каждого объединения  $P_j \cup Q_i$  константна, а значит  $P \oplus Q$  есть объединение  $(n-2)(m-2)$  многоугольников константной сложности (каждой пары). Отсюда следует, что сложность объединения есть  $O(n^2m^2)$  (обычная оценка, нет свойства псевдодисков).  $\square$

## 20 3 18: Вероятностный алгоритм мин.охв.окружности FLYINGLEAFE

Рассмотрим задачу: у нас есть множество  $P = \{p_1, \dots, p_n\}$  точек на плоскости. Нужно построить окружность такую, чтобы все точки из  $P$  лежали бы внутри нее или на границе, причем из таких окружностей надо выбрать минимальную.

**Идея:** строим окружность итеративно, рассматривая точки по одной. В этом нам очень поможет следующая лемма.

**Лемма 20.1** (О добавлении точки в минимальную окружность). *Определим  $P_i = \{p_1, \dots, p_i\}$ , а  $D_i$  - мин. охват. окружность для  $P_i$ . Рассмотрим точку  $p_i$ . Верно следующее:*

1. Если  $p_i \in D_{i-1}$ , то  $D_i = D_{i-1}$
2. Иначе  $p_i$  лежит на границе  $D_i$

**Доказательство.** Докажем эту лемму, как следствие следующей (по сути, следующая – это переформулировка этой)  $\square$

**Лемма 20.2** (О точках, лежащих внутри и на границе). *Пусть  $P$  – множество точек на плоскости,  $R$  – тоже (возможно, пустое) Обозначим как  $md(P, R)$  наименьшую окружность, охватывающую  $P$  и имеющую все точки  $R$  на границе. Пусть  $p \in P$ . Тогда:*

1. Если  $md(P, R)$  существует, то он единственен.
2. Если  $p \in md(P \setminus \{p\}, R)$ , то  $md(P, R) = md(P \setminus \{p\}, R)$
3. Если  $p \notin md(P \setminus \{p\}, R)$ , то  $md(P, R) = md(P \setminus \{p\}, R \cup \{p\})$

**Доказательство.** 1. Если  $|R| > 2$ , то это очевидно невозможно – потому что по 3 точкам окружность строится единственным образом. Пусть тогда  $|R| > 2$  и существуют 2 минимальные окружности  $D_0$  и  $D_1$  с радиусом  $r$  и центрами  $x_0$  и  $x_1$  соответственно.

Тогда  $P \subset D_0 \cap D_1$ ,  $q_0$  и  $q_1$  – точки пересечения  $D_0$  и  $D_1$ , и  $R \subset \{q_0, q_1\}$ . Но если мы построим окружность с центром точно посередине  $q_0$  и  $q_1$ , она будет включать в себя  $D_0 \cap D_1$  и на ее границе будет лежать  $R$ . И по построению ее радиус будет меньше, чем  $r$ . Значит,  $D_0$  и  $D_1$  не являются минимальными охватывающими окружностями.

2. Очевидно.
3. Обозначим  $D_0 = md(P \setminus \{p\}, R)$  и  $D_1 = md(P, R)$ . Это две окружности, очевидно, гомотопически эквивалентны. Обозначим их центры и радиусы как  $x_0, r_0, x_1$  и  $r_1$  соответственно.

Построим между ними кратчайшую гомотопию следующим образом:

$$\begin{aligned} D(\lambda) &= \{x(\lambda), r(\lambda)\} \\ x(\lambda) &= (1 - \lambda)x_0 + \lambda x_1 \\ r(\lambda) &= \|z - x(\lambda)\| \end{aligned}$$

Тут  $z$  – одна из точек пересечения  $D_0$  с  $D_1$ .

Замечание: точки пересечения всегда есть, когда  $R$  непусто, а если оно пусто, то они должны быть из соображений минимальности.

Очевидно, что  $\forall \lambda \in [0, 1] : P \subset D(\lambda), R \subset \partial D(\lambda)$ , ведь это верно для пересечения  $D_0$  и  $D_1$ , которое по построению в себя включает каждая из  $D(\lambda)$ . Тогда существует некая  $\lambda^*$ ,  $0 < \lambda^* \leq 1$ , такая, что  $p \in \partial D(\lambda^*)$ . Но по построению  $r(\lambda^*) \leq r_1$ , и если  $\lambda^* < 1$ , то  $D_1$  не является  $md(P, R)$ , так как ей является  $D(\lambda^*)$ . Противоречие! Значит,  $\lambda^* = 1$ , из чего следует, что  $p \in \partial D_1$ , что и требовалось доказать.  $\square$

```

Function make0( $n$ )
  Result: Возвращает минимальную охватывающую окружность множества точек
   $D_2 \leftarrow$  окружность на диаметре между точками  $p_1$  и  $p_2$ ;
  Перебираем точки с  $p_3$  по  $p_n$ ;
  if  $p_i \in D_{i-1}$  then
    |  $D_i = D_{i-1}$ ;
  else
    |  $D_i = make1(i, p_i)$ ;
  end

Function make1( $k, p$ )
  Result:  $md(\{p_1, \dots, p_k\}, \{p\})$ 
   $D_1 \leftarrow$  окружность на диаметре между точками  $p_1$  и  $p$ ;
  Перебираем точки с  $p_2$  по  $p_k$ ;
  if  $p_i \in D_{i-1}$  then
    |  $D_i = D_{i-1}$ ;
  else
    |  $D_i = make2(i, p, p_i)$ ;
  end

Function make2( $k, p, q$ )
  Result:  $md(\{p_1, \dots, p_k\}, \{p, q\})$ 
   $D_0 \leftarrow$  окружность на диаметре между точками  $p$  и  $q$ ;
  Перебираем точки с  $p_1$  по  $p_k$ ;
  if  $p_i \in D_{i-1}$  then
    |  $D_i = D_{i-1}$ ;
  else
    |  $D_i$  строится единственным образом по трем точкам –  $p, q$  и  $p_i$ ;
  end

```

**Алгоритм 9:** Алгоритм поиска минимальной охватывающей окружности

#### Корректность

Доказанная лемма гарантирует, что окружность, которая ищется при вызове  $make1$  и  $make2$ , всегда существует. Кроме того, она показывает, что построенная на каждом шаге  $make0$  окружность является корректной. Значит, и весь алгоритм корректен.

#### Асимптотика

$make2(n, p, q)$  всегда работает за  $O(n)$ .

$make0(n)$  и  $make1(n, p)$  работают тоже за  $O(n)$ , если не учитывать вызовы нижележащих функций. Но их нужно учитывать! Из этого можно заключить, что верхней оценкой на время выполнения является  $O(n^3)$ . На практике же (на случайных точках) алгоритм работает существенно быстрее.

Разберемся, почему. Для этого рассмотрим работу алгоритма "задом наперед". Сначала рассмотрим функцию  $make1(n, p)$

Пусть у нас есть результатирующая окружность. Начнем удалять из множества точек в обратном порядке и сжимать окружность, когда это возможно.

Вероятность того, что на каком-то шаге окружность сожмется, равна вероятности того, что на этом шаге при обычном исполнении будет вызвана  $make2(i, p, p_i)$ . Какова эта вероятность? Окружность может "опираться" на 2, 3 или более точек, одна из которых всегда  $q$  (которую мы удалить не можем). В первом случае удаление только 1 точки может спровоцировать сжатие окружности, во втором - одной из 2, в третьем - окружность не сожмется в любом случае. Итого, на каждом шаге есть не более 2 точек, удаление одной из которых приведет к вызову  $make2$ . Вероятность удаления одной из этих точек -  $\frac{2}{i}$ .

Итого, ожидаемое время работы функции  $make1(n, p)$ :

$$O(n) + \sum_{i=2}^n O(i) \frac{2}{i} = O(n)$$

Применив аналогичные рассуждения, докажем линейное ожидаемое время работы для функции  $make0$ .

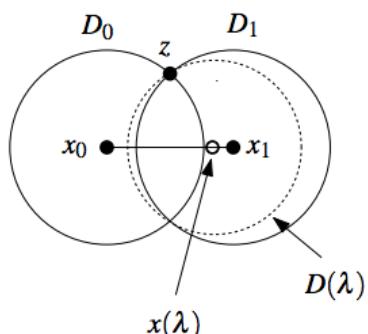


Рис. 45: Иллюстрация к доказательству единственности

## 21 2 19: Граф видимости и планирование движения VOLHOVM

Задача поставлена следующим образом: есть объект, точечный или нет, нужно провести его через полигональные препятствия (все в  $\mathbb{R}^2$ ). Известность карты – тоже входной параметр.

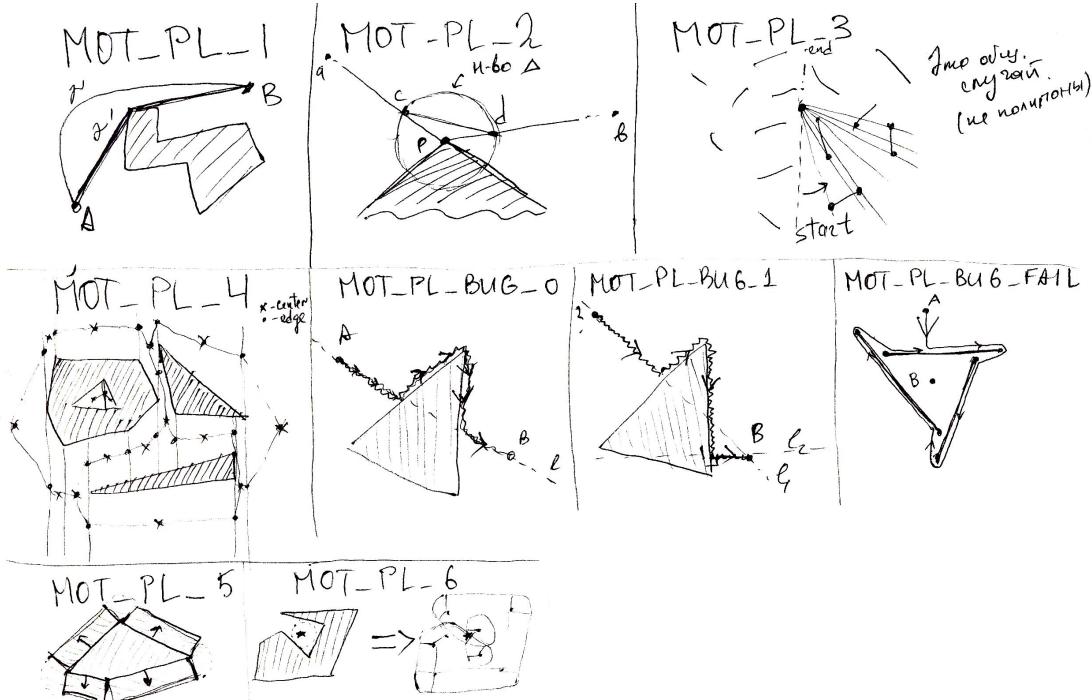


Рис. 46: Иллюстрации к теме про Motion Planning

### 21.1 2 Точечный объект

Решим задачу для точечных объектов. Пусть у нас есть поле, точки  $A$  и  $B$ . Нужно попасть из первой во вторую, не пересекаясь с препятствиями.

#### 21.1.1 3 Граф видимости

Первая тривиальная идея, которая приходит в голову – это построить граф, в котором узлы – это вершины полигонов, составляющих карту, а ребра между двумя вершинами  $u, v$  строим в том случае, если  $uv$  не пересекается ни с одним полигоном из данных. Такой граф называется картой видимости. Можно его обойти дейкстрой. Получаем  $O(n^2)$  и памяти и времени на запрос (если использовать дейкстру без кучи). Предподсчет будет занимать втупую  $O(n^3)$ , то есть для каждой пары точек проверить пересечение со всеми отрезками полигонов (их  $n$  штук).

Для начала заметим, что оптимальный путь – ломаная (доказательство от противного, пусть есть какая-то кривая, огибающая препятствие, тогда спрямим ее, получим прямую меньшей длины) – см. MOT\_PL\_1. Более того, кратчайший путь между двумя точками в поле с полигональными препятствиями содержит только начало, старт и точки полигонов в качестве своих точек. Отсюда такой граф видимости будет содержать оптимальный путь.

Рассмотрим следующую оптимизацию: если рассмотреть вершину полигона  $P$ , то путь из двух ребер (входящее в нее  $aP$  и исходящее  $Pb$ ) неоптimalен, если угол  $aPb < 180^\circ$ . См. MOT\_PL\_2. Доказательство простое – рассмотрим такой угол. Тогда возьмем две любые точки  $c \in aP$ ,  $d \in Pb$  (можно взять их как точки пересечения окружности с центром в  $P$  с прямыми  $aP$  и  $Pb$ , при этом окружность взять радиуса меньше чем каждый из отрезков), получим по неравенству треугольника что путь  $acdb$  короче чем  $aPb$ . Такие ребра можно не добавлять. Оптимизация не понижает асимптотику, а только уменьшает константу, но делает это неплохо.

Препроцессинг можно уменьшить с  $O(n^3)$  до  $O(n^2 \log n)$  с помощью алгоритма Ли (Lee's algorithm, заматающая прямая). Алгоритм представляет из себя модификацию Б-О (см. MOT\_PL\_3):

Для очередной точки  $P$  найдем, какие отрезки из нее исходят вправо (предполагаем, что все отрезки влево уже были добавлены на предыдущем шаге). Для этого рассмотрим все ребра, которые пересекают

прямые, начиная от  $P$  и вниз и вправо против часовой стрелки вверх на  $180^\circ$ . Типа рассмотрели зону видимости "вправо". Формально мы все отрезки, которые заканчиваются правее нашей точки берем, раскладываем на события (стандартные Б-О ивенты типа начало отрезка, пересечение, конец отрезка) и сортируем по углу поворота относительно  $P$ . Дальше перебираем их всех против часовой стрелки, храня стейт всех отрезков, которые пересекают наш луч. Первый отрезок в стейте будет отрезком, который "виден" из  $P$  – будем по ходу дела добавлять концы видимых отрезков в ответ.

Ли работает за  $O((n+k)\log n)$ , где  $n$  – количество отрезков, а  $k$  – количество пересечений. Таким образом для всех точек оцениваем сверху препроцессинг до  $O(n^2 \log n)$ .

Препроцессинг на самом деле уменьшается до  $O(n^2)$ , но это древняя магия (есть какие-то статьи).

Заметим, что можно не хранить ребра, а создавать их только когда мы пришли в вершину. Вроде как достраивать граф по ходу дела. Динамически отвечать на запросы, чтобы снизить память. Я так понимаю, это когда мы минимум выбираем (за  $O(n)$ ), то ищем вершины, потом релаксируем, храним предка чтобы знать путь. Вот короче динамически так будем строить граф.

Есть много других интересных подходов, в том числе алгоритм Митчелла, который снижает память до  $O(n \log n)$ , причем на запрос времени  $O(n + \log^2 n)$ , где логарифм от локализации в планарном графе. Общая идея там похожа на принцип Гюйгенса-Френеля, если я все правильно понимаю: мы запускаем из очередной точки сферические волны и смотрим на те точки плоскости, где в  $\alpha - \varepsilon$  волна остановилась, а в  $\alpha + \varepsilon$  она идет, типа границы видимости. И запускаемся дальше от таких (бесполезное знание)

Есть также похожая задача, суть которой состоит в разбиении плоскости на зоны возможной скорости объекта. Там закон Снеллиуса о преломлении и решение работает за  $O(n^8 \log n)$ .

### 21.1.2 3 Трапецидная карта, наивное решение

Воспользуемся трапецидной картой для решения задачи.

Построим сначала обычную трапецидную карту для набора точек  $S$ , которые формируют полигоны, а затем удалим те трапециды, которые лежат внутри полигонов (можно пройтись по DCEL'у, полагаю, и удалить). Для каждого трапецида из результата добавим в набор точек пути его центр и для каждого соседнего левого или правого трапецида добавим точку на середине вертикального ребра, которое их соединяет (`MOT_PL_4`,  $\times$  – середина трапецида,  $\cdot$  – середина соединяющего соседние трапециды ребра). Для каждого трапецида соединим его середину с серединами соседних ребер. Назовем получившийся график дорожной картой.

Как реализовывать запросы с трапецидной картой? Пусть имеются точки  $A, B$ :

1. Принадлежат одному трапециду – проведем прямую между точками.
2. Иначе определим трапециды  $\Delta_A, \Delta_B$ , содержащие точки  $A$  и  $B$  соответственно, пойдем от  $A$  к центру  $\Delta_A$ , потом по дорожной карте дойдем до центра  $\Delta_B$ , а дальше проведем прямую от центра к  $B$ .

Корректность пути относительно предиката "нет столкновений" очевидна – для путей внутри трапецидов это верно по построению, для каждого элемента дорожной карты тоже.

Оценка времени такова: поиск трапецидов, в которых содержатся  $A$  и  $B$  занимает  $O(\log n)$  с помощью структуры точечной локализации, но можно проверить и за  $O(n)$ , так как весь алгоритм работает за  $O(n)$ . Будем находить путь между трапецидами поиском в глубину, что займет  $O(n)$  шагов, так как в графе и число трапецидов (а значит и их центров) линейно, и число соединений между двумя трапецидами линейно (следует из планарности). Итого  $O(n \log n)$  на препроцессинг,  $O(n)$  на запрос и мы не можем гарантировать оптимальность. Чтобы это сделать, нам необходимы более сложные структуры данных.

### 21.1.3 2 Решение с помощью триангуляции

Формально, вместо трапецидной карты можно пользоваться триангуляцией – памяти столько же, а строить проще.

Можно попробовать сократить память до  $O(n)$ , триангулировав множество вершин полигонов с учетом видимости. Потом рассмотрим двоственный триангуляции граф (взяв в качестве точек центры треугольников).

- Путь получается достаточно плохой, необходимо сгладить ребра:
  - Жадным способом их поменять. Например, если при локализации мы прошли по двум соседним ребрам и они могут быть спрятлены, то так и сделаем.
  - На используемых вершинах подобавлять, типа уточнить путь на графике.

- Подразбивать ребра, добавить вершины и провести их них еще какие-нибудь другие ребра.
- Мы умеем бросать из точки отрезок и смотреть что он пересекает (препятствие или нет). Количество пересеченных треугольников будет  $O(\sqrt{n})$ . Это асимптотика пересечения числа треугольников это  $\sqrt{n}$ . Типа если рандомную точку кинуть, то до нее ребер будет столько.
- Добавляем еще сетку и дополнительно триангулируем по ней, это уменьшает длины ребер треугольников (это хорошо почему-то)
- Зная среднюю и максимальную длину ребра, можем кидать отрезки, кратные ей и таким образом ограничивать количество просматриваемых треугольников.

#### 21.1.4 2 Слепой жук: точечный объект, нет знания карты

Задача сформулирована так же, как предыдущие, но в этом случае у нас нету возможности заранее что-либо предподсчитать.

Рассмотрим наивное **нерабочее** решение: **BUG\_FAIL**. Пусть наш жук будет обходить препятствие до первого поворота, а дальше направляться в сторону конца. Контипример изображен на (**MOT\_PL\_BUG\_FAIL**): в этом случае мы зациклимся и не достигнем финиша.

Рассмотрим два несложных решения.

- **BUG0:** Зафиксируем прямую из начальной в конечную точку  $l$ . Будем придерживаться этой прямой. При встрече с препятствием фиксируем положение обходим его в одну фиксированную сторону до тех пор, пока не окажемся снова на этой линии (**MOT\_PL\_BUG\_0**).
- **BUG1:** Пусть зафиксирована прямая движения  $l_1$ . Будем следовать ей, а при встрече с препятствием обойдем и отметим ситуацию, когда мы находимся ближе всего к точке  $B$ . В этот момент сформируем прямую  $l_2 = pB$ , где  $p$  – текущее положение жука, и пойдем по  $l_2$  дальше, следуя алгоритму (**MOT\_PL\_BUG\_1**).

В некоторых случаях **BUG1** значительно лучше **BUG0**: представим себе спираль, в центре которой финиш. Начав вне спирали с алгоритмом **BUG1** мы после первого шага будем идти внутрь спирали, уменьшая расстояние до финиша. С **BUG0** алгоритм будет часто делать лишние шаги по спирали и покажет себя значительно хуже.

Вот еще забавный алгоритм: **CBUG**. Выбираем любой алгоритм из предыдущих двух и дополняем его таким образом: на старте и на финише строим эллипс как на центрах. Добавляем его в список преград, то есть. При первом столкновении с какой-либо поверхностью запоминаем точку. Может произойти так, что мы пройдемся по ограничивающему эллипсу и вернемся назад. Если такое произошло, увеличим эллипс в два раза.

Есть еще уйма классных алго для роботов. К примеру, есть модификация для робота со зрением. Это примерно аналогично роботу без зрения, только "прощупывание" стены визуальное.

## 21.2 3 Неточечный объект

Для неточечного объекта чаще всего задача сводится к какому-то расширению препятствий и сведению к предыдущей задаче с точечным объектом.

В общем случае задача делится на две по критерию "можно ли поворачивать объект". Для круга этот вопрос не имеет смысла, поэтому он вынесен в отдельную подзадачу.

### 21.2.1 3 Задача для круга

Эта задача делится на две:

Пусть полигоны выпуклые. Расширим прямые полигонов вовне на радиус круга. Свяжем расширенные прямые в узлах разрыва каким-то приближением кругов (выставим некоторое количество точек, образовав вписанный многоугольник). См. **MOT\_PL\_5**. Потом, если не будем пользоваться графикой видимости, новые полигоны объединим. Если же решение с трапецидной картой или триангуляцией, то это нужно сделать, чтобы не связывать лишние вершины.

Если полигоны невыпуклые – тоже хотим расширить, но возникают проблемы с самопересечением (**MOT\_PL\_6** – у нас внутри полигона может поместиться круг, хотя туда нельзя его провести извне). Есть два варианта – не учитывать буферную зону пересечения (более-менее просто) или строить честный straight-skeleton (уже нетривиально, хотя и были лекции, которых нет в программе экзамена). Первое можно решить так: для каждого отрезка расширить его на радиус описанного многоугольника, и все такие расширения (они будут прямоугольниками) пообъединять (вместе с основной фигурой и скруглениями).

Также утверждается, что первое решается за  $O(n^2)$  с помощью priority queue, на уровне "добавляем события и аккуратно смотрим за пересечениями". Про это мне нечего сказать.

### 21.2.2 3 Задача для полигона без вращения

Пусть мы проводим через поле невращающийся полигональный выпуклый объект. Выберем некоторую точку в полигоне и будем думать в сторону построения суммы Минковского полигонов поля относительно нашего агента с зафиксированной точкой.

На этом этапе предполагается, что все рассуждения о том, как нужно строить сумму Минковского, относятся к соответствующей теме, поэтому просто выпишем некоторые тезисы:

Для невыпуклых многоугольников их нужно разбить на выпуклые (триангулировать), для каждого построить сумму Минковского, а затем их объединить.

Сложность объединения невыпуклого полигона с выпуклым  $O(nm)$ , невыпуклого в невыпуклым  $O(n^2m^2)$ .

Для выпуклого многоугольника с  $n$  точками построение суммы Минковского с агентом из  $m$  точек занимает  $O(nm)$  времени (наивно с помощью выпуклой оболочки), а если пользоваться методом калиперов, то  $O(n + m)$ .

На триангуляцию многоугольника с  $m$  вершинами уходит  $O(m \log m)$  времени (можно и за  $O(m)$  с очень сложным алгоритмом). Тогда для всех многоугольников это можно сделать за  $n \log n$ .

Асимптотика мерджа расширенных суммой Минковского агентов с треугольниками триангуляции составляет  $O(n \log^2 n)$ : выбирать какие треугольники мерджить можно с помощью divide-and-conquer (это  $O(\log n)$  операций, а один шаг мерджа занимает  $O(n \log n)$  по линейности количества треугольников).

Запрос будет реализовываться все так же за  $O(n)$ .

### 21.2.3 3 Задача для полигона с вращением

Хорошая практика в этом вопросе решать его неточно (точные решения занимают  $O(n^4)$  памяти)). Определим некоторый дискретный набор углов, на которые мы будем поворачивать нашего агента. Построим много карт для разных углов.

Имея некоторое количество карт, хочется понять, как локализовываться в них. Для локализации хочется получить какую-то общую для всех уровней структуру. Хорошее предложение – для каждого угла поворота нарисовать трапецидную карту и сплинковать каждые соседние слои.

Линковка происходит просто: для каждого двух соседних (соответствующих соседним углам дискретизации поворота) трапецидных карт будем смотреть их пересечение. Для двух пересекающихся трапецидов из разных уровней добавляем в граф еще одну вершину как центр пересечения трапецидов и соединяем ее с центрами двух пересекаемых трапецидов. В пересечении двух трапецидов мы можем повернуть объект из одного угла дискретизации в другой. Так, прошивая каждые два соседние слоя – мы получим карту и roadmap, с которой можно работать как раньше – BFS'ом или Дейкстрой.

Памяти всего будет  $O(n + m)$  на слой если все выпуклое и  $O(n^2m^2)$ , если невыпуклое. Суммарно еще умножить на количество слоев по дискретизации угла, которых обычно берут  $O(n^2)$ , выйдет:

Препятствия	Агент	Память
Невыпуклые	Невыпуклый	$O(n^4m^2)$
Невыпуклые	Выпуклый	$O(n^3m)$
Выпуклые	Невыпуклый	$O(n^3m)$
Выпуклые	Выпуклый	$O(n^2(n + m))$

Поправка: такой алгоритм не всегда верен, как можно догадаться – поворот объекта внутри пересечения трапецидов не обязан не задевать препятствия. Можно увеличить количество углов поворота, но и это не будет гарантировать корректность. Чтобы превратить все true negative в false positive, мы можем считать карту для слоя для модифицированного робота: зафиксируем у него точку, повернем его на `+angle` и `-angle`, возьмем выпуклую оболочку получившегося робота вместо него самого.

## 22.1 3 Оптимальная триангуляция

Пусть поставлена задача интерполировать множество точек в  $\mathbb{R}^2$ . Для каждой точки определено значение функции (скажем, высота ландшафта), и по множеству хочется восстановить значение функции в каждой другой точке плоскости. В этом случае нам поможет триангуляция, но какая именно?

**Определение.** Для триангуляции  $T$  с  $t$  треугольниками будем называть отсортированный по возрастанию вектор всех углов во всех треугольниках  $A(T) = \alpha_1 \dots \alpha_{3n}$  вектором углов триангуляции. Отношение  $<$  на векторах-углах определяется лексикографически.

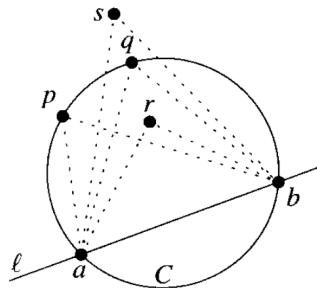
**Определение.** Будем говорить, что триангуляция  $T$  множества точек  $G$  оптимальна относительно углов, если для каждой другой триангуляции  $T'$  верно  $A(T) \geq A(T')$ .

Для задачи с интерполяцией множества точек лучше всего подходит именно оптимальная с точки зрения углов (далее просто "оптимальная") триангуляция, так как она соединяет близкие точки и не допускает слишком "вытянутых" треугольников.

### Теорема 22.1.

Пусть  $C$  – окружность,  $l$  – прямая, пересекающая  $C$  в точках  $a, b$ . Точки  $p, q, r, s$  лежат по одну сторону  $l$ , причем  $p, q$  лежат на окружности,  $r$  внутри нее, а  $s$  – вне окружности. Тогда верно:

$$\angle arb > \angle apb = \angle aqb > \angle asb$$



Рассмотрим флип – операцию на двух соседних треугольниках, меняющую одну диагональ на другую. Будем называть ребро плохим, если при флипе мы уменьшаем минимум среди всех шести углов. Очевидно, что последовательно находя плохие ребра и флипая их, мы будем увеличивать оптимальность триангуляции (так как мы увеличиваем лексикографический порядок, избавляясь от малых углов).

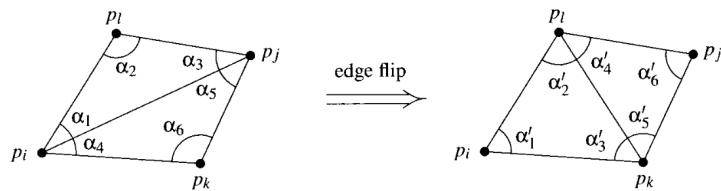


Рис. 47: Замена плохого ребра  $p_i p_j$  с помощью флипа.

### Теорема 22.2.

Пусть дан треугольник, и вокруг него описана окружность. Если существует другая точка, лежащая внутри этой окружности так, что ее от треугольника отграничивает ребро  $e$ , то такое ребро – плохое.

Более того, если четыре точки не лежат на одной окружности и формируют четырехугольник, то ровно одна из двух диагоналей плохая.

**Доказательство.** Поймем, что из второго утверждение следует первое.

Второе можно показать с помощью матана и какого-нибудь деформирующего непрерывного преобразования квадрата. Вроде того: возьмем квадрат  $abcd$ , опишем возле него окружность. Покажем, что двигая одну из точек в любом направлении, хотя бы одна окружность смещается и необходимая диагональ становится плохой.

В деБERGE доказательства нету. □

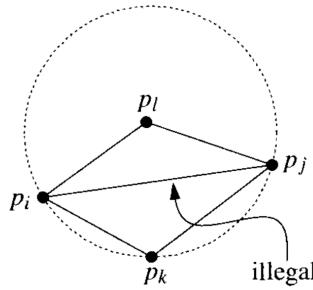


Рис. 48: Ребро  $p_i p_j$  – плохое.

## 22.2 3 Триангуляция Делоне

**Определение** (Триангуляция Делоне). Триангуляция Делоне – это такое разбиение множества точек на треугольники, что в описанной вокруг каждого окружности нету других точек из этого множества.

Становится ясно, что триангуляция Делоне есть оптимальная, то есть может быть получена в результате применения флипов на все плохие ребра к любой триангуляции, при условии конечности процесса флипов. Последнее, кстати, требует отдельного внимания. Перед тем, как показать конечность числа флипов и существование триангуляции Делоне, представим один удобный механизм работы с триангуляциями.

Рассмотрим четыре точки  $a, b, c, d$ . Как проверить, принадлежит ли  $d$  окружности? Введем функцию

$$\phi((p_x, p_y)) = (p_x, p_y, p_x^2 + p_y^2)$$

. Эта функция "поднимает" точку на параболоид в нуле. Тогда заметим, что  $\phi(a), \phi(b), \phi(c)$  образуют плоский треугольник в  $\mathbb{R}^3$ , такой, что плоскость через него проходящая, отсекает от параболоида ровно поднятие круга описанного вокруг  $\Delta abc$  на параболоид. Отсюда становится понятно, что для проверки принадлежности точки  $d$  окружности достаточно проверить знак определителя (поворот в  $\mathbb{R}^3$ ):

$$D = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix}$$

### Теорема 22.3.

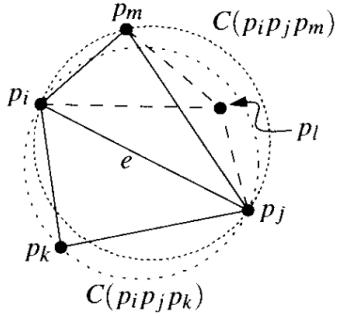
Алгоритм, флипающий все плохие ребра, терминируется.

**Доказательство.** Несложно показать, что происходит при флипе на проекции триангуляции на параболоид. Плохое ребро на параболоиде выглядит как "долина" в оригами – диагональ четырехугольника вдавливается внутрь параболоида. Флип же меняет "долину" на "гору" тем самым уменьшая объем подграфика параболоида, ограниченного подъемом точек, смежных бесконечно большой грани триангуляции (на параболоиде они самые высокие). Уменьшение строгое, поэтому каждый раз объем строго уменьшается. Значит значение объема как функции имеет минимум. Алгоритм не может уменьшить это значение до нуля, потому что это противоречит физическому устройству флипов – невозможно получить сколь угодно малый объем подграфика. Значит, алгоритм терминируется и триангуляция Делоне существует.  $\square$

### Теорема 22.4.

Алгоритм, флипающий ребра, строит триангуляцию Делоне (из локальноальной оптимальности следует глобальная).

**Доказательство.** Напомним, что в оптимальности ничего не говорится о **всех** точках, лишь только о тех, что формируют смежный четырехугольник. Покажем, что если  $T$  – триангуляция без "плохих" ребер, то это триангуляция Делоне.



Докажем от противного. Пусть все ребра хорошие, но триангуляция не обладает свойством Делоне. Значит, есть треугольник  $\Delta p_i p_j p_k$  такой, что в его описанной окружности  $C_1$  есть точка  $p_l$ . Назовем хорошее ребро  $p_i p_j$  (такое, что треугольник  $p_i p_l p_j$  не пересекается с первоначальным)  $e$ . Теперь выберем среди всех таких треугольников и неправильных точек  $p_l$  такую пару, которая максимизирует  $\angle p_i p_l p_j$ . Рассмотрим треугольник  $\Delta p_i p_j p_m$ , смежный с выбранным по ребру  $e$ . По оптимальности  $T$  ребро  $e$  правильное, значит  $p_m$  не лежит в описанной вокруг  $p_i p_j p_k$  окружности.

Заметим, что  $p_l$  также лежит в окружности, описанной вокруг  $p_i p_j p_m$  ( $C(p_i p_j p_m)$  содержит полностью часть окружности  $C(p_i p_j p_k)$ , которая лежит с внешней стороны  $e$ ).

Пусть, без потери общности, точка  $p_l$  лежит по внешнюю сторону ребра  $p_m p_j$  треугольника  $\Delta p_i p_j p_m$ . Тогда  $\angle p_m p_l p_j > \angle p_i p_l p_j$ , поскольку  $p_m p_j$  опирается на большую дугу окружности. Последнее утверждение противоречит тому, что угол максимальен.  $\square$

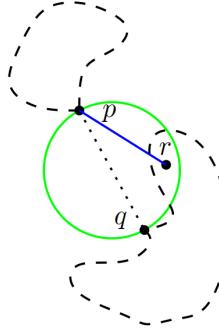
### 22.3 3 Использование триангуляции Делоне

Триангуляция Делоне помогает решать некоторые практические задачи. Кроме уже упомянутой интерполяции, решим задачу поиска минимального оствового дерева в евклидовом пространстве.

Задача: построить минимальное оствовое дерево для набора точек, где вес ребра – расстояние по евклидовой метрике между точками.

**Лемма 22.1.** *Все ребра евклидового МСТ лежат в триангуляции Делоне.*

*Доказательство.* От противного: пусть это не так. Рассмотрим ребро  $e = pq$ , не принадлежащее триангуляции Делоне. Поскольку  $e$  – не ребро Делоне, то в окружности его охватывающей есть другие точки (пусть  $r$ ).



Тогда пусть без потери общности  $r$  принадлежит части оства по сторону точки  $q$ . Тогда заменив  $e$  на  $pr$ , мы не уменьшим количество ребер, но уменьшим длину ребра, поэтому МСТ – не минимально.  $\square$

Воспользуемся системой непересекающихся множеств. Будем действовать согласно алгоритму, очень похожему на алгоритм Крускала: добавим все точки в СНМ, будем обходить ребра Делоне (отсортированные) и добавлять, если для очередного ребра  $e$  верно  $get(e.start) \neq get(e.end)$ .

## 23 2 21: Делоне: алгоритм + корректность

VOLHOVM

Существуют следующие алгоритмы построения триангуляции Делоне:

1. Итеративными флипами из произвольной триангуляции.
2. Построением выпуклой оболочки поднятия точек на параболоид.
3. Инкрементальное добавление точек в триангуляцию (рандомизированное).
4. Конвертирование из диаграммы Вороного.

Первые два алгоритма были рассмотрены в теоретическом билете. Этот билет целиком о третьем алгоритме. Четвертый способ будет рассмотрен в билете про диаграмму Вороного.

Частично этот билет есть в дБерге, частично тут. И тут.

### 23.1 2 Изменение статической триангуляции

Для начала решим подзадачу – поймем, как, имея триангуляцию Делоне, добавить точку внутрь. Предположим, что точка всегда находится внутри триангуляции (этого легко достичь, зная ограничения на точки: первоначально построить треугольник, охватывающий все поле, а в конце удалить).

У нас всего есть два варианта, куда может попасть точка при локализации – внутрь треугольника или на ребро. Добавим три ребра, если точка попала внутрь треугольника, и два, если на ребро (см. иллюстрацию). Осталось только нормализовать ребра, которые могли стать плохими. Потенциально для случая с тремя вершинами нам нужно нормализовать  $p_i p_j$ ,  $p_j p_k$ ,  $p_k p_i$  – все три ребра треугольника, в который мы вставили вершину. Для случая с ребром – это все четыре стороны четырехугольника.

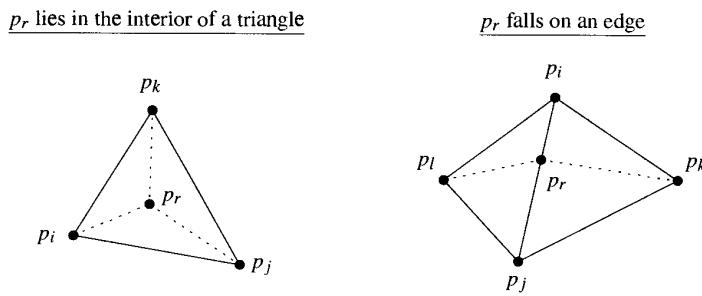


Рис. 49: Возможные варианты положения точки  $q$  относительно триангуляции

Нормализовать ребро – значит заменить его флипом. Разберемся, какие конкретно флипы следует делать. Заметим, что ребро  $e$  может стать плохим только в том случае, если изменился один из треугольников, который на  $e$  опирался. Поэтому будем проверять только ребра новых треугольников.

Алгоритм нормализации ребра таков: пусть новая точка –  $q$  а мы должны нормализовать  $e$ . Найти четырехугольник  $qabc$ , такой что  $e = ac$  и флипнуть это ребро, заменив его на  $qb$ . Повторить процедуру относительно ребер  $bc$  и  $cd$ , так как для них изменились внешние треугольники. Ребра  $qa$  и  $qc$  – новые, добавленные при первоначальной вставке, поэтому мы их не трогаем. Несложно показать, что не нужно будет никогда больше трогать и новое ребро  $qb$ .

Корректность алгоритма следует напрямую из леммы о том, что проверять и флипать следует лишь ребра, у которых изменились смежные треугольники.

### 23.2 2 Алгоритм

Пусть дан набор точек  $S$ . Наша задача – построить триангуляцию Делоне  $DT(S)$  и эффективно поддерживать ее относительно операций удаления или вставки. Отметим, что

Будем использовать локализационную структуру, используя подход Бернуллевских множеств:

$$S = S_0 \subseteq S_1 \subseteq S_2 \subseteq \dots \subseteq S_k$$

$$\text{Probability}(p \in S_{i+1} | p \in S_i) = \frac{1}{\alpha} \in [0, 1]$$

Структура данных достаточно простая. Будем отсылаться к  $S_i$  как к  $i$ -му уровню структуры. Треугольник как структура данных будет знать о своих смежных треугольниках и вершинах, а каждая

вершина – свой треугольник. Число уровней структуры не фиксируется и зависит исключительно от  $\alpha$ .

Чтобы локализовать точку  $q$  в триангуляции, будем начинать локализовываться на  $k$ -м уровне от точки  $v_{k+1}$ , находящейся на  $k+1$ -м. Далее найдем точку  $v_k$ , ближайшую к  $q$  точку из  $DT_k$ . Поскольку все точки на  $k$ -м уровне есть и на  $k-1$ -м, то найдя  $v_i$  остается только спуститься вниз и продолжить поиск. Так будем спускаться до нулевого уровня, на котором найдем нужный треугольник.

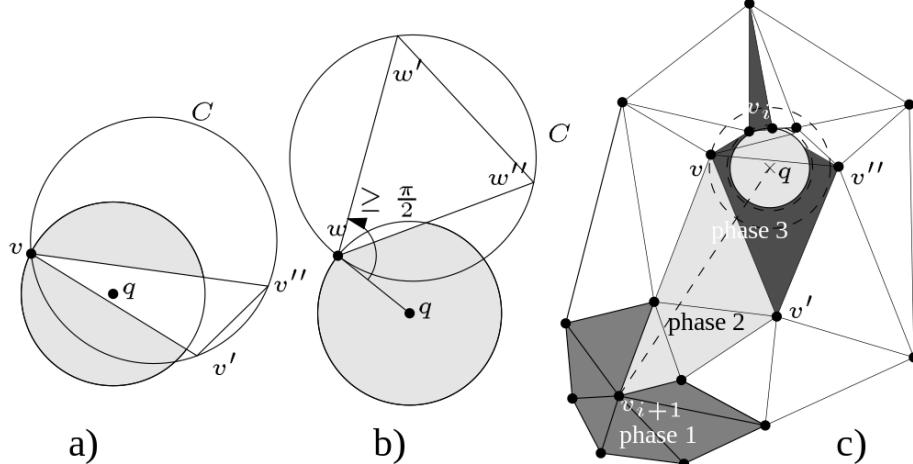


Рис. 50: Иллюстрации к локализации в  $i$ -ом уровне структуры

На уровне  $i$  поиск проходит в три этапа:

1. Рассматривается треугольник, который содержит вершину  $v_{i+1}$ , все смежные ему треугольники проверяются на наличие отрезка  $v_{i+1}q$ .
2. Посещаются по очереди все треугольники в  $DT_i$ , которые пересекает отрезок  $v_{i+1}q$ . Находим треугольник  $t_i$ , содержащий точку  $q$ .
3. Найдем жадно точку, ближайшую к  $q$ : посетим соседей треугольника  $t_i$  по следующему алгоритму. Пусть  $t_i = vv'v''$ , и, без потери общности, ближайшая вершина к  $q$  есть  $v$ . Тогда либо  $v_i$  есть  $v$ , либо  $v_i$  находится внутри круга с центром в  $q$  и радиусом  $qv$ . Следовательно, следует проверить только смежных ребрам  $vv'$  и  $vv''$  треугольники, не проверяя тот, что смежен  $v'v''$  (окружность его не пересекает ввиду того, что  $v$  – ближайшая к  $q$ ). См. иллюстрацию пункт  $a$ . Для каждого такого треугольника будем запоминать ближайшую к  $q$  точку  $w$  и переходить к следующему треугольнику, смежному по ребру  $ww'$ , только если угол  $\angle qww' < \frac{\pi}{2}$ . На иллюстрации пункт  $b$  изображает выбор ребра  $ww''$  и отбрасывает переход по  $ww'$ .

Алгоритм локализации целиком изображен на иллюстрации пункт  $c$ . После локализации в структуре будем изменять ребра так, как это делалось в статическом алгоритме, а затем с вероятностью  $\frac{1}{\alpha}$  прорабатывать вершину наверх и добавлять ее там так же.

### 23.3 2 Асимптотика

#### Теорема 23.1.

*В графе Делоне с  $n$  точками верхняя оценка на количество ребер  $3n - 6$ , а на количество граней  $2n - 5$ .*

*Доказательство.* Воспользуемся теоремой Эйлера для планарного графа:  $m_v - m_e + m_f = 2$ . Для графа Делоне эта формула выглядит так:

$$n - n_e + n_f = 2$$

Заметим также, что каждая грань имеет степень хотя бы три, а каждое ребро связывает между собой всегда две грани:

$$2n_e \geq 3n_f$$

Вместе с предыдущим утверждением решение системы доказывает утверждение теоремы.  $\square$

Сначала оценим количества новых треугольников, создаваемых при добавлении точки.

**Лемма 23.1.** На каждом шаге алгоритма для статической триангуляции с  $k$  при добавлении точки  $p_r$  создается максимум 9 треугольников.

*Доказательство.* В начале добавления точки мы создаем либо 3 (в случае попадания внутрь треугольника) либо 4 (попали на ребро) треугольника. Каждый последующий флип при нормализации создает 2 новых треугольника. Напомним, что такой флип создает ребро, смежное с вершиной  $p_r$ . То есть, если степень вершины  $p_r$  после вставки есть  $k$ , то максимум может создаться  $2(k - 3) + 3 = 2k - 3$  новых треугольников. Степень вершины – количество ребер ей инцидентных, назовем это число  $\deg(q, DG_q)$ , где  $DG_q = p_1, p_2, \dots, p_r \cup a, b, c$  (второе – это внешние три точки, которые ограничивают все остальные). Из теоремы об оценке количества ребер графа Делоне, количество ребер в  $D_r$  есть  $3(r + 3) - 6$ . Три ребра из этих принадлежат самому внешнему треугольнику, тогда общая степень вершин в графе меньше чем  $2[3(r + 3) - 9] = 6r$ . Это значит, что средняя ожидаемая степень новой вершины – 6.

Оценим тогда количество треугольников, создаваемых на шаге  $r$ :

$$EX \leq E[2\deg(p_r, DG_r) - 3] = 2E[\deg(p_r, DG_r)] - 3 \leq 2\dot{6} - 3 = 9$$

□

**Замечание.** Для простого алгоритма без использования ступенчатой структуры локализации количество созданных треугольников есть  $1 + 9n$ .

Оценим теперь время локализации, так как в него упирается весь алгоритм (мы уже поняли, что на каждая вставка – это  $O(1)$ ). Пусть  $S$  – это множество вершин текущей триангуляции,  $q$  – вставляемая вершина. В силу рандомизированности  $q$  – случайный элемент  $S \cup q$ . Будем называть  $n_i = |S_i|$  (где  $S_i$  – набор точек на уровне  $i$ ),  $R_i = S_i \cup q$ . В силу рандомизированности тут  $q$  – случайный элемент  $R_i$ , а само  $R_i$  – случайное множество размера  $n_i + 1$ . Можно думать об  $R_i$  как об случайному подмножестве  $R_{i-1}$ . Будем использовать нотацию  $E(X)$  для обозначения матожидания.

Напомним, что в алгоритме локализации есть три фазы. Стоимость первой фазы, то есть посещения всех смежных  $v_{i+1}$  треугольников, есть степень  $v_{i+1}$  в  $DG_i$ . Стоимость второй фазы равна количеству ребер, пересекаемых отрезком  $v_{i+1}q$ . Стоимость третьей фазы – количество вершин-кандидатов на роль ближайшей во время поиска  $v_i$  после нахождения  $t_i$ .

**Лемма 23.2.** Ожидаемая степень  $v_i$  в  $DT_{i-1}$  есть  $O(1)$ .

*Доказательство.* Пусть  $NN$  – nearest-neighbor граф  $R_i$ , то есть граф, в котором вершины – точки  $R_i$ , а ребро между  $v$  и  $q$  есть тогда и только, если  $v = NN(q)$ , то есть  $v$  – ближайшая вершина к  $q$  или  $q = NN(v)$ .

$NN$  есть подграф  $DT(R_i)$  (следует из оптимальности триангуляции, т.к. она максимизирует углы, делая все ребра равномерными по длине), а максимальная степень  $NN$  есть 6. Последнее доказывается достаточно просто. Следует лишь нарисовать 7 точек – правильный шестиугольник с центром, и предположить, что максимальная степень центральной вершины больше 6. Далее несложно показать, что седьмая ближайшая к центру точка на самом деле окажется ближе всего к соседу, так как в правильном шестиугольнике треугольники вида  $OAB$ , где  $O$  – центр, а  $A$  и  $B$  – соседние вершины, равносторонние.

Будем называть  $\deg_G(v)$  степенью вершины  $v$  в графе  $G$ . Напомним также, что точка  $q$  – случайная в  $R_i$ .

Сложность доказательства состоит в том, что вероятность получения  $NN(q)$  не равномерна, даже если это верно для самой  $q$ .

$$\begin{aligned} E(\deg_{DT_{i-1}}(NN(q))) &= E\left(\frac{1}{|R_i|} \sum_{q \in R_i} \deg_{DT_{i-1}}(NN(q))\right) \\ &= \frac{1}{|R_i|} E\left(\sum_{v \in R_i} \sum_{q \in \{r: v = NN(r)\}} \deg_{DT_{i-1}}(v)\right) \\ &\leq \frac{1}{|R_i|} E\left(\sum_{v \in R_i} 6 \times \deg_{DT_{i-1}}(v)\right) \\ &\leq 36 \end{aligned}$$

□

**Лемма 23.3.** Пусть  $w \in R_i$ , тогда ожидаемое количество вершин  $q$  из  $R_i$ , таких что  $w$  принадлежит кругу с центром в  $q$ , проходящему через  $NN(q)$  в  $R_{i+1}$ , меньше чем  $6\alpha$ .

**Доказательство.** Пусть  $w \in R_i$ . Рассмотрим  $Q = q_1 \dots q_k$  – точки  $R_i$ , лежащие в любом секторе с центром  $w$  шириной  $\frac{\pi}{3}$ , отсортированные по возрастанию расстояния от  $w$ . Очевидно, что окружность с центром в  $q_l$ , проходящая через  $q_j$  не может содержать  $w$ , если  $j < l$ . Тогда если  $q = q_l$ , необходимое условие для того, чтобы  $w$  содержалась в диске между  $q$  и  $NN(q)$  в  $R_{i+1}$ , есть то, что ни одна из точек  $\{q_1 \dots q_{l-1}\}$  не содержится в уровне выше, то есть в  $R_{i+1}$ . Вероятность такого события  $(1 - \frac{1}{\alpha})^l$ .

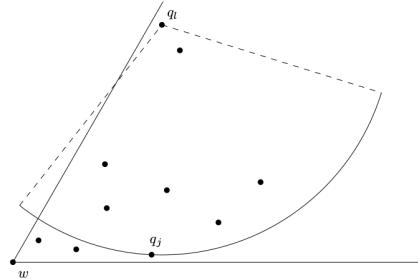


Рис. 51: Иллюстрация к доказательству леммы 3

Просуммируем результат по всем шести секторам и по всем  $q \in R_i$ , получим результат  $6\alpha$ . Для одного сектора:

$$EX = \sum_{k=0}^{\infty} k(1-\alpha)^{k-1}\alpha = \alpha \left( \sum_{k=0}^{\infty} (1-\alpha)^k k \right) = \frac{1}{p} = \alpha$$

Отметим также, что окружность с центром в  $q$  проходящий через  $NN(q)$  содержит внутри себя окружность диаметром  $qNN(q)$ . Отсюда ожидаемое количество вершин  $q \in R_i$  в окружности с диаметром  $qNN(q)$ , меньше  $6\alpha$ .  $\square$

**Лемма 23.4.** Ожидаемое количество ребер  $DT_i$ , пересекаемое отрезком  $v_{i+1}q$ , есть  $O(\alpha)$ .

**Доказательство.** Пусть  $e$  – ребро  $DT_i$ , которое пересекает отрезок  $v_{i+1}q$ . Если  $e$  не существует в  $DT_{R_i}$ , значит  $e$  – внутреннее ребро, исчезнувшее после триангуляции, которую алгоритм совершает при вставке  $q$ . Матожидание количества таких ребер 3, поскольку  $q$  – случайная точка в  $R_i$ , а число равно средней степени вершины (6) минус 3.

Если  $e$  принадлежит  $DT_{R_i}$ , один его конец  $w$  должен принадлежать окружности диаметром  $qv_{i+1} = disk[qv_{i+1}]$ . Иначе любая окружность проходящая через концы  $e$  должна содержать  $q$  или  $v_{i+1}$ , что противоречит критерию Делоне  $DT_{R_i}$ .

Ожидаемое количество ребер  $DT_{R_i}$ , пересекающей  $disk[qv_{i+1}]$  ограничено сверху суммой степеней вершин внутри этой окружности. В матожидании мы должны просуммировать по всем  $q$ , а внутри по всем  $w$  внутри  $disk[qv_{i+1}]$  степень вершин. Но можем переставить местами суммы и раскрыть внутреннюю по предыдущей лемме, тогда остается сумма по всем  $w \in R_i$  степени вершины умноженной на  $6\alpha$ , что в свою очередь меньше  $36\alpha$ , как средняя степень вершины в графе Делоне.

Итого мы ограничили количество ребер, просматриваемых во втором этапе, числом  $36\alpha$ .  $\square$

**Лемма 23.5.** Ожидаемое количество треугольников, которые просматриваются во время поиска  $v_i$  из треугольника  $t_i$ , есть  $O(\alpha)$ .

**Доказательство.** Все треугольники  $t$ , просматриваемые алгоритмом, имеют вершину в круге с центром в  $q$  проходящим через  $v_{i+1}$ . Поэтому для них оценка тоже верна: используя предыдущую лемму, можно показать, что матожидание  $\leq 36\alpha$ .  $\square$

**Теорема 23.2.**

Время вставки  $n$ -точкой точки в структуру есть  $O(\alpha \log_\alpha n)$ . Суммарно алгоритм создания триангуляции Делоне из  $n$  точек работает за  $O(n \alpha \log_\alpha n)$  при рандомизированной вставке, не учитывая распределение точек на плоскости.

**Доказательство.** Из лемм выше следует, что локализация на каждом уровне занимает  $O(\alpha)$  времени. Уровней в среднем  $\log_\alpha n$ . Вставка точки занимает  $O(1)$ , что было доказано в самом начале параграфа. Прокидывание точек на верхние уровни дает нам дополнительную асимптотику, но в среднем меньшую, чем  $\log_\alpha n$ .

Второе утверждение является следствием первого.  $\square$

## 23.4 2 Удаление

Задача стоит удалить из триангуляции внутреннюю точку  $q$ . После удаления точки и смежных ей ребер многоугольник останется звездным. Но это нам (я так понял) не очень помогает, замечание в никуда.

Отметим, что мы хотим флипать ребра в особенном порядке. И этот порядок – обратный порядку при добавлении точки  $q$  (не единственный этот, конечно). Подумаем, как можно его восстановить.

Пусть после удаления  $q$  остался многоугольник  $ABCDE$ . Тогда спроектируем его на параболоид. Предположим, что его триангуляция существует. Тогда, поднимаясь из точки  $(q_x, q_y, -\infty)$  в  $(q_x, q_y, q_x^2 + q_y^2)$  (назовем этот луч  $l$ ), мы будем переставить видеть грани в порядке удаления, вместе с приближением точки к верхней границе  $l$ .

Отсюда алгоритм: возьмем очередь с приоритетом, запихнем туда все уши с  $\alpha < 180^\circ$ , компаратор – пересечение с прямой  $l$ . Будем брать, пока берутся. Отметим, что уши, которые плохие, смотрят внутрь параболоида, поэтому они не исчезнут до того момента, как  $l$  пересечет параболоид.

Статья на викиконспектах

### 24.1 3 Определения и свойства

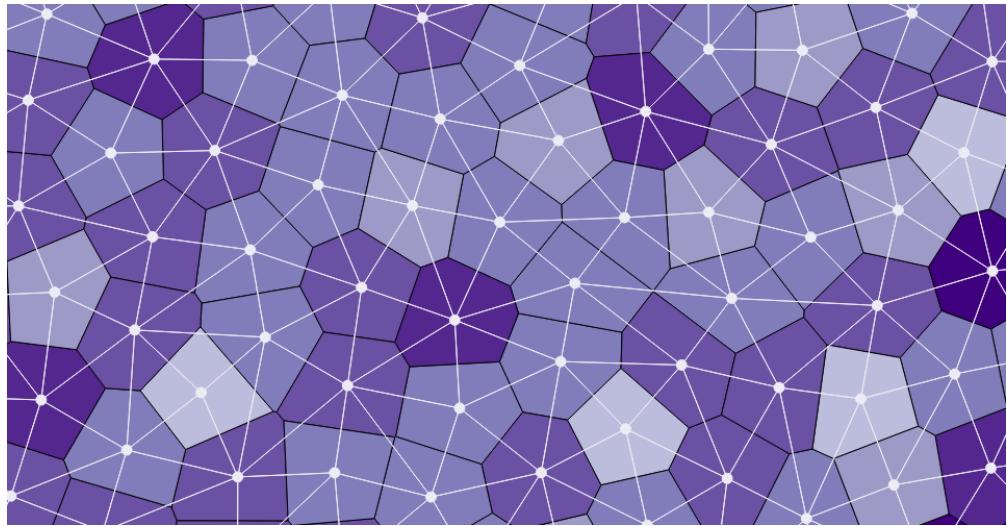


Рис. 52: Офигительно красивая диаграмма Вороного, которая мотивирует ботать

**Определение** (Диаграмма Вороного). *Диаграммой Вороного для множества точек (далее сайтов)  $P$  будем называть такое разбиение пространства на области  $\{Q_i\}$ , что для каждого  $i$  все точки в  $Q_i$  ближе по метрике к  $P_i$ , чем к  $P_j$  для  $j \neq i$ . Будем называть  $Q_i$   $i$ -й клеткой диаграммы.*

Отметим отдельно, что метрика в определении не зафиксирована (и даже размерность пространства, хотя мы будем рассматривать  $\mathbb{R}^2$ ). Будем далее рассматривать диаграммы Вороного для евклидовой метрики ( $\rho(a, b) = \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}$ ), но диаграмму можно строить для любой метрики вообще (парижская, манхэттенская, даже на википедии есть картиночки).

**Замечание.** Каждая клетка  $V_i$  диаграммы есть область, образованная пересечением плоскостей, проходящих через середину  $p_i p_j$ , для сайтов с  $j \neq i$ , и перпендикулярных  $p_i p_j$ .

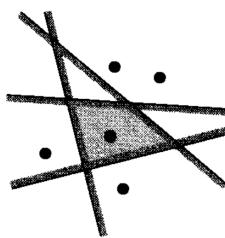


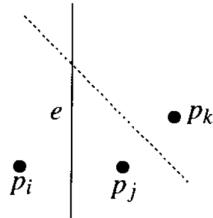
Рис. 53: Ячейка Вороного, образованная пересечением полуплоскостей

#### Теорема 24.1.

Пусть  $P$  – множество из  $n$  сайтов. Тогда  $Vor(P)$  (граф, образованный диаграммой) – это:

1.  $n - 1$  параллельная прямая, если все сайты лежат на одной прямой.
2. Набор из отрезков и лучей, иначе.

**Доказательство.** Первый пункт доказывается достаточно просто. Предположим, что сайты не лежат на одной прямой.



От противного: пусть есть прямая  $e$ , разделяющая клетки  $V(p_i)$  и  $V(p_j)$ . Найдем точку, не лежащую на прямой  $p_ip_j - p_k$  (не умоляя общности, пусть она лежит справа  $e$ , как на изображении). Тогда серединные перпендикуляры между  $p_ip_j = e$  и  $p_jp_k$  не могут быть параллельны. Но часть прямой  $e$  выше точки пересечения с другим серединным перпендикуляром лежит ближе к  $p_k$ , что противоречит предположению.

Связность  $Vor(p)$  следует из того, что компоненты связности несвязного графа разделялись бы одной бесконечной ячейкой, такой что она содержала бы своими сторонами две прямые, что невозможно по предыдущему пункту теоремы.  $\square$

**Лемма 24.1.** Для всех  $n \geq 3$  диаграмма Вороного из  $n$  сайтов содержит максимум  $2n - 5$  вершин и  $3n - 6$  ребер.

*Доказательство.* Смотри доказательство оценки для триангуляции Делоне. Для правильного чтения нужно: заменить "сайты" на "вершины" а "вершины" на "грани". Почему это работает, можно понять, поняв двойственность Делоне-Вороной. Алсо еще нужно представить, что все уходящие на бесконечность ребра смыкаются в одной дополнительной точке, иначе не применить формулу Эйлера.

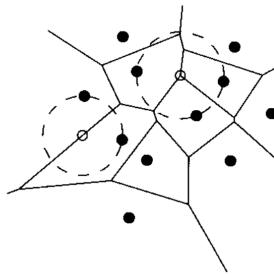
Вот краткое доказательство: из Эйлера имеем  $(n_v + 1) - n_e + n = 2$ , из того, что каждое ребро содержит две точки  $- 2n_e \geq 3(n_v + 1)$ . Решим систему.  $\square$

## 24.2 3 Двойственность Делоне

**Теорема 24.2.**

Для набора сайтов  $P$  и диаграммы Вороного  $Vor(P)$  верно следующее:

1. Точка  $q$  есть вершина  $Vor(P) \leftrightarrow$  максимальная окружность, не содержащая точек внутри себя, содержит ровно 3 сайта на границе.
2. Серединный перпендикуляр порождает элемент диаграммы Вороного между сайтами  $p_j$  и  $p_i$   $\leftrightarrow$  на нем есть точка  $q$ , такая что максимального радиуса окружность в центре  $q$ , не содержащая внутри себя точек, имеет на границе ровно  $p_j$  и  $p_i$ .



*Доказательство.* 1. Доказательства справа налево очевидно – возьмем эту окружность. Поскольку внутри нее нету других точек, то  $q$  должна лежать на границе всех трех клеток, а значит она есть вершина диаграммы Вороного. С другой стороны, каждая вершина инцидентна как минимум трем граням, а значит, как минимум, клеткам  $V(p_i)$ ,  $V(p_j)$ ,  $V(p_k)$ . Поскольку точка  $q$  по определению диаграммы лежит на одинаковой дистанции от трех сайтов, то внутри окружности не будет других точек.

2. Пусть есть точка  $q$  с указанным свойством. Тогда если соседние сайты –  $p_i$  и  $p_j$ , то верно  $\forall k \in [0, n], \rho(q, p_i) = \rho(q, p_j) \leq \rho(q, p_k)$ . Тогда по определению  $q$  лежит на ребре или вершине графа Вороного. По первой части теоремы  $q$  не может быть точкой, тогда  $q$  лежит на ребре.

Обратно, пусть  $p_i$  и  $p_j$  задают два соседних сайта, тогда для любой точки на ребре максимальная пустая окружность будет содержать эти сайты и ничего другого по определению диаграммы.  $\square$

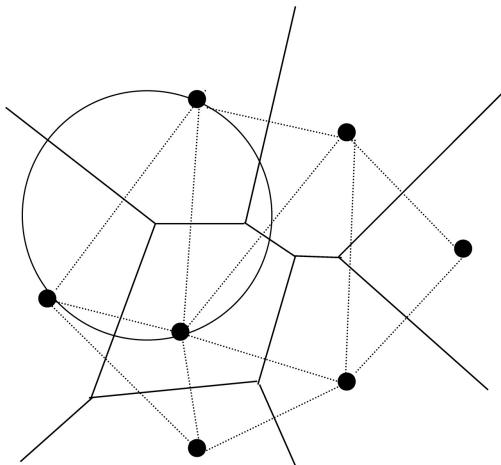


Рис. 54: Пример изоморфизма диаграммы и триангуляции

Таким образом, если соединить все сайты, соответствующие смежным ячейкам диаграммы Вороного, мы получим триангуляцию Делоне. Это несложно доказать, ведь ребро смежных сайтов образовано серединным перпендикуляром.

#### 24.2.1 2 Построение из триангуляции диаграмму

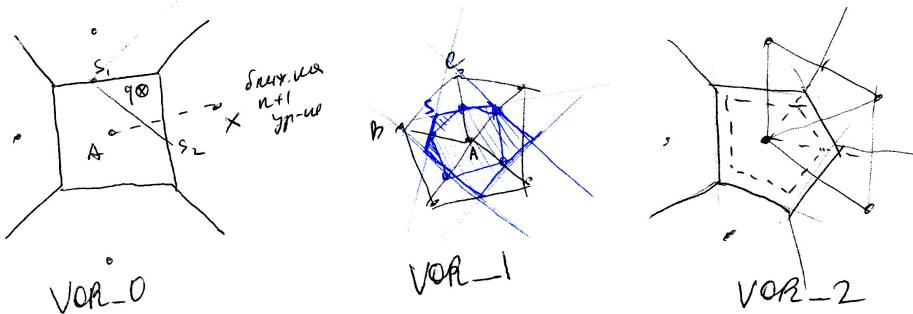


Рис. 55: Иллюстрации с лекции про диаграмму Вороного

Как построить из триангуляции Делоне диаграмму Вороного? Возьмем диаграмму, выделим какую-то точку  $A$ . Построим серединные перпендикуляры для каждого ребра, исходящего из  $A$ , пересечем их всех. Поймем, что получившееся пересечение сер. перпендикуляров образует ячейку Вороного. (См. VOR\_1)

Покажем, что такая ячейка конечна. Рассмотрим треугольник  $ABC$ . По определению, этот треугольник – треугольник Делоне, поэтому точка пересечения серединных перпендикуляров лежит внутри, и расстояние от  $S$  до точек прямоугольника минимально, если точка есть пересечение серединных перпендикуляров. Более того, по свойству Делоне, в окружности не лежит никаких других точек.

Любой отрезок созданной ячейки Вороного принадлежит ей, потому что ячейки диаграммы Вороного выпуклые. Отсюда, поскольку точки отрезка лежат в ячейке, отрезок тоже лежит. Типа сама точка  $A$  лежит ближе всего к себе. Точка пересечения серединных перпендикуляров тоже лежит в ячейке, тогда для каждой двух соседних точек прямая между ними тоже лежит, так как ячейка вороного – выпуклый многоугольник.

#### 24.2.2 2 Построение из диаграммы триангуляции

Возьмем диаграмму Вороного и построим **разбиение** Делоне – то есть могут получиться не треугольники (расположите точки, к примеру, в вершинах квадрата). Но в этом случае несложно показать, что любой такой многоугольник можно триангулировать любым образом, при этом свойство Делоне останется.

Почему в общем случае граф, в котором мы соединили сайты соседних граней, получится разбиением Делоне? Возьмем в DCEL'е узел, который соединяет три грани. Берем их сайты, соединяем. Это получится треугольник. Прогоним обратное следствие в нужную сторону.

## 24.3 2 Алгоритм построения и асимптотика

Наивный алгоритм – брать для каждого сайта все остальные и пересекать полуплоскости, образованные серединными перпендикулярами. Пересечение строится за  $n \log n$ , поэтому суммарно все будет работать за  $n^2 \log n$ . Но это медленно, разумеется.

Антону больше нравится инкрементальный алгоритм построения диаграммы Вороного, так как он похож на Делоне, поэтому его тут и опишем. Перед прочтением этой статьи почти обязательно понимать инкрементальный алгоритм построения триангуляции Делоне.

У нас есть все те же  $O(\log(n))$  уровней, где есть какие-то сабсеты, для каждого мы можем построить за  $O(1)$  новую диаграмму. Разумеется, можно не пользоваться многоуровневой бернулевой системой локализации, тогда алгоритм будет работать за  $O(n^2)$  – для каждой точки локализация будет за  $O(n)$ .

На первом шаге (самом верхнем уровне), где точек  $O(1)$ , чтобы локализоваться, достаточно найти ближайшую точку, посчитав метрику. Как с помощью  $n + 1$  уровня найти ближайшую точку на  $n$ -м уровне?  $X$  – ближайшая точка на  $n + 1$  уровне.  $A$  – точка, которую мы хотим вернуть, то есть ближайшая к  $q$  на  $n$ -м уровне. Проведем отрезок  $XA$  и проверим все соседние грани точки  $X$ , выберем ту, которую пересекает  $XA$ .  $XA$  также может пересекать какую-то точку триангуляции. Тогда нужно перебрать все соседние прямые, исходящие из этой точки и выбрать такие две, между которыми проходит  $XA$ . Локализация, словом, очень похожа на ту, что рассматривалась в Делоне.

Как достроить диаграмму Вороного, если мы уже локализовались? Рассмотрим **VOR\_0**. Построим между  $q$  и  $A$  серединный перпендикуляр, пересечь его с фейсом вершины  $A$ . Будем дальше идти по соседним DCEL'ам и заворачивать ячейку, строя серединные перпендикуляры, прямые вокруг  $q$ . Таким образом, построим грань для вершины  $q$ .

Асимптотика (inb4 можно это делать, строя двойственную триангуляцию):

- Вставка: посчитаем среднюю степень, проведем регрессионный анализ, как в алгоритме Делоне. Получится за  $O(1)$ .
- Локализация: пересечем  $O(1)$  ребер на каждом уровне. Это доказательство тоже копируется с Делоне. Можно сказать, что мы пройдем по количеству DCEL'ов которые не добавились на более высокий уровень. Поскольку слои диаграммы – это множество Бернулли, то на каждом шаге мы добавим не больше чем сколько-то точек, а они экспоненциально убывают.

## 24.4 2 Высшие порядки

**Определение** (Ячейка Вороного  $k$ -го порядка). Ячейка Вороного  $k$ -го порядка  $V_k(p_1, p_2, \dots, p_k)$  – такое множество точек, что ближайшие к каждой точке  $k$  сайтов есть ровно  $p_1, p_2, \dots, p_k$ .

**Определение** (Диаграмма Вороного  $k$ -го порядка). Диаграмма Вороного  $k$ -го порядка – разбиение плоскости на ячейки Вороного  $k$ -го порядка.

Представим два алгоритма построения диаграммы  $k$ -го порядка. Один из них опишем только для второго уровня, другой в общем случае. Также для  $n - 1$  порядка есть отдельный алгоритм, который можно почитать на вики.

### 24.4.1 2 Алгоритм с удалением точек

Хотим уметь удалять точку из диаграммы Вороного. Рассмотрим **VOR\_2**. Возьмем сайт, его фейс. Будем двигать стороны внутрь по серединным перпендикулярам. Тогда в какой-то момент стороны ячейки схлопнутся. На практике нужно искать пару серединных перпендикуляров, которые схлопнутуться первыми (это только соседние), и пересекать. Вопрос "что значит первые" имеет, в моем понимании, ответ "смотря триангуляцию Делоне". В принципе, можно построить двойственную триангуляцию, удалить точку, а потом вернуть Вороного. Это будет за  $O(1)$  все, но, полагаю, можно сделать проще.

Диаграмма второго порядка – это диаграмма первого, в которой каждая ячейка подразбита на подячейки в зависимости от того, какая вторая точка ближе. Рассмотрим изображение. Допустим, что точка 6 удалена, а ее ячейка размечена на 5 зон. К примеру, в зоне III точки находятся ближе всего к 6 и 2. Тогда запомним ребра, получившиеся после удаления ячейки, а также продлим все серединные перпендикуляры между точкой 6 и остальными. Полученные на втором изображении зоны будут искомым разбиением для диаграммы Вороного 2 порядка. Такую операцию нужно проделать для каждого сайта.

### 24.4.2 3 Алгоритм с пересечениями

Строить диаграмму  $k$ -го порядка будем следующим образом: возьмем диаграмму  $k - 1$  порядка, для каждой ячейки  $V_i$  найдем  $k - 1$  точек, которые обозначим как  $S$ . Пересечем каждую такую ячейку

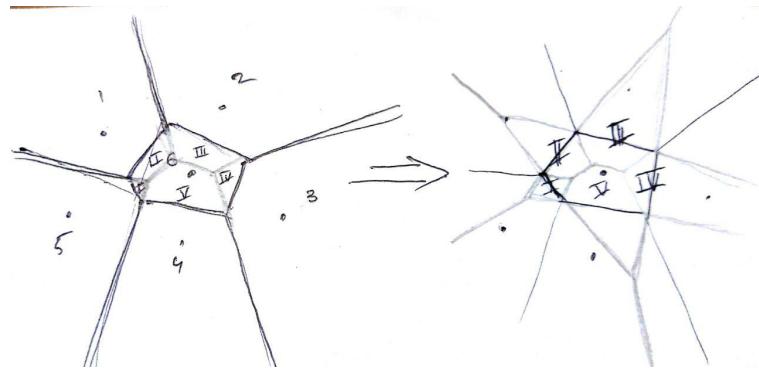


Рис. 56: Построение диаграммы 2 порядка через удаление точки

с множеством ячеек, построенных на множестве сайтов  $P \setminus S$ . Такое пересечение ( $V_{k-1}(S)$  с ячейкой  $V_1(p_i)$ ) генерирует ячейку для  $S \cup \{p_i\}$ . После того, как мы все попрессекали, нужно объединить ячейки, отвечающие за одинаковые наборы сайтов (они могут иметь только одно общее ребро).

Алгоритм работает за  $O(kn^2 \log n)$  – на каждом из  $k$  шагов мы создаем  $n$  диаграмм, каждую за  $O(n \log n)$ , пересекаем  $O(n)$  ячеек с  $O(n)$  других за  $O(n)$ , и еще мерджим за  $O(n)$  (максимально ребер  $O(n)$ , а конкретно объединение в DCEL'е за  $O(1)$ ).

## 24.5 3 Диаграмма минус первого порядка

`./figures/VORONOI_FARTHEST_POINT.gif`

Отдельного рассмотрения заслуживает диаграмма Вороного  $n - 1$ -го порядка – это набор таких сайтов, что для каждого есть  $n - 1$  точка самая ближайшая, значит в частности для каждой точки в ячейке существует одна самая далекая.

Граф Делоне двойственный диаграмме  $n - 1$  порядка – это верхняя крышка проекции диаграммы на параболоид. Это следует из того, что сайт, который не лежит на выпуклой оболочке, не может иметь ячейки. Также несложно заметить, что каждая точка на выпуклой оболочке имеет ячейку.

Я искренне надеюсь, что алгоритм построения farthest-point диаграммы не будет входить в экзамен.

## 25 Бонусные задачи и нетронутые темы

### 25.1 Из множества прямых произвольных восстановить DCEL

Можно делать инкрементально. Для трех понятно, как строить. Дальше кидаем прямую. Берем первое пересечение, локализуем точку на прямой за  $O(n)$ , дальше обходим соседние фейсы DCEL'a пока не найдем точку пересечения нашей прямой с какой-то другой. И так пока все не пересечем. Можно показать, что асимптотика будет норм –  $O(n^2)$ .

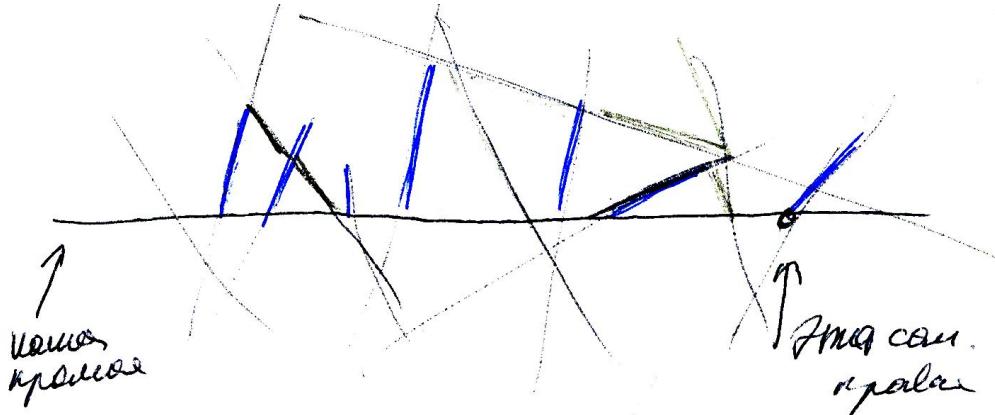


Рис. 57: Восстановление DCEL из набора прямых

Рассмотрим BON-0.  $l$  – наша прямая. Утверждается, что от пересечения нашей прямой с какой-то другой до следующего пересечения нужно пробежать не более чем  $O(n)$  ребер.

В среднем заметим, что у нас  $O(n^2)$  ребер и  $O(n^2)$  фейсов. Тем не менее, из этого не следует, что на каждый фейс приходится  $O(1)$  ребер, может быть так, что какие-то фейсы жирные, а какие-то нет.

Мы всегда знаем направление, в котором мы будем двигаться от точки.

Рассмотрим все. Покрасим точки с положительным наклоном относительно прямой синим цветом, а серым – с отрицательным. Будем считать только те ребра, которые лежат в DCEL'ах, которые пересекают наша прямая  $l$ .

Посчитаем, сколько таких цветных ребер есть (кстати, синих и серых ребер будет одинаковое количество).

Прямая  $l$  придет через  $O(n)$  ребер. Выкинем самую правую прямую, которая имеет синий отрезок. Тут типа индукция. Тогда есть  $\exists c$ , что прямая пересекает не более  $c * (n - 1)$ . Попробуем ее заново добавить. Пусть прямая имеет серый наклон вправо. Тогда такая прямая подразобьет не более чем  $O(1)$  ребер.

### 25.2 Поиск касательной точки и многоугольника

Тут Славик рассказал способ найти касательную точки и многоугольника с помощью подразбиения многоугольника на подмногоугольники (каждый вложенный берет точки предыдущего через одну). Потом он типа ищет для самого вложенного треугольника касательную, а потом передвигается к более богатым многоугольникам, сдвигая касательную влево или вправо на одну вершину. Тоже алгоритм за  $\log(n)$ . Типа на каждом шаге есть step, мы рассматриваем текущего кандидата на касательную + step и -step. Выбираем лучшего, переходим к нему и делим шаг на два.

### 25.3 Масштабирование дорог

Задача поставлена так: существует некоторый масштаб карты, в условиях которого выражена дорога, представляющая собой полилинию. Требуется выразить эту дорогу в более мелком масштабе, то есть так сократить количество точек, чтобы новая полилиния в некотором смысле была похожа на старую. Сглаженность определяется рациональным числом  $\varepsilon$ : необходимо, чтобы новая прямая не выходила из  $\varepsilon$ -коридора старой прямой. Пример показан на изображении, в нашем случае красная прямая – решение.

Предлагается два решения этой задачи:

1. Будем использовать сумму Минковского и уже разобранные алгоритмы планирования движения. С помощью суммы расширим прямую правильным многоугольником, выбрав радиус описанной окружности соответствующий нужному масштабу, а затем решим задачу об оптимальном

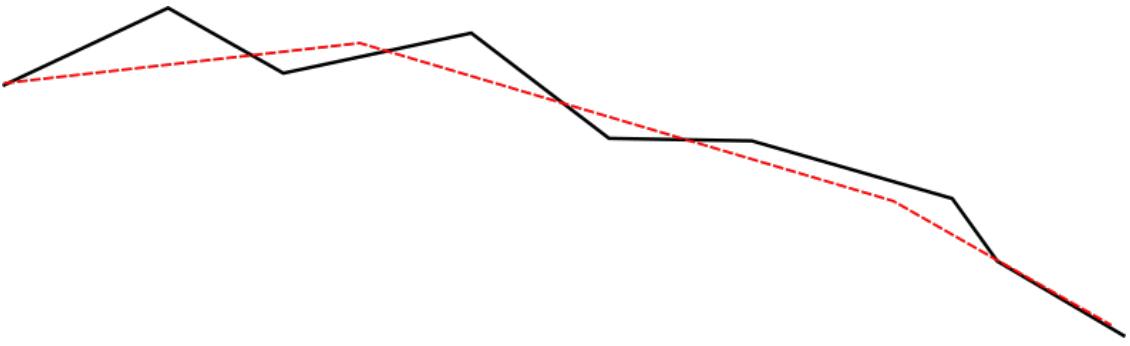


Рис. 58: Пример решения задачи о масштабировании дороги

прохождении роботом коридора, который и представляет собой расширение. Напомним, что в этом случае при использовании, например, графа видимости, асимптотика решения составит  $O(n^2 \log n)$  на препроцессинг и  $O(n)$  на решение.

2. Применим стандартную тактику "разделяй и властвуй" и решим задачу за амортизированное  $O(n \log n)$  ( $O(n^2)$  в худшем случае, алгоритм Дугласа-Пекера):

```

Function Douglas-Peucker(points,  $\varepsilon$ )
  Data: Набор точек, задающих полилинию  $\{p_i\}$ 
  Коэффициент расширения  $\varepsilon$ 
  Result: Набор точек, задающих решение (полилинию)
  range  $\leftarrow [1, n]$ ;
  while range.length  $> 0$  do
    segment  $\leftarrow \langle \text{points}[\text{range.start}], \text{points}[\text{range.end}] \rangle$ ;
    pmax  $\leftarrow \min(\text{points}[\text{range.start}], \text{points}[\text{range.end}])$ , компаратор по дистанции до seg);
    if dist(pmax, segment) < ε then
      return (segment.start, segment.end);
    else
      p1  $\leftarrow$  Douglas-Peucker(points[0..pmax],  $\varepsilon$ );
      p2  $\leftarrow$  Douglas-Peucker(points[pmax..n],  $\varepsilon$ );
      return union(p1, p2);
    end
  end

```

Алгоритм 10: Алгоритм Дугласа-Пекера для масштабирования полилинии

## 25.4 Звездные многоугольники

**Определение** (Звездный многоугольник). *Многоугольник звездный, если существует точка  $q$ , что для любой вершины многоугольника  $a$  отрезок  $qa$  не пересекает ни одного ребра многоугольника. То есть из точки  $q$  "видно" все остальные вершины.*

**Определение** (Ядро звездного многоугольника). *Ядром звездного многоугольника – это множество точек, из которых видны все вершины многоугольника.*

Заметим, что ядро всегда выпукло, так как является пересечением полуплоскостей всех ребер многоугольника. Отсюда и алгоритм поиска (за  $O(n)$ ).

Умеем триангулировать звездный многоугольник.

1. Если есть ядро, выберем любую точку, соединим ее со всеми вершинами и флипами будем менять, пока  $q$  нельзя будет убрать.  
Можно также отрезать уши по очереди, отрезая немного от ядра. Уши нужно выбирать только "выпуклые то есть те, которые имеют внутренний угол  $< 180^\circ$ . Можно отрезать все уши, избегая отрезания того, в котором  $q$ . Отрезать его последним.
2. Если нет ядра, то можно построить за  $O(n)$ . Если не искать, то можно *как-то* за  $O(n)$  или  $O(n \log n)$ .

## 26 Источники



Рис. 59: Spasibo za wnimanie))