МОДИФИЦИРОВАННЫЙ АЛГОРИТМ ЧЕНА ПОСТРОЕНИЯ ВЫПУКЛОЙ ОБОЛОЧКИ НА ПЛОСКОСТИ

Р.В. ЧАДНОВ, А.В. СКВОРЦОВ, Н.С. МИРЗА

Задача построения выпуклой оболочки является фундаментальной задачей вычислительной геометрии. Важность этой задачи объясняется не только огромным количеством приложений (в распознавании образов, обработке изображений, базах данных, в задаче раскроя и компоновки материалов, математической статистике), но также и полезностью выпуклой оболочки как инструмента решения множества задач вычислительной геометрии.

Эта задача позволяет разрешить целый ряд других, иногда с первого взгляда не связанных с ней вопросов: построение диаграмм Вороного, построение триангуляций и т.д. Построение выпуклой оболочки конечного множества точек на плоскости довольно широко исследовано и имеет множество приложений. Очень широко алгоритмы построения выпуклой оболочки используются в геонформатике и геоинформационных системах.

Задача построения выпуклой оболочки имеет давнюю историю. Она является одной из первых задач вычислительной геометрии, с которой начала зарождаться эта наука. В настоящее время известно достаточно большое число алгоритмов построения выпуклой оболочки. Основная масса этих алгоритмов может быть разделена на два типа:

1. Алгоритм Грэхема, его модификации, алгоритм «разделяй и властвуй», быстрый алгоритм построения выпуклой оболочки. Эти алгоритмы имеют трудоемкость $O(n \log n)$.

2. Обход Джарвиса. Если за h обозначить количество вершин выпуклой оболочки, то этот алгоритм имеет трудоемкость O(nh). Такие алгоритмы, трудоемкость которых зависит и от n (размера входных данных), и от h (размера выходных данных), называются *чувствительными к результату*.

Стоит заметить две вещи. С одной стороны, алгоритмы первого типа оптимальны в среднем. Однако если предположить, что h – количество вершин выпуклой оболочки, мало, то трудоемкость алгоритма Джарвиса меньше, чем алгоритмов первого типа. К примеру, если h известно заранее или постоянно, то алгоритм Джарвиса имеет линейную трудоемкость, тогда как алгоритмы из первой группы всё равно имеют логарифмическую трудоемкость. С другой стороны, если h гораздо больше, чем log n, то алгоритм Джарвиса проигрывает. К примеру, если $h = n$, то этот алгоритм показывает квадратичную трудоемкость.

Эти рассуждения наводят на следующую проблему: существует ли алгоритм, по крайней мере такой же быстрый, как оба вышеприведенных класса алгоритмов на любых входных данных?

Алгоритм Чена

Оказывается, существуют даже более быстрые алгоритмы. Это было показано в 1986 году Г. Киркпатриком и Р. Зейделем [1]. Они описали алгоритм с трудоемкостью $O(n \log h)$. Однако этот алгоритм имел чрезвычайно сложную логику.

Позже, в 1994 году, Тимоти Чен [2, 3, 4] предложил *чрезвычайно простой алгоритм* с трудоемкостью $O(n \log h)$. Алгоритм Чена является оригинальным обобщением алгоритмов Джарвиса и Грэхема.

Предположим, что нам известно количество вершин выпуклой оболочки h. Тогда алгоритм Чена можно описать следующим образом:

Шаг 1. Произвольно разделим исходное множество S на подмножества $S_1, S_2, \ldots, S_{\lceil n/h \rceil}$, каждое (кроме, возможно, последнего) размером h.

Шаг 2. Для каждого i, $1 \le i \le \lceil n/h \rceil$, найдем выпуклую оболочку множества точек S_i и обозначим её C_i.

Шаг 3. Теперь переходим к вычислению выпуклой оболочки всего множества *S*, используя методику «заворачивания подарка». Алгоритм начинает свою работу с нахождения точки p_0 из *S*, которая гарантированно является вершиной выпуклой оболочки *S*. Обычно за точку p_0 берется самая нижняя из точек множества *S*. Если *S* содержит несколько точек с минимальной ординатой, то берется самая левая из них. Очевидно, что точка p_0 может быть найдена за время O(n). Пусть p_1 – точка множества *S* такая, что ребро $p_0\,p_1$ является ребром выпуклой оболочки, направленным против часовой стрелки. Далее рассмотрим, каким образом следует находить p_1 (рис. 1).

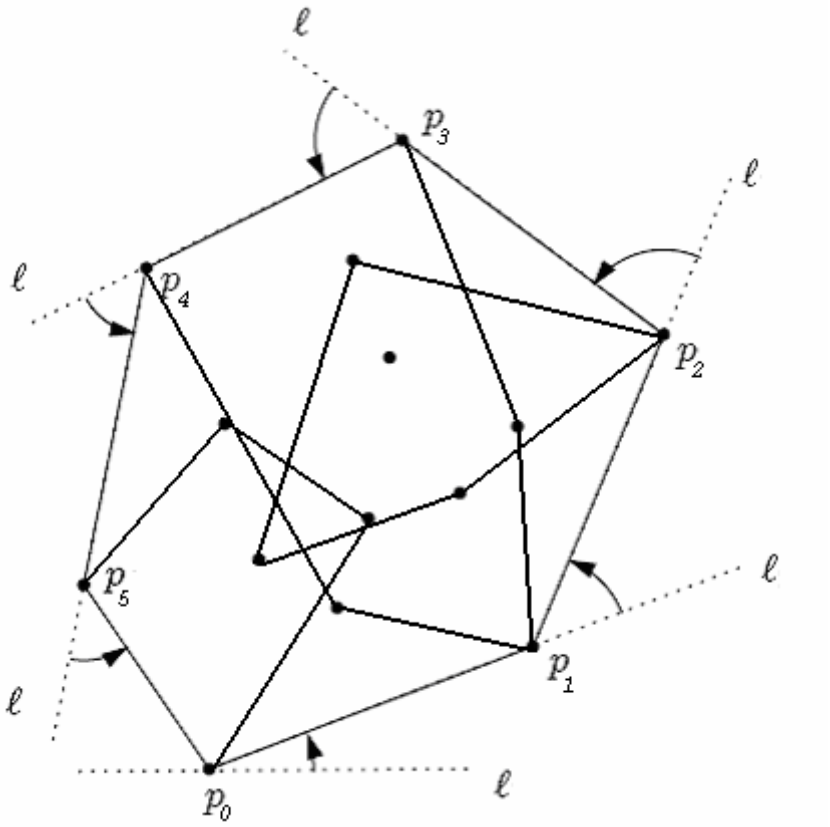

Рис. 1. Алгоритм Чена

Пусть *l* – луч, идущий из p_0 в сторону положительного направления оси абсцисс. Заметим, что все точки множества *S* находятся левее этого луча или на нем. Для каждого i, $1 \le i \le \lceil n/h \rceil$, мы находим точку $q_i \in S_i$, которая является первой точкой, которой коснется луч *l*, если его вращать против часовой стрелки. Далее, за линейное время мы можем найти точку q_j из множества $\{q_1, q_2, ..., q_{\lceil n/h \rceil}\}$, для

которой угол между l и p_0q_j минимален (если таких точек несколько, то выбирается из них та, расстояние от которой до точки p_0 максимально). Очевидно, что точка q_j и является искомой точкой p_1.

Найдя p_1, мы таким же образом находим точку p_2: пусть l' – луч, направленный из p_0 и проходящий через p_1. Как и ранее, все точки множества S находятся левее луча или на нем. Для каждого i, $1 \le i \le \lceil n/h \rceil$, мы выполняем «запрос на экстремальность», то есть находим точку $q_i \in S_i$, которая является первой точкой, которой коснется луч l', если его вращать против часовой стрелки относительно точки p_0. Тогда p_2 – это та точка x из $\lceil n/h \rceil$ точек q_i, для которой угол между l' и p_1x минимален. Мы продолжаем выполнять такие шаги «заворачивания подарка», пока не вернемся к исходной точке p_0. *Конец алгоритма.*

Подсчитаем трудоемкость вышеприведенного алгоритма. Шаг 1 занимает время O(n). Выпуклая оболочка для каждого из исходных подмножеств S_i может быть найдена за время $O(|S_i|\log|S_i|) = O(h\log h)$. Таким образом, шаг 2 в целом имеет трудоемкость

$$O\left(\frac{n}{h}\log h\right) = O(n\log h).$$

Рассмотрим шаг 3. Начальная точка p_0 может быть найдена за линейное время. Для нахождения p_1 мы выполняем «запрос на экстремальность» в каждом из подмножеств S_i. Если осуществлять эти запросы с использованием метода бинарного поиска, то каждый такой запрос будет иметь трудоемкость $O(\log|S_i|) = O(\log h)$.

Таким образом, все запросы вместе займут время $O\left(\frac{n}{h}\log h\right)$.

Это даст нам $\lceil n/h \rceil$ точек-кандидатов на попадание в выпуклую оболочку. За время $O(n/h)$ мы выбираем из них точку p_1. Следовательно, имея p_0, мы находим p_1 алгоритмом с трудоемкостью $O(n/h\,\log h)$.

Так как в выпуклой оболочке h вершин, то общая трудоемкость их нахождения равна $O\left(n + h \times \frac{n}{h}\log h\right) = O(n \log h)$.

То есть весь алгоритм нахождения выпуклой оболочки множества S имеет трудоемкость $O(n \log h)$.

Однако заметим, что это выполняется только при условии априорной известности h. Как должен поступать алгоритм, если h не известно? Трюк состоит в том, что мы «подберем» необходимое значение h следующим специальным способом.

Предположим, что H – это текущее предполагаемое значение h. Мы запускаем вышеприведенный алгоритм, заменив h на H. Шаги 1 и 2 выполняются за время $O(n \log H)$. На шаге 3 могут получиться две различные ситуации:

1. За H шагов «заворачивания подарка» мы вернулись в точку p_0. Это может произойти, только если $h \le H$. В этом случае выпуклая оболочка S с учетом трудоемкости шага 3 алгоритма может быть построена за время

$$O\left(n + h\frac{n}{H}\log H\right) = O(n \log H).$$

2. После H шагов «заворачивания подарка» мы не достигли точки p_0. Это может произойти, только если $h > H$, то есть в нашем предположении количество вершин выпуклой оболочки было слишком мало. В этом случае мы останавливаем алгоритм после H шагов. При этом мы затратили время $O(n \log H)$, но так и не нашли выпуклой оболочки множества S. Так как в нашем предположении величина h была слишком мала, мы увеличиваем H и запускаем алгоритм снова.

Какие значения необходимо давать H на первом и последующих шагах приближения? Обычно начинают с $H = 4$ или любого другого небольшого числа. Каждый раз, когда обнаруживается, что $h < H$, будем принимать $H = H^2$.

Пусть H_f – финальное значение H. То есть, запуская алгоритм в предположении, что количество вершин выпуклой оболочки равно H_f, мы успешно заканчиваем построение выпуклой оболочки не более чем за H_f шагов. Тогда мы знаем,

что $H_f \geq h$. Предыдущее предполагаемое значение, которое было равно $\sqrt{H_f}$, оказалось слишком мало, то есть $H_f < h^2$.

Теперь мы можем вычислить трудоемкость алгоритма в целом. Для каждого предполагаемого значения H мы тратим времени не больше $c \cdot n \log H$. Следовательно, на финальном шаге мы тратим времени не больше $c \cdot n \log H_f$. Трудоемкость предпоследнего шага не больше $c \cdot n \log\left(\sqrt{H_f}\right) = \frac{1}{2} c \cdot n \log H_f$.

В общем для i-го шага подбора H мы тратим времени не больше

$$c \cdot n \log\left(\sqrt[2^i]{H_f}\right) = \left(\frac{1}{2}\right)^i c \cdot n \log H_f .$$

Тогда общее время работы алгоритма равно

$$\sum_{i \geq 0} \left(\frac{1}{2}\right)^i c \cdot n \log H_f \leq 2c \cdot n \log H_f \leq 2c \cdot n \log h^2 = 4c \cdot n \log h = O(n \log h).$$

Полученная трудоемкость говорит о том, что цель – найти алгоритм, имеющий преимущества методов Грэхема и Джарвиса, достигнута. В среднем этот алгоритм имеет трудоемкость $O(n \log h)$. В худшем случае этот алгоритм будет иметь трудоемкость $O(n \log n)$.

Достоинством алгоритма Чена является тот факт, что он имеет трудоемкость $O(n \log h)$ в среднем, при этом в худшем случае он является оптимальным. К недостаткам же можно отнести то, что алгоритм не является открытым, т.е. необходимо априорное знание всего множества точек.

Алгоритм бинарного поиска

В описанном выше алгоритме предполагается, что выполнение «запроса на экстремальность» имеет логарифмическую трудоемкость. Эта трудоемкость может быть достигнута только с использованием специализированной структуры данных. Структура, удовлетворяющая вышеприведенным требованиям, была описана Д. Киркпатриком и Д. Добкиным в [5].

Рассмотрим следующую проблему: имеется множество *S*, состоящее из *n* точек. Необходимо хранить это множество в такой структуре данных, которая позволяет отвечать на «запросы на экстремальность».

Смысл такого запроса заключается в следующем. Пусть даны некая точка *q* на плоскости и луч *l*, направленный из *q* вправо. Все точки множества *S* лежат или на луче *l*, или слева от него. Ответом на запрос будет являться точка, которую *l* коснется первой, если его вращать вокруг точки *q* против часовой стрелки (рис. 2).

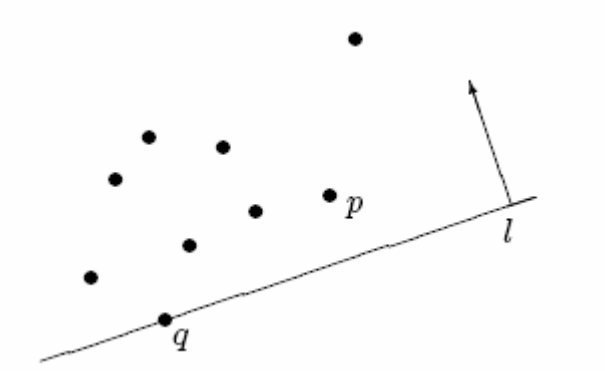

Рис. 2. Если вращать *l* вокруг *q*, то первой точкой, которой коснется *l*, будет точка *p*

Очевидно, что точка *p* может быть ответом на «запрос на экстремальность» только если она является вершиной выпуклой оболочки множества *S*. Следовательно, в искомой структуре данных необходимо хранить только точки выпуклой оболочки. Пусть $P = (p_0, p_1, \ldots, p_{h-1})$ – вершины выпуклой оболочки *S*, пронумерованные в порядке против часовой стрелки. Будем хранить эти точки, используя *иерархическое представление P*. Это представление выглядит следующим образом: пусть P_0=*P*. Для каждого $i > 0$ определим P_i как последовательность, полученную из P_{i-1} удалением каждой второй точки, начиная с точки, имеющей нулевой индекс. Пусть *k* – такой индекс, что P_k содержит ровно два элемента. Тогда список последовательностей $H(P) = (P_0, P_1, \ldots, P_k)$ называется иерархическим представлением последовательности *P* (рис. 3). Очевидно, что $k < \log h$.

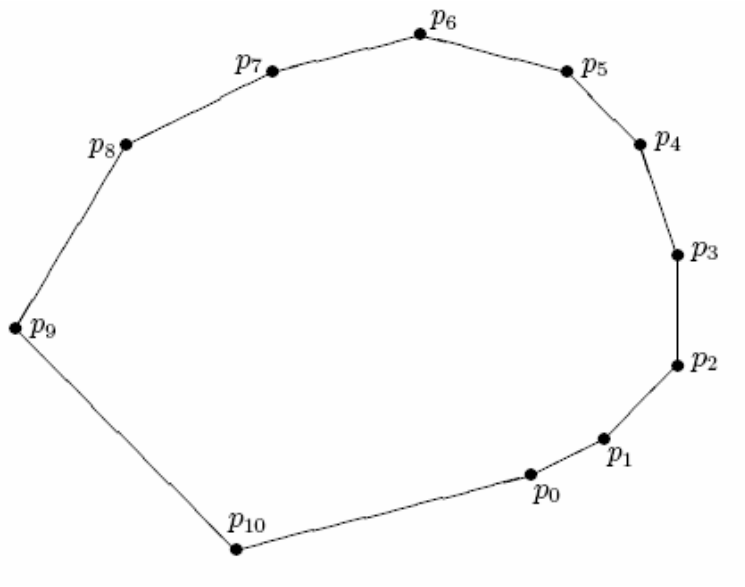

Рис. 3. Иерархическое представление выпуклого многоугольника P состоит из $P_0=P$, $P_1=(p_0, p_2, p_4, p_6, p_8, p_{10})$, $P_2=(p_0, p_4, p_8)$, $P_3=(p_0, p_8)$

Процедура построения иерархического представления такова:

1. Вычислим выпуклую оболочку S, используя, к примеру, обход Грэхема. Это даст нам последовательность P_0.

2. Для каждого i =1,2,… построим последовательность P_i на основе $P_{i\text{-}1}$, удаляя каждую вторую точку.

Первый шаг этого алгоритма имеет трудоемкость $O(n \log n)$. Трудоемкость второго шага равна $O\left(\sum_{i=0}^{k} h_i\right) = O(n)$. Таким образом, иерархическое представление какого-либо множества может быть получено за время $O(n \log n)$.

Однако даже вышеприведенный алгоритм не годится для применения на практике из-за большого количества используемой памяти. Авторами было предложено хранить только число k – количество уровней в иерархическом представлении, а для выполнения экстремальных запросов использовать следующий алгоритм, имитирующий прохождение по слоям иерархического представления:

Шаг 1. Зная k, сравнить углы для точек p_0 и p_{2^k}. Используя формулу (4), выбрать точку p, сохранить её индекс r.

Шаг 2. Для каждого $e = k-1, k-2,...,0$, повторяем следующие действия: сравниваем углы у точек $p=p_r$, p_{r+2^e} и p_{r-2^e}, ту из них, которая имеет наименьший угол по отношению к l, сохраняем в p, а её индекс – в r.

После окончания этого алгоритма в p будет лежать точка, являющаяся ответом на экстремальный запрос к исходному множеству точек.

Модифицированный алгоритм

Проанализировав алгоритм Чена, который имеет оптимальную трудоемкость $O(n \log h)$, авторы пришли к выводу, что этот алгоритм можно модифицировать, увеличив его быстродействие.

Несмотря на приемлемую теоретическую трудоемкость, алгоритм Чена делает слишком много лишних действий, которые, несомненно, уменьшают его практическое быстродействие, так как каждый шаг подбора значения h включает в себя построение выпуклых оболочек частей исходного множества точек. Однако результаты этого построения учитываются однобоко – только для построения текущего участка выпуклой оболочки. При этом в алгоритме не предусмотрено какого-либо влияния результатов одного шага на другой.

Весьма целесообразным было бы использовать результаты построения выпуклых оболочек частей исходного множества для отсечения точек, которые заведомо не попадут в выпуклую оболочку множества в целом. Очевидно, что точки, отброшенные во время построения выпуклой оболочки какой-либо части исходного множества, должны быть отброшены и из точек – потенциальных вершин выпуклой оболочки всего множества. Рассмотрим рис. 4. Мы можем отбросить точки p_1, p_4, p_8, p_{10}, p_{11}, p_{12}, p_{18}. Использование этого отсечения позволит увеличить скорость работы алгоритма Чена.

Данная модификация позволяет не только увеличить скорость, но и уменьшить трудоемкость алгоритма Чена. Трудоемкость первого шага алгоритма равна $O(n \log 4) = O(n)$. Начиная с первого шага, количество точек уменьшается, и после

каждого шага оно равно сумме количеств точек во всех выпуклых оболочках подмножеств исходного множества *S*.

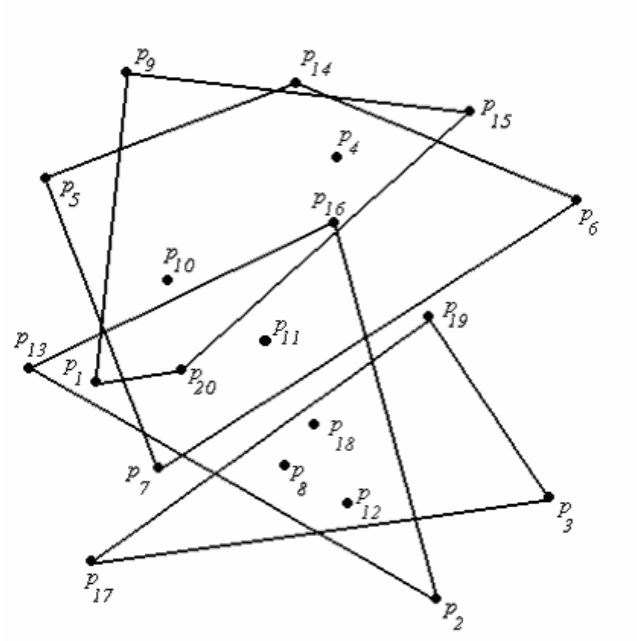

Рис. 4. Отсечения в алгоритме Чена

Согласно [6], для точек, равномерно распределенных внутри единичного круга, количество точек на выпуклой оболочке равно $O(n^{1/3})$. Таким образом, если точки были разбиты случайным образом, в рассмотрении после отбрасывания по результатам первого шага останется $4 \cdot \sqrt[3]{n/4}$ точек.

Если проводить отбрасывание только после первого шага, то получим, что действия, сделанные начиная со второго шага, соответствуют действиям для построения выпуклой оболочки множества из $4 \cdot \sqrt[3]{n/4}$ точек. Так как первый шаг имел линейную трудоемкость, то общая трудоемкость алгоритма равна

$$O(4 \cdot (n/4)^{1/3} \log h) + O(n) = O(n^{1/3} \log h) + O(n) = O(\max(n^{1/3} \log h\,, n)).$$

После построения части выпуклой оболочки, если *H* оказалось меньше реального *h*, алгоритм Чена увеличивает *H* и начинает построение оболочки снова. Если же сохранять часть оболочки, полученную на предыдущих шагах, то можно увеличить скорость работы алгоритма, однако его трудоёмкость останется той же. Таким образом, эта модификация является исключительно практически значимой,

так как очевидно, что такой подход может увеличивать скорость работы алгоритма почти вдвое.

Алгоритм Чена, использующий приведенные улучшения: первоначальное отсечение точек прямоугольником, отсечение точек на каждом шаге работы алгоритма, а также сохранение уже построенных граней выпуклой оболочки, был назван *модифицированным алгоритмом Чена*.

Заключение

Авторами был проведен количественный анализ предложенного алгоритма путём моделирования работы различных алгоритмов построения выпуклых оболочек на различных распределениях точек. При этом анализировалась скорость работы алгоритмов.

Для сравнения алгоритмов в различных условиях в экспериментах строились наборы данных, располагавшихся в единичном квадрате $[0,1)^2$, со следующими характеристиками:

1.Равномерное распределение. Все точки были распределены равномерно и независимо в единичном интервале.

2.Нормальное распределение. Все точки размещались в единичном квадрате по нормальному распределению с центром в точке (0,5; 0,5) и среднеквадратическим отклонением 0,1. Точки, попадавшие за пределы единичного квадрата, отбрасывались.

3. Распределение Лапласа. Все точки размещались в единичном квадрате по распределению Лапласа с центром в точке (0,5; 0,5). Точки, попадавшие за пределы единичного квадрата, отбрасывались.

4. Распределение точек на окружности. Все точки были равномерно распределены по окружности радиуса 0,5 с центром в точке (0,5; 0,5).

5. Кластерное распределение. Внутри единичного квадрата случайно выбиралось (равномерно и независимо по обеим координатам) случайное количество точек, которые становились центрами кластеров. Внутри кластеров точки распреде-

лялись по равномерному распределению в окружности случайного радиуса, описанной вокруг центра кластера.

Результаты проведенных экспериментов показали, что различные алгоритмы ведут себя на различных классах входных данных. Предложенный алгоритм показал лучшую скорость работы на всех распределениях. Однако, несмотря на использование множества улучшений, модифицированный алгоритм Чена остался чувствительным к результату, что показали итоги его тестирования.

Таким образом, можно сказать, что модифицированный алгоритм Чена является наиболее быстрым, практически применимым алгоритмом, чувствительным к результату.

ЛИТЕРАТУРА

1. Kirkpatrick D.G., Seidel R. The ultimate planar convex hull algorithm? // SIAM Journal on Computing. – 1986. – Vol. 15. – P. 287–299.
2. Chan T.M. Output-Sensitive Construction of Convex Hulls: Ph.D. thesis. – Department of Computer Science: University of British Columbia, 1995. – 104 p.
3. Chan T.M. Optimal output-sensitive convex hull algorithms in two and three dimensions // Discrete & Computational geometry. – 1995.
4. Chan T.M. Output-sensitive results on convex hulls, extreme points and related problems // Proceedings of the 11th Annual ACM symposium on Computational Geometry. – 1995. – P. 10–19.
5. Dobkin D.P., Kirkpatrick D.G. Fast detection of polyhedral intersection // Theoretical computer science. – 1983. – Vol. 27. – P. 241–253.
6. O'Rourke J. Computational geometry in C. – Cambridge University Press, 1994. – 376 p.