

## Лекции 3-4. Алгоритмы геометрического поиска

### 1. Задача поиска

*Задача поиска* в общей абстрактной форме: есть набор данных, именуемый *файлом*, и некоторый новый элемент данных, именуемый *образцом*, нужно установить связь между образцом и файлом.

Пример - задача двоичного поиска числа  $x$  в отсортированном по возрастанию массиве  $a[0..n]$ . Алгоритм двоичного поиска использует упорядоченность элементов массива. Для заданного подмассива  $a[n_1..n_2]$  ключ поиска  $x$  сравнивается с элементом  $a[m]$ , расположенным примерно посередине между  $n_1$  и  $n_2$ . Если  $x$  не больше, чем  $a[m]$ , то двоичный поиск рекурсивно продолжается в левом подмассиве  $a[n_1..m]$ , в противном случае (при  $x$  больше  $a[m]$ ) рекурсивный двоичный поиск продолжается для правого подмассива  $a[m+1..n_2]$ . В конечной ситуации  $n_1=n_2$  и возвращается значение  $n_1$ , если  $x=a[n_1]$ , или -1, если  $x \neq a[n_1]$ .

*Алгоритм двоичного поиска в отсортированном массиве*

```
FUNCTION BinarySearch(x:INTEGER; a:SortArray;  
                     n1,n2:INTEGER): INTEGER;  
  
VAR m: INTEGER;  
BEGIN  
  IF n1=n2 THEN  
    IF a[n1]=x THEN Result:=n1  
    ELSE Result:=-1  
  ELSE  
    BEGIN  
      m:=(n1+n2) DIV 2;  
      IF (x<=a[m]) THEN Result:=BinarySearch(x,a,n1,m)  
      ELSE Result:=BinarySearch(x,a,m+1,n2)  
    END;  
  END;  
END;
```

Двоичный поиск числа  $x$  в массиве  $a[0..n]$  осуществляется обращением к функции  
 $i := \text{BinarySearch}(x, a, 0, n);$

Время работы двоичного поиска в наихудшем случае определяется рекуррентным соотношением

$$T(n) = 2T(n/2) + a \quad \text{при } n > 1$$

$$T(n) = b \quad \text{при } n = 1$$

Нетрудно показать, аналогично тому, как это делалось для алгоритма сортировки массива, что  $T(n) \in \Theta(\log n)$ .

### 2. Геометрический поиск

*Геометрический поиск* имеет особенности, связанные со специфическим характером данных. Во-первых, они отображают сложные структуры (полигоны, полиэдры и т.п.), а во вторых, результатом поиска может оказаться не элемент файла, а положение образца относительно файла.

Поисковое сообщение, в соответствии с которым ведется просмотр файла, называется *запросом*. Разовый запрос называется *уникальным*. Запросы, обработка которых повторяется многократно на одном и том же файле, называются *массивом*. В последнем случае,

возможно, стоит расположить информацию в виде структуры, облегчающей поиск. Анализ требуемого ресурса оценивается по четырем критериям:

- 1) *Время запроса*. Сколько времени необходимо на один запрос?
- 2) *Память*. Сколько памяти необходимо для структуры данных?
- 3) *Время предобработки*. Сколько времени необходимо для организации данных перед поиском?
- 4) *Время корректировки*. Сколько времени потребуется на включение элемента данных в структуру или на удаление их нее?

Главные модели геометрического поиска:

А) *Задача локализации*, когда файл представляет собой разбиение геометрического пространства на области, а запрос является точкой. Нужно определить область, содержащую эту точку.

В) *Задача регионального поиска*. Файл содержит набор точек пространства, а запрос есть некоторая стандартная геометрическая фигура, произвольно перемещаемая в этом пространстве. Нужно извлечь (задача отсчета) или подсчитать (задача подсчета) все точки внутри запросной области.

### 3. Локализация точки в простом многоугольнике при уникальном запросе

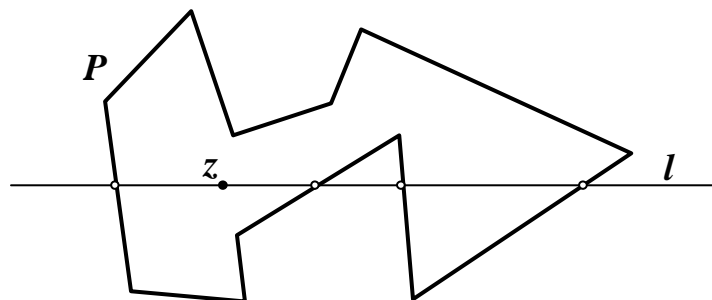
*Простой* многоугольник - не имеющий самопересечений.

**Задача 1** (Принадлежность точки простому многоугольнику). Даны простой многоугольник  $P$  и точка  $z$ . Определить, находится ли  $z$  внутри  $P$ .

Рассмотрим случай уникального запроса.

Проведем через точку  $z$  горизонталь  $l$ . Если  $l$  не пересекает  $P$ , то  $z$  - внешняя точка.

Пусть теперь  $l$  пересекает  $P$ .



Рассмотрим сначала случай, когда  $l$  не проходит через вершины  $P$ . Пусть  $L$  - число точек пересечения  $l$  с границей  $P$  левее  $z$ . Поскольку  $P$  ограничен, левый конец  $l$  лежит вне  $P$ . Двигаясь из  $-\infty$  направо, при первом пересечении границы попадаем внутрь  $P$ , при втором - выходим наружу из  $P$ , при третьем - снова внутрь и т.д. Поэтому  $z$  лежит внутри  $P$  тогда и только тогда, когда  $L$  нечетно.

Теперь рассмотрим вырожденный случай, когда  $l$  проходит через вершины  $P$ . Бесконечно малый поворот  $l$  вокруг  $z$  против часовой стрелки не изменит локализации точки, но устранил вырожденность. Теперь видно следующее. Если обе вершины ребра, принадлежат  $l$ , то его не следует учитывать. Если же только одна вершина ребра лежит на  $l$ , то пересечение следует учесть, если это вершина с большей ординатой, и игнорировать в противном случае.

#### *Алгоритм локализации точки в простом многоугольнике*

```
BEGIN  $L := 0$  ;  
  FOR  $i := 1$  TO  $N$  DO (* цикл по всем ребрам *)  
    IF (ребро( $i$ ) не горизонтально)  
      AND (ребро( $i$ ) пересекает  $l$  нижним концом слева от  $z$ )  
      THEN  $L := L + 1$  ;
```

```

IF ( $L$  нечетно) THEN  $z$  внутри ELSE  $z$  снаружи;
END;

```

Очевидно, что требуемое время выполнения этого алгоритма -  $O(n)$ , где  $n$  - число вершин многоугольника. Таким образом, доказана

**Теорема.** Принадлежность точки внутренней области простого  $n$ -угольника можно установить за время  $O(n)$  без предобработки.

#### 4. Локализация точки в выпуклом многоугольнике при массовом запросе

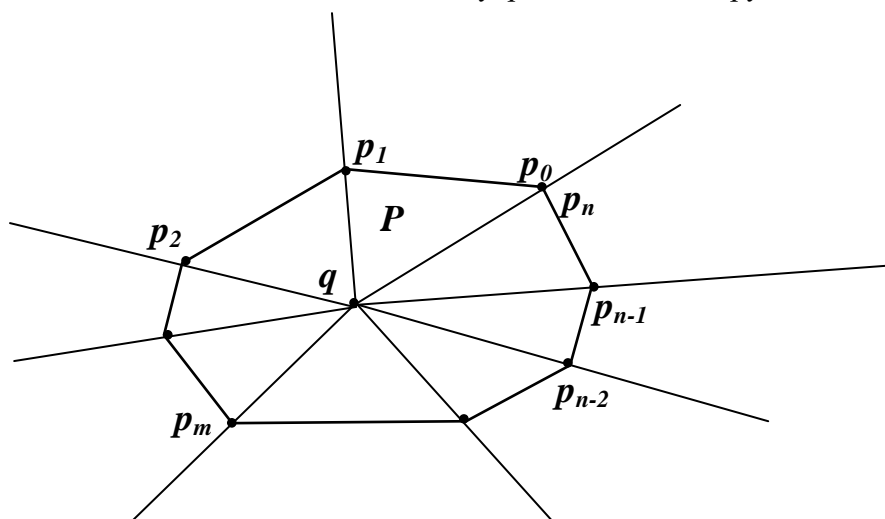
*Выпуклый* многоугольник - целиком лежащий по одну сторону относительно любого своего ребра.

**Задача 2** (Принадлежность точки выпуклому многоугольнику). Даны выпуклый многоугольник  $P$  и точка  $z$ . Определить, находится ли  $z$  внутри  $P$ .

Рассматривается случай массового запроса.

Вершины выпуклого многоугольника упорядочены по полярным углам относительно любой внутренней точки. В качестве таковой можно взять, например, середину любой диагонали (если вершин больше трех), либо центр тяжести любого треугольника, образованного вершинами полигона. Обозначим эту внутреннюю точку  $q$ .

Рассмотрим  $n$  лучей, выходящих из  $q$  и проходящих через вершины многоугольника  $P$ . Эти лучи разбивают плоскость на  $n$  клиньев, а каждый клин разбивается ребром полигона на две части. Одна из этих частей лежит целиком внутри полигона, а другая - целиком снаружи.



Таким образом мы можем сначала отыскать клин, в котором лежит точка  $z$ , а затем сравнить  $z$  с единственным ребром, разделяющим этот клин. Поиск клина, содержащего  $z$ , выполняется методом двоичного поиска.

Таким образом, нужно выполнить предобработку, которая состоит в следующем. Нужно найти точку  $q$  и расположить вершины полигона в какой-нибудь структуре данных, пригодной для двоичного поиска. В качестве такой структуры подойдет вектор, компонентами которого являются вершины, перечисленные в направлении против часовой стрелки  $(p_0, p_1, \dots, p_n)$ , причем  $p_0 = p_n$ .

Проверка попадания точки  $z$  в клин, определяемый углом  $(p_i, q, p_{i+1})$ , осуществляется следующим образом. Если этот угол меньше  $\pi$ , то достаточно убедиться, что углы  $(p_i, q, z)$  и  $(z, q, p_{i+1})$  положительны. Если же угол  $(p_i, q, p_{i+1})$  больше  $\pi$ , то точка  $z$  попадает в него, если она не попадает в угол  $(p_i, q, p_{i+1})$ . Знак угла  $(p, q, r)$  совпадает со знаком векторного произведения направляющих векторов его сторон  $[(q, p) \times (q, r)]$ .

*Алгоритм локализации точки в выпуклом многоугольнике*

```

TYPE Vector = ARRAY [0..n] OF Point;

```

```

VAR P: Vector;
    q: Point; (* q.x:=P[0].x+P[1].x+P[2].x;
               q.y:=P[0].y+P[1].y+P[2].y; *)

FUNCTION PositivAngle(p,q,r: Point): BOOLEAN;
VAR ax,ay,bx,by: DOUBLE;
BEGIN
    ax:=p.x-q.x; ay:=p.y-q.y;
    bx:=r.x-q.x; by:=r.y-q.y;
    Result:=(ax*by-ay*bx>=0);
END;

FUNCTION FindAngle(P: Vector; q,z: Point): INTEGER;
VAR m: INTEGER;
BEGIN
    n1:=0; n2:=n;
    WHILE (n2-n1>1) DO
        BEGIN
            m:=n DIV 2;
            IF PositivAngle(P[n1],q,P[m]) THEN
                IF PositivAngle(P[n1],q,z)
                    AND PositivAngle(z,q,P[m]) THEN n2:=m
                ELSE n1:=m
            ELSE
                IF PositivAngle(P[m],q,z)
                    AND PositivAngle(z,q,P[n2]) THEN n1:=m
                ELSE n2:=m;
            Result:=n1;
        END;
    END;

FUNCTION PointInTriangle(q,p,r,z: Point): BOOLEAN;
BEGIN
    Result:=(PositivAngle(q,p,r)=PositivAngle(z,p,r));
END;

FUNCTION PointInConvexPolygon(P: Vector; q,z: Point): BOOLEAN;
VAR i: INTEGER;
BEGIN
    i:=FindAngle(P,q,z);
    Result:=PointInTriangle(q,P[i],P[i+1],z);
END;

```

Предобработка в этом алгоритме состоит в поиске точки  $q$  и в размещении полигона в векторе, что требует время  $O(n)$ . Выполнение запроса требует для поиск сектора, содержащего точку  $z$ , времени  $O(\log n)$  и постоянного времени для локализации точки внутри треугольника в секторе  $O(1)$ . Значит, общие затраты на запрос составляют  $O(n)$ . Следовательно, справедлива

**Теорема.** Время ответа на запрос о принадлежности точки выпуклому  $n$ -угольнику равно  $O(\log n)$  при затратах  $O(n)$  памяти и  $O(n)$  времени на предобработку.

## 5. Планарные графы

Граф  $G=(V,E)$  (где  $V$  - множество вершин, а  $E$  - множество ребер) называется *планарным*, если его можно уложить на плоскости без самопересечений. *Плоская укладка* планарного графа - это отображение каждой вершины из  $V$  в точку на плоскости, а каждого ребра из  $E$  - в простую линию, соединяющую пару образов концевых вершин этого ребра так, чтобы образы ребер пересекались только в своих концевых точках. Известно, что любой планарный граф можно уложить на плоскости так, чтобы все ребра отобразились в прямолинейные отрезки.

*Прямолинейная укладка* плоского графа определяет разбиение плоскости, называемое *планарным подразбиением* или *картой*. Плоскость разбивается на связные области, называемые *гранями*.

Пусть в планарном подразбиении  $v$  - число вершин,  $e$  - число ребер,  $f$  - число граней (включая единственную бесконечную грань такого подразбиения). Справедлива следующая классическая

**Теорема Эйлера.** В планарном подразбиении связного графа  $v-e+f=2$ .

### Доказательство.

Рассмотрим последовательно графы  $G_0, G_1, G_2, \dots, G$ , образованные из вершин и ребер графа  $G$  по следующему правилу. Граф  $G_0$  состоит из одной вершины. Граф  $G_1$  получается из  $G_0$  добавлением одного ребра, инцидентного этой вершине, и второй вершины этого ребра. Далее граф  $G_{k+1}$  получается из графа  $G_k$  добавлением одного ребра такого, что хотя бы одна его вершина принадлежит  $G_k$ , и второй вершины этого ребра (если она еще не принадлежала  $G_k$ ).

Для графа  $G_0$  имеем  $v_0=1, e_0=0, f_0=1$  - утверждение теоремы справедливо  $v_0-e_0+f_0=2$ . Пусть для  $G_k$  оно справедливо  $v_k-e_k+f_k=2$ . Покажем, что тогда и для  $G_{k+1}$  оно тоже выполняется. Рассмотрим два возможных случая положения нового ребра, вошедшего в  $G_{k+1}$  и отсутствующего в  $G_k$ . Одна вершина этого ребра принадлежит  $G_k$ .

Рассмотрим сначала случай, когда вторая вершина, инцидентная этому ребру, не принадлежит  $G_k$ . Тогда число ребер в  $G_{k+1}$  по сравнению с  $G_k$  увеличивается на 1, число вершин - тоже на 1, а число граней остается прежним, т.е. имеет место

$$v_{k+1}-e_{k+1}+f_{k+1}=(v_k+1)-(e_k+1)+f_k=2$$

и утверждение теоремы выполняется.

Теперь рассмотрим второй случай, когда вторая вершина вводимого в  $G_{k+1}$  ребра также как и первая принадлежит  $G_k$ . В этом случае ребро делит какую-то грань карты  $G_k$  на две грани. Поэтому  $v_{k+1}=v_k, e_{k+1}=e_k+1, f_{k+1}=f_k+1$  и

$$v_{k+1}-e_{k+1}+f_{k+1}=v_k-(e_k+1)+(f_k+1)=2,$$

что и завершает доказательство теоремы.

Степенью  $\rho(a)$  вершины  $a$  называется число ребер, инцидентных этой вершине.

**Теорема.** В планарном подразбиении связного графа, у которого степень каждой вершины не менее 3, выполняются следующие неравенства:

$$1) v \leq \frac{2}{3}e, \quad 2) e \leq 3v-6,$$

$$3) e \leq 3f-6, \quad 4) f \leq \frac{2}{3}e,$$

$$5) v \leq 2f-4, \quad 6) f \leq 2v-4.$$

### Доказательство.

Докажем неравенство 1.

Из определения степени вершины и из условия теоремы следует  $\sum_{a \in V} \rho(a) = 2e \geq 3v$ ,

откуда получаем требуемое неравенство.

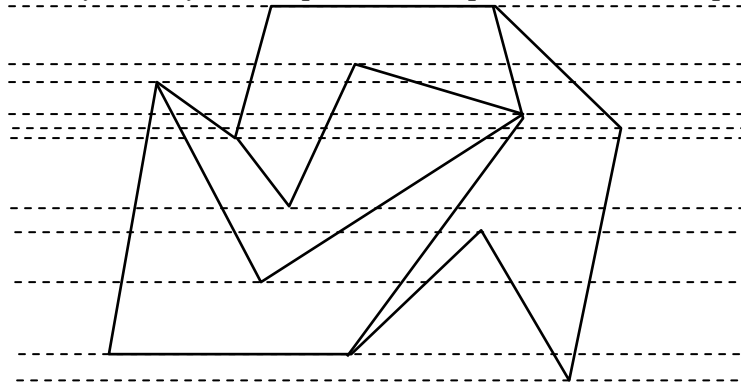
**Следствие.** Величины  $v, e, f$  асимптотически попарно пропорциональны.

## 6. Локализация точки в планарном подразбиении

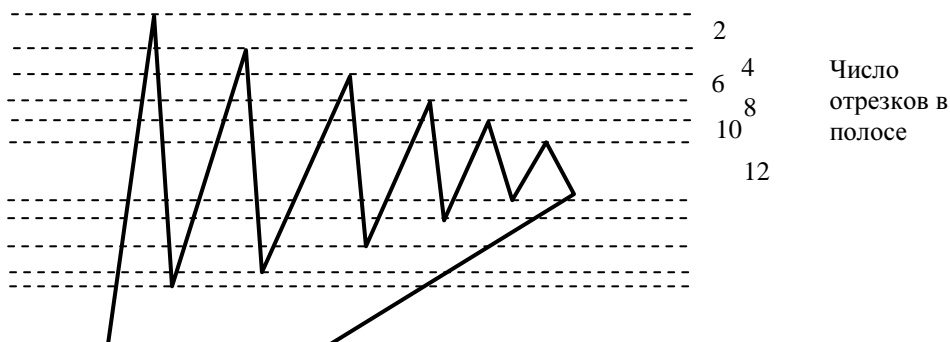
Планарный граф, уложенный на плоскости так, что все его ребра являются прямолинейными отрезками, называется *плоским прямолинейным графом* (ППЛГ). Любой ППЛГ задает подразбиение плоскости на простые многоугольники. Будем рассматривать далее связные ППЛГ, у которых нет вершин степени меньше 2. Тогда согласно предыдущему разделу общее число ребер в ППЛГ составит величину  $O(n)$ .

Задача локализации точки в ППЛГ состоит в поиске многоугольника, содержащего эту точку. Методы локализации основаны на идее создания новых геометрических объектов, допускающих двоичный поиск. В качестве примера метода решения задачи рассмотрим метод полос.

Пусть задан ППЛГ  $G$ , имеющий  $n$  вершин. Тогда согласно предыдущему разделу общее число ребер в ППЛГ составит величину  $O(n)$ . Проведем горизонтальные прямые через каждую его вершину. Они разделяют плоскость на  $n+1$  горизонтальных полос. Если провести сортировку этих полос по координате  $y$  на этапе предобработки, то появится возможность найти ту полосу, в которой лежит пробная точка, за время  $O(\log n)$ .



Рассмотрим пересечение одной из полос с графом  $G$ . Оно состоит из отрезков ребер графа  $G$ . Эти отрезки определяют множество трапеций и треугольников. Эти отрезки можно упорядочить в каждой полосе слева направо и использовать двоичный поиск для нахождения трапеции или треугольника, содержащего пробную точку. Таким образом, можно обеспечить время запроса  $O(\log n)$ . Однако затраты памяти для размещения информации о полосах составят  $O(n^2)$ . Соответствующий случай иллюстрируется следующим ППЛГ.



Простая реализация идеи метода полос состоит в сортировке трапеций и треугольников в каждой полосе. Поскольку количество полос  $O(n)$ , а в каждой полосе число трапеций составляет также  $O(n)$ , получаем, что общее время составит  $O(n^2 \log n)$ . Однако существует более экономичный алгоритм, позволяющий сократить время предобработки до  $O(n^2)$  [Препарата, Шеймос, стр.64-66].

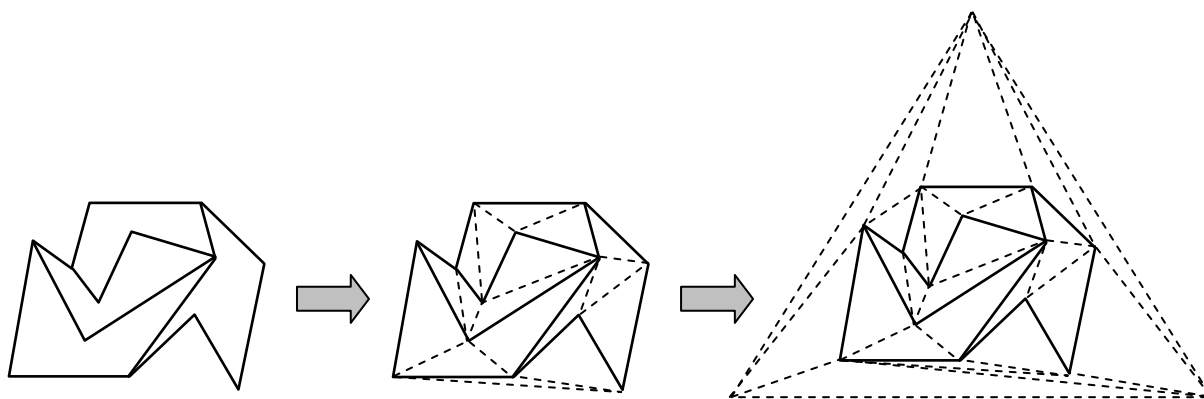
Таким образом, метод полос позволяет сформулировать следующее утверждение.

**Теорема.** Локализация точки в  $n$ -вершинном ППЛГ реализуется за  $O(\log n)$  время с использованием  $O(n^2)$  памяти, если  $O(n^2)$  времени ушло на предобработку.

Данный результат не является пределом. Следует упомянуть замечательный алгоритм Киркпатрика, описанный там же (стр.75-79), обеспечивающий тоже логарифмическое время запроса при памяти  $O(n)$  и времени предобработки  $O(n \log n)$ .

## 7. Алгоритм Киркпатрика

1. Сведение ППЛГ к триангуляции (триангуляция граней).  
Время сведения  $O(n \log n)$ , память  $O(n)$ .
2. Построение охватывающего треугольника.



Далее строится последовательность триангуляций  $S_1, S_2, \dots, S_{h(n)}$ , в которой  $S_1 = G$ , а  $S_i$  получается из  $S_{i-1}$  по следующим правилам.

**Шаг 1.** Удалим некоторое множество независимых (т.е. несмежных) неграничных вершин триангуляции  $S_{i-1}$  и инцидентные к ним рёбра.

**Шаг 2.** Вновь триангулируем многоугольники, образовавшиеся в результате удаления вершин и рёбер.

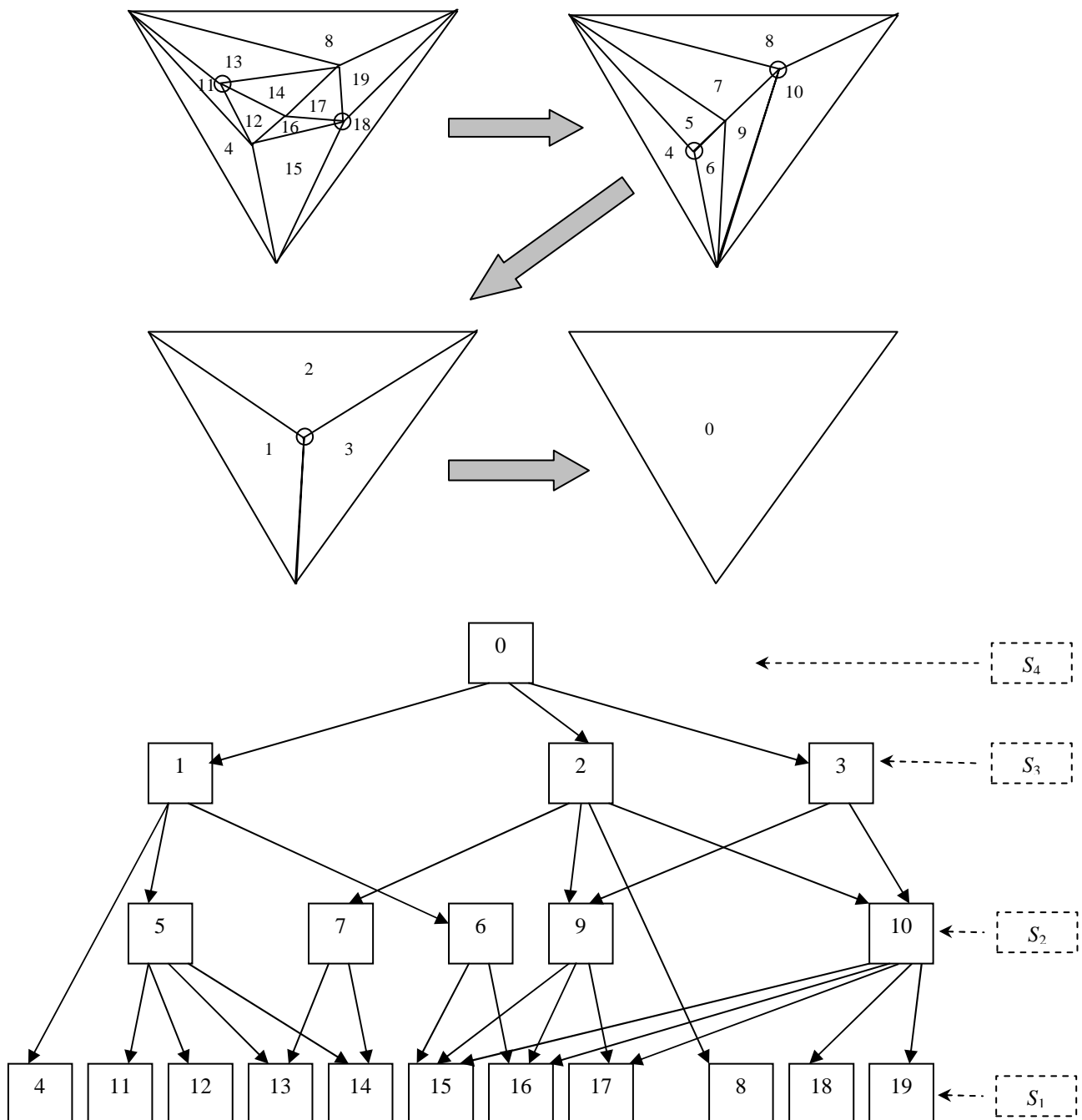
Процесс выполняется до момента, когда в  $S_{h(n)}$  не остаётся внутренних рёбер.

Обозначим треугольник  $R_j$ .

Введём отношение принадлежности треугольников триангуляциям:  $R_j \in S_i$ , если  $R_j$  создан на шаге 2 при построении  $S_i$ .

**Структура данных  $T$**  для поиска – это дерево, узлы которого соответствуют треугольникам.  $T$  – ациклический ориентированный граф. От узла  $R_k$  к  $R_j$  проводится дуга, если при построении  $S_i$  после  $S_{i-1}$  выполнены условия:

- 1)  $R_j$  удаляется из  $S_{i-1}$  на шаге (1)
- 2)  $R_k$  создаётся в  $S_i$  на шаге (2)
- 3)  $R_j \cap R_k \neq \emptyset$ .



#### Алгоритм поиска

Обозначим:

$\mathbf{z}$  – точка-запрос,

$\Gamma(\mathbf{v})$  – список потомков узла  $\mathbf{v}$ ,

**треугольник( $\mathbf{v}$ )** – это треугольник, отнесённый к узлу  $\mathbf{v}$ .

Геометрический поиск осуществляется следующей процедурой.

**PROCEDURE Локализация точки( $\mathbf{z}$ );**

**BEGIN**

**IF ( $\mathbf{z} \notin \text{треугольник(корень)}$ ) THEN**

**Result(« $\mathbf{z}$  лежит в бесконечной области»)**

**ELSE**



```

V:=корень;
WHILE ( $\Gamma(v) \neq \emptyset$ ) DO
    FOR каждый  $u \in \Gamma(v)$  DO
        IF ( $z \in \text{треугольник}(u)$ ) THEN
            V:=u;
        END IF;
    END FOR;
END WHILE;
Result(«z лежит в треугольнике(v)»)
END ELSE;

```

**END Локализация точки.**

**Выбор множества вершин для удаления**

Предположим, что можно выбрать это множество так, чтобы выполнялись следующие свойства:

Свойство 1.  $n_i = \alpha_i \cdot n_{i-1}$ , где  $n_i$  число вершин в триангуляции  $S_i$ ,  $\alpha_i \leq \alpha < 1$  для  $i = 2, \dots, h(n)$ .

Свойство 2. Каждый треугольник  $R_j \in S_i$  пересекается не более, чем с  $H$  треугольниками из  $S_{i-1}$  и наоборот.

Из свойства 1 следует, что время поиска есть  $O(\log n)$ :

$$\begin{aligned}
 n_i = \alpha_i \cdot n_{i-1} &\Rightarrow n_1 = n; \\
 n_2 &= \alpha_2 \cdot n_1 \leq \alpha \cdot n; \\
 n_3 &= \alpha_3 \cdot n_2 \leq \alpha^2 \cdot n; \\
 &\dots\dots\dots \\
 n_k &= \alpha_k \cdot n_{k-1} \leq \alpha^{k-1} \cdot n
 \end{aligned}$$

Процесс заведомо завершится при  $k$  таком, что

$$k: \alpha^{k-1} \cdot n \leq 1 \Rightarrow (k-1) \log \alpha + \log n \leq 0 \Rightarrow k \leq 1 + \frac{\log n}{\log \frac{1}{\alpha}} = O(\log n).$$

Таким образом, высота графа поиска  $O(\log n)$ .

Из свойств 1 и 2 следует, что память  $O(n)$ .

Действительно, из теоремы Эйлера (из  $f \leq 2v-4$ ) следует, что число треугольников  $f_i$  в триангуляции  $S_i$  ограничено  $f_i \leq 2n_i$ . Отсюда следует оценка общего числа узлов

$$f_1 + f_2 + \dots + f_{h(n)} \leq 2(n_1 + n_2 + \dots + n_{h(n)}) \leq 2n_1 (1 + \alpha + \alpha^2 + \dots + \alpha^{h(n)-1}) \leq \frac{2n_1}{1-\alpha}.$$

$$\text{Поскольку каждый узел имеет не более } H \text{ указателей, то и память есть } O\left(\frac{2n}{1-\alpha} \cdot H\right),$$

т.е.  $O(n)$ .

**Время предобработки**

Триангуляция на  $n_1, n_2, \dots, n_{h(n)} = 1$  вершинах требует время

$$n_1 \log n_1, n_2 \log n_2, \dots, n_{h(n)} \log n_{h(n)}.$$

$$\sum_{i=1}^{h(n)} n_i \log n_i \leq \log n \cdot \sum_{i=1}^{h(n)} n_i \leq \log n \cdot \sum_{i=1}^{h(n)} \alpha^{i-1} \cdot n = n \log n \cdot \sum_{i=1}^{h(n)} \alpha^{i-1} \leq n \log n \cdot \frac{1}{1-\alpha}, \text{ т.е.}$$

время предобработки  $O(\log n)$ .

### **Критерий выбора множества удаляемых вершин**

Правило, при котором выполняются свойства 1 и 2, следующее:

*Удалить несмежные вершины со степенью меньше  $K$ . При этом порядок просмотра несущественен.*

Свойство 2 выполняется очевидно. Удаление вершины приводит к появлению многоугольной грани с числом вершин рёбер меньше  $K$ . Значит, на этом месте образуется не более  $K - 2$  новых треугольников в следующей триангуляции. Следовательно, каждый старый треугольник пересекает не более  $K - 2$  новых. Положим  $H = K - 2$  и свойство 2 выполнится.

Свойство 1 устанавливается так. Все вершины имеют степень не ниже 3 (и граничные в том числе). Из теоремы Эйлера число рёбер  $e \leq 3n - 6$ .

Поскольку рёбер не более  $3n - 6$  и каждое инцидентно двум вершинам, то сумма степеней

всех  $n$  вершин меньше  $6n$ . Значит, не менее  $\frac{n}{2}$  вершин имеют степень меньше 12. Положим

$K = 12$ . Пусть  $\nu$  - число выброшенных вершин. Поскольку каждой инцидентно не более 11 рёбер, то не более 11 вершин являются с ней смежными. Значит, удаление вершины из  $\frac{n}{2}$

оставляет более, чем  $\frac{n}{2} - 12$  вершин. Следовательно, удаляется не менее  $\nu \geq \frac{1}{12} \left( \frac{n}{2} - 3 \right)$ ,

т.е.  $\nu \geq \frac{1}{24} n - \frac{1}{4}$  и число оставшихся вершин есть  $n - \nu \leq n - \frac{1}{24} n + \frac{1}{4} \leq \frac{23}{24} n + 1$ , тогда

$$\alpha \approx \frac{23}{24} < 0.959 < 1.$$

Эта оценка грубая и заниженная. Численные эксперименты показывают лучший результат.

Таким образом, доказана

**Теорема.** Локализация точки в  $n$ -вершинном ППЛГ можно осуществить за  $O(\log n)$  время с использованием  $O(n)$  памяти, если  $O(n \log n)$  времени ушло на предобработку.