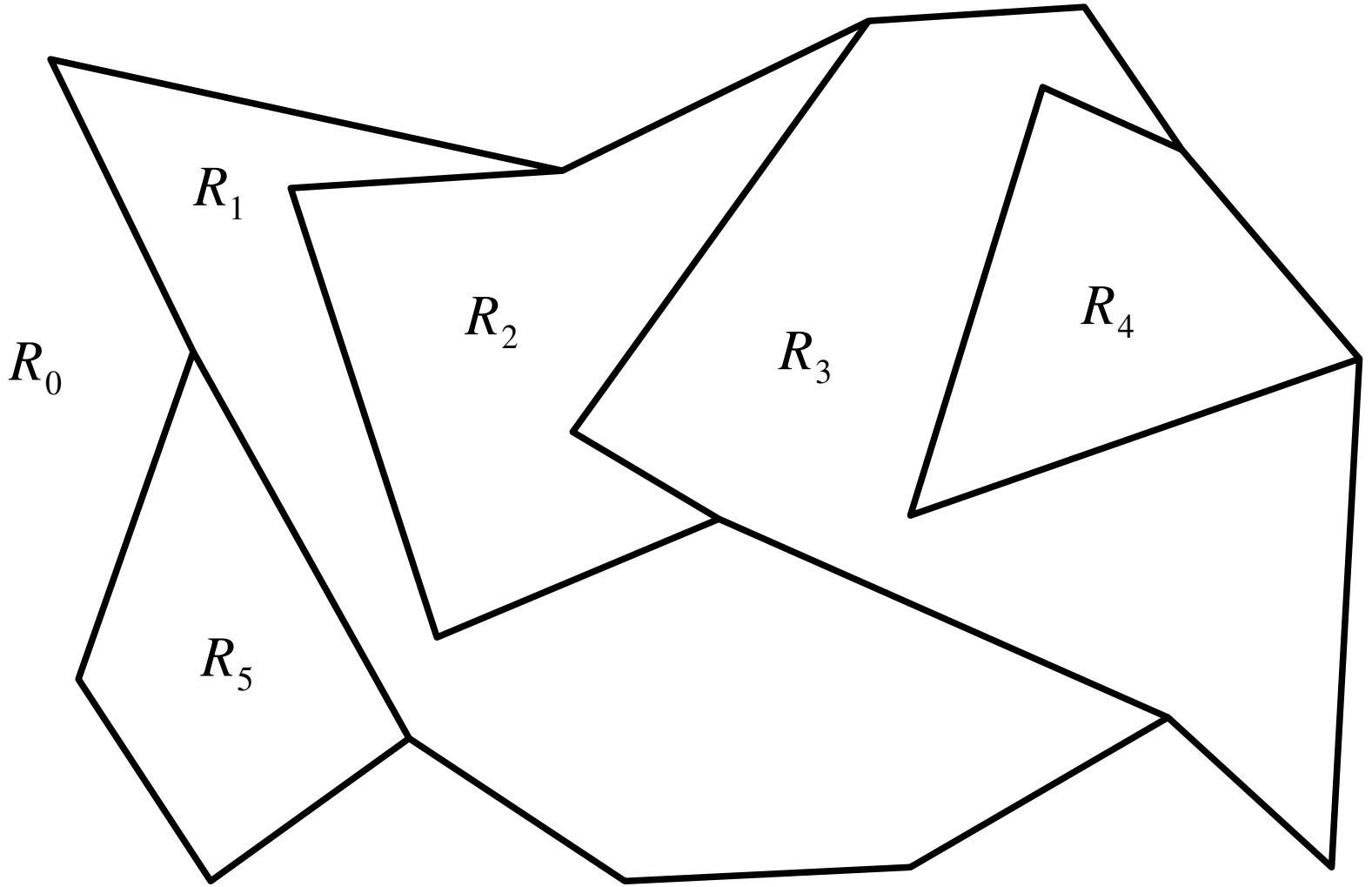
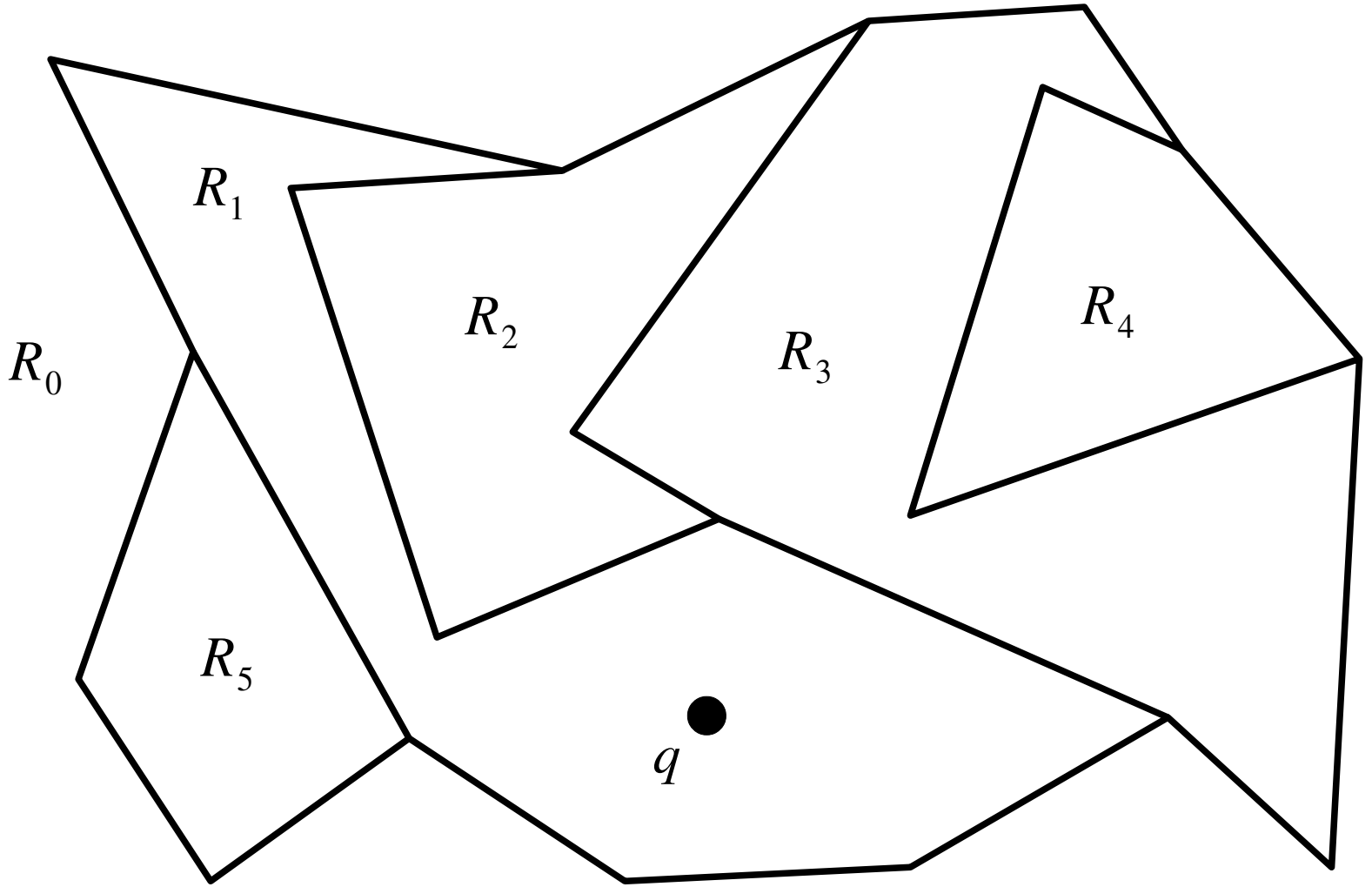
# Punktlokalisierung

# 1. Beschreibung der Aufgabenstellung

Die Ebene ist zerlegt in Regionen.



Wir suchen die Region, in der $q$ liegt.

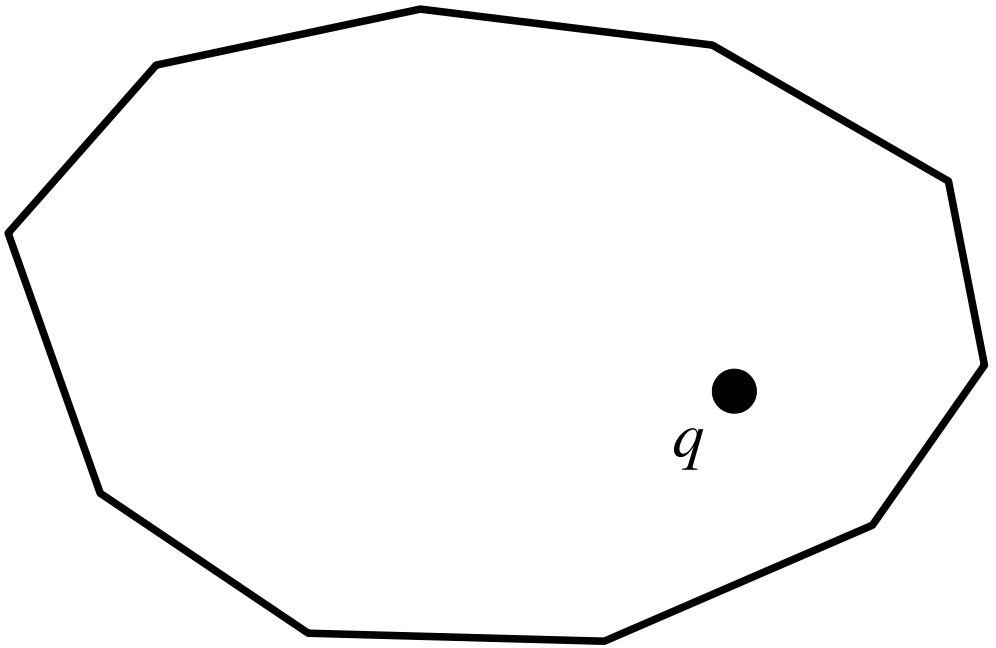


# 2. Einfache Situationen

Frage: Liegt $q$ in einem **Rechteck** $R$?


Solche Anfragen lassen sich für beliebige Punkte leicht in $O(1)$ Zeit beantworten.

Frage: Liegt $q$ in einem **konvexen Polygon** $P$ mit $n$ Ecken?



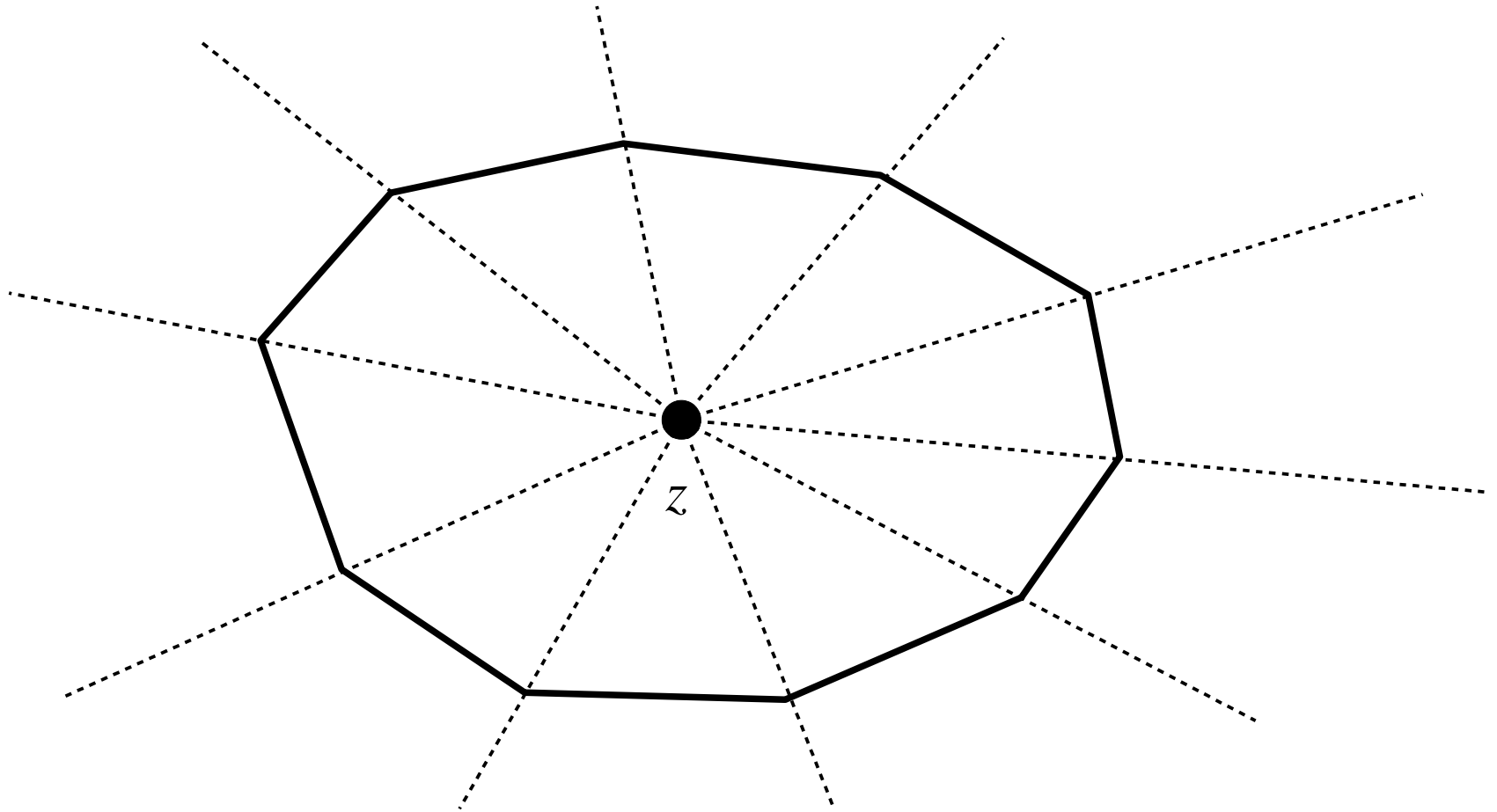

Ohne weitere Vorbereitung kann eine solche Anfrage für einen beliebigen Punkt $q$ in $O(n)$ Zeit beantwortet werden.

Wir haben aber eher folgendes Szenario vor Augen: Für **dasselbe** $P$ werden sehr **viele Anfragen** gestellt.

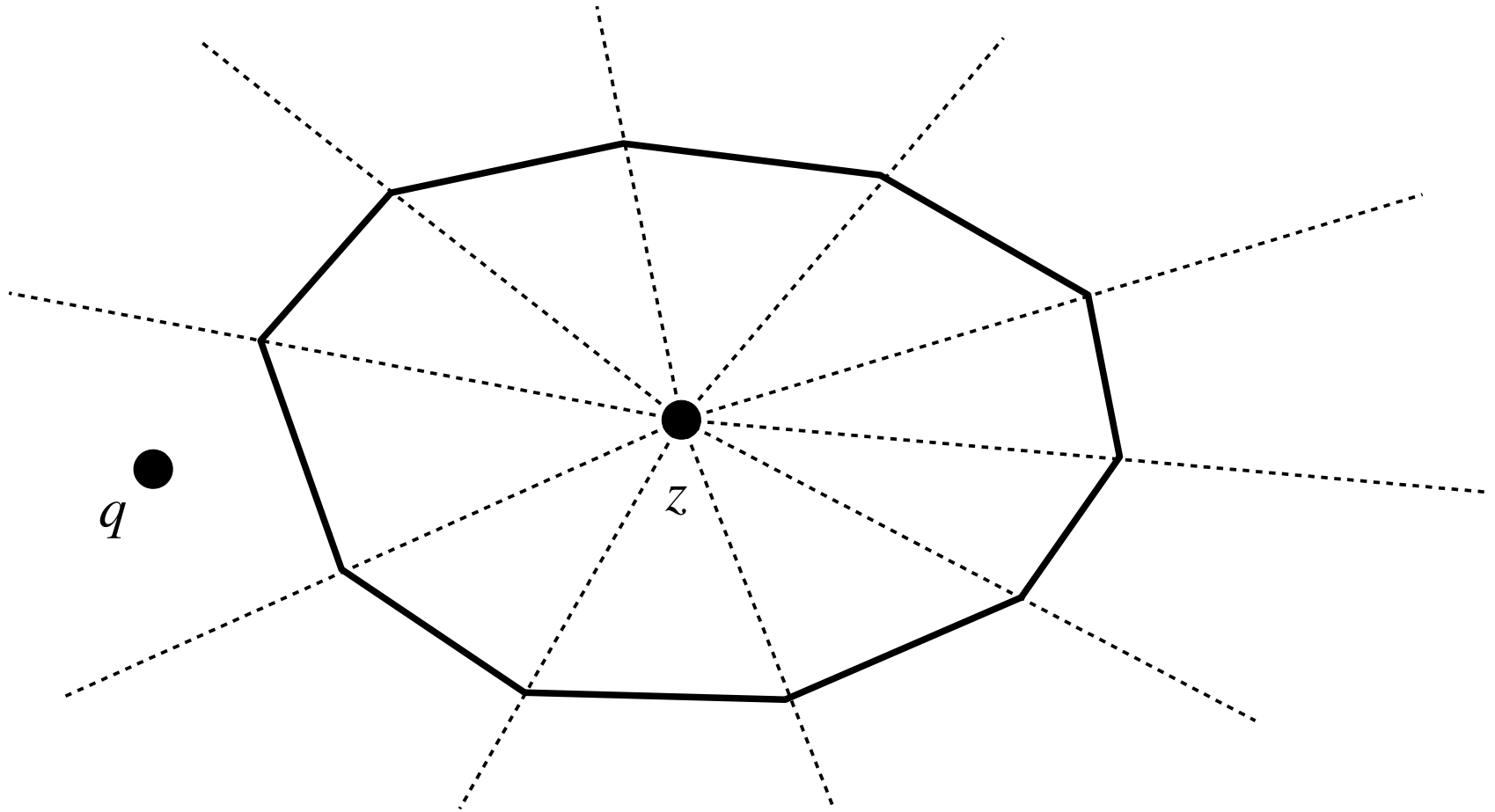
Dann könnte es sich lohnen, $P$ für diese Anfragen so **vorzubereiten**, dass für einen beliebigen Punkt $q$ die Antwort in $O(\log n)$ Zeit gegeben werden kann.

Die Idee besteht nun darin, eine Art **binäre Suche** zu ermöglichen.

Dazu wählen wir einen Punkt $z$ im Inneren von $P$ und konstruieren die Folge der Strahlen mit Anfangspunkt $z$ durch die Ecken von $P$.



Jetzt lokalisieren wir zuerst das **Tortenstück**, in dem $q$ liegt. Das geht mit binärer Suche in $O(\log n)$ Zeit. Anschließend kann man in $O(1)$ Zeit prüfen ob $q$ in $P$ liegt. Der Aufwand zur Vorbereitung ist in $O(n)$.



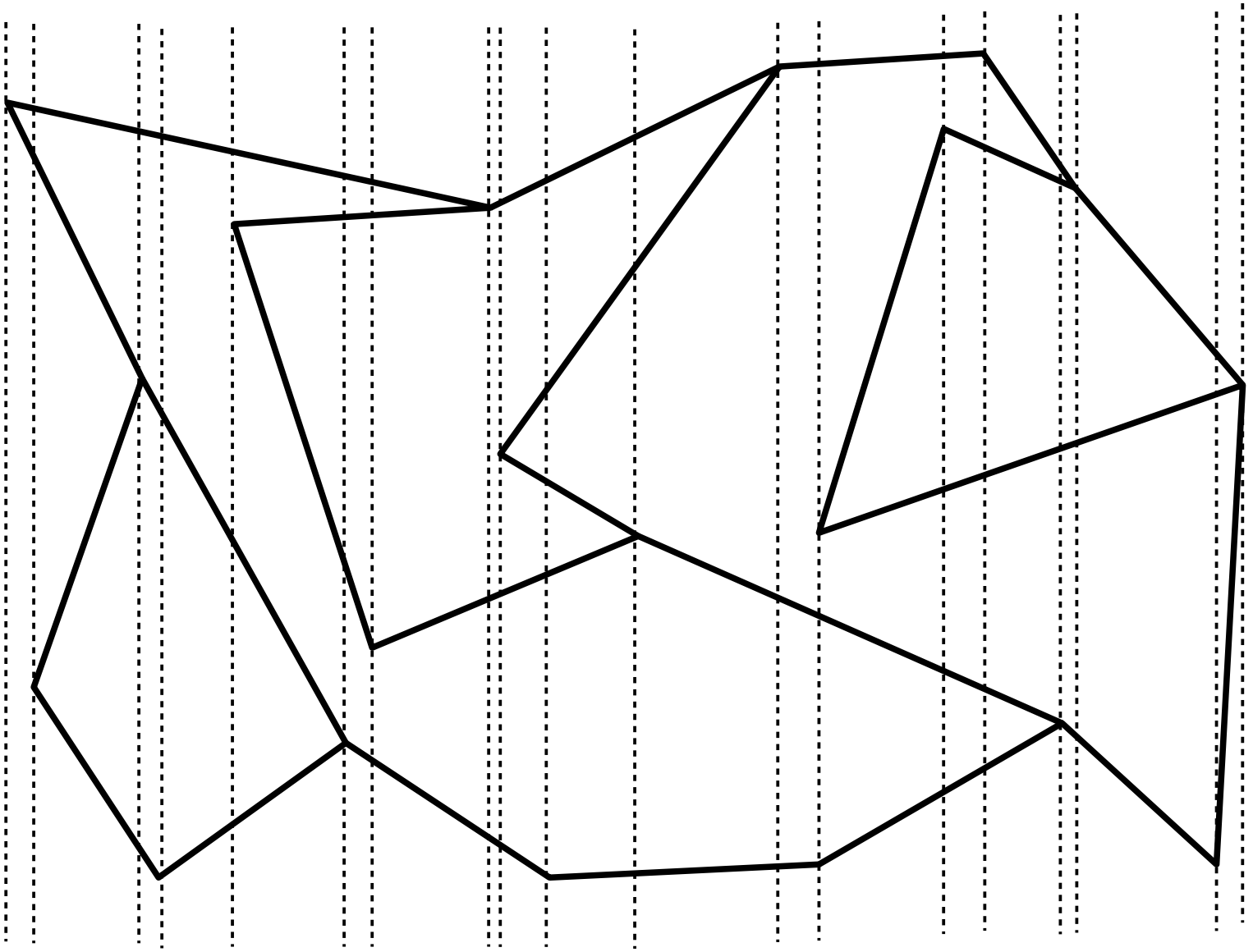
# 3. Einfache Verfahren

Es gibt nun auch einfache Verfahren, die es uns erlauben, **allgemeine polygonale Regionen** so vorzubereiten, dass Punktlokalisierungsanfragen in $O(\log n)$ Zeit bearbeitet werden können. Dabei ist $n$ die Gesamtzahl der Ecken in den Regionen.

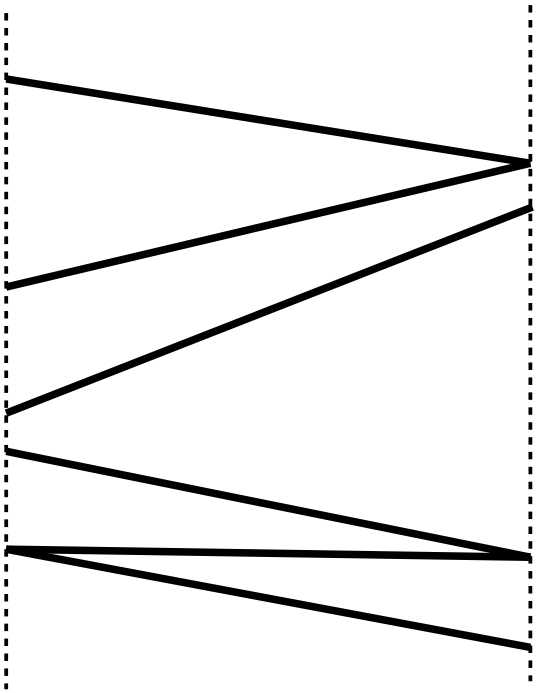
Der Nachteil dieser Verfahren ist, dass der **Aufwand zur Erstellung** entsprechender Datenstrukturen sehr groß ist oder diese Datenstrukturen sehr viel Speicherplatz benötigen.

Trotzdem wollen wir uns ein solches Verfahren etwas genauer ansehen, weil es uns die Idee für eine effizientere Herangehensweise liefern wird.

Wir legen durch **jede Ecke** eine senkrechte Gerade.



Schauen wir uns den **Bereich zwischen zwei** aufeinander folgenden Geraden etwas genauer an.



Da zwischen den zwei Senkrechten keine Eckpunkte liegen, kann man die Kantenstücke dazwischen **von unten nach oben sortieren**.

Damit kann man einen Punkt $q$ wie folgt lokalisieren:

1. Finde mit **binärer Suche** die beiden Senkrechten, zwischen denen $q$ liegt.

2. Lokalisiere $q$ mit **binärer Suche** unter den Kanten zwischen den beiden gefundenen Senkrechten.

Beide Schritte sind natürlich in $O(\log n)$ Zeit machbar und damit kann man Anfragen in der gewünschten Zeit beantworten.

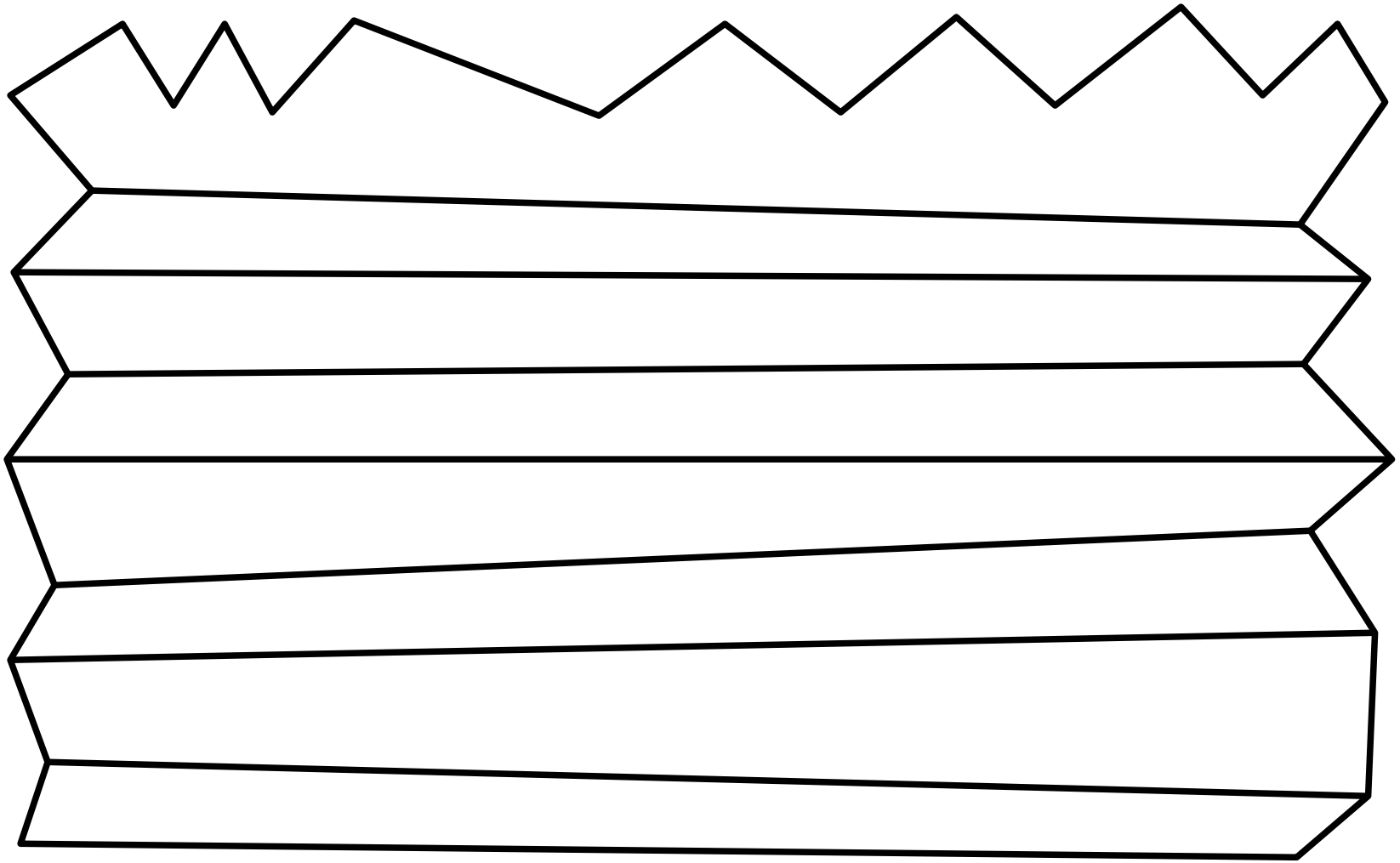
Allerdings ist der Aufwand zur Konstruktion und der Speicherbedarf bei dem eben betrachteten Verfahren **sehr hoch**.

Zur Konstruktion kann man einen Gleitgeradenalgorithmus verwenden.

Dieser würde zwar in $O(n \log n)$ Zeit laufen, aber man muss bei jedem Ereignispunkt den **kompletten Inhalt der Statusstruktur** auslesen und ablegen.

Dies führt im worst case zu einer Laufzeit in $\Omega\left(n^2\right)$.

Die Regionen könnten nämlich so aussehen:



# 4. Ideen für Verbesserungen

Das eigentlich Aufwändige bei der geschilderten Vorgehensweise sind das **Auslesen und Ablegen** der kompletten Statusstruktur bei jedem Ereignispunkt.

Unsere Statusstruktur ist eine Wörterbuchimplementierung, also etwa ein **Rot-Schwarz-Baum**.

Um unser Verfahren zu verbessern, könnten wir versuchen, einen Weg zu finden, das **Auslesen des Wörterbuchs zu vermeiden**.

Hier helfen uns so genannte **persistente Wörterbücher** weiter.

In der uns interessierenden Anwendung ist es ja so, dass wir mit einem **leeren Wörterbuch starten** und dann bei $n$ Ereignispunkten jeweils konstant viele Operationen (Suchen, Einfügen, Löschen) auf dem Wörterbuch ausführen.

Letztlich wird also beginnend bei einem leeren Wörterbuch eine **Folge von $O(n)$ Operationen** auf dem Wörterbuch ausgeführt.

Bei einem persistenten Wörterbuch haben wir nun die Möglichkeit **zu jedem Zeitpunkt** $j > i$ auf das Wörterbuch in dem Zustand zuzugreifen, in dem es sich unmittelbar nach der $i$-ten Operation befand.

Operationen: $O_1, O_2, \ldots, O_m$ mit $m \in O(n)$

Nachdem diese $m$ Operationen ausgeführt wurden, kann man für ein beliebiges $i \in \{1, \ldots, m\}$ zugreifen auf den Zustand desWörterbuchs unmittelbar nach Ausführung von Operation $O_i$.

Dazu übergebe ich der Suchprozedur nicht nur den Schlüssel, nach dem ich suche, sondern auch das $i$.

Das Schöne ist nun, dass man persistente Wörterbücher so implementieren kann, dass alle Operationen weiterhin in $O(\log n)$ Zeit möglich sind und der Speicherbedarf in $O(n)$ bleibt. Wie man dies genau erreicht, soll hier nicht weiter ausgeführt werden.

## Zusammenfassung der Idee

Wir führen wieder den Gleitgeradenalgorithmus aus, verwenden aber als Statusstruktur die Implementierung eines **persistenten** Wörterbuchs.

Damit können wir nun folgendermaßen einen Punkt $q$ lokalisieren:

1. Finde die beiden Senkrechten, zwischen denen $q$ liegt. Damit ist auch klar, auf **welche Version** des Wörterbuchs man im Folgenden zugreifen muss.

2. Lokalisiere mit Hilfe der entsprechenden **Version des Wörterbuchs** den Punkt $q$ unter den Kanten zwischen den beiden Senkrechten.

## Analyse des verbesserten Verfahrens

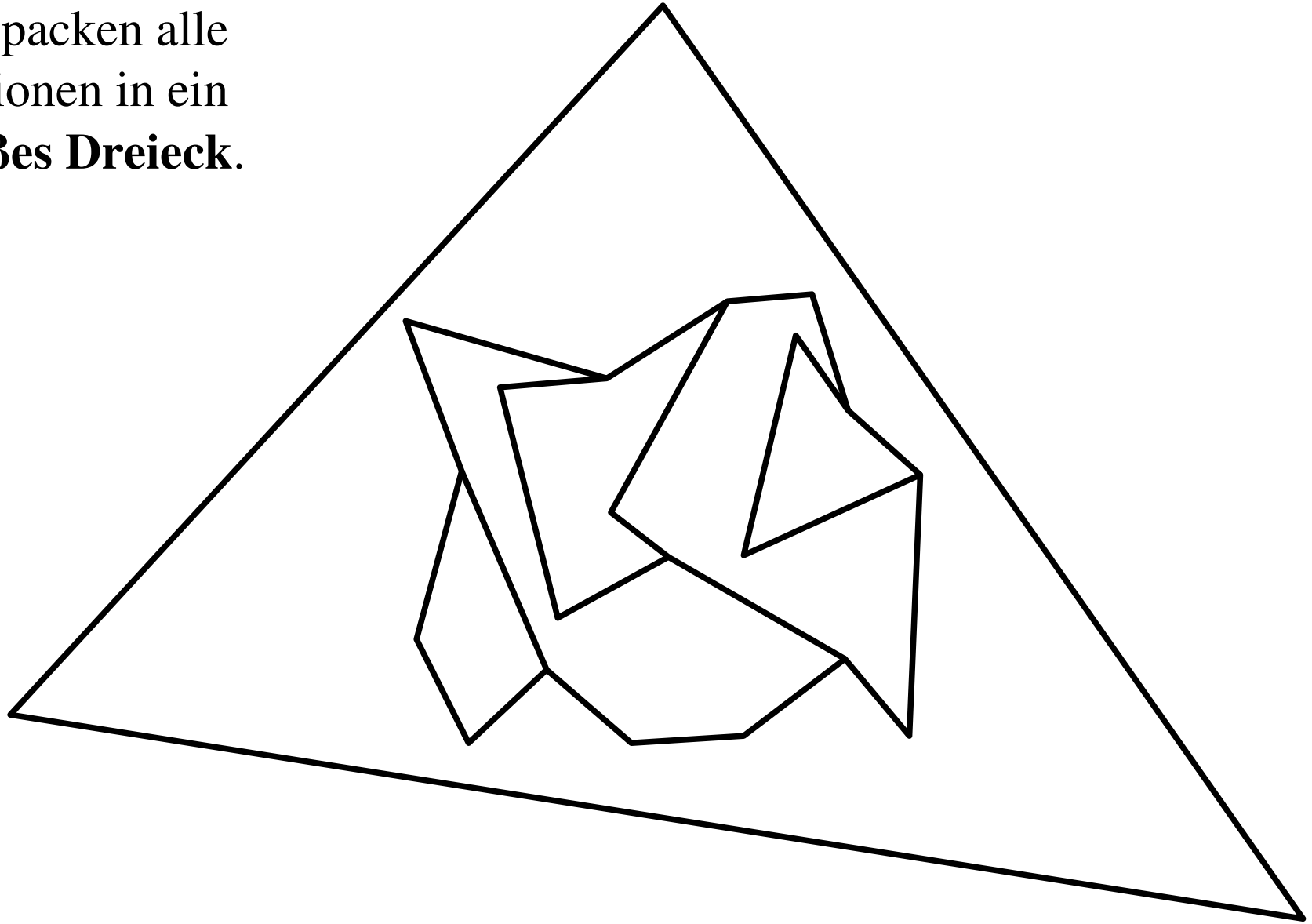
Bearbeitung einer Anfrage: $O(\log n)$

Aufbau der Datenstruktur: $O(n \log n)$

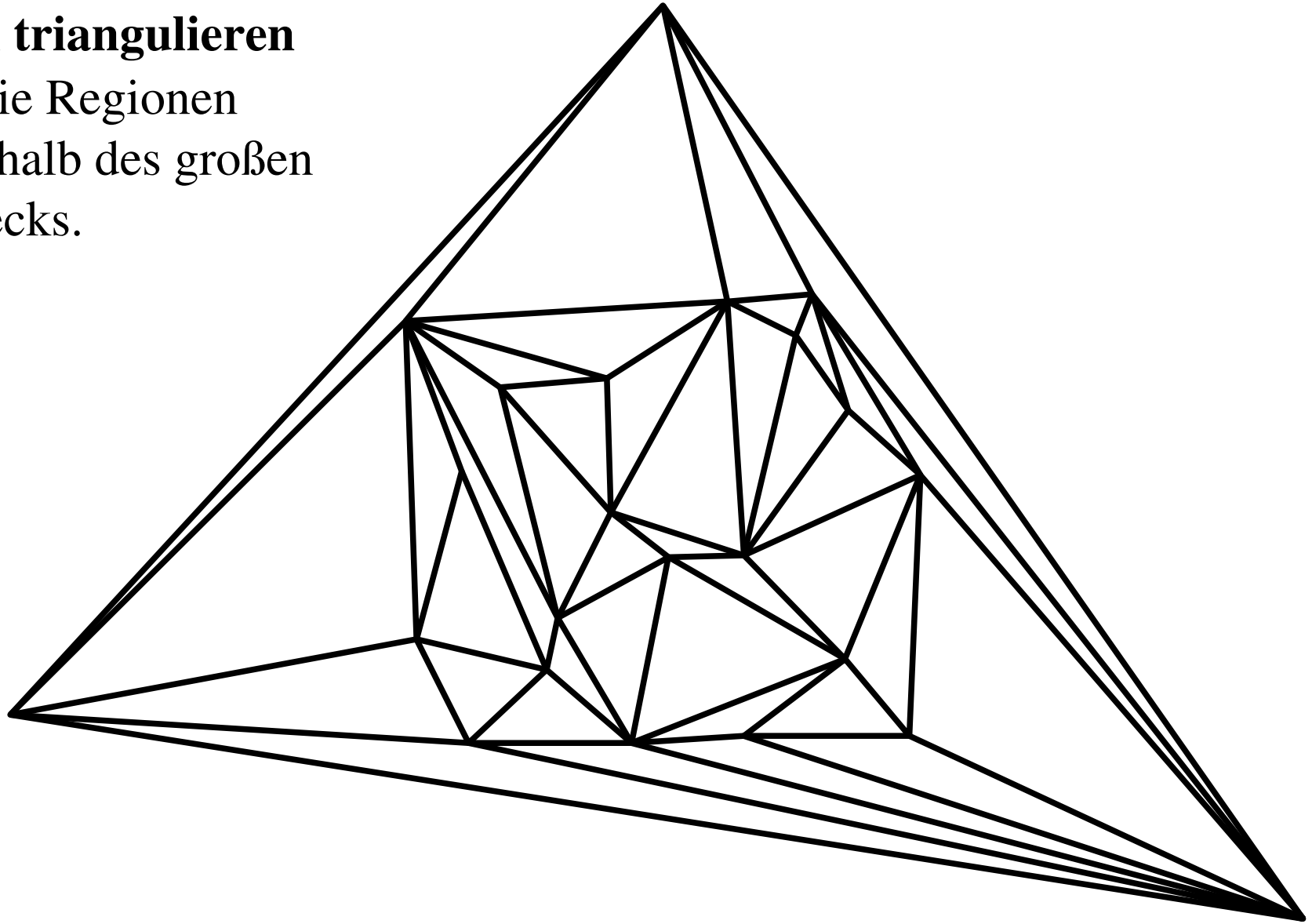
Speicherplatzbedarf: $O(n)$

# 5. Triangulationsbasiertes Verfahren

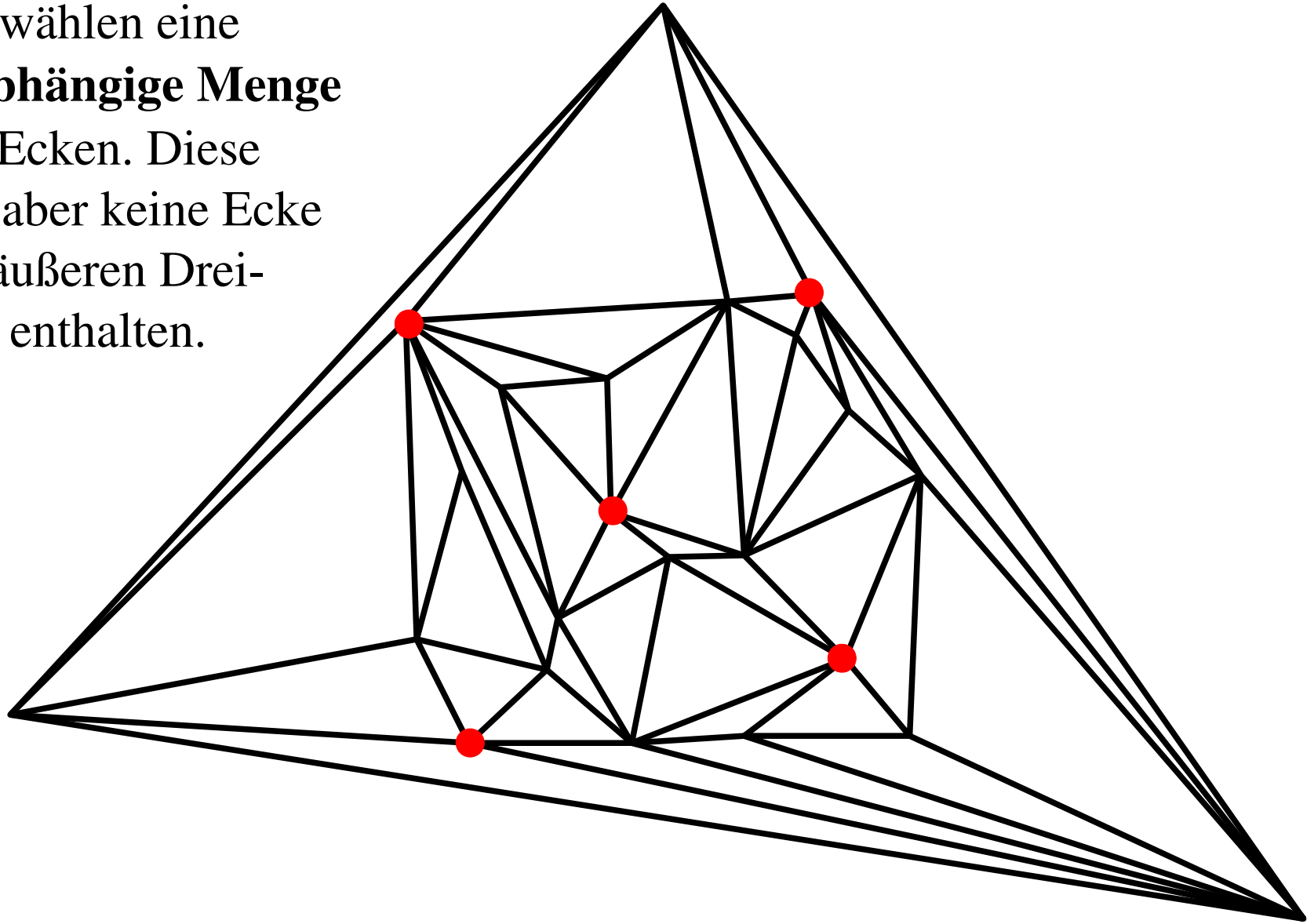
Wir packen alle Regionen in ein **großes Dreieck**.



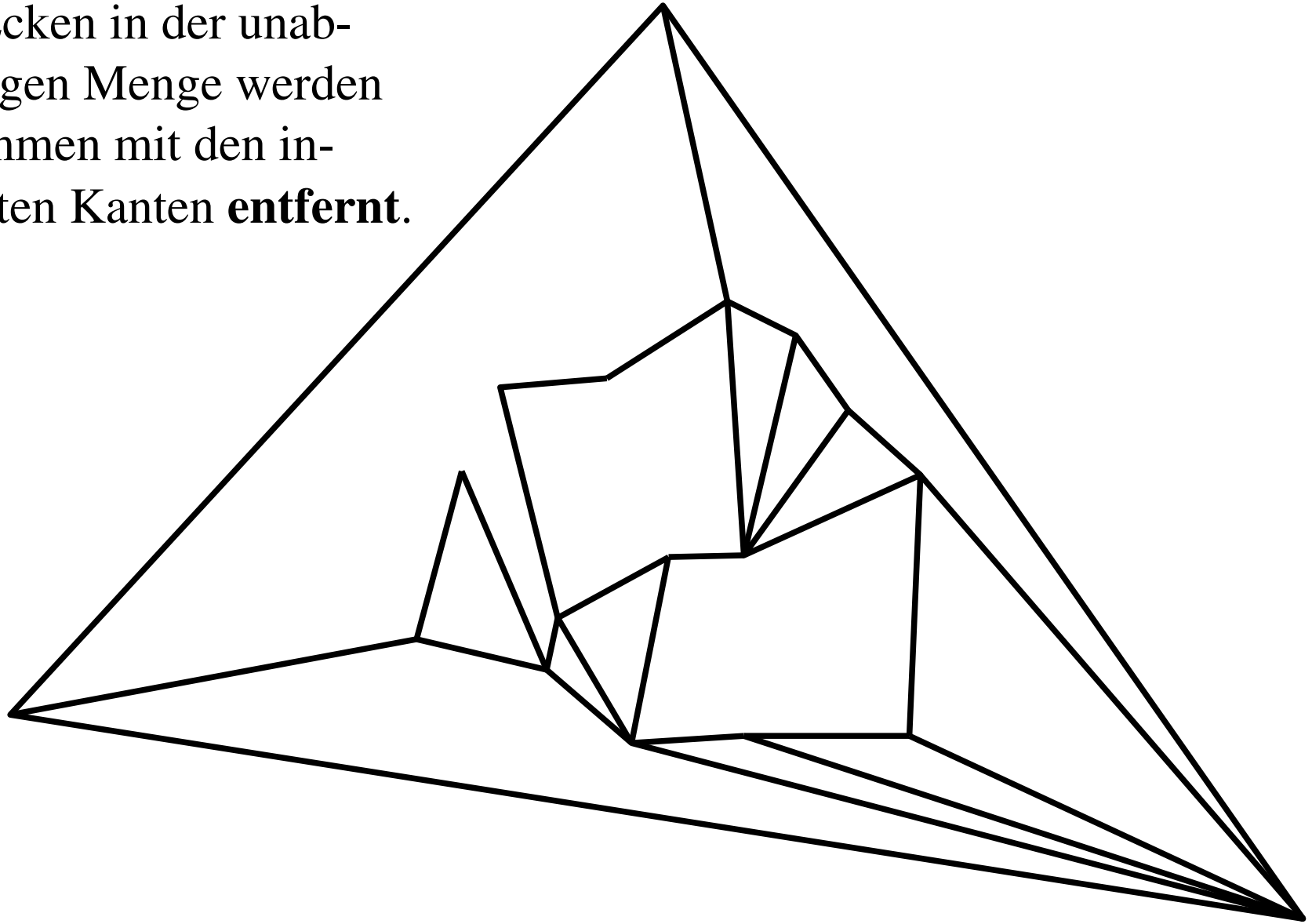
Dann **triangulieren** wir die Regionen innerhalb des großen Dreiecks.



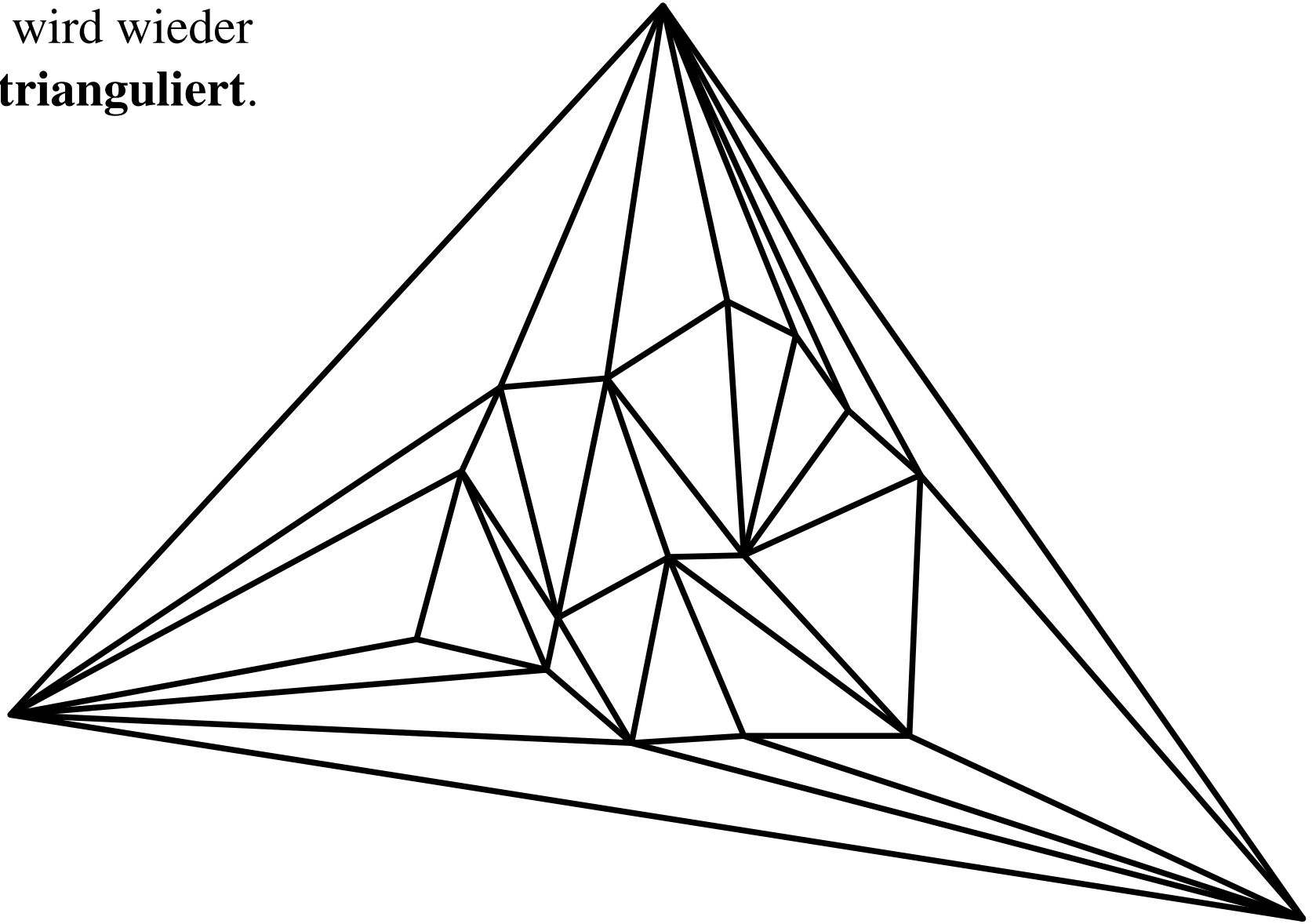
Wir wählen eine **unabhängige Menge** von Ecken. Diese darf aber keine Ecke des äußeren Dreiecks enthalten.



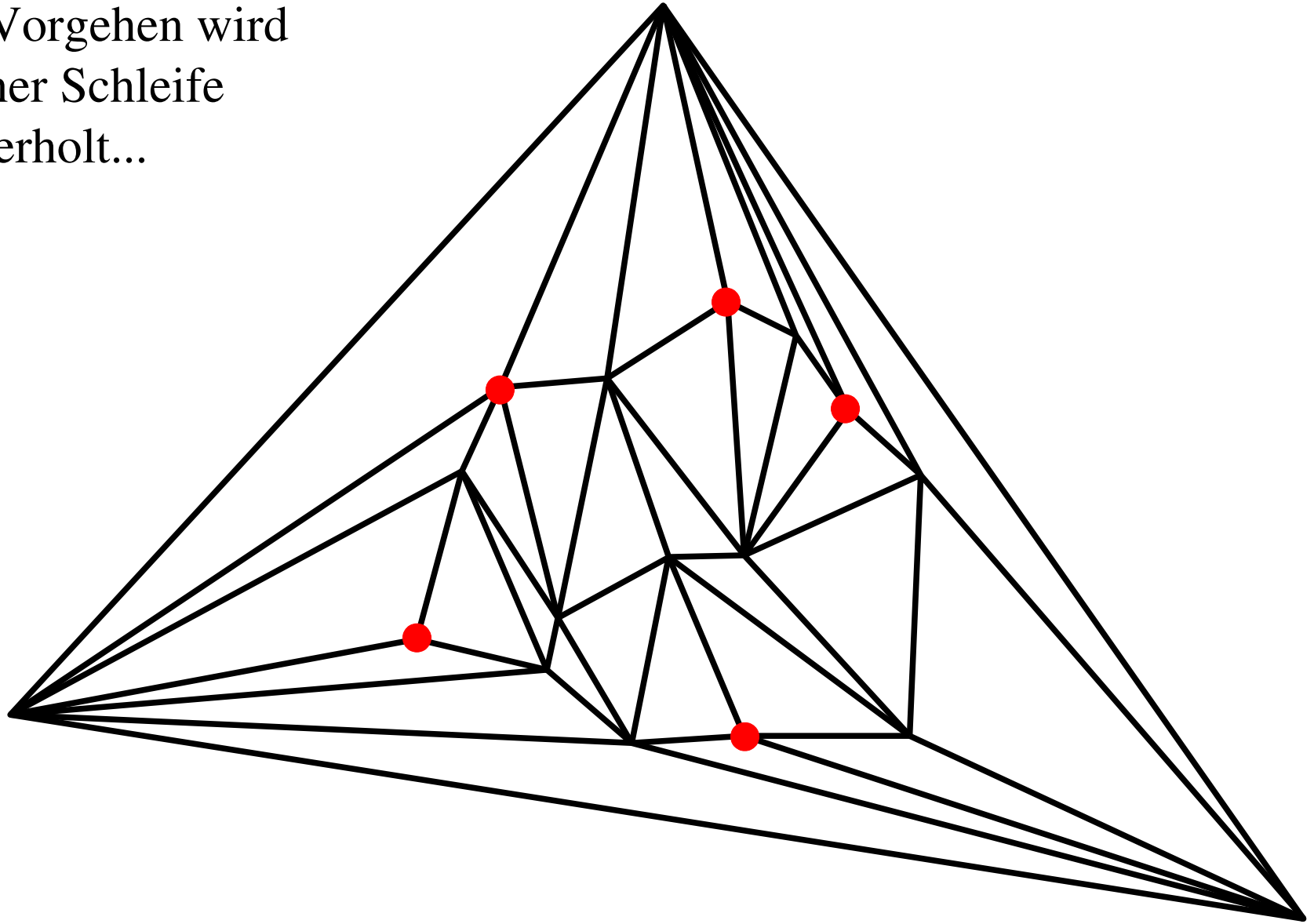
Die Ecken in der unabhängigen Menge werden zusammen mit den inzidenten Kanten **entfernt**.

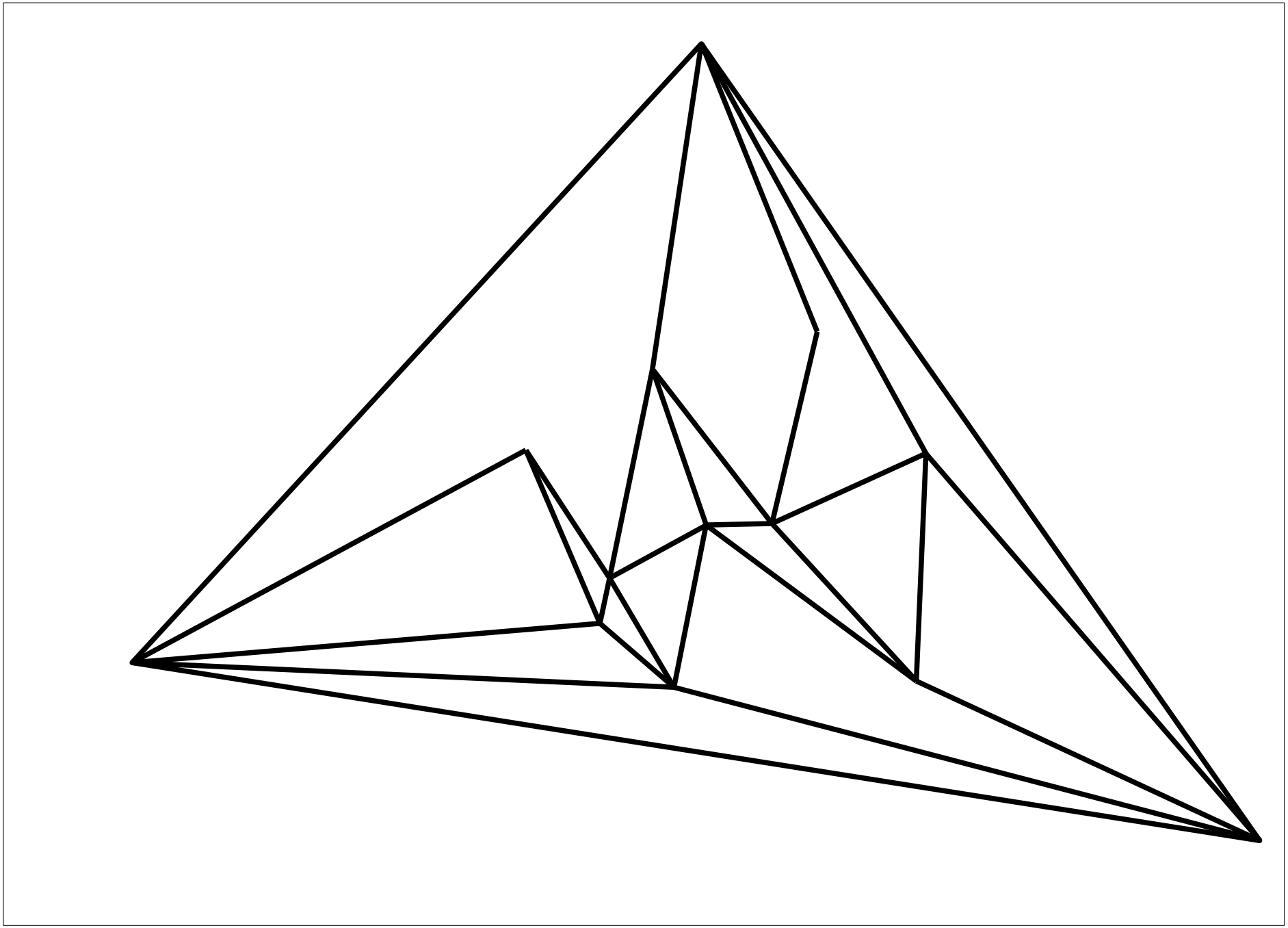


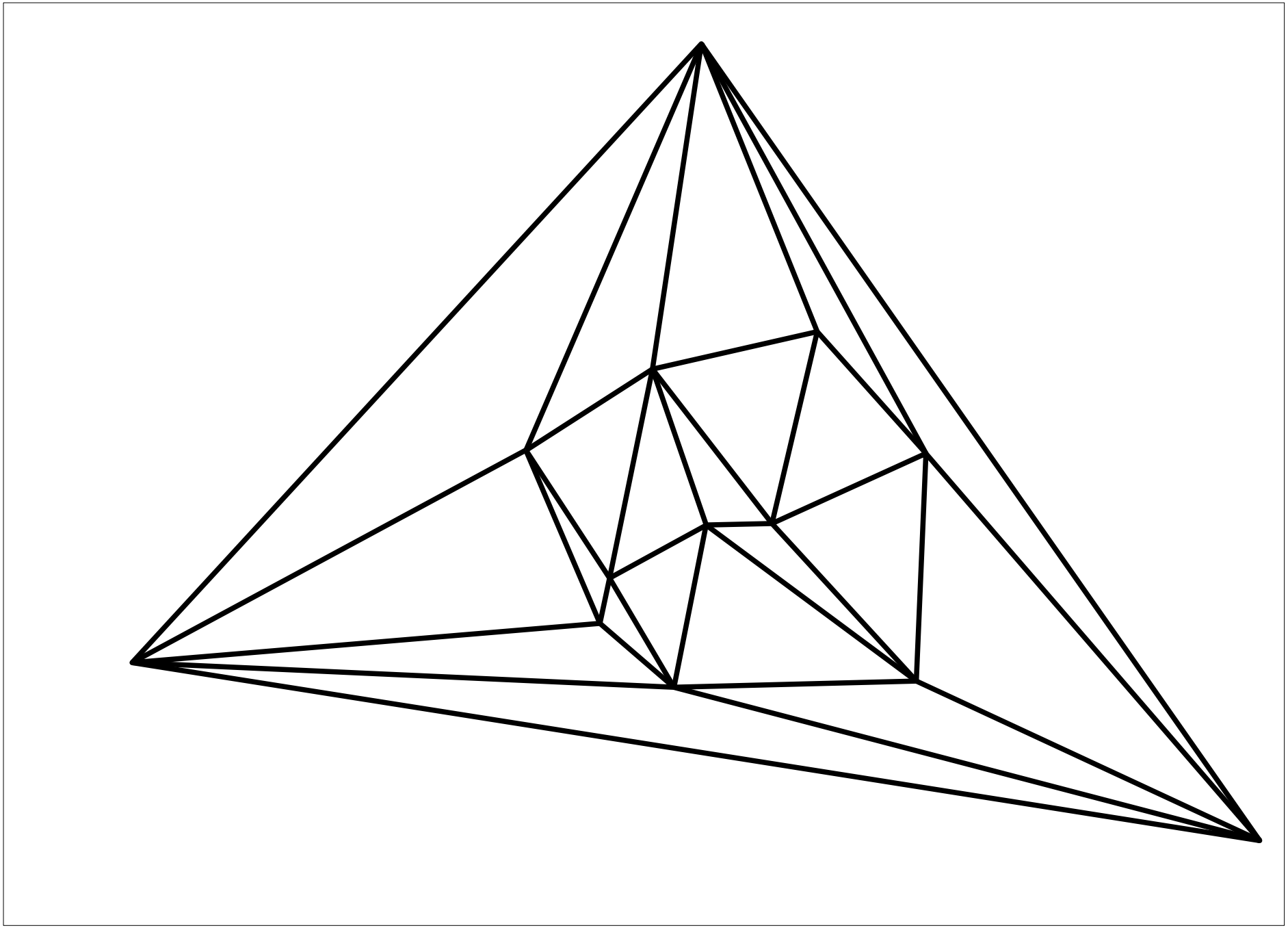
Jetzt wird wieder
**neu trianguliert**.

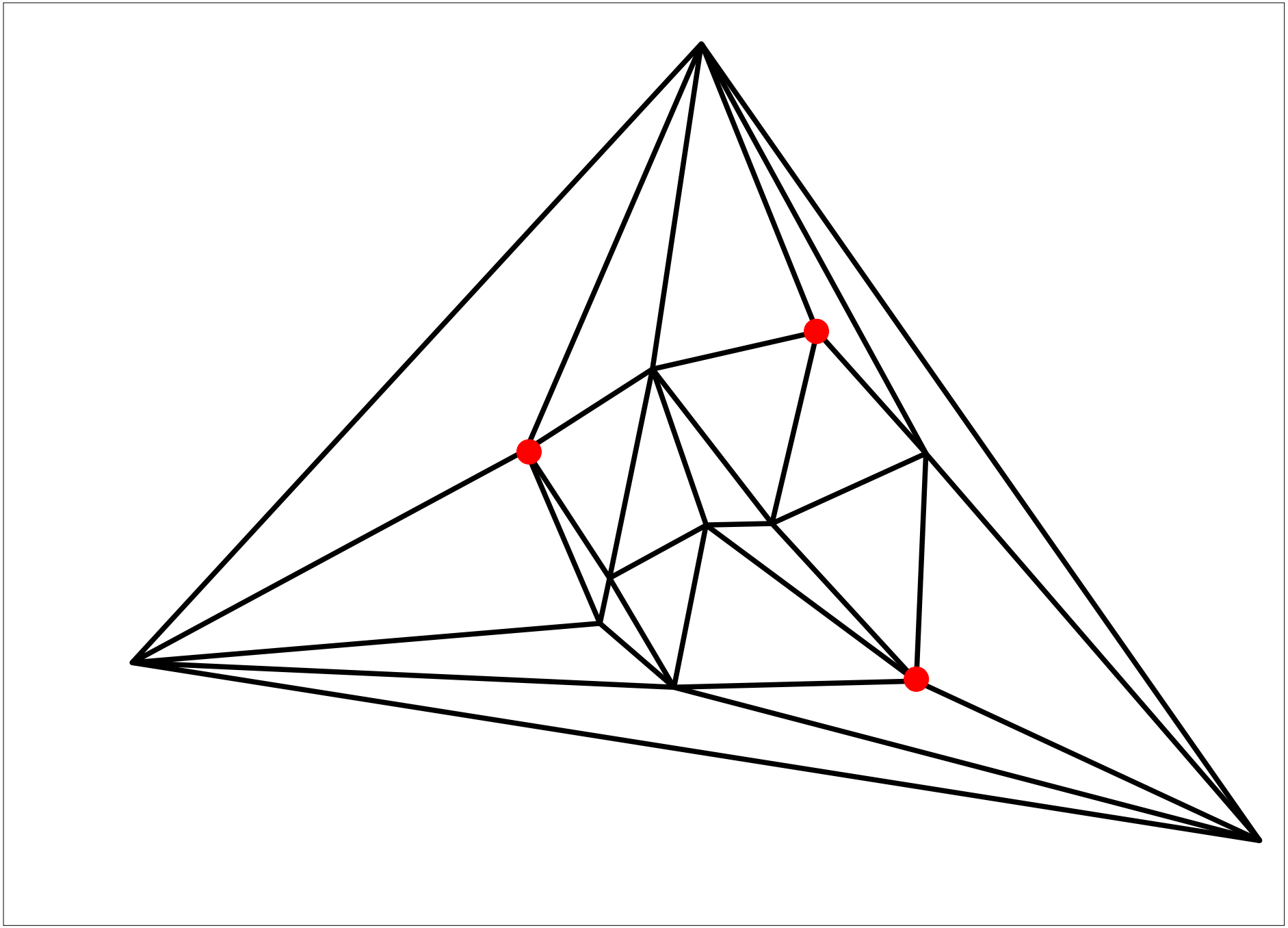


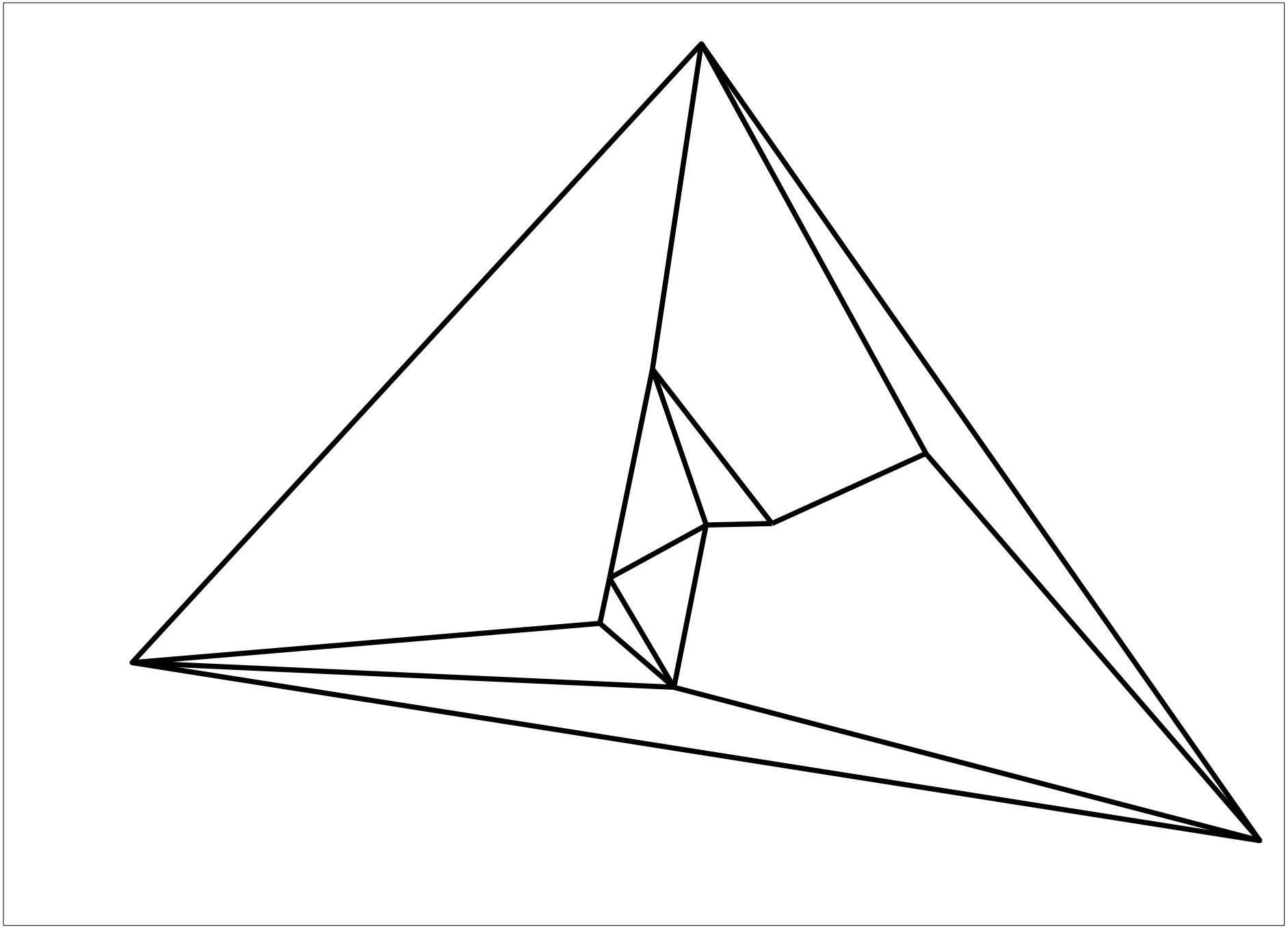
Das Vorgehen wird in einer Schleife wiederholt...

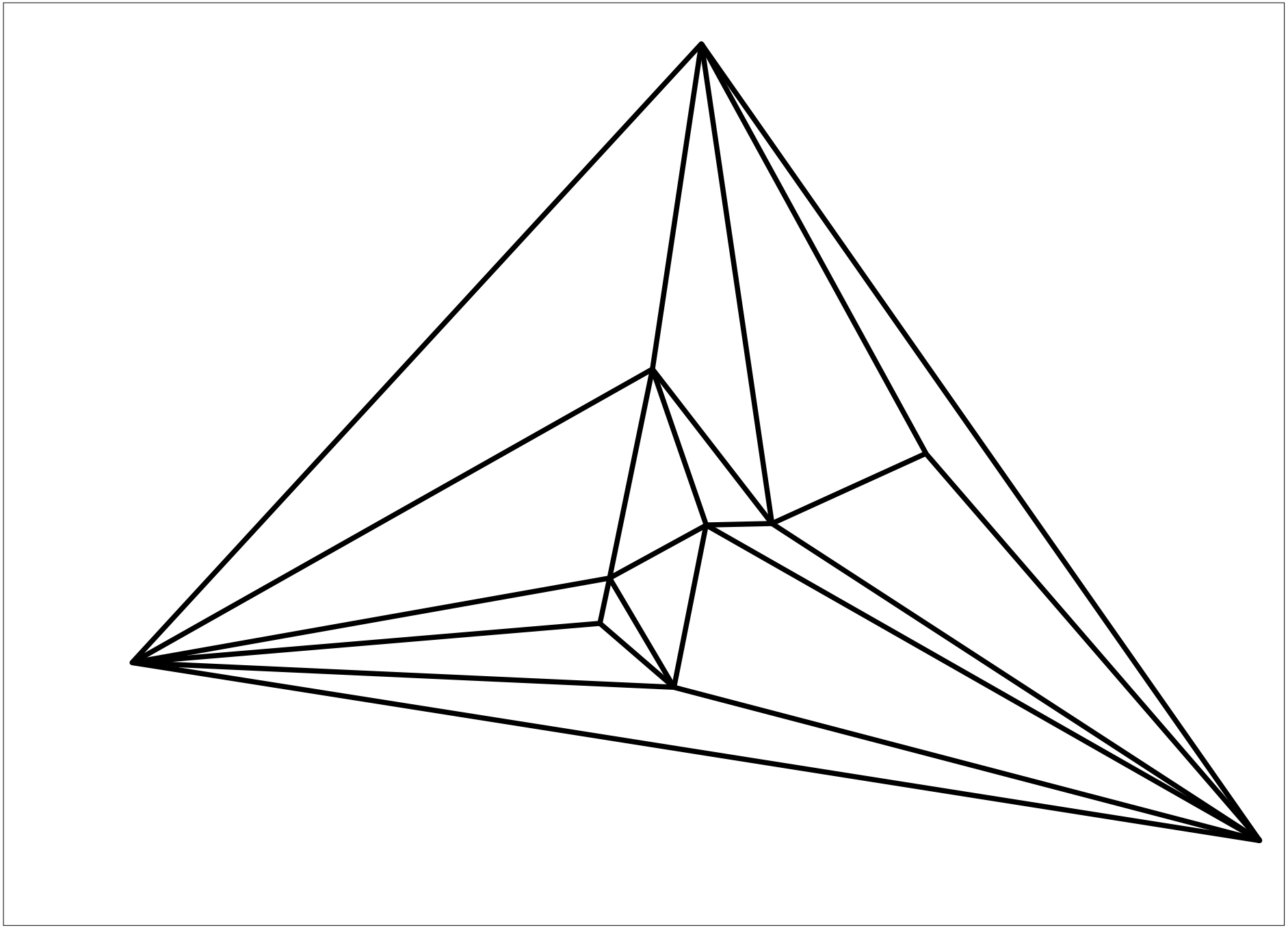


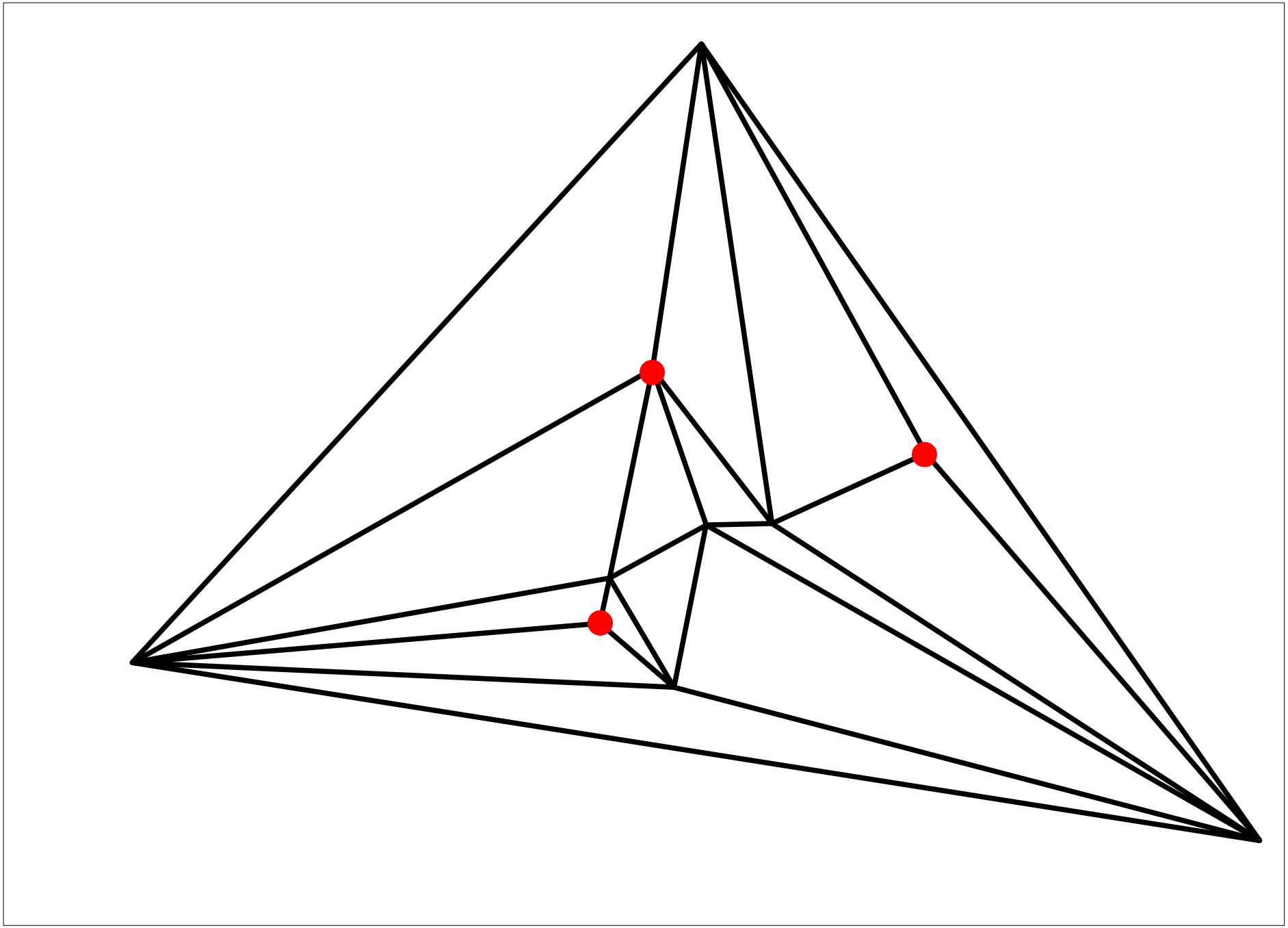


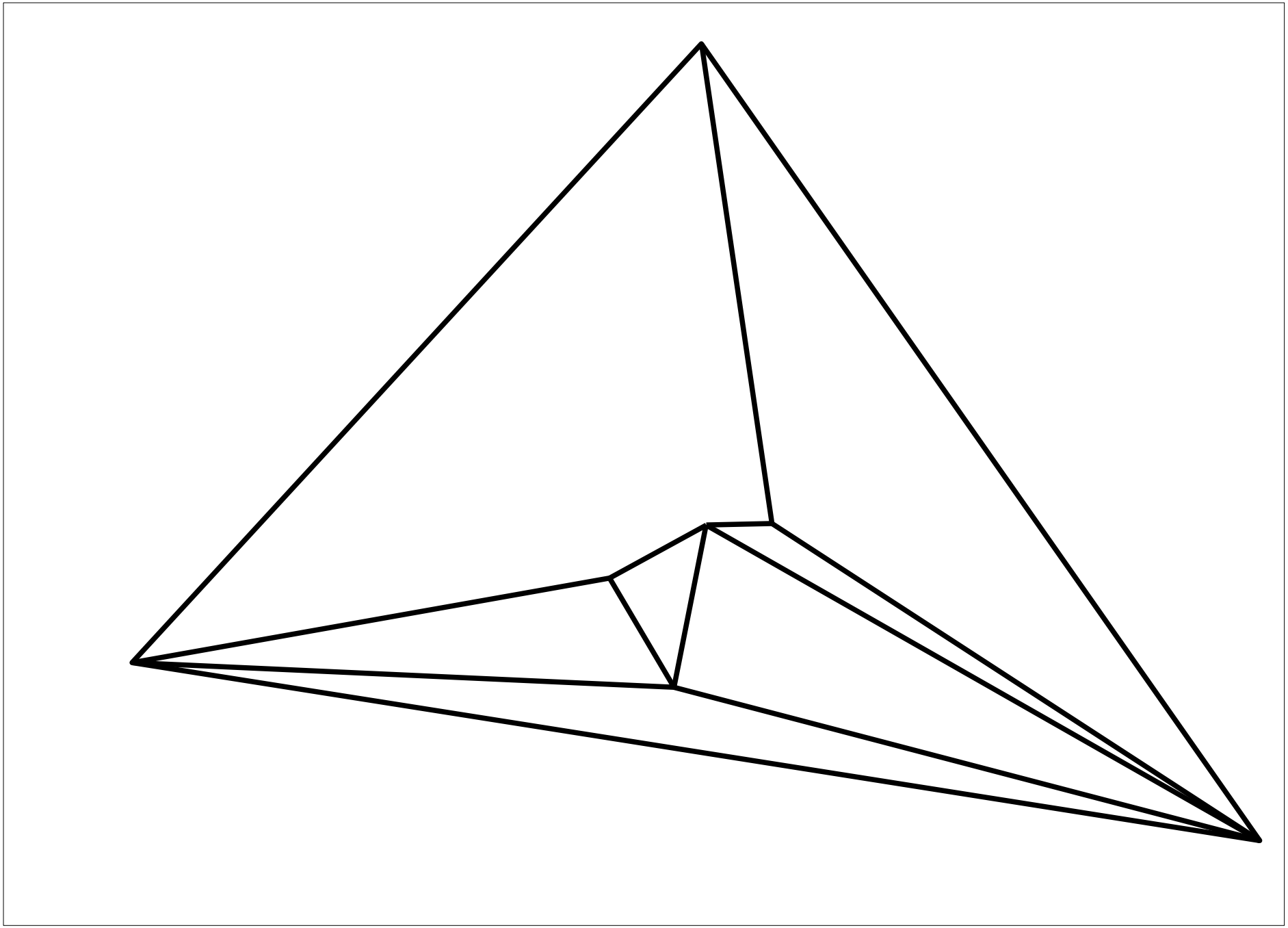


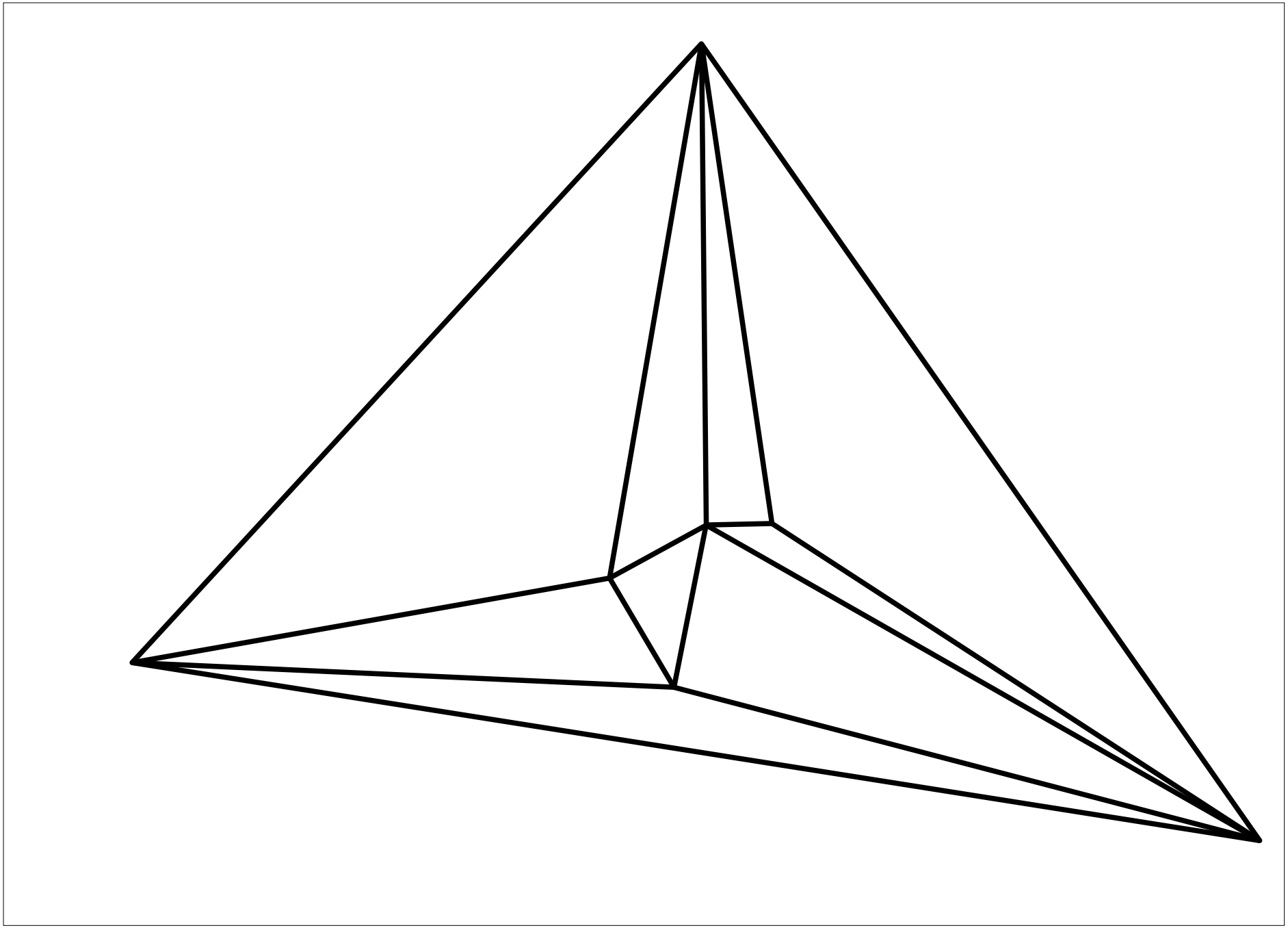


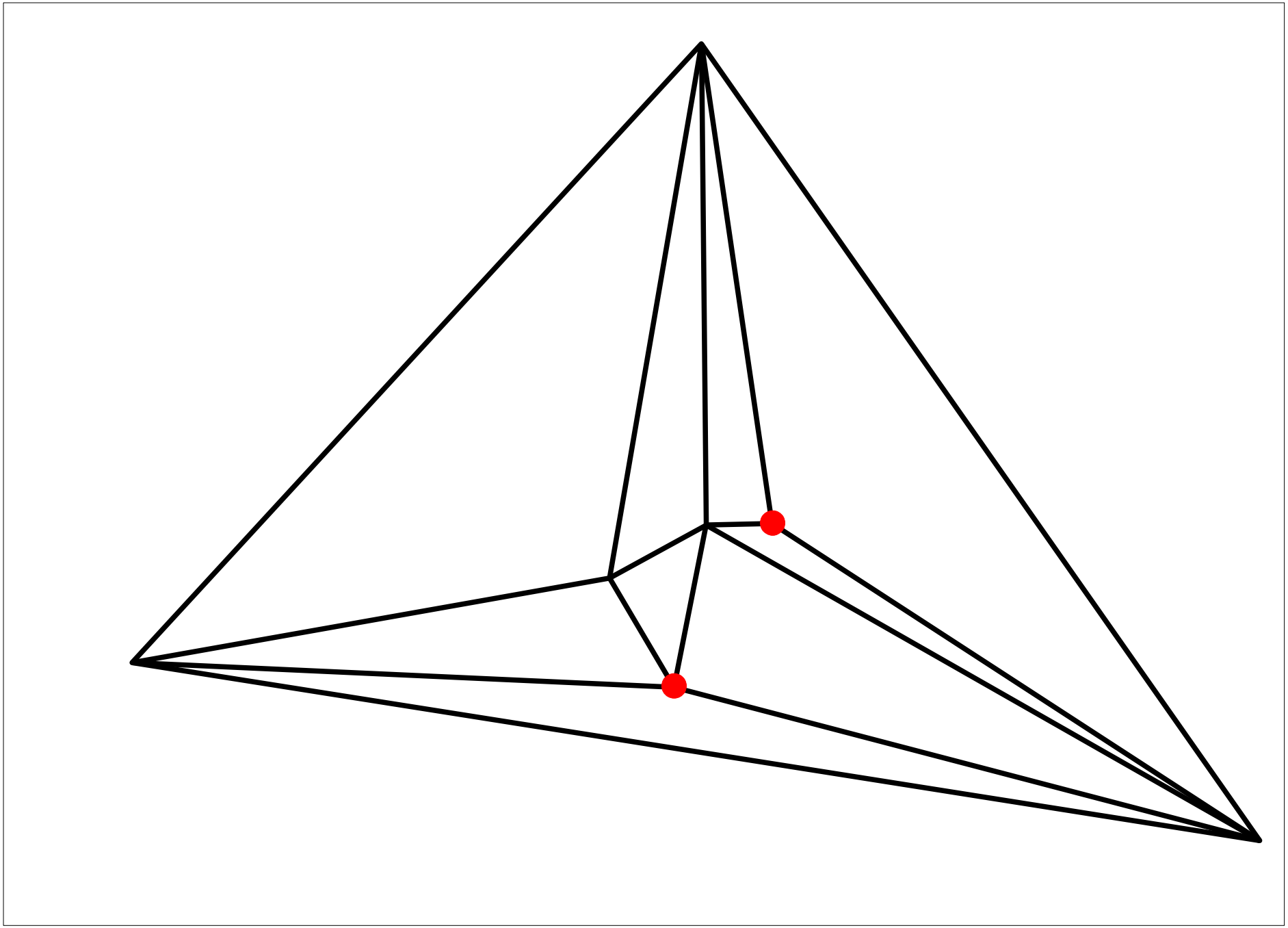


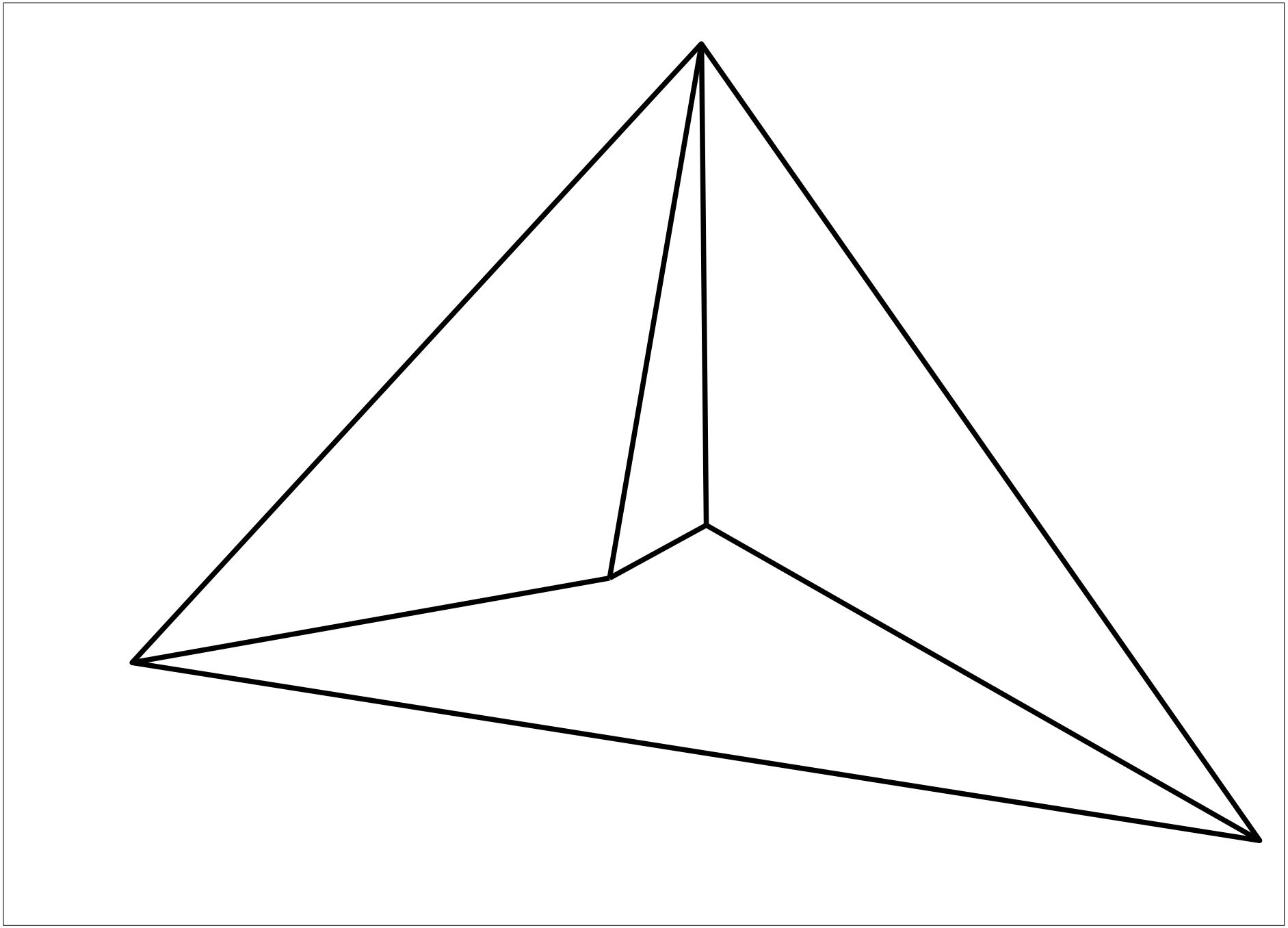


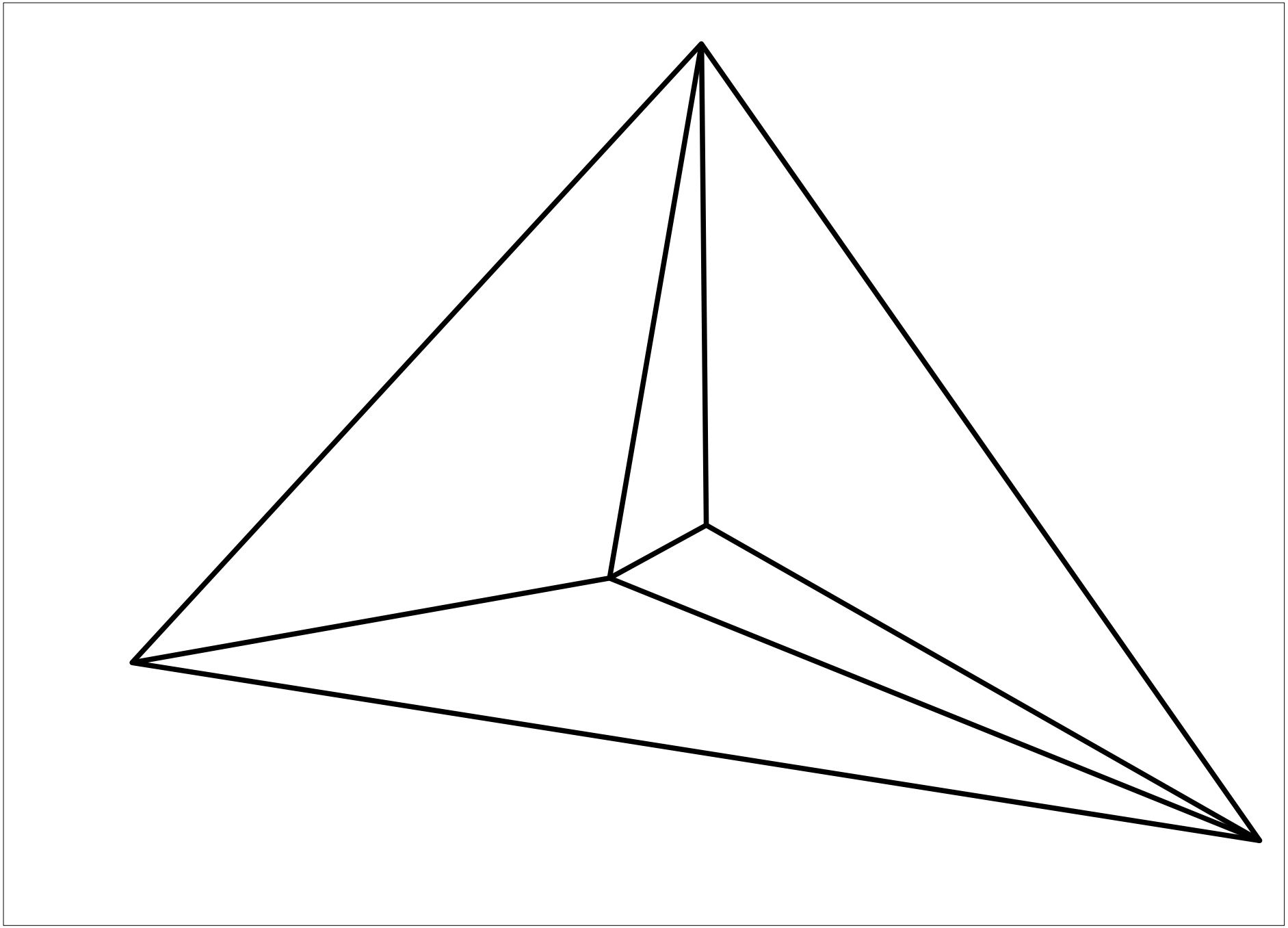


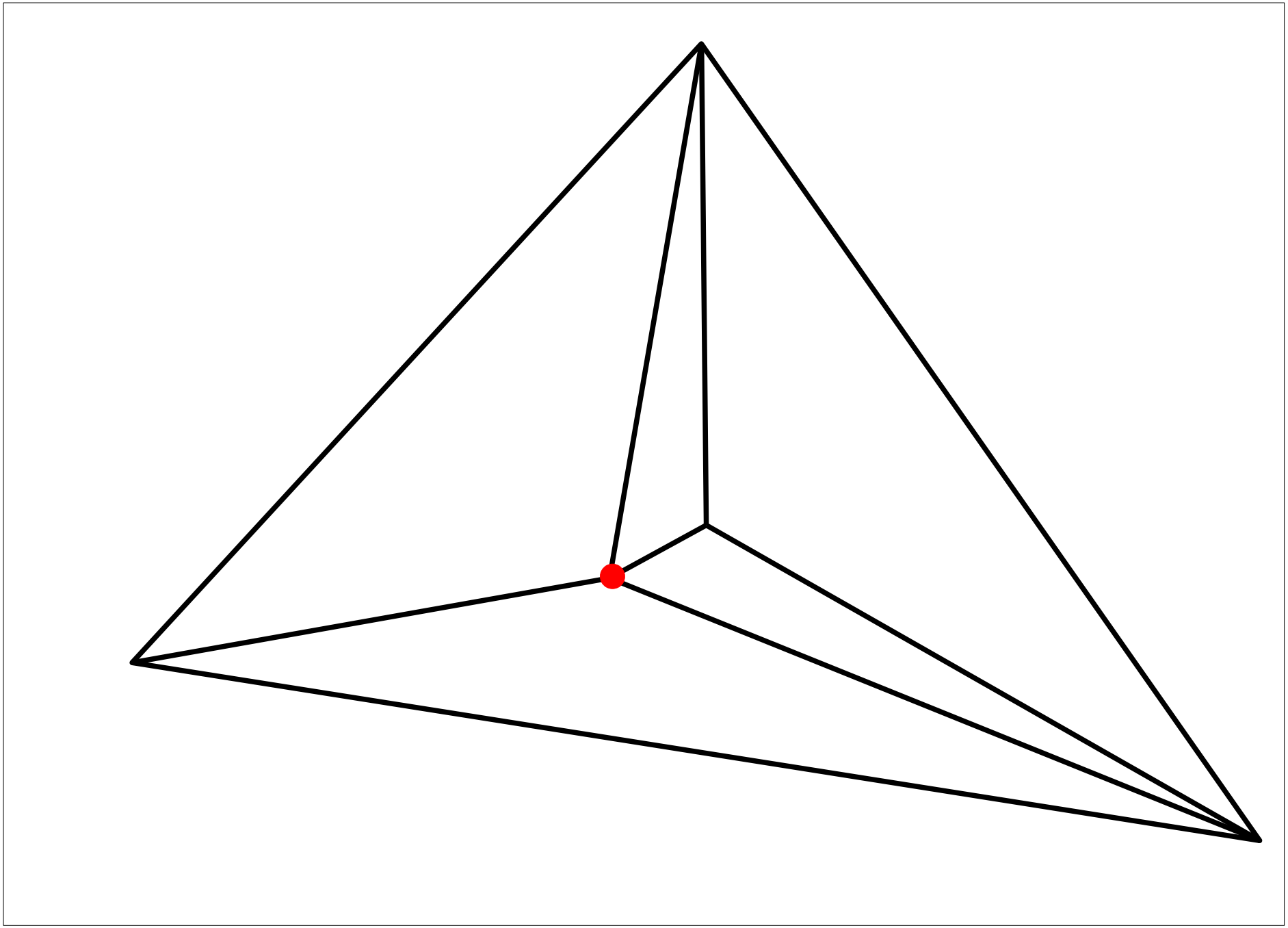


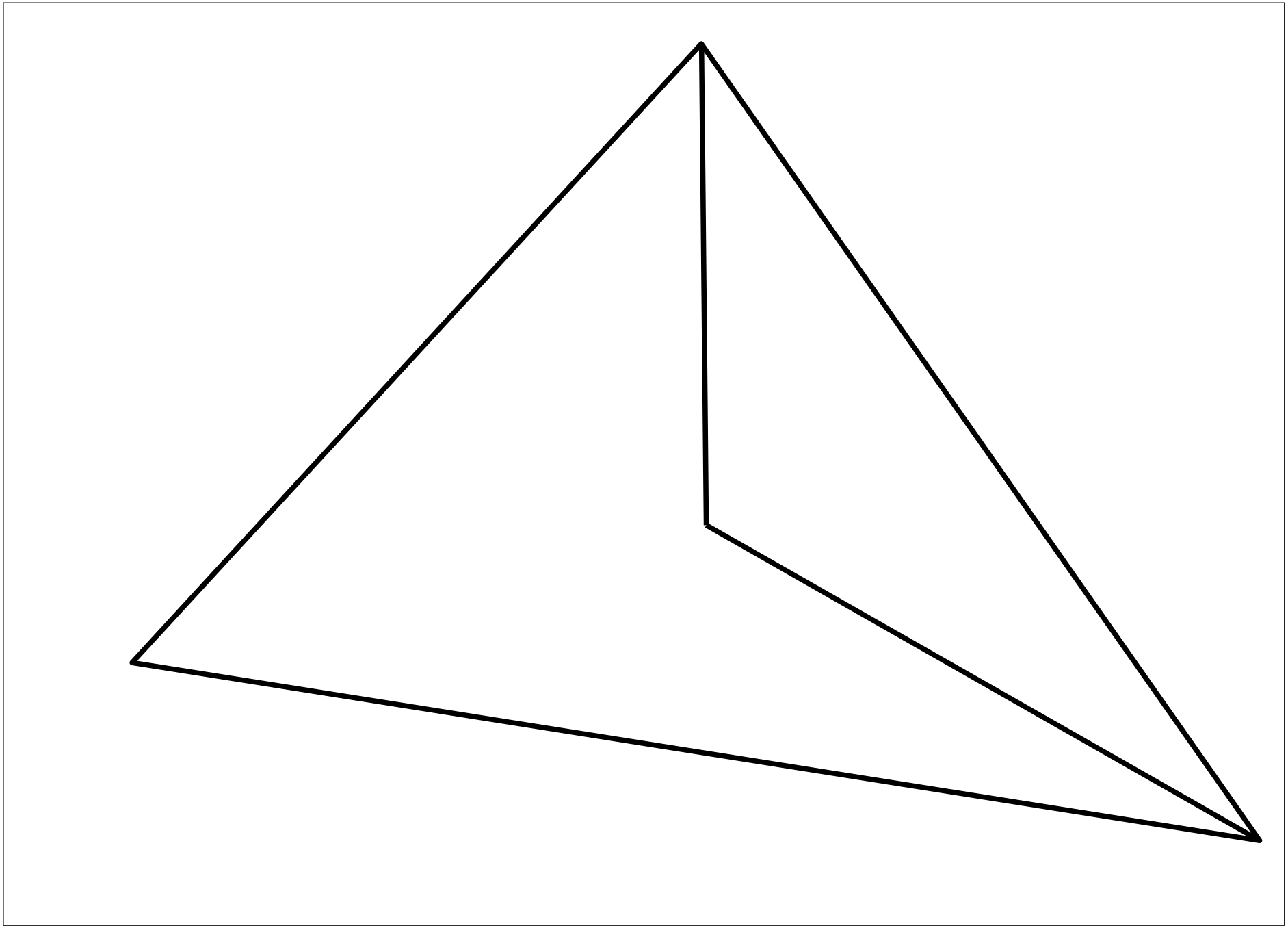


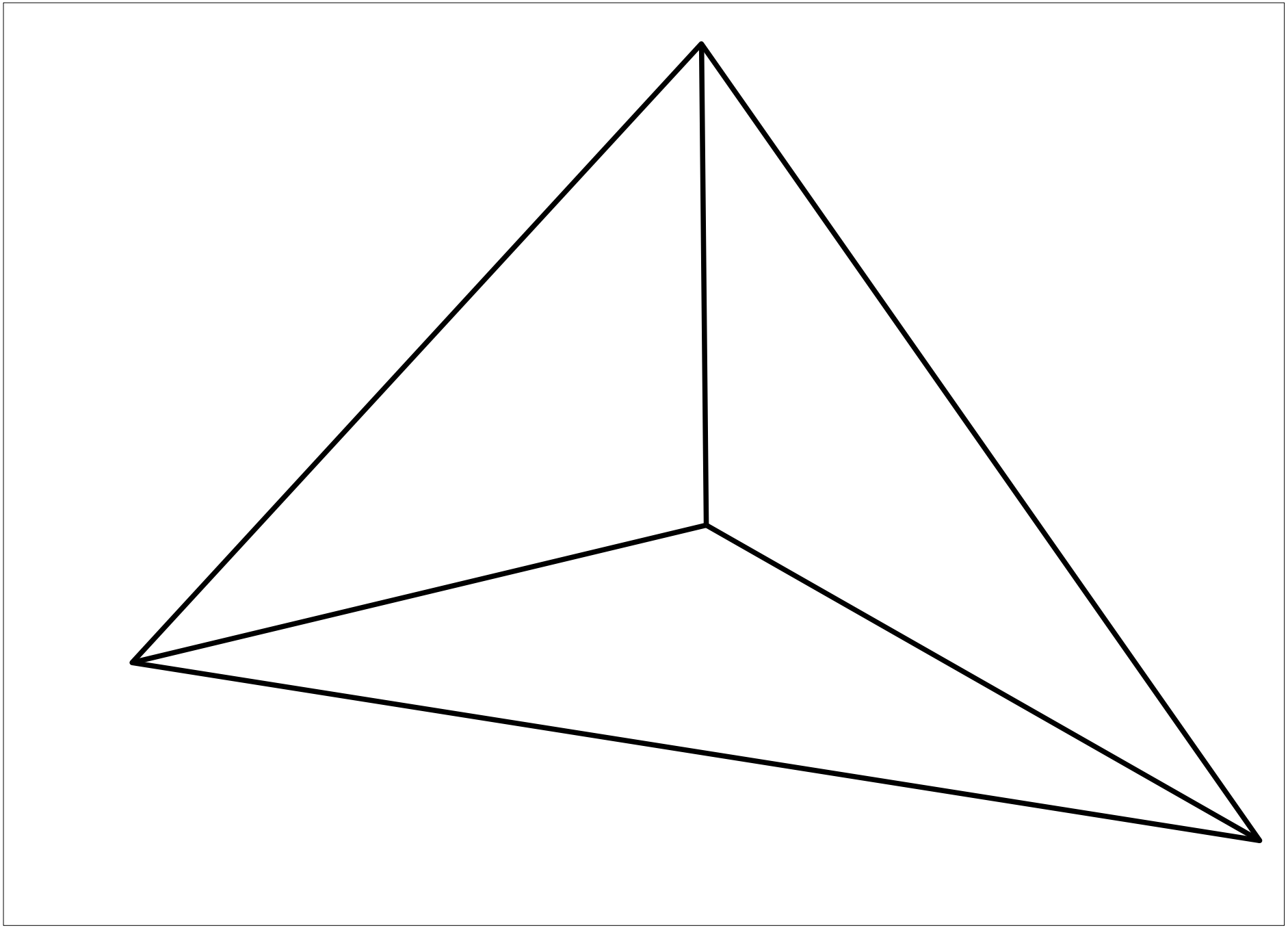


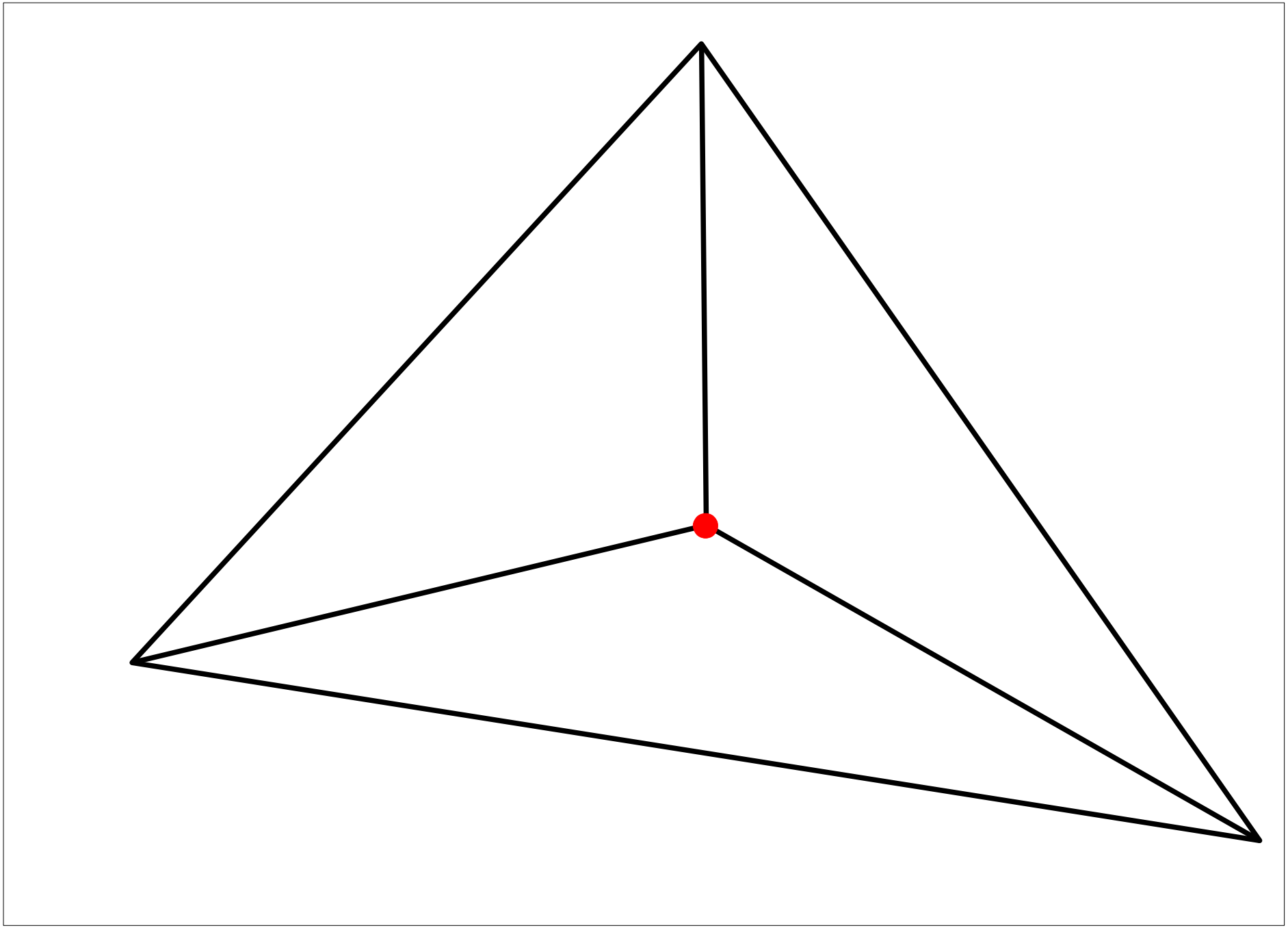




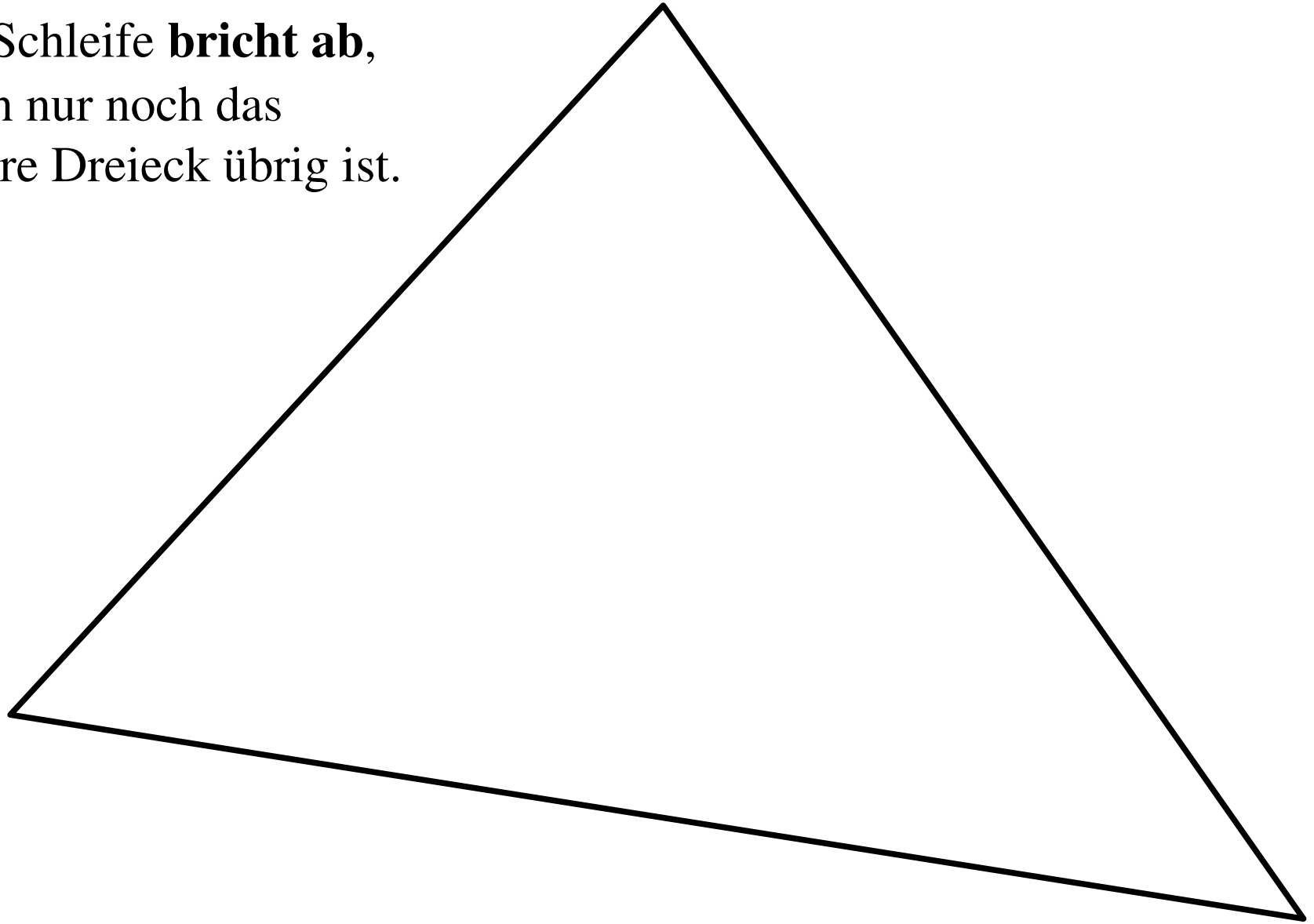








Die Schleife **bricht ab**,
wenn nur noch das
äußere Dreieck übrig ist.



Auf diese Weise erhalten wir eine **Folge von Triangulationen**:

$$T_1, T_2, ..., T_k$$

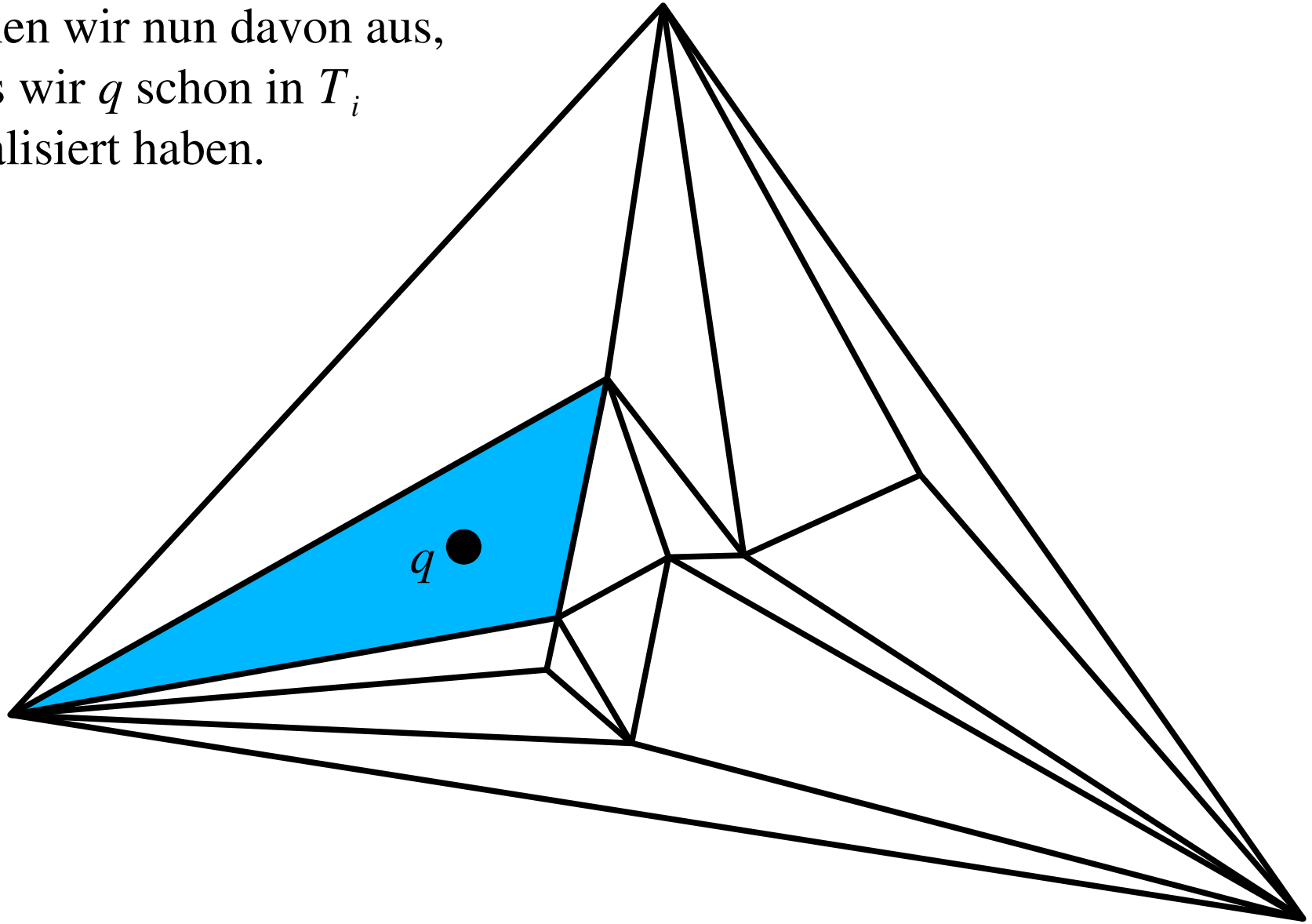
$T_1$ ist die Ausgangstriangulation.

$T_k$ besteht aus nur einem Dreieck.

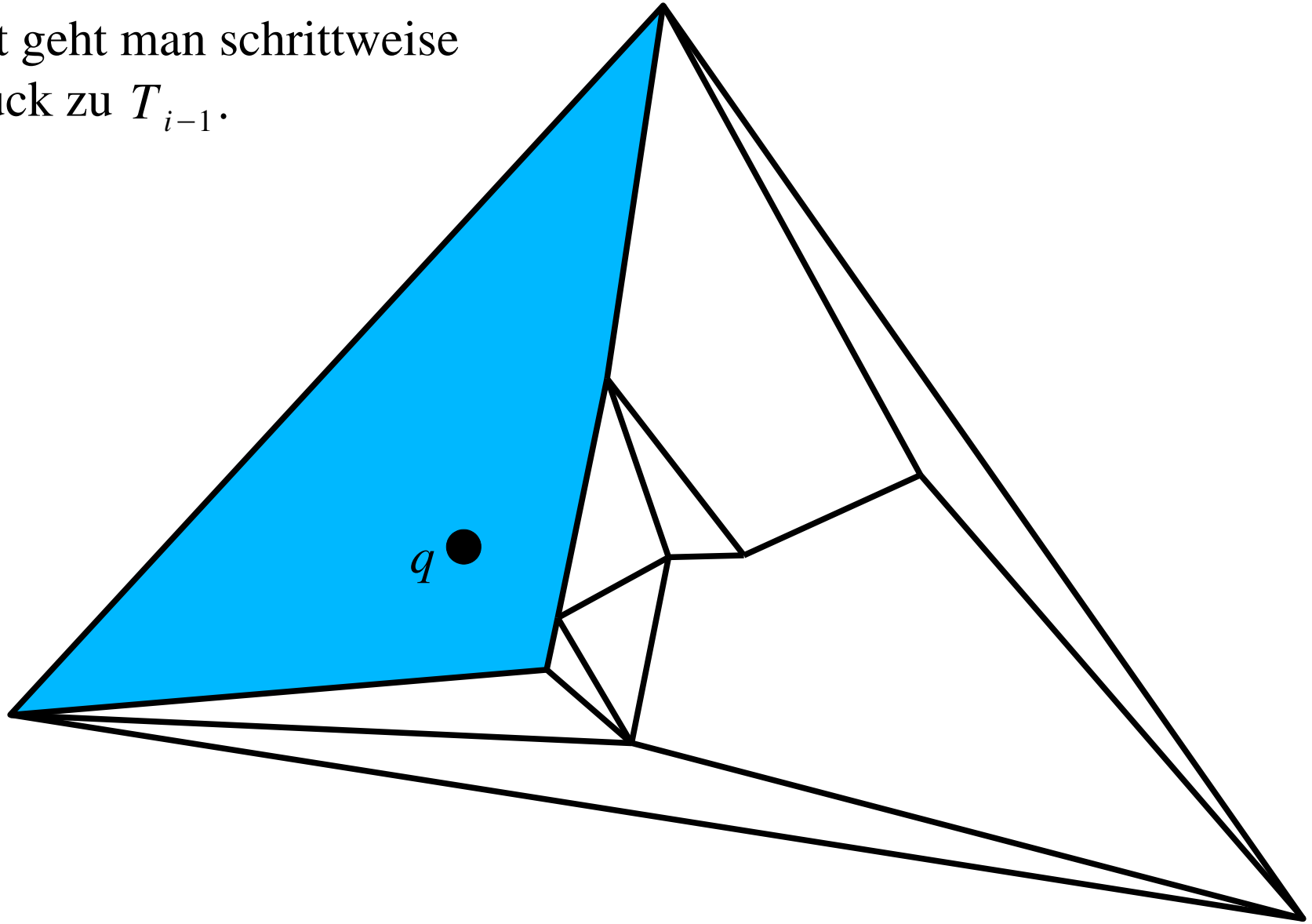
Wir wollen einen Punkt $q$ in der Triangulation $T_1$ lokalisieren.

Dazu lokalisieren wir ihn zuerst in $T_k$. Das ist aber einfach.

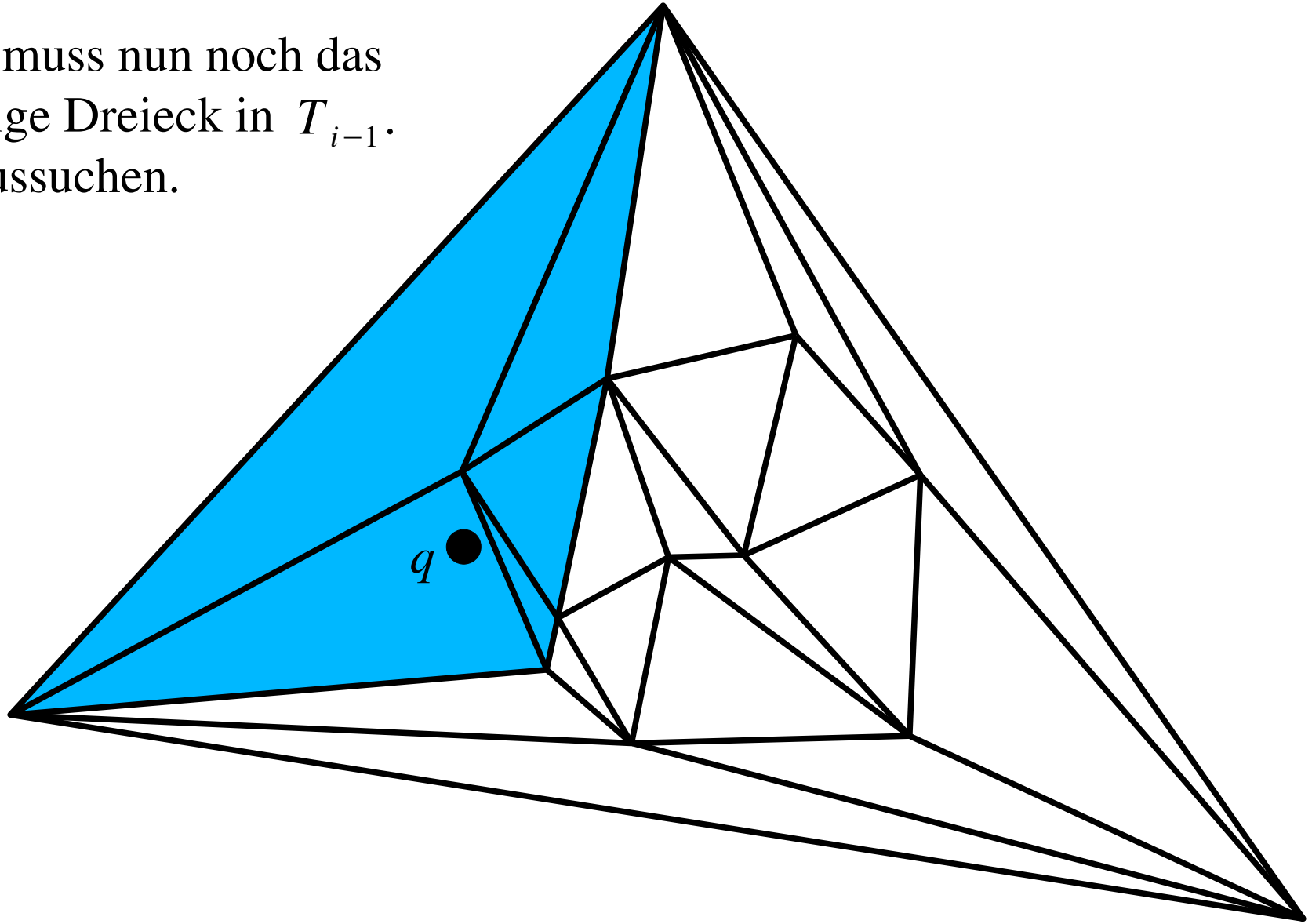
Gehen wir nun davon aus, dass wir $q$ schon in $T_i$ lokalisiert haben.



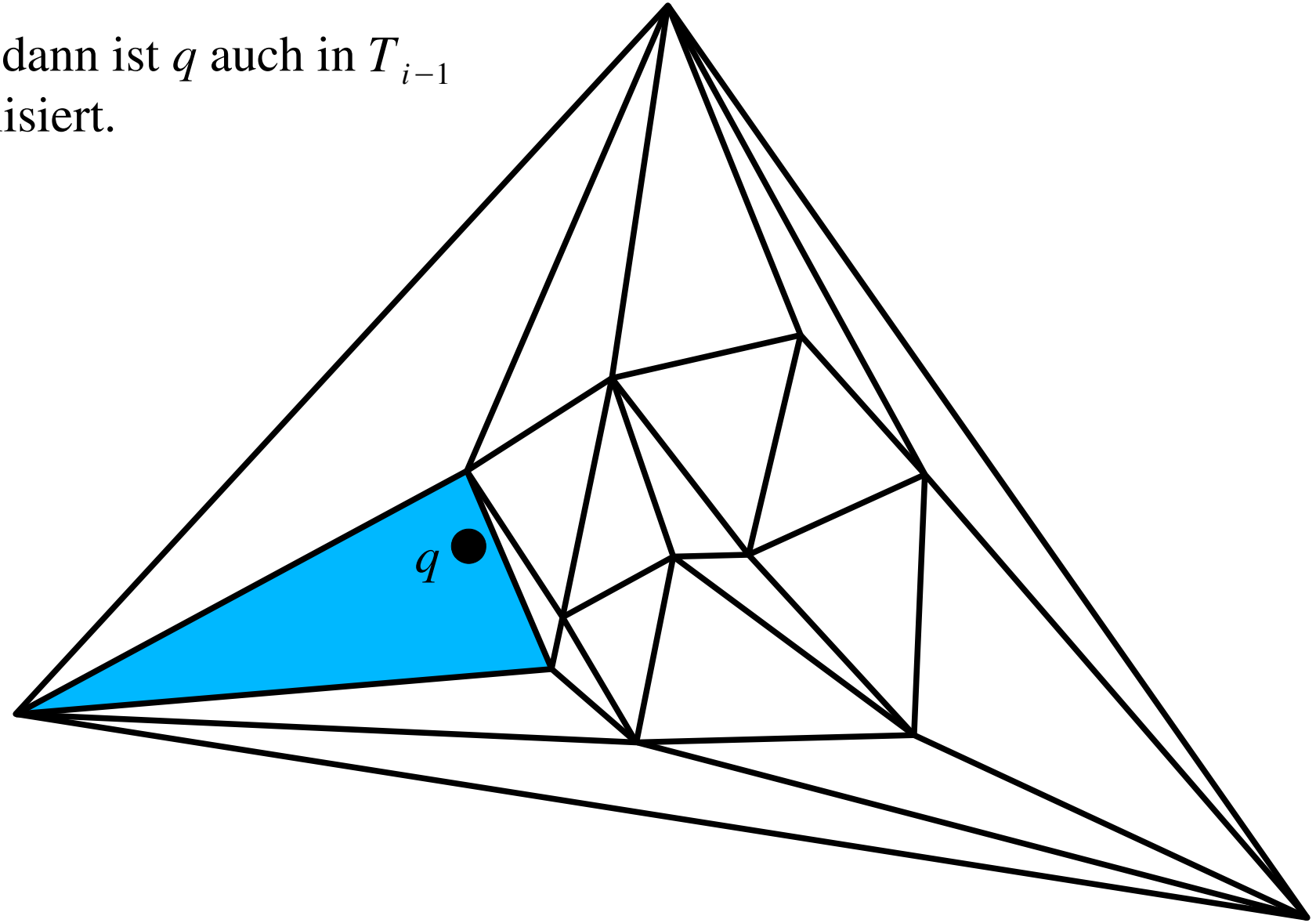
Jetzt geht man schrittweise zurück zu $T_{i-1}$.



Man muss nun noch das richtige Dreieck in $T_{i-1}$. heraussuchen.



Und dann ist $q$ auch in $T_{i-1}$ lokalisiert.



Damit unser Verfahren überhaupt effizient arbeitet, darf die **Anzahl der Triangulationen** $k$ natürlich nicht zu groß sein.

Beim Übergang von $T_i$ zu $T_{i-1}$ würden wir am liebsten einfach alle in Frage kommenden **Dreiecke durchtesten**.

Dies ist aber nur dann sinnvoll, wenn die **Anzahl dieser Dreiecke** ebenfalls nicht zu groß ist.

Es stellt sich uns also im Folgenden die Aufgabe, **nicht irgendeine Folge** von Triangulationen zu konstruieren, sondern eine, die möglichst gut ist, im Sinne der oben gemachten Bemerkungen.

Lemma 1

Wir betrachten eine Triangulation mit $n$ Ecken, bei der es genau drei äußere Kanten gibt. Dann gibt es in dieser Triangulation insgesamt $m = 3n - 6$ Kanten.

Beweis:

Führen wir als Erstes noch eine weitere Bezeichnung $d$ für die Anzahl der Dreiecke in der Triangulation ein.

Dann gilt nach der **Eulerschen Formel**: $n - m + d = 1$

Außerdem können wir die Kanten in der Triangulation auf zwei Arten abzählen: $3d = 2m - 3$

Das stellen wir nach $d$ um: $d = \frac{2}{3}m - 1$

Setzen es in die Eulersche Formel ein: $n - m + \frac{2}{3}m - 1 = 1$

Stellen es nach $m$ um: $m = 3n - 6$ ■

Lemma 2

Sei $c$ eine positive ganze Zahl. Dann gibt es in einer Triangulation mit $n$ Ecken und drei äußeren Kanten mindestens $\left(1-\frac{6}{c}\right)n$ Ecken mit weniger als $c$ inzidenten Kanten.

Beweis:

Doppeltes Abzählen liefert: $\sum_{\substack{\text{Ecke } v \\ deg(v) \geq c}} deg(v) \leq 2m$

Also ist die Zahl der Ecken mit mindestens $c$ inzidenten Kanten nicht größer als:

$$\frac{2m}{c}$$

Damit ist die Zahl der Ecken mit weniger als $c$ inzidenten Kanten mindestens:

$$n - \frac{2m}{c}$$

Anwendung von Lemma 1 liefert:

$$\left(1 - \frac{6}{c}\right) n + \frac{12}{c}$$

■

## Lemma 3

Sei $c$ eine positive ganze Zahl. Dann gibt es in einer Triangulation mit $n$ Ecken und drei äußeren Kanten eine unabhängige Menge, die nur Ecken mit weniger als $c$ inzidenten Kanten enthält und mindestens $\frac{1}{c}\left(\left(1-\frac{6}{c}\right)n-3\right)$ Elemente enthält. Eine solche unabhängige Menge kann in $O(n)$ Zeit berechnet werden.

## Beweis:

Wir suchen zuerst alle Ecken mit weniger als $c$ inzidenten Kanten heraus, welche keine Ecken des äußeren Dreiecks sind.

Dann wählen wir immer eine solche Ecke und alle ihre Nachbarn.

Damit verbrauchen wir jeweils höchstens $c$ Ecken mit weniger als $c$ inzidenten Kanten.

Nach Lemma 2 können wir also diesen Schritt mindestens

$$\frac{1}{c}\left(\left(1-\frac{6}{c}\right)n-3\right)$$

mal anwenden, bis die Menge der Ecken mit weniger als $c$ inzidenten Kanten erschöpft ist. ■

Wir wollen nun die Konstante $c$ so wählen, dass die durch das in Lemma 3 angedeutete Verfahren gewonnene unabhängige Menge **möglichst viele Elemente** hat.

Dazu untersuchen wir die Funktion $f(x) = \frac{1}{x}\left(1 - \frac{6}{x}\right)$.

Diese hat eine Maximumstelle bei $x = 12$.

Wir wählen also günstigerweise $c = 12$.

Damit erhalten wir in einer Triangulation mit $n$ Ecken immer eine unabhängige Menge von Ecken mit **weniger als 12 inziden-**

**ten Kanten**, die mindestens $\frac{1}{24}n - \frac{1}{4}$ Elemente umfasst.

Man rechnet leicht nach, dass für $n > 11$ gilt: $\frac{1}{24}n - \frac{1}{4} \geq \frac{1}{48}n$

Damit schaffen wir es beim Übergang von einer Triangulation

zur nächsten immer **mindestens** $\frac{47}{48}$ der noch vorhandenen

Ecken los zu werden.

Daraus folgt, dass die Anzahl der Triangulationen $k$ in $O(\log n)$ liegt.

Als Speicherbedarf ergibt sich: $O(n)$.

Da $c = 12$ ist, können wir es uns auch leisten, bei einer Punktlokalisierung beim Übergang von einer Triangulation zur vorherigen immer **alle Kandidatendreiecke** (höchstens 11) durchzutesten.

Die Zeit zur Konstruktion der Datenstruktur hängt davon ab, wie schnell ich triangulieren kann.

# Zusammenfassung

Wenn sich die Regionen, für die eine Punktlokalisierungs-Datenstruktur aufgebaut werden soll, in $O(n)$ Zeit triangulieren lassen, dann erhalten wir:

| | |
|---|---|
| Aufbau der Datenstrukur: | $O(n)$ |
| Speicherbedarf: | $O(n)$ |
| Anfragen: | $O(\log n)$ |