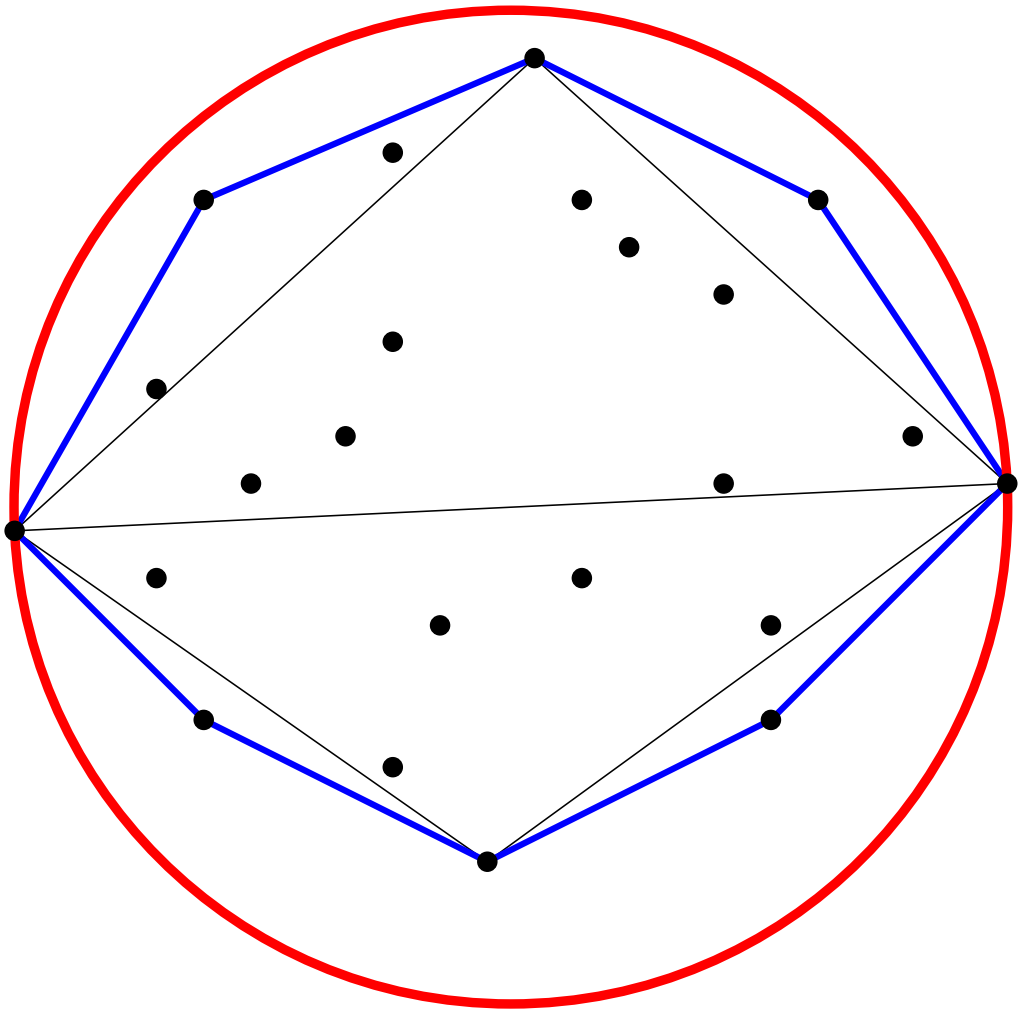
FernUniversität in Hagen

Die konvexe Hülle und der kleinste Kreis

Aufgabenbeschreibung für das Programmierpraktikum 1580/1582/1584
im Sommersemester 2012

Lehrgebiet Kooperative Systeme



Fakultät für Mathematik und Informatik

Inhalt

Vorwort

Herzlich willkommen zum *Programmierpraktikum* 1580/1582/1584 im Sommersemester 2012!

Ziel dieses Praktikums ist es, eine größere Programmieraufgabe selbständig zu lösen. Ihre konkrete Aufgabe besteht darin, ein Programm für zwei anschauliche Optimierungsaufgaben zu erstellen, wobei das Programm möglichst *effizient* arbeiten soll, damit es auch größere Eingaben in annehmbarer Zeit bewältigen kann. Um Programme dieser Art schreiben zu können, benötigen Sie effiziente Datenstrukturen und Algorithmen, die Sie teilweise schon aus den Grundkursen kennen und zum anderen Teil durch diesen Text kennen lernen werden.

Ein gutes Gelingen und viel Freude an der Bearbeitung dieser Aufgabe wünschen Ihnen die Betreuer!

Betreuung

Für das Praktikum haben wir auf dem News-Server `feunews.fernuni-hagen.de` zwei *Newsgruppen* eingerichtet. Newsgruppen

Die Newsgruppe

`feu.informatik.kurs.1580+82+84.betreuung.ss` Organisatorisches

ist das Forum für Ankündigungen und Organisatorisches im Praktikum. Darin werden wir aktuelle organisatorische Informationen bekanntgeben. Hier *müssen* Sie regelmäßig mitlesen, um nichts zu verpassen.

Die Gruppe

`feu.informatik.kurs.1580+82+84.diskussion.ss` Diskussionen

ist das Diskussionsforum für alle Teilnehmer. Hier wird unter den Teilnehmern und mit den Betreuern über die Aufgabe, Lösungsmöglichkeiten, die Programmierumgebung usw. diskutiert.

Bitte benutzen Sie zur Kommunikation möglichst diese Foren, damit immer alle anderen Teilnehmer mitlesen und von den Antworten profitieren können.

Für Angelegenheiten in Zusammenhang mit Ihrer persönlichen Teilnahme wenden Sie sich bitte an:

`Andrea.Frank@fernuni-hagen.de`

Die Webseite zum Praktikum ist: Webseite

http://www.fernuni-hagen.de/mathinf/studium/lehre/praktika/programmierpraktikum/2012_ss.shtml

Wettbewerb

Die abgegebenen Programme werden von den Betreuern über das vorgeschriebene Test-Interface auf Korrektheit überprüft. Dazu wird noch die Programm-Laufzeit für eine ziemlich große Eingabe ermittelt. Wer das Programm abgibt, das in der kürzesten Zeit das korrekte Ergebnis zurückgibt, erhält zur Belohnung ein Informatik-Fachbuch.

Preis zu gewinnen

alternative Algorithmen

Zur Verbesserung der Laufzeit ist es möglich, *alternative Algorithmen* (entweder selbst erfundene oder aus anderen Quellen ermittelte) als die in diesem Text vorgestellten zu verwenden, aber natürlich nur selbst programmierte. Man sollte aber beachten, dass die hier vorgestellten Verfahren gerade wegen ihrer Einfachheit und Robustheit ausgesucht wurden.

Zeitlicher Ablauf

Empfohlen wird ein zügiger Start in die Praktikumsarbeit gleich nach Erhalt dieser Aufgabe. Insbesondere die Unklarheiten sollten möglichst bald durch Inanspruchnahme der Betreuung beseitigt werden. Kurz vor den Abgabeterminen wird es erfahrungsgemäß auch mit der Betreuungsmöglichkeit sehr eng. Die angegebenen *Termine* sind unbedingt einzuhalten. Eine vorzeitige Abgabe ist zu jeder Zeit möglich.

Termine einhalten!

Arbeitsbeginn: 26. März 2012
Abgabe des ersten Teils spätestens bis: 10. Juni 2012
Korrekturhinweise zum ersten Teil bis: 7. Juli 2012
Nachbesserungsfrist für den ersten Teil bis zum: 20. Juli 2012
Abgabe des zweiten Teils spätestens bis: 26. August 2012
Korrekturhinweise zum zweiten Teil bis: 14. September 2012
Nachbesserungsfrist für den zweiten Teil bis zum: 20. September 2012
Ggfs. persönliche Vorstellung des eigenen Programms in Hagen: ab 24. September 2012
Bekanntgabe des Wettbewerbs-Siegers: ca. 28. September 2012
Zusendung der Leistungsnachweise: ca. 30. September 2012

Kontrollen

nur eigene Programme einreichen!

Ein ernst gemeinter Appell an alle Teilnehmerinnen und Teilnehmer: Seien Sie fair und kein Plagiator, nutzen Sie auch keine Ghostwriter, die eigene Leistung zählt! Nur jeweils selbständig erstellte Programme dürfen abgegeben werden, Übernahmen aus fremden Quellen werden als Täuschung gewertet und führen zum sofortigen Nichtbestehen bzw. bei nachträglichem Bekanntwerden mindestens zur Aberkennung der Leistung.

automatische Kontrollen

Die abgegebenen Programme werden intensiv kontrolliert. So werden zum Beispiel alle abgegebenen Programme automatisch untereinander und mit verschiedenen anderen Quellen verglichen und auf Ähnlichkeiten untersucht. Wir behalten uns vor, einzelne Teilnehmer gegen Ende September nach Hagen zu bitten und uns ihr Programm persönlich vorzustellen.

Programmierumgebung

Computer

Für das Praktikum benötigen Sie einen *Computer* mit Internetanschluss. Als Betriebssysteme sind auf jeden Fall Windows, Linux (diverse Distributionen) und MacOSX geeignet.

Java

Die zu verwendende Programmiersprache ist *Java*, Vorkenntnisse zur Java-Programmierung sind unbedingt erforderlich. Wenn Sie noch kein Java auf Ihrem Computer installiert haben, dann installieren Sie es bitte von Ihrem Linux-Installationsmedium oder laden sich das aktuelle *Java SE Development Kit (JDK)* für Windows von `java.sun.com`. Unter MacOSX ist Java praktisch immer installiert.

Sie dürfen nur eigene Klassen und solche aus der Java-Standard-Bibliothek verwenden. Die Nutzung anderer Klassenbibliotheken ist nicht zulässig.

Eclipse

Die zu verwendende Programmierumgebung ist *Eclipse*, das man sich von der Webseite `www.eclipse.org` herunterladen kann. Empfohlen wird, die aktuelle Version von *Eclipse IDE for Java Developers* zu benutzen. In vielen Linux-Distributionen ist Eclipse schon enthalten.

CVS

Verwendet wird außerdem die *Versionskontrolle mit CVS* auf einem zentralen CVS-Server. Größere Programmierprojekte sind heute ohne Versionskontrolle kaum vorstellbar und sollten jedem Informatiker geläufig sein.

Die Dateien, die Sie in Ihrem CVS-Bereich verwalten, sind nur für Sie selbst und für die Praktikumsbetreuer lesbar. Angeschaut werden aber normalerweise nur die von Ihnen abgegebenen (mit `Tag`[1] versehenen) Programmversionen.

CVS-Funktionen

Praktischerweise ist in Eclipse ein CVS-Client eingebaut. Die meisten *CVS-Funktionen* von Eclipse finden sich in den mit der rechten Maustaste erreichbaren Untermenüs `Team` (`Share`, `Update`, `Commit`, `Tag`, ...) und `Compare with`, dazu noch z. B. in der `import`-Funktion und bei `New Project`.

[1] tag (engl.): Etikett, Kennzeichen

Java-Projekt

Zu Beginn sollten Sie ein neues *Java-Projekt* erzeugen und in Ihrem CVS-Bereich sichern (mit `Team->Share`):

```
Host: grisrock.fernuni-hagen.de
Path: /export/homes/CVSpropra
User: (hier q<Matrikelnummer> eintragen, z. B. q1234567)
Connection type: pserver
Password: (wird per E-Mail mitgeteilt)
Name des Moduls und des Projekts: q<Matrikelnummer>
```

Auch die Projektdatei (`.project`) von Eclipse wird mit eingecheckt (passiert normalerweise automatisch).

Nun können Sie direkt aus der Programmierumgebung den Stand Ihrer Arbeit sichern, so oft Sie wollen, (mit `Team->Commit`, Versionenkommentare zum eigenen Nutzen nicht vergessen) und später die fertigen Programmversionen abgeben (mit `Team->Tag`).

Für alle eigenen Java-Packages verwenden Sie bitte Namen mit dem Präfix

```
de.feu.propra12.q<Matrikelnummer>.
```

So können die Packages immer eindeutig zugeordnet werden.

zweites Projekt

Für das vorgeschriebene Java-Interface und zum Testen benötigen Sie unbedingt noch unser vorbereitetes Modul als zweites Projekt in Ihrem Eclipse-Workspace und aus einem zweiten CVS-Repository:

```
Host: grisrock.fernuni-hagen.de
Path: /export/homes/CVSpropra/shared
User: anonymous
Connection type: pserver
Password: (nichts)
Module name: Tester
```

Verknüpfen Sie die beiden Projekte, indem Sie die *Properties* von Ihrem Projekt aufrufen, darin den *Java Build Path* und *Libraries*, dann *Add JARs* anklicken und die Datei `Tester/ProPraTester.jar` aus dem Tester-Projekt auswählen. Wenn Sie außerdem noch zu dieser `jar`-Datei als *Javadoc location* das Verzeichnist `Tester/doc` eintragen, dann kann Eclipse die passende Dokumentation im jeweiligen Kontext anzeigen.

Testmöglichkeiten

Das Tester-Projekt sollten Sie regelmäßig und insbesondere nach einer entspechenden Aufforderung in der Newsgruppe aktualisieren (mit `Team->Update`), um zusätzliche *Testmöglichkeiten* zu bekommen.

fertige Programme abgeben

Das *fertige Programm* zu Teil 1 kennzeichnen Sie bitte in Ihrem CVS-Bereich mit dem Tag `Teil1`. Sie können also in Ruhe an Teil 2 weiterarbeiten und weitere Versionen generieren, ohne die abgegebene Version zu verändern. Das fertige Programm zu Teil 2 kennzeichnen Sie später bitte entsprechend mit dem Tag `Teil2`.

Test-Interface

In `Tester/doc/de/feu/propra12/interfaces` ist die Beschreibung des zu implementierenden *Interfaces* `ISmallestCircleCalculator` zu finden.

Interface

Ihr Programm sollte nun folgendermaßen arbeiten. Wird ihm kein Parameter übergeben, soll die normale Oberfläche mit der graphischen Benutzeroberfläche (siehe weiter unten) erscheinen. Wird dem Programm der Parameter `-t` übergeben, sollte die Testklasse der Bibliothek `ProPraTester.jar` verwendet werden, um die Rückgabewerte zu testen. Um dies zu erreichen, könnten die ersten Zeilen der `main`-Methode wie folgt aussehen:

```
if (args.length > 0 && "-t".equals(args[0])) {
   ISmallestCircleCalculator calculator =
                                  new SmallestCircleCalculator();
   Tester tester = new Tester(args, calculator);
   System.out.println(tester.test());
}
else // Benutzeroberfläche
```

Dabei ist der Name `SmallestCircleCalculator` durch den Namen Ihrer Implementierung des Interfaces `ISmallestCircleCalculator` zu ersetzen.

Teil 1

Die konvexe Hülle

Stellen wir uns eine Menge von einzelnen Punkten in der Ebene anschaulich als *Nägel* vor, die aus der Ebene herausragen. Nun legen wir ein *Gummiband* um diese Nägel herum und lassen es sich zusammenziehen, so dass es so kurz wie möglich wird. Diese Figur, die das Gummiband annimmt, ist der kürzeste Rundweg, der alle Punkte einschließt. Wir nennen ihn die *konvexe Hülle* der Punktmenge; siehe Abbildung 1.1. Einige der Nägel werden vom Gummiband berührt, so dass es dort einen Knick macht; diese Punkte nennen wir *Extrempunkte*.

Nägel

Gummiband

konvexe Hülle

Extrempunkte

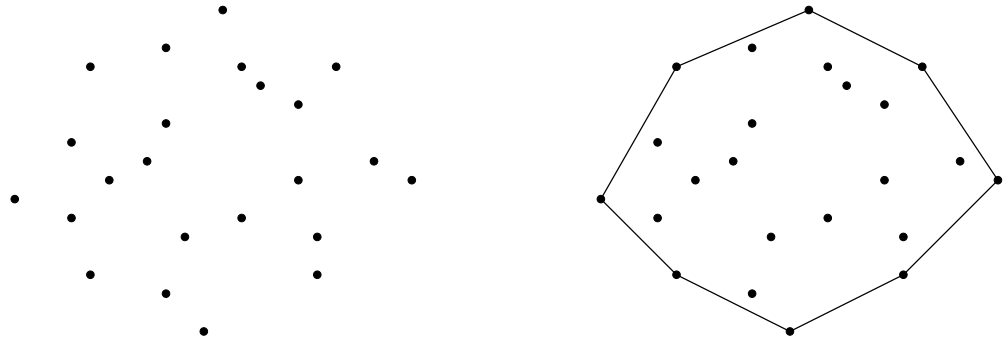


Abbildung 1.1: Eine Menge von Punkten (links) und dieselben Punkte mit ihrer konvexen Hülle (rechts).

Zur *Berechnung der konvexen Hülle* $KH(S)$ einer Punktmenge S von n Punkten müssen alle Extrempunkte gefunden und in der Reihenfolge entlang des Randes (des Gummibands) ausgegeben werden; dies ist die erste Aufgabe im Praktikum.

Berechnung der konvexen Hülle

1.1 Das Konturpolygon-Verfahren

Genauere und formal korrekte Definitionen verschieben wir auf Abschnitt 1.2, zunächst wollen wir aus der großen Auswahl von bekannten Algorithmen zur Konstruktion der konvexen Hülle einer ebenen Punktmenge ein besonders einfaches, leicht verständliches und *worst-case*-optimales Verfahren beschreiben, das *Konturpolygon*-Verfahren. Hier und im Folgenden verzichten wir auf Be-

Konturpolygon

weise und beschränken uns darauf, die nötigen Fakten kurz und anschaulich zusammenzutragen. Wer sich für die Hintergründe und weitere interessante Probleme dieser Art interessiert, sei auf Kurs 1840 *Algorithmische Geometrie* [1] verwiesen.

zwei Phasen

Das Verfahren läuft in *zwei Phasen* ab:

- Zuerst wird das sogenannte *Konturpolygon* von S konstruiert, dessen Ecken zu S gehören. Das Konturpolygon stellt bereits einen Rundweg dar, der jeden Punkt von S einschließt; dies ist aber nicht unbedingt schon der kürzeste Rundweg und daher nicht unbedingt konvex.
- Im zweiten Schritt wird die konvexe Hülle des Konturpolygons gebildet, siehe Abbildung 1.2.

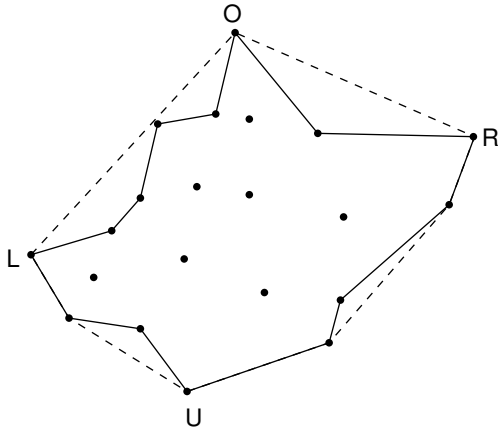


Abbildung 1.2: Zur Konstruktion der konvexen Hülle der Punktmenge wird zunächst ihr Konturpolygon bestimmt.

Das Konturpolygon ist von besonders einfacher Gestalt. Anschaulich entsteht es folgendermaßen: Wir stellen uns wieder vor, dass die Punkte in S als Nägel aus der Ebene herausragen. Links von S liegt ein senkrechter Wollfaden. Er wird nun von einem Gebläse nach rechts gegen die Punkte getrieben. Zuerst trifft er dabei auf den linken Extrempunkt L von S, d. h. auf den Punkt mit minimaler X-Koordinate. Dann schmiegt sich der Faden den Nägeln in der in Abbildung 1.2 gezeigten Weise an, bis er den oberen Extrempunkt O und den unteren Extrempunkt U erreicht hat; dort wird er abgeschnitten. Dasselbe Vorgehen wird von rechts wiederholt. Man nennt die beiden Fäden in ihrer endgültigen Lage die *linke* und die *rechte Kontur der Punktmenge* S, beide zusammen das Konturpolygon. Übrigens sind die Extrempunkte L, R, U, O auf jeden Fall schon Ecken der konvexen Hülle.

linke Kontur
rechte Kontur

Abbildung 1.3 zeigt, dass sich die beiden Konturen außer an ihren Endpunkten O und U auch an anderen Stellen berühren und dass manche Extrempunke zusammenfallen können; das stört uns aber nicht.

Das Konturpolygon wird folgendermaßen bestimmt. Zuerst sortiert man die Punkte nach aufsteigenden X-Koordinaten. Dann bewegt man eine senkrechte Gerade (*sweep line*) von links nach rechts über die Ebene und merkt sich die

sweep line

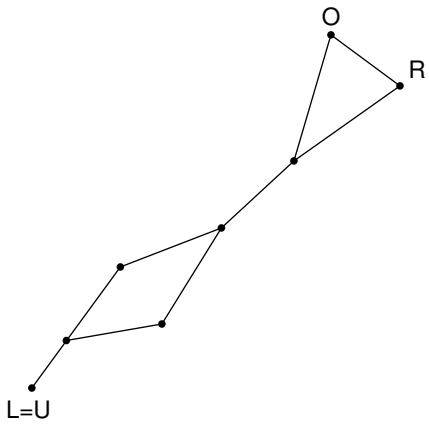


Abbildung 1.3: Ein Konturpolygon mit flachen Stellen.

Punkte *MinYSoFar* und *MaxYSoFar* mit minimaler und mit maximaler Y-Koordinate, die bisher angetroffen wurden.

Am Anfang stimmen beide Punkte mit dem linken Extrempunkt L überein, am Ende ist $MaxYSoFar = O$ und $MinYSoFar = U$. Wann immer ein neuer Punkt *MaxYSoFar* entdeckt wird, verbindet man ihn durch eine Kante mit seinem Vorgänger; dasselbe geschieht mit *MinYSoFar*.

Auf diese Weise lässt sich die linke Kontur berechnen. Mit der rechten Seite verfährt man analog, durch einen *sweep* von rechts nach links.

Jetzt brauchen wir nur noch die konvexe Hülle von P zu bilden. Dazu nehmen wir uns jedes der Fadenstücke zwischen den vier Extrempunkten vor und ziehen es stramm. Es genügt, dieses Verfahren am Beispiel des oberen Teils der linken Kontur zwischen L und O zu demonstrieren; siehe Abbildung 1.4.

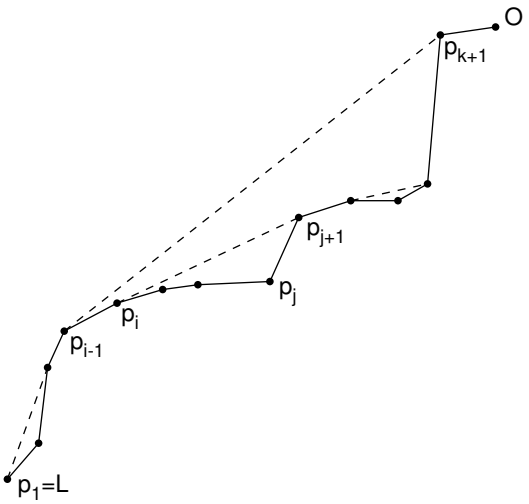


Abbildung 1.4: Berechnung der konvexen Hülle am oberen Teil der linken Kontur.

Die Ecken dieses Randstücks seien von links nach rechts mit $L = p_1, p_2, \ldots$ bezeichnet. Wir durchlaufen sie in dieser Reihenfolge. Wann immer wir an eine hereinragende Ecke p_j kommen, laufen wir so weit zurück, bis wir auf p_1 stoßen

oder vorher auf einen Eckpunkt p_i, für den p_{j+1} rechts von der Geraden durch $p_{i-1}p_i$ liegt; vergleiche Abbildung 1.4. Das gesamte Randstück von p_1 bzw. p_i bis p_{j+1} wird durch das Liniensegment zwischen diesen Punkten ersetzt. Damit ist die spitze Ecke p_j verschwunden, und bis zum Punkt p_{j+1} haben wir eine konvexe polygonale Kette mit Ecken in S erzeugt, die die alte Kontur von oben umschließt. Bei jedem Test benutzen wir den sogenannten *Links-Rechts-Test*, der die Frage beantwortet, ob ein Punkt links oder rechts einer Geraden durch zwei Punkte liegt; dieser Test wird in Abschnitt 1.3 genauer beschrieben.

Links-Rechts-Test

Im Beispiel in Abbildung 1.4 wird das Liniensegment p_ip_{j+1} wieder verschwinden, wenn später $p_{i-1}p_{k+1}$ eingeführt wird.

Abschließend (nur für die, die es wirklich wissen wollen) können wir über die *Laufzeit* des Konturpolygonverfahrens sagen, dass es $O(n \log n)$ Zeit benötigt, und zwar schon durch das Sortieren im ersten Schritt, nach dem Sortieren nur noch $O(n)$ Zeit. Dies ist optimal im schlimmsten Fall (*worst-case*).

Laufzeit

1.2 Begriffe

Jetzt wollen wir aber doch ein paar Begriffe genau erklären, damit eindeutig und nicht nur anschaulich klar ist, wovon geredet wird.

Alles in diesem Text spielt sich im zweidimensionalen Raum ab, also dem $\mathbb{R}^2$ bzw. der *Ebene*, höhere Dimensionen kommen nicht vor.

Ebene

Ein *Punkt* $A = (A_x, A_y)$ ist ein Paar bestehend aus einer X- und einer Y-Koordinate aus den reellen Zahlen ($\mathbb{R}$). Im Programm werden wir nur Punkte mit ganzzahligen Koordinaten behandeln.

Punkt

Eine *Gerade* durch zwei verschiedene Punkte A und B ist anschaulich die gerade Verbindung zwischen den beiden Punkten und deren gerade Verlängerung in beide Richtungen. Mathematisch exakter ausgedrückt ist es die Menge

Gerade

$$\{\, tA + (1-t)B;\ t \in \mathbb{R} \,\}$$

von Punkten in der Ebene, die Gerade ist natürlich durch die beiden Punkte eindeutig bestimmt.

Drei Punkte sind *kollinear*, wenn Sie auf derselben Geraden liegen.

kollinear

Ein *Polygon* ist ein Vieleck (Dreieck, Viereck, Fünfeck, ...), also eine geschlossene Kurve in der Ebene, die nur aus endlich vielen geraden Stücken besteht. Da, wo zwei solche Stücke zusammenstoßen, hat das Polygon seine *Ecken*, die geraden Stücke zwischen den Ecken heißen *Kanten*. Eindeutig bestimmt ist ein Polygon durch die Folge seiner Ecken. Wir betrachen meist nur sogenannte *einfache Polygone*, d. h. zwei nicht benachbarte Kanten schneiden sich nicht, siehe Abbildung 1.5.

Polygon

Ecke

Kante

einfaches Polygon

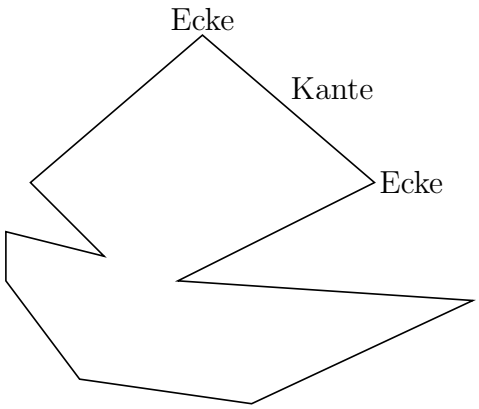


Abbildung 1.5: Ein einfaches Polygon.

Dreieck
$\triangle(A, B, C)$

Ein *Dreieck* ist ein Polygon mit drei Ecken und wird kurz $\triangle(A, B, C)$ geschrieben, wenn A, B und C die Ecken sind.

konvex

Eine Menge von Punkten in der Ebene heißt *konvex*, wenn für je zwei Punkte der Menge die gerade Verbindung ganz in der Menge liegt.

konvexes Polygon

Ein einfaches Polygon heißt konvex, wenn sein Inneres (das eingeschlossene Gebiet) konvex ist. Ein Dreieck z. B. ist immer konvex, wie auch jede Kreisfläche.

konvexe Hülle
konvexes Polygon

Die *konvexe Hülle* $KH(S)$ einer Punktmenge S von n Punkten ist die kleinste konvexe Menge, die S enthält; $KH(S)$ ist immer ein *konvexes Polygon* mit höchstens n Ecken und stellt auch den kürzesten Rundweg dar, der S einschließt.

Begriffe für Teil 2

Die folgenden Begriffe werden erst in Teil 2 benötigt und sind hier schon einmal der Vollständigkeit halber genannt:

Distanz
Abstand

Die *Distanz* (der *Abstand*) zwischen zwei Punkten ist die Länge der geraden Verbindung zwischen den beiden Punkten.

Kreis
Radius
Durchmesser

Der *Kreis* mit Mittelpunkt M und Radius r ist die Menge aller der Punkte, die den Abstand r von M haben. Der *Radius* eines Kreises ist der Abstand vom Mittelpunkt zum Rand, der *Durchmesser* ist der doppelte Radius.

Umkreis
$\bigcirc(A, B, C)$

Der *Umkreis* von drei nicht kollinearen Punkten A, B, und C ist der eindeutig bestimmte Kreis $\bigcirc(A, B, C)$, der durch die drei Punkte geht.

1.3 Links-Rechts-Test

Gegeben sind drei Punkte A, B, C in der Ebene. Man will wissen, ob C links oder rechts von der Geraden g durch A und B liegt, die in Richtung von A nach B orientiert (gerichtet) ist, siehe Abbildung 1.6.

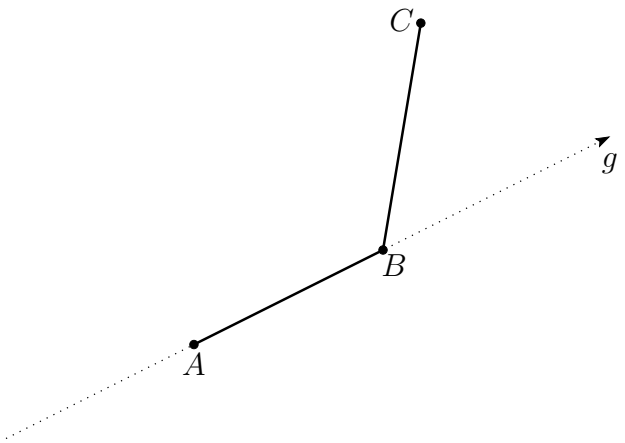


Abbildung 1.6: Hier liegt Punkt C links von der Geraden g durch A und B, die Determinante ist positiv.

Determinante

Dafür benötigen wir die Formel für die *Determinante*:

$$\det(A,B,C) = \begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_x & C_y & 1 \end{vmatrix}$$
$$= (C_x - A_x)(C_y + A_y) + (B_x - C_x)(B_y + C_y) + (A_x - B_x)(A_y + B_y)$$

Hat die Determinante einen positiven Wert, so liegt C links von der orientierten Geraden g, bei einem negativen Wert rechts. Der Wert ist genau dann gleich 0, wenn C auf g liegt, also genau dann, wenn die drei Punkte kollinear sind.

Dreiecksfläche mit Vorzeichen

Wir bezeichnen diese Formel auch als *Dreiecksfläche mit Vorzeichen*, denn sie ergibt den doppelten Flächeninhalt des Dreiecks $\triangle(A,B,C)$, falls A, B, C in dieser Reihenfolge entgegen dem Uhrzeigersinn auf dem Dreiecksrand liegen, bzw. den negativen doppelten Flächeninhalt im Uhrzeigersinn.

ganzzahlige Operationen

große ganze Zahlen

Wichtig ist, dass dieser Test nur Rechenoperationen im Bereich der ganzen Zahlen benutzt, wenn die eingegebenen Koordinaten selbst auch ganze Zahlen sind. Aber Vorsicht: Hier treten schon recht *große ganze Zahlen* auf; bei Verwendung von `int`-Koordinaten sollte die Berechnung der Determinante (inklusive der Zwischenrechnungen!) mindestens als `long` erfolgen. Ein Überlauf von `int`-Zahlen wird von Java nicht bemerkt!

1.4 Anforderungen an das Programm, Teil 1

Interface

Zur Lösung dieser Praktikumsaufgabe schreiben Sie bitte eine Java-Application, die das vorgegebene *Interface* implementiert und die über folgende Funktionen verfügt.

Zeichenfläche

Punkte editieren

Der Benutzer sieht zunächst eine leere *Zeichenfläche* und kann durch Mausklicks nach Belieben Punkte (der Punktmenge S) setzen, löschen und verschieben.

Bildschirmkoordinaten

Die Eingabekoordinaten sind vom Typ `int`, es können also die *Bildschirmkoordinaten* von Mausklicks ohne jede Umrechnung direkt übernommen werden. Bildschirmkoordinaten sind allerdings üblicherweise in der Vertikalen von oben nach unten orientiert, so dass sich auch die Rollen von Links und Rechts vertauschen.

keine Rechenungenauigkeiten

Alle Berechnungen benutzen ganze Zahlen, so dass das Ergebnis *keine Rechenungenauigkeiten* enthalten kann.

konvexe Hülle

dynamische Anzeige

Das Programm berechnet für jede Konstellation sofort die *konvexe Hülle* und zeigt sie an. Das gilt auch für das Verschieben: Während mit der Maus ein Punkt verschoben wird, geht die Hülle immer mit, falls sich daran durch die Verschiebung etwas ändert.

zufällige Eingaben

Das Programm kann *zufällige Eingaben* von 10, 50, 100, 500 und 1000 Punkten erzeugen, die im Bereich der Zeichenfläche liegen sollen, und diese Beispielaufgaben durchrechnen.

Dateien einlesen und abspeichern

Das Programm kann Punktmengen aus *Dateien einlesen und abspeichern*, die Daten werden im Textformat abgelegt, wobei in jeder Zeile ein Punkt mit durch Leerzeichen getrennten ganzzahligen x- und y-Koordinaten steht. Zum Beispiel bedeutet

```
123 453
237 135
42 357
```

die Punktmenge $\{(123, 453), (237, 135), (42, 357)\}$. Alle Zeilen, die sich nicht in diesem Format einlesen lassen, werden ignoriert (Kommentarzeilen).

selbst programmieren

Java-Library

Der Algorithmus zur Berechnung der konvexen Hülle (Konturpolygon-Verfahren oder auch ein anderes nach Wahl) wird selbst programmiert. Für das Sortieren können allerdings fertige Funktionen aus der *Java-Library* benutzt werden. Es wird kein Programmcode aus anderen Quellen verwendet.

Dokumentation mit Javadoc

Überlegen Sie sich einen einheitlichen Kommentarkopf mit Javadoc-Elementen, den Sie vor jeder Klasse und Methode einfügen, die kommentiert werden soll. In diesem Kommentarkopf beschreiben Sie den Zweck der verwendeten Parameter, die Aufgabe, die von der Klasse bzw. Methode erfüllt wird, und die Einordnung der Klasse/Methode in den Gesamtkontext des Programmes. Benutzen Sie das Javadoc-Werkzeug, um die Parameter und Rückgabewerte auszuzeichnen. Halten Sie die Beschreibung knapp, zwei bis drei prägnante Sätze zur Aufgabe und Einordnung der Klasse bzw. Methode sollten ausreichen.

Eclipse unterstützt Javadoc

Nutzen Sie die Unterstützung von *Eclipse* für Javadoc, das macht es für Sie bequemer. Am Ende muss mittels des Javadoc-Systems eine HTML-Dokumentation des Programms erzeugt werden können, die so ausführlich ist, dass auf einen separaten Text als Programmdokumentation verzichtet werden kann.

Teil 2

Der kleinste Kreis

2.1 Motivation

Was ist der günstigste Standort für eine Feuerwache? Die Feuerwehr soll natürlich möglichst schnell zu jedem Haus in einem bestimmten Gebiet gelangen können. Deshalb muss der Standort einer Feuerwache sorgfältig gewählt werden.

Die Feuerwehr soll auch im schlimmsten Fall, also zum am weitesten entfernten Haus, einen möglichst kurzen Weg haben. Gesucht ist also der zentrale Standort, so dass dieser weiteste Weg möglichst kurz ist.

Der Einfachheit halber stellen wir uns vor, dass die Häuser und die Feuerwache Punkte sind und nur die Distanz (Luftlinie) zwischen Feuerwache und den Häusern berücksichtigt werden soll. Der Standort der Feuerwache ist dann der Mittelpunkt eines Kreises, innerhalb dessen alle Häuser liegen. Das Haus mit der größten Entfernung liegt auf dem Kreisrand und bestimmt den Kreisradius; dieser Radius, d. h. der Weg der Feuerwehr im schlimmsten Fall, soll möglichst kurz sein.

Eine andere Anwendung, die auf dasselbe geometrische Problem führt, ist die Aufstellung einer Funkstation, z. B. für Mobilfunk oder WLAN. Die Station soll alle Kunden in einem Gebiet versorgen, andererseits soll die Sendeleistung aus Kostengründen möglichst klein sein. Die Sendeleistung hängt aber nur davon ab, wie weit der am weitesten entfernte Kunde von der Station weg liegt. Es wird also ein Standort gesucht, der diese Entfernung minimiert.

2.2 Umsetzung als geometrisches Optimierungsproblem

Die Häuser oder Kunden modellieren wir als Punkte in der Ebene, davon gibt es, sagen wir, n Stück. Wählen wir zunächst einen beliebigen Standort M und messen die Distanz von M zu jedem der n Punkte; die maximale Distanz wird bei einem Punkt P erreicht. Zeichnen wir nun einen Kreis mit Mittelpunkt M durch P, dann liegen alle n Punkte innerhalb dieses Kreises, siehe zum

Beispiel Abbildung 2.1. Der Radius des Kreises ist genau die Distanz zwischen M und P.

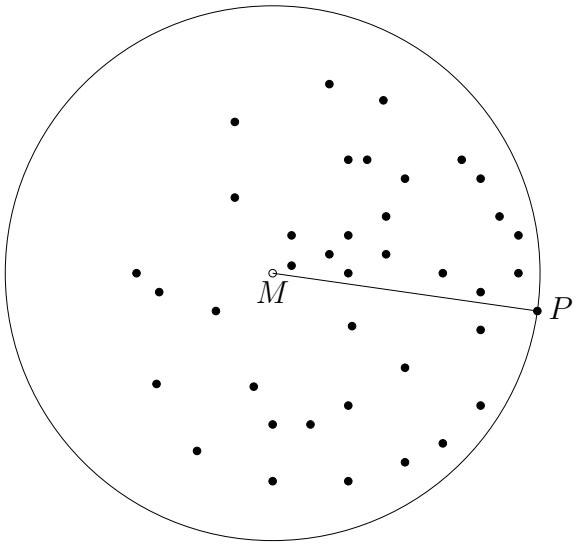


Abbildung 2.1: Der Kreis um M durch P enthält alle Punkte.

Wie man an Abbildung 2.1 leicht sieht, kann man den Kreismittelpunkt meistens noch verschieben, so dass der alles umschließende Kreis kleiner wird. Der gesuchte optimale Standort ist der Mittelpunkt des *kleinsten Kreises*, der alle Punkte enthält.

kleinster umschließender Kreis

Das zu lösende Problem *kleinster umschließender Kreis* (KUK) kann also formal folgendermaßen beschrieben werden:

Problem *KUK*: Gegeben ist eine Menge $S = \{p_1, p_2, \ldots, p_n\}$ von $n \geq 2$ Punkten in der Ebene. Gesucht ist der kleinste Kreis $KUK(S)$, der alle Punkte von S enthält.

Folgende Fakten sind über dieses Problem bekannt:

eindeutig lösbar

- KUK ist immer *eindeutig lösbar*, es gibt also immer genau eine Lösung bestehend aus Mittelpunkt und Radius des optimalen Kreises.

Randpunkte

- Der optimale Kreis $KUK(S)$ enthält mindestens zwei Punkte aus S auf seinem Rand.

Durchmesser

- Wenn $KUK(S)$ nur zwei Punkte aus S auf dem Rand enthält, dann ist deren Verbindungsstrecke der *Durchmesser* des Kreises, und der Kreismittelpunkt liegt genau in der Mitte der beiden Randpunkte; siehe Abbildung 2.2.

- Wenn $KUK(S)$ drei oder mehr Punkte aus S auf dem Rand enthält, dann ist jedes Kreisrandstück zwischen zwei solchen Randpunkten kürzer als ein Halbkreis, siehe Abbildung 2.3.

- Die Punkte aus S auf dem Rand von $KUK(S)$ gehören alle zur konvexen Hülle $KH(S)$ (aber natürlich nicht umgekehrt). Die konvexe Hülle ist in den beiden Abbildungen gestrichelt dargestellt und wurde schon in Teil 1 genauer besprochen.

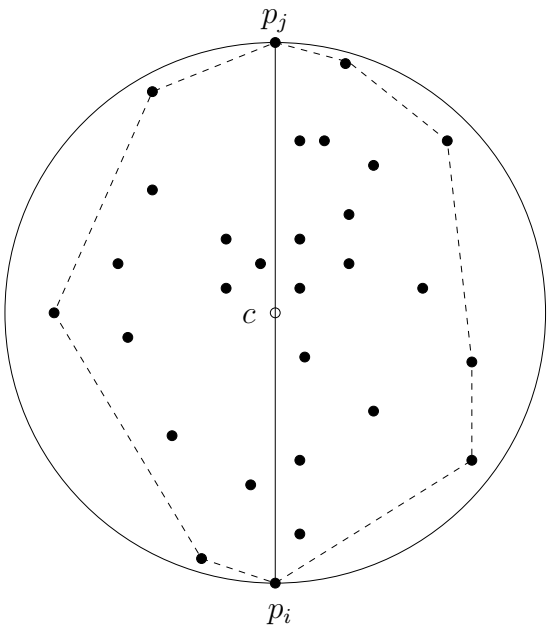


Abbildung 2.2: Der kleinste Kreis $KUK(S)$ geht hier nur durch zwei Punkte.

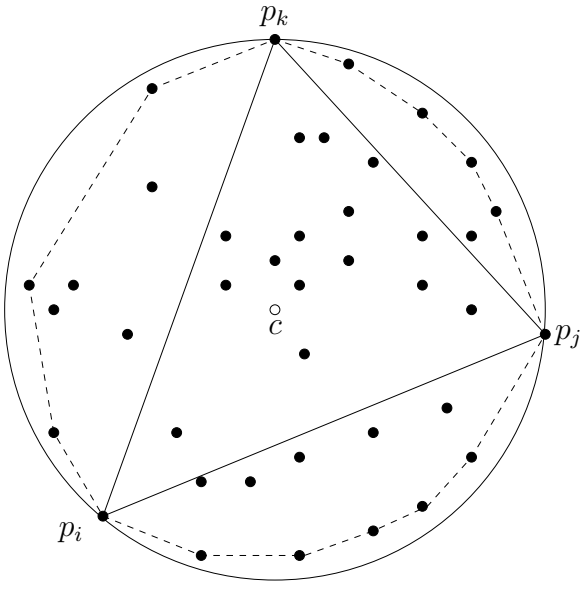


Abbildung 2.3: $KUK(S)$ geht durch die drei Punkte p_i, p_j und p_k aus S.

Schauen wir uns nun Algorithmen für unser Problem an. Die dazu benötigten Grundoperationen und Formeln werden wir später in Abschnitt 2.5 kennen lernen.

2.3 Ein naives und ineffizientes Verfahren

Mit den in Abschnitt 2.2 genannten Eigenschaften kann man auf die Idee kommen, den optimalen Kreis zu berechnen, indem man alle Möglichkeiten von Randpunkten ausprobiert:

- Für alle Paare (P_i, P_j) von Punkten aus S testen wir, ob der Kreis mit dem Durchmesser von P_i nach P_j alle anderen Punkte enthält.
- Für alle Tripel (P_i, P_j, P_k) von Punkten aus S testen wir, ob der Umkreis $\bigcirc(P_i, P_j, P_k)$ alle anderen Punkte enthält.
- Von allen dabei gefundenen Kreisen, die alle Punkte enthalten, nehmen wir den kleinsten.

brute-force-Methode: alle Möglichkeiten durchprobieren

Dieses Verfahren ist zwar korrekt, denn tatsächlich werden alle denkbaren Möglichkeiten berücksichtigt, aber es braucht als so genannte *brute-force-Methode* viel mehr Zeit als nötig: Allein das Testen aller $O(n^3)$ Punktetripel mit allen anderen Punkten benötigt $O(n^4)$ Zeit. Bei einer Verdopplung der Anzahl der Punkte ist die Rechenzeit 16-mal so lang! So ist es praktisch ausgeschlossen, größere Probleme in annehmbarer Zeit durchzurechnen.

Der „Vorteil“ der *brute-force-Methode* liegt nur in ihrer Einfachheit: man benötigt praktisch überhaupt keine Datenstrukturen. Allenfalls zum Testen mit kleinen Beispieldaten kommt sie in Frage: die Ergebnisse dieses naiven und eines komplexeren, aber effizienteren Verfahrens müssen übereinstimmen, sonst ist etwas falsch.

2.4 Ein einfaches und effizientes Verfahren

Es gibt eine Reihe von Algorithmen zur Bestimmung des kleinsten umschließenden Kreises, darunter sogar eines mit $O(n)$ Laufzeit [2], das aber für eine Implementierung, insbesondere im Rahmen dieses Praktikums, wegen seiner Kompliziertheit weniger geeignet ist.

Wir wollen hier das besonders einfache Verfahren von Skyum [3] benutzen. Dieser Algorithmus hat die Laufzeit $O(n \log n)$ und ist verhältnismäßig leicht zu programmieren, da er im Wesentlichen nur Umkreisradien vergleicht, bis der kleinste Kreis gefunden ist. Quasi nebenbei berechnet das Verfahren auch eine *Triangulation der konvexen Hülle*, also eine Zerlegung in Dreiecke, die wir zur Kontrolle auch anzeigen werden.[1]

Triangulation der konvexen Hülle

Das eigentliche Verfahren sehen wir uns in diesem Abschnitt an, die dafür benötigten Grundlagen verschieben wir auf Abschnitt 2.5.

Zunächst nutzen wir die in Abschnitt 2.2 genannte Eigenschaft aus, dass nur Eckpunkte der konvexen Hülle $KH(S)$ als Randpunkte des kleinsten umschließenden Kreises in Frage kommen. Wie wir diese bekommen können, haben wir ja schon in Teil 1 gesehen, jedenfalls gehen wir jetzt davon aus, dass alle übrigen Punkte aus der Menge S entfernt wurden.

[1] Für Insider: es handelt sich um die sogenannte inverse Delaunay-Triangulation der Punkte auf der konvexen Hülle.

Danach haben wir also die Punkte $S = \{P_1, P_2, \ldots, P_n\}$ in der Ebene, die in dieser Reihenfolge im Uhrzeigersinn ein *konvexes Polygon* bilden. Wir setzen auch voraus, dass keine drei aufeinander folgenden Punkte kollinear sind, d. h. auf einer Geraden liegen, sonst kann man immer den mittleren der drei Punkte entfernen, ohne das konvexe Polygon zu verändern.

konvexes Polygon

Sei $|S|$ die Anzahl der Punkte der Menge S, also zunächst $|S| = n$.

Wir bezeichnen mit $\mathrm{Nach}(P)$ den *Nachfolger* einer Ecke P und mit $\mathrm{Vor}(P)$ ihren *Vorgänger* im Uhrzeigersinn; das Ganze ist zyklisch gemeint, also der Nachfolger der letzten Ecke ist wieder die Erste, der Vorgänger der Ersten ist die Letzte usw.

Nachfolger
Vorgänger

Seien $A \neq B$ und $B \neq C$ Punkte, dann bezeichnen wir mit $\mathrm{Radius}(A, B, C)$ den Radius des Umkreises der Punkte. Für den besonderen Fall, dass die beiden Punkte A und C identisch sind, sei $\mathrm{Radius}(A, B, C)$ die Hälfte der Distanz $d(A, B)$, siehe auch Abbildung 2.4; wir werden gleich sehen, warum das Sinn macht.

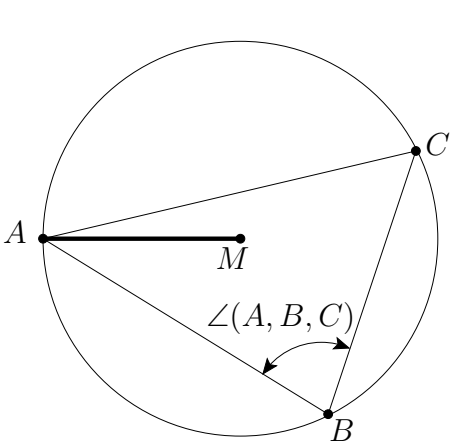
$\mathrm{Radius}(A, B, C)$

Mit $\angle(A, B, C)$ bezeichnen wir den Winkel zwischen den Liniensegmenten AB und BC. Für $A = C$ gelte $\angle(A, B, C) = 0$. Für jede Ecke P aus S gilt nun immer

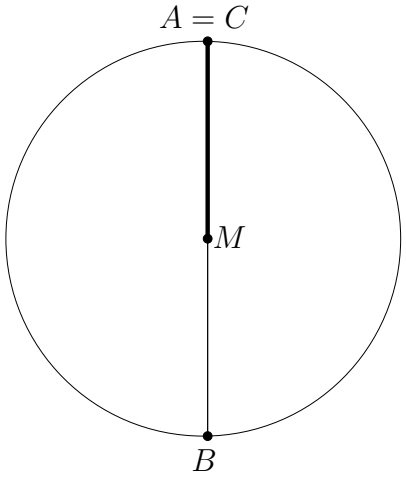
$\angle(A, B, C)$

$$0 \leq \angle\left(\mathrm{Vor}(P), P, \mathrm{Nach}(P)\right) < 180^\circ,$$

da die drei aufeinander folgenden Ecken $\mathrm{Vor}(P)$, P und $\mathrm{Nach}(P)$ auf der konvexen Hülle liegen und nicht kollinear sind.



$\mathrm{Radius}(A, B, C) = d(M, A)$



$\mathrm{Radius}(A, B, C) = d(M, A) = d(A, B)/2$
$\angle(A, B, C) = 0$

Abbildung 2.4: Der Umkreis des Dreiecks $\triangle(A, B, C)$ hat den Radius $d(M, C)$, und die Größe des Winkels zwischen AB und BC liegt zwischen 0° und 180°.

Ordnung für Umkreise

Es wird nun eine *Ordnung für Umkreise* definiert:

$$\bigcirc(A,B,C) < \bigcirc(E,F,G) \iff$$
$$\text{Radius}(A,B,C) < \text{Radius}(E,F,G)$$
$$\text{oder}$$
$$\Big(\text{Radius}(A,B,C) = \text{Radius}(E,F,G) \text{ und } \angle(A,B,C) < \angle(E,F,G) \Big)$$

Die Kreise werden also in erster Priorität nach ihrem Radius geordnet, bei gleichem Radius wird dann der Winkel zum Vergleich herangezogen (lexikographische Ordnung).

Der Algorithmus von Skyum [3] sucht immer wieder mittels der oben beschriebenen Ordnung den größten vorkommenden Umkreis von drei aufeinander folgenden Ecken $\text{Vor}(P), P, \text{Nach}(P)$. Interessanterweise gilt nun, dass dieser größte Umkreis schon der gesuchte optimale Kreis ist, wenn der Winkel $\angle(\text{Vor}(P), P, \text{Nach}(P))$ kleiner oder gleich 90° ist. Wenn er aber größer als ein rechter Winkel ist, dann kann auf den mittleren der drei Punkte verzichtet werden, der kleinste umschließende Kreis der Punktemenge ändert sich durch die Entfernung dieses Punktes nicht:

Algorithmus von Skyum

Algorithmus von Skyum

Punktmenge S gegeben mit $|S| \geq 2$.

Entferne alle Punkte aus S, die nicht auf der konvexen Hülle liegen und sortiere die verbliebenen Punkte im Uhrzeigersinn.

finish := false;

repeat

(1) finde den Punkt $P_{\max}$ in S, so dass das Maximum

$$\max_{P \in S} \Big(\bigcirc (\text{Vor}(P), P, \text{Nach}(P)) \Big)$$

bei $P_{\max}$ erreicht wird.

(2) **if** $\angle(\text{Vor}(P_{\max}), P_{\max}, \text{Nach}(P_{\max})) > 90°$ **then**
Kante $\text{Vor}(P_{\max})$ zu $\text{Nach}(P_{\max})$ gehört zur Triangulation, merken
entferne $P_{\max}$ aus S
else
finish := true
fi;

until finish;

return $\Big(\bigcirc (\text{Vor}(P_{\max}), P_{\max}, \text{Nach}(P_{\max})) \Big)$

Nachdem zunächst sichergestellt ist, dass die verbliebenen Punkte ein konvexes Polygon bilden, wird in (1) der Punkt $P_{\max}$ in S gesucht, so dass der Umkreis $\bigcirc(\text{Vor}(P), P, \text{Nach}(P))$ nach dem beschriebenen Ordnungskriterium am größten wird.

Falls nun der Winkel

$$\angle\left(\mathrm{Vor}(P_{\max}), P_{\max}, \mathrm{Nach}(P_{\max})\right) \leq 90^\circ \ ,$$

nicht größer als ein rechter Winkel ist, dann haben wir in (2) den kleinsten umschließenden Kreis von S gefunden und der Algorithmus terminiert. Sonst wird $P_{\max}$ aus S entfernt, so dass die Anzahl der Punkte in S um 1 abnimmt. Bei jeder solchen Entfernung haben wir nebenbei auch eine Kante der Triangulation gefunden, nämlich vom Vorgänger zum Nachfolger von $P_{\max}$. Spätestens bei $|S| = 2$ terminiert der Algorithmus, denn da ist der Winkel 0.

Abbruchbedingung

Praktischerweise wurde so eine Fallunterscheidung zwischen dem Zwei-Punkte-Fall (Abbildung 2.2) und dem Drei-Punkte-Fall (Abbildung 2.3) vermieden. Der Algorithmus liefert als Lösung immer den Umkreis $\bigcirc(A, B, C)$ von drei Punkten, wobei der Zwei-Punkte-Fall auftritt, falls die beiden Punkte A und C identisch sind.

Falls übrigens mehr als drei Punkte auf dem kleinsten Kreis liegen, dann bemerkt der Algorithmus diesen Sonderfall nicht, sondern er entfernt so lange weitere Punkte, bis er zu einem Dreieck kommt. Das Ergebnis ist trotzdem richtig, denn alle ursprünglichen Punkte liegen ja in dem Umkreis oder auf dessen Rand.

Wie soll aber in (1) das jeweilige Maximum in jeder Runde *effizient* bestimmt werden? Dafür benutzen wir eine Datenstruktur, die eine Menge von Objekten sortiert verwalten kann, indem sie die Operationen *Einfügen*, *Entfernen* und *Maximum* bereitstellt. Ein balancierter Baum z. B. bietet diese Operationen jeweils in Zeit $O(\log n)$ an.

Maximum bestimmen

Wir benutzen jetzt solch eine Datenstruktur zur Verwaltung der Punkte *geordnet nach dem Umkreisradius*, oder wir speichern darin statt der Punkte gleich die Umkreise: Hat das konvexe Polygon S also die Ecken P_1, P_2, …, P_n, dann wird die Struktur anfangs mit allen Umkreisen $\bigcirc(P_1, P_2, P_3)$, $\bigcirc(P_2, P_3, P_4)$, …, $\bigcirc(P_n, P_1, P_2)$ gefüllt. Jeweils wenn der aktuell größte Kreis $\bigcirc(P_{i-1}, P_i, P_{i+1})$ gefunden ist, also $P_{\max} = P_i$, werden *alle drei Umkreise entfernt*, an denen der Punkt P_i beteiligt ist, und *zwei neue Umkreise* aufgenommen, die jetzt von drei aufeinanderfolgenden Punkten gebildet werden, nämlich $\bigcirc(P_{i-2}, P_{i-1}, P_{i+1})$ und $\bigcirc(P_{i-1}, P_{i+1}, P_{i+2})$, wobei alle Indizes natürlich modulo n gerechnet werden.

geordnet nach dem Umkreisradius

drei Umkreise entfernt

zwei neue Umkreise

Wichtig ist, dass so pro Iteration nur konstant viele Änderungen an der Struktur vorgenommen werden. Auf keinen Fall wird jedes Mal das Maximum dadurch bestimmt, dass alle Umkreise einmal angeschaut werden, denn dann kann die Gesamtlaufzeit nicht mehr in $O(n \log n)$ sein.

2.5 Formeln und grundlegende Operationen

In diesem Abschnitt listen wir die benötigten Formeln auf und erklären ein paar Begriffe aus der Geometrie und grundlegende geometrische Operationen.

2.5.1 Distanz

Die Distanz zwischen zwei Punkten A und B ist die Zahl

$$d(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}\ ,$$

Distanzenvergleich

wobei man oft auf die Berechnung der Wurzel verzichtet, denn zum Vergleich zweier Distanzen kann man effizienter deren Quadrate verwenden, wobei wir bei ganzahliger Eingabe auch im Bereich der ganzen Zahlen bleiben:

$$d^2(A,B) = (A_x - B_x)^2 + (A_y - B_y)^2$$

2.5.2 Umkreis

Der Umkreis von drei nicht-kollinearen Punkten A, B und C ist immer eindeutig bestimmt. Sein Radius und sein Mittelpunkt lassen sich aus den Koordinaten der drei Punkte berechnen.

Umkreisradius

In der Formel für den *Umkreisradius* steht der Betrag der eben erwähnten Determinante $\det(A,B,C)$, die ja bei nicht-kollinearen Punkten nicht 0 ist, im Nenner und die Seitenlängen des Dreiecks $\triangle(A,B,C)$ im Zähler:

$$R = \frac{d(A,B)\cdot d(B,C)\cdot d(C,A)}{2\cdot|\det(A,B,C)|}\ ,$$

Den Radius selbst benötigen wir so aber nur ganz zum Schluss, weil wir meistens nur zwei Radien miteinander vergleichen werden; hierfür könnten wir wieder auf die Wurzeln verzichten und nur die Quadrate vergleichen:

$$R^2 = \frac{d^2(A,B)\cdot d^2(B,C)\cdot d^2(C,A)}{4\cdot\det^2(A,B,C)}$$

Hier kommt allerdings immer noch eine Division vor, die wir auch lieber vermeiden wollen. Ganz ohne Rechenungenauigkeit ist der Vergleich zweier Radien $\text{Radius}(A,B,C)$ und $\text{Radius}(E,F,G)$, wenn die Koordinaten aller Punkte A bis F ganzzahlig sind (z. B. durch Mausklick erzeugte Bildschirmkoordinaten)

Radienvergleich

und wir durch folgenden Test im Raum der ganzen Zahlen bleiben:

$$\text{Radius}(A,B,C) < \text{Radius}(E,F,G) \iff$$

$$\det^2(E,F,G)\cdot d^2(A,B)\cdot d^2(B,C)\cdot d^2(C,A) < \det^2(A,B,C)\cdot d^2(E,F)\cdot d^2(F,G)\cdot d^2(G,E)$$

extrem große ganze Zahlen!

Hier ist allerdings Vorsicht angebracht! Die Rechenergebnisse können extrem groß werden, so dass die Java-Typen `int` und `long` nicht mehr ausreichen. Statt dessen sollte man, wo es nötig ist, `BigInteger` verwenden.

Für das Endergebnis benötigen wir ganz zum Schluss neben dem Radius auch den *Umkreismittelpunkt* $M = (M_x, M_y)$, dieser hat natürlich keine ganzzahligen Koordinaten mehr:

Umkreismittelpunkt

$$M_x = \frac{1}{2} \cdot \frac{(A_x^2 + A_y^2)(B_y - C_y) + (B_x^2 + B_y^2)(C_y - A_y) + (C_x^2 + C_y^2)(A_y - B_y)}{A_y(C_x - B_x) + B_y(A_x - C_x) + C_y(B_x - A_x)}$$

$$M_y = \frac{1}{2} \cdot \frac{(A_x^2 + A_y^2)(C_x - B_x) + (B_x^2 + B_y^2)(A_x - C_x) + (C_x^2 + C_y^2)(B_x - A_x)}{A_y(C_x - B_x) + B_y(A_x - C_x) + C_y(B_x - A_x)}$$

2.5.3 Winkelvergleich

Nun wollen wir zwei Winkel $\angle(A, B, C)$ und $\angle(E, F, G)$ vergleichen. Im Prinzip gilt ja nach dem Kosinussatz

$$\angle(A, B, C) = \arccos\left(\frac{d^2(A, B) + d^2(B, C) - d^2(A, C)}{2 \cdot d(A, B) \cdot d(B, C)}\right) ,$$

aber diese aufwändige und mit Rechenungenauigkeiten behaftete Formelauswertung wollen wir so nicht durchführen lassen. Der genaue Winkel interessiert uns doch gar nicht, sondern nur, welcher von zwei Winkeln größer ist!

Folgendes Kriterium ist von der genannten Formel abgeleitet, aber leichter auszuwerten. Zunächst seien

$$W_{ABC} = d^2(A, B) + d^2(B, C) - d^2(A, C)$$

und

$$W_{EFG} = d^2(E, F) + d^2(F, G) - d^2(E, G)$$

sowie

$$\text{sign}(x) = \begin{cases} 1 & \\ 0 & \text{falls } x = 0 \\ -1 & \end{cases} \quad \begin{array}{l} x > 0 \\ \\ x < 0 \end{array}$$

Nun gilt:

Vergleich zweier Winkel

$$\angle(A, B, C) \; < \; \angle(E, F, G) \iff$$

$$\Big(\text{sign}(W_{ABC}) > \text{sign}(W_{EFG})\Big) \text{ oder}$$

$$\Big(\text{sign}(W_{ABC}) = \text{sign}(W_{EFG}) \text{ und } \text{sign}(W_{ABC}) \cdot W_{ABC}^2 \cdot d^2(E, F) \cdot d^2(F, G) > \text{sign}(W_{EFG}) \cdot W_{EFG}^2 \cdot d^2(A, B) \cdot d^2(B, C)\Big)$$

Wieder bleiben wir bei ganzzahliger Eingabe auch immer ganzzahlig, wie auch beim Radienvergleich, aber wieder treten sehr große Zahlen auf!

Viel einfacher ist der Test, ob ein Winkel nicht größer als ein rechter Winkel ist:

Vergleich mit rechtem Winkel

$$\angle(A, B, C) \leq 90^\circ \iff W_{ABC} \geq 0$$

2.6 Anforderungen an das Programm, Teil 2

Zur Lösung der Praktikumsaufgabe erweitern Sie bitte die Java-Application aus Teil 1 um folgende Funktionen, die bisherigen Anforderungen wie zufälliges Erzeugen, Speichern/Laden, Interface und Dokumentation bleiben natürlich weiter bestehen. Das fertige Programm kennzeichnen Sie bitte in Ihrem CVS-Bereich mit dem Tag `Teil2`.

konvexe Hülle
kleinster Kreis
Triangulation

Das Programm berechnet für jede Konstellation der eingegebenen Punkte sofort die *konvexe Hülle und* den *kleinsten umschließenden Kreis* (*KUK*) und zeigt sie zusammen mit den Kanten der *Triangulation* an.

dynamische Anzeige
Beispiel

Das gilt auch für das Verschieben: Während mit der Maus ein Punkt verschoben wird, geht die Hülle, deren Triangulation und der kleinste Kreis immer mit, falls sich daran durch die Verschiebung etwas ändert. Auf der Titelseite dieser Aufgabenstellung finden Sie ein Beispielbild mit allen diesen Elementen: Punkte, konvexe Hülle, Triangulation und kleinster Kreis.

exakte Rechnungen

Alle Zwischenrechnungen werden exakt mit ganzen Zahlen und dem passenden Datentyp durchgeführt. Nur der Radius und der Mittelpunkt des kleinsten Kreises werden zum Schluss über die vorgegebene Schnittstelle als `Double`-Werte zurückgegeben.

Undo/Redo

Es gibt eine *Undo/Redo*-Funktion: alle vorhergehenden Einfügungen, Verschiebungen usw. können durch sukzessives *Undo* rückgängig und durch *Redo* entsprechend wieder gültig gemacht werden. Undo und Redo ändern aber nichts an abgespeicherten Dateien.

selbst programmieren
Java-Library

Auch der Algorithmus von Skyum (oder ein anderer nach Wahl...) wird selbst programmiert. Für das Sortieren und den balancierten Baum können fertige Funktionen aus der *Java-Library* verwendet werden.

Literatur

[1] R. Klein, C. Icking. *Algorithmische Geometrie*. Kurs 1840 der FernUniversität in Hagen, 2012.

[2] N. Megiddo. *Linear-Time Algorithms for Linear Programming in* $\mathbb{R}^3$ *and related Problems*. SIAM J. Comput. 12 (1983) 759–776.

[3] S. Skyum. *A Simple Algorithm for Computing the Smallest Enclosing Circle*. Information Processing Letters 37 (1991) 121–125.