Aufgabe 3: Hund mit Leine

Team-ID: 00572

Team-Name: Ralli

Bearbeiter/-innen dieser Aufgabe:
David Adam

06. November 2025

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabe besteht darin, die maximale Länge einer Hundeleine zu berechnen, sodass die Leine bei einem bestimmten Spazierweg nicht durch einen See geht.

Meine Lösungsidee basiert auf der Erkenntnis, dass die Leine ein Segment zwischen zwei Punkten des Spazierwegs ist. Die Leine darf weder einen See schneiden noch durch einen See gehen. Die maximale erlaubte Länge ist also der minimale Abstand zwischen irgendeinem Wegsegment und irgendeiner Seekante.

Ich betrachte jedes Segment des Spazierwegs und jeden See einzeln. Für jede Kombination prüfe ich:

- Liegt ein Endpunkt des Wegsegments im See? → Leinenlänge ist 0
- Schneidet das Wegsegment eine Kante des Sees? → Leinenlänge ist 0
- Sonst: Berechne den minimalen Abstand zwischen dem Wegsegment und allen Kanten des Sees

Das globale Minimum über alle diese Abstände ist die maximale Leinenlänge.

2 Umsetzung

Zuerst implementiere ich grundlegende Geometrie-Helfer. Mit `dot`, `cross`, `sub` und `dist` lassen sich die nötigen Vektoroperationen berechnen. Darauf aufbauend prüfen `orientation`, `on_segment` und `segments_intersect`, ob sich Segmente schneiden. Die Abstände bestimme ich mit `dist_point_segment` und daraus `segment_segment_distanc` Für Punkt-in-Polygon verwende ich Ray-Casting in `point_in_polygon`; Punkte auf Kanten zählen als innen. In `compute_max_leash_length` behandle ich zunächst die harten Fälle (Endpunkt im See oder Schnitt → Ergebnis 0), ansonsten nehme ich das Minimum aller Segment–Kanten-Abstände. Das `main` liest die Eingabe und gibt die Länge mit sechs Nachkommastellen aus (bzw. `inf`, wenn es keine Seen gibt).

3 Werkzeuge

math-Bibliothek: Für die Distanzberechnung nutze ich `math.hypot`, was numerisch stabiler ist als `sqrt(dx**2 + dy**2)`.

Geometrie-Algorithmen: Ich habe mir verschiedene Online-Ressourcen zu geometrischen Algorithmen angeschaut, besonders zu Segment-Schnitt-Tests und dem Ray-Casting-Algorithmus. Die Grundideen stammen aus verschiedenen Quellen, aber die konkrete Implementierung habe ich selbst geschrieben.

VS Code mit Inline-Vervollständigung: Die Codevervollständigung hat mir beim Schreiben der vielen ähnlichen geometrischen Funktionen geholfen, besonders bei den Koordinatenzugriffen (`a[0]`, `a[1]`).

Epsilon-Vergleiche: Um numerische Ungenauigkeiten bei Fließkommazahlen zu behandeln, verwende ich eine kleine Konstante `EPS = 1e-9`. Statt `x == 0` prüfe ich `abs(x) < EPS`. Diese Technik habe ich aus verschiedenen Geometrie-Algorithmus-Implementierungen übernommen.

Vorgehensweise: Ich habe zuerst auf Papier verschiedene Fälle gezeichnet (Segmente, die Seen schneiden, Segmente parallel zu Seen, etc.). Dann habe ich die grundlegenden geometrischen Funktionen implementiert und mit einfachen Testfällen getestet. Danach kam die Hauptlogik, die alle Fälle kombiniert. Am schwierigsten war es, alle Sonderfälle korrekt zu behandeln, besonders wenn Punkte genau auf Kanten liegen.

4 Beispiele

4.1 hund01.txt

```
4.036037
```

4.2 hund02.txt

```
4.209878
```

4.3 hund03.txt

```
2.236068
```

4.4 hund04.txt

```
1.671258
```

4.5 hund05.txt

```
50.320350
```

4.6 hund06.txt

```
199.572502
```

5 Quellcode (Auszug)

```python
def segments_intersect(a, b, c, d):
    """Prueft ob sich zwei Segmente schneiden"""
    o1 = orientation(a, b, c)
    o2 = orientation(a, b, d)
    o3 = orientation(c, d, a)
    o4 = orientation(c, d, b)

    # Kollineare Sonderfaelle
    if abs(o1) < EPS and on_segment(a, b, c): return True
    if abs(o2) < EPS and on_segment(a, b, d): return True
    if abs(o3) < EPS and on_segment(c, d, a): return True
    if abs(o4) < EPS and on_segment(c, d, b): return True

    # Allgemeiner Fall
    return (o1 > 0) != (o2 > 0) and (o3 > 0) != (o4 > 0)


def dist_point_segment(p, a, b):
    ab = sub(b, a)
    ap = sub(p, a)
    ab2 = dot(ab, ab)
    if ab2 < EPS:
        # a und b sind fast gleich
        return dist(p, a)
    t = dot(ap, ab) / ab2
    if t < 0:
        return dist(p, a)
    elif t > 1:
        return dist(p, b)
    proj = (a[0] + t * ab[0], a[1] + t * ab[1])
    return dist(p, proj)


def compute_max_leash_length(segments, lakes):
    if not lakes:
        return float("inf")

    best = float("inf")

    for a, b in segments:
        for poly in lakes:
            n = len(poly)

            # Endpunkte im See?
            if point_in_polygon(a, poly) or point_in_polygon(b, poly):
                return 0.0

            for i in range(n):
                c = poly[i]
                d = poly[(i + 1) % n]

                # Schnitt mit einer Polygonkante?
                if segments_intersect(a, b, c, d):
                    return 0.0

                d_seg = segment_segment_distance(a, b, c, d)
                if d_seg < best:
                    best = d_seg

    if best == float("inf"):
        # sollte praktisch nicht vorkommen, falls es Seen gibt
        return 0.0
    return best
```

Listing 1: Geometrische Kernfunktionen