Aufgabe 4: Krocket

Team: HochgradigTalentiertWenigKrips, ID: 00476

Jurek Engelmann, Lennart Peters

17. November 2024

Inhaltsverzeichnis

1 Übersicht

Das Ziel des Algorithmus ist es, zu überprüfen, ob es möglich ist, eine Gerade zu finden, die mit einem einzigen Schuss durch alle gegebenen Tore führt, und wenn dies möglich ist den Startpunkt und die Richtung des Schusses zu bestimmen. Dabei wird berücksichtigt, dass der Ball einen bestimmten Radius besitzt.

Die Lösung gliedert sich in zwei Hauptphasen:

1. **Test zur richtigen Reihenfolge der Tore**
2. Ein geometrischer Ansatz prüft, ob die Tore überhaupt in der vorgegebenen Reihenfolge durchschossen werden können, ohne den Schuss zu blockieren. Dies geschieht mithilfe des Linear Ordering Tests, der die Relationen zwischen den Toren untersucht.
3. **Finden der optimalen Geraden**
4. Durch systematisches Testen von Linien, die durch die Torpfosten verlaufen, wird eine Gerade gesucht, die alle Tore in der richtigen Reihenfolge schneidet. Dabei wird zunächst der Ballradius ignoriert, um die Komplexität zu verringern. Anschließend wird überprüft, ob diese Linie „aufgeblasen“ werden kann, um den Ballradius zu berücksichtigen.

2 Test zur richtigen Reihenfolge der Tore (Linear Ordering Test)

Der Linear Ordering Test dient dazu, vorab zu prüfen, ob es geometrisch möglich ist, alle Tore in der vorgegebenen Reihenfolge zu durchschießen. Dabei wird die Richtung des Schusses zwischen aufeinanderfolgenden Toren analysiert, indem die Winkelbeziehungen zwischen den Pfosten berücksichtigt werden.

2.1 Berechnung der Winkel

Für jedes Tor werden die folgenden Winkel berechnet:

- Winkel zwischen dem ersten Pfosten des aktuellen Tores, dem zweiten Pfosten des aktuellen Tores und einem beliebigen Pfosten des nächsten Tores.
- Winkel zwischen dem zweiten Pfosten des aktuellen Tores, dem ersten Pfosten des aktuellen Tores und einem beliebigen Pfosten des nächsten Tores.

2.2 Interpretation der Winkel

Die Winkel bestimmen die Richtung des Schusses zwischen zwei aufeinanderfolgenden Toren. Es gibt drei mögliche Fälle:

1. Beide Winkel sind größer als 180°:
    1. In diesem Fall verläuft die Verbindung zwischen den Toren auf einer Seite, Wenn dies zum ersten Mal auftritt, wird diese Richtung als Referenzrichtung festgelegt.
    2. Falls sich diese Richtung in späteren Tests ändert, ist das Durchschießen unmöglich.
2. Beide Winkel sind kleiner als 180°:
    1. Auch hier wird beim ersten Auftreten die Richtung festgelegt.
    2. Falls sich diese Richtung ändert, ist das Durchschießen unmöglich.
3. Ein Winkel ist größer als 180° und der andere kleiner:
    1. Hier kann keine klare Aussage zur Richtung gemacht werden. Der Algorithmus setzt die Analyse fort, um zu prüfen, ob andere Verbindungen eindeutige Hinweise liefern.

2.3 Ergebnis des Linear Ordering Tests

- **Schuss ist unmöglich**: Falls sich die Richtung zwischen zwei Toren ändert, wird der Test beendet, und der Algorithmus gibt aus, dass es keine Lösung gibt.
- **Unklar**: Wenn der Test keine widersprüchlichen Hinweise liefert, geht der Algorithmus zur vollständigen Lösung über.

3 Finden der optimalen Geraden

Falls der Linear Ordering Test keinen Widerspruch ergibt, wird versucht, eine Gerade zu finden, die alle Tore in der richtigen Reihenfolge schneidet. Der Ballradius wird dabei zunächst ignoriert, was bedeutet, dass die Tore als einfache Geradensegmente betrachtet werden. Eine Lösung wird schrittweise verfeinert.

3.1 Beobachtung: Verwendung von Ankerpunkten

Jede mögliche Gerade, die alle Tore in der richtigen Reihenfolge schneidet, muss durch mindestens zwei Pfosten verlaufen. Diese Pfosten werden als Ankerpunkte bezeichnet. Die Reduktion auf Geraden durch *Ankerpunkte* vereinfacht die Suche erheblich.

3.2 Brute-Force-Suche nach Geraden

Der Algorithmus untersucht systematisch alle möglichen Kombinationen von Pfosten:

1. Kombination der Pfosten:
    1. Für jede mögliche Kombination von zwei Pfosten wird eine Gerade berechnet.
2. Schnittprüfung:
    1. Es wird überprüft, ob die Gerade alle Tore in der richtigen Reihenfolge schneidet. Tore, deren Pfosten auf der Geraden liegen, werden automatisch als „durchschossen“ gewertet.
3. Optimierung der Suche:
    1. Der Algorithmus beginnt die Suche bei Pfosten nahe der Mittellinie der Tore und arbeitet sich nach außen vor. Dies erhöht die Wahrscheinlichkeit, schneller eine Lösung zu finden. Pfosten desselben Tores werden nicht miteinander verbunden, da solche Geraden keine praktikablen Lösungen darstellen.

3.3 Berücksichtigung des Ballradius („Aufblasen“ der Geraden)

Falls eine passende Gerade gefunden wurde, wird überprüft, ob sie auch mit dem Ballradius durch alle Tore passt. Hierzu wird die Gerade „aufgeblasen“, was bedeutet, dass sie auf den Durchmesser des Balls erweitert wird.

Für die „aufgeblasene“ Gerade werden vier mögliche Konfigurationen geprüft:

1. Eine Parallele rechts der ursprünglichen Geraden.
2. Eine Parallele links der ursprünglichen Geraden.
3. Eine diagonale Verbindung von der rechten oberen zur linken unteren Parallele.
4. Eine diagonale Verbindung von der linken oberen zur rechten unteren Parallele.

Falls eine dieser Konfigurationen alle Tore durchschneidet, ist die Lösung gültig.

3.3.1 Optimierungen und Abbruchbedingungen

Dynamischer Abbruch: Sobald eine gültige Lösung gefunden wurde, stoppt der Algorithmus.

3.3.2 Ausgabe der Lösung

Um Startpunkt und Richtung zu ermitteln muss die Linie in der Mitte von den beiden Lösungsparallelen berechnet werden. Der Startpunkt ist die Überkreuzung dieser Linie mit dem ersten Tor und die Richtung ist die Überkreuzung mit dem letzten Tor.

4 Beispielaufgaben

1. Aufgabe 1
   - There exists a solution for exercise 1:
   - The startpoint is: (11.729885477259762, 5.295973443292018)
   - And the direction is: (238.55306247854304, 120.7448958690949)
2. Aufgabe 2
   - There doesn't exist a solution for exercise 2
   - The Linear Ordering Test found misaligned goals.
3. Aufgabe 3
   - There doesn't exist a solution for exercise 3!
4. Aufgabe 4
   - There doesn't exist a solution for exercise 4!
5. Aufgabe 5
   - The startpoint is: (1308.4142482202403, 14671.90783474263)
   - And the direction is: (48892.86645053272, 2767.446634655332)

5 Umsetzung

5.1 Winkeltest

Die Umsetzung des Linear Ordering Tests dient dazu, schnell zu überprüfen, ob ein durchgehender Schuss durch alle Tore möglich ist, indem die Winkel zwischen benachbarten Toren analysiert werden. Die Funktion iteriert dabei durch die Liste der Tore und berechnet für jedes Tor zwei Winkel: einen am ersten und einen am zweiten Pfosten, jeweils mit einem Pfosten des nächsten Tors. Die Winkel werden so berechnet, dass sie immer gegen den Uhrzeigersinn beschrieben werden, um konsistente Vergleiche zu ermöglichen. Anschließend wird überprüft, ob beide Winkel größer oder kleiner als 180° sind, was auf eine vorwärts- oder rückwärtsgerichtete Bewegung hinweist. Beim ersten Auftreten wird die Bewegungsrichtung festgelegt; bei jeder weiteren Überprüfung muss diese Richtung beibehalten werden, ansonsten wird die Funktion abgebrochen und False zurückgegeben. Zeigen die Winkel in unterschiedliche Richtungen, kann keine Aussage getroffen werden, und die Richtung bleibt undefiniert. Am Ende gibt die Funktion True zurück, wenn keine widersprüchlichen Richtungsänderungen festgestellt wurden. Dieser Test ermöglicht eine schnelle Vorabentscheidung, ob eine detaillierte Prüfung des Schusspfads notwendig ist, was Zeit in der Berechnung spart.

```
def linear_ordering_test(list_goals: list[tuple[(float, float), (float, float)]]) -> bool:
    richtung = None

    for i, goal in enumerate(list_goals):
        if i == len(list_goals) - 1:
            break

        # fr ein Tor berechne ich den Winkel an dem ersten Pfosten von dem Tor und dem zweiten Pfosten von dem Tor
        # und irgendeinem Pfosten vom nchsten Tor und den Winkel an dem zweiten Pfosten von dem Tor
        # und dem ersten Pfosten von dem Tor und dem anderen Pfosten von dem nchsten Tor.
        a1 = goal[1]
        x1 = goal[0]
        b1 = list_goals[i + 1][0]

        a2 = goal[0]
        x2 = goal[1]
        b2 = list_goals[i + 1][1]

        # Zur Berechnung der Winkel muss der Winkel gegen den Uhrzeigersinn beschrieben werden.
        # Wenn der erste Punkt auf der y-Achse hher ist als der zweite Punkt desselben Tors mssen die Winkel so
        # beschrieben werden, dass
        if x1[1] > x2[1]:
            angle1 = calculate_angle(b1, x1, a1)
            angle2 = calculate_angle(a2, x2, b2)
        else:
            angle1 = calculate_angle(a1, x1, b1)
            angle2 = calculate_angle(b2, x2, a2)

        # wir erhalten zwei Winkel, die zwei Tore verbinden. Wenn diese beiden Innenwinkel grer als 180 sind, wird
        # beim ersten Mal die Richtung festgelegt, und bei jedem weiteren Tor berprft, ob die Richtung sich gendert
        # hat. Wenn die beiden Innenwinkel in jeweils andere Richtungen zeigen, bleibt die Richtung None.
        # Ab der ersten Richtungsbereinstimmung beider Innenwinkel muss jede weitere Richtungsbereinstimmung
        # der beiden Innenwinkel dieselbe Richtung vorgeben. Ansonsten ist der Pfad unmglich. Zeigen beide Innenwinkel
        # in jeweils unterschiedliche Richtungen ist unklar, ob der Schusspfad existiert und deswegen kann er hier nicht
        # verworfen werden.
        if angle1 > 180 and angle2 > 180:
            if richtung is None:
                richtung = "backwards"

            if richtung == "forwards":
                return False

        elif angle1 < 180 and angle2 < 180:
            if richtung is None:
                richtung = "forwards"

            if richtung == "backwards":
                return False

    return True
```

5.2 bruteforce_alg

Die Funktion bruteforce_alg führt einen Brute-Force-Algorithmus aus, um zu prüfen, ob ein Schuss mit einem Ballradius durch alle Tore möglich ist. Zunächst wird ein Set checked_lines erstellt, um doppelte Überprüfungen der gleichen Linien zu vermeiden. Ein Generator iterate_from_middle wird verwendet, um die Tore von der Mitte aus zu durchlaufen, wobei für jedes Tor jeder Pfosten mit jedem Pfosten der anderen Tore überprüft wird. Die Funktion berechnet die lineare Funktion (Steigung m und Achsenabschnitt b) zwischen den Pfosten, um eine Linie zu definieren. Es wird überprüft, ob diese Linie alle Tore schneidet. Wenn dies der Fall ist, wird die Funktion check_path aufgerufen, um zu überprüfen, ob der Ball den Schuss entlang der Linie ausführen kann, wobei der Ballradius berücksichtigt wird. Sobald eine gültige Lösung gefunden wird, gibt die Funktion die Lösung zurück und stoppt. Wenn keine Lösung gefunden wird, gibt die Funktion ein leeres Tupel zurück.

```
def bruteforce_alg(list_goals, ball_radius) -> tuple:
    checked_lines = set()

    def iterate_from_middle(list_goals):
        # generator function that returns the items of the list from the middle outwards
        if not list_goals:
            return

        middle = len(list_goals) // 2
        yield list_goals[middle]

        for shift in range(1, middle + 1):
            if middle - shift >= 0:
                yield list_goals[middle - shift]
            if middle + shift < len(list_goals):
                yield list_goals[middle + shift]

    # fr jeden pfosten in jedem tor jeden anderen pfosten in jedem anderen tor berprfen
    for goal in iterate_from_middle(list_goals):
        for post in goal:
            for second_goal in list_goals:
                for second_post in second_goal:
                    if post == second_post:
                        continue

                    # get the linear function of the line
                    m, b = linear_function(post, second_post)
                    if m is None and b is None:
                        continue
                    line_identifier = (m, b)

                    if line_identifier in checked_lines:
                        continue

                    checked_lines.add(line_identifier)
                    line_works = True

                    # Check if the line passes through all goals
                    for third_goal in list_goals:
                        is_intersection, _ = check_intersection(
                            (-1e6, m * (-1e6) + b),
                            (1e6, m * 1e6 + b),
                            third_goal[0],
                            third_goal[1]
                        )
                        if not is_intersection:
                            line_works = False
                            break

                    if line_works:
                        solution = check_path((post, second_post), ball_radius, list_goals)
                        if solution:
                            return solution # Stop after finding the first solution

    return tuple()
```

```
def iterate_from_middle(list_goals):
    # generator function that returns the items of the list from the middle outwards
    if not list_goals:
        return

    middle = len(list_goals) // 2
    yield list_goals[middle]

    for shift in range(1, middle + 1):
        if middle - shift >= 0:
            yield list_goals[middle - shift]
        if middle + shift < len(list_goals):
            yield list_goals[middle + shift]
```

5.3 iterate_from_middle

Die Funktion iterate_from_middle ist ein Generator, der eine Liste von Zielen von der Mitte aus durchläuft und die Elemente abwechselnd nach außen hin liefert. Ziel ist es, von der Mitte der Liste aus sowohl nach links als auch nach rechts zu iterieren. Dies kann nützlich sein, wenn du eine symmetrische Reihenfolge von Elementen durchgehen möchtest.

5.4 check_path

Die Funktion check_path überprüft, ob es möglich ist, eine Linie, die durch alle Tore verläuft, so zu erweitern, dass der Ball mit dem angegebenen Radius diese Linie entlang rollen kann. Zunächst werden vier mögliche Linien erzeugt: zwei parallele Linien zur ursprünglichen Linie (eine links und eine rechts) und zwei Linien, die durch Kombination der parallelen Linien und der Originallinie entstehen. Für jede dieser Linien wird überprüft, ob sie alle Tore schneidet.

Falls eine Linie durch alle Tore verläuft, wird eine Lösung ermittelt, indem der Mittelpunkt zwischen den Schnittpunkten der Linien mit dem ersten und letzten Tor berechnet wird. Das Ergebnis ist ein Startpunkt und eine Richtung für den Schuss. Die Funktion berücksichtigt alle Fälle, in denen der Ball entlang der Linie mit dem angegebenen Ballradius fließen könnte.

```
def check_path(line, ball_radius, list_goals) -> tuple:

    def get_goal_intersection(line, goal):
        # this function returns the intersection point of a line and a goal
        m, b = linear_function(line[0], line[1])

        is_intersection, intersection_point = check_intersection(
            (−1e6, m ∗ (−1e6) + b),
            (1e6, m ∗ 1e6 + b),
            goal[0],
            goal[1]
        )
        return intersection_point

    def midpoint(point1, point2):
        # this function returns the midpoint of a line
        x1, y1 = point1
        x2, y2 = point2
        return (x1 + x2) / 2, (y1 + y2) / 2

    def get_shot_solution(line1, line2, first_goal, last_goal):
        # this function returns a shot given two lines
        # the shot is the line that runs parallel and in the middle of the two other lines
        line1_first_goal = get_goal_intersection(line1, first_goal)
        line1_last_goal = get_goal_intersection(line1, last_goal)
        line2_first_goal = get_goal_intersection(line2, first_goal)
        line2_last_goal = get_goal_intersection(line2, last_goal)

        start_point = midpoint(line1_first_goal, line2_first_goal)
        direction = midpoint(line1_last_goal, line2_last_goal)

        return start_point, direction

    # this function checks if it's possible to expand the line that was found to the ball diameter
    diameter = ball_radius ∗ 2

    # list of all possible lines (4 cases)
    possible_lines = []
    # Case 1: parallel to the left
    parallel_left = LineString(line).offset_curve(diameter)
    possible_lines.append(parallel_left)

    # Case 2: parallel to the right
    parallel_right = LineString(line).offset_curve(−diameter)
    possible_lines.append(parallel_right)

    # Case 3:
    first_point = parallel_left.coords[0]
    last_point = parallel_right.coords[1]
    case3_line1 = LineString((first_point, line[1]))
    case3_line2 = LineString((line[0], last_point))
    possible_lines.append(case3_line1)
    possible_lines.append(case3_line2)

    # Case 4:
    first_point = parallel_right.coords[0]
    last_point = parallel_left.coords[1]
    case4_line1 = LineString((first_point, line[1]))
    case4_line2 = LineString((line[0], last_point))
    possible_lines.append(case4_line1)
    possible_lines.append(case4_line2)

    working_lines = []

    # loop through every possible line
    for possible_line in possible_lines:
        line_works = True

        # get the linear function of the possible line
        m, b = linear_function(possible_line.coords[0], possible_line.coords[1])

        # iterate over every goal to check if the possible line goes through the goal
        for goal in list_goals:
            is_intersection, _ = check_intersection(
                (−1e6, m ∗ (−1e6) + b),
                (1e6, m ∗ 1e6 + b),
                goal[0],
                goal[1]
            )
            if not is_intersection:
                line_works = False
                break
```

```
def get_goal_intersection(line, goal):
    # this function returns the intersection point of a line and a goal
    m, b = linear_function(line[0], line[1])

    is_intersection, intersection_point = check_intersection(
        (−1e6, m ∗ (−1e6) + b),
        (1e6, m ∗ 1e6 + b),
        goal[0],
        goal[1]
    )
    return intersection_point
```

5.5 get_goal_intersection

Die Funktion get_goal_intersection wurde so implementiert, dass sie den Schnittpunkt einer gegebenen Linie mit einem Tor (repräsentiert durch zwei Punkte) berechnet. Sie verwendet eine Kombination aus der Berechnung der linearen Gleichung der Linie und der Schnittpunktprüfung mit der check_intersection-Funktion.

```
def midpoint(point1, point2):
    # this function returns the midpoint of a line
    x1, y1 = point1
    x2, y2 = point2
    return (x1 + x2) / 2, (y1 + y2) / 2
```

5.6 midpoint

Die Funktion baut darauf auf indem der Durchschnitt der x- und y-Koordinaten der beiden Punkte gebildet wird. Dies bedeutet, dass der Mittelpunkt der Mittelwert der x-Koordinaten und der Mittelwert der y-Koordinaten der beiden gegebenen Punkte ist.

```
def get_shot_solution(line1, line2, first_goal, last_goal):
    # this function returns a shot given two lines
    # the shot is the line that runs parallel and in the middle of the two other lines
    line1_first_goal = get_goal_intersection(line1, first_goal)
    line1_last_goal = get_goal_intersection(line1, last_goal)
    line2_first_goal = get_goal_intersection(line2, first_goal)
    line2_last_goal = get_goal_intersection(line2, last_goal)

    start_point = midpoint(line1_first_goal, line2_first_goal)
    direction = midpoint(line1_last_goal, line2_last_goal)

    return start_point, direction
```

5.7 get_shot_solution

Die Funktion get_shot_solution berechnet eine Schusslösung, die in einem bestimmten geometrischen Kontext (z. B. bei einem Spiel wie Krocket oder einem ähnlichen Szenario) genutzt wird. Sie bestimmt den besten Startpunkt und die Richtung eines Schusses basierend auf zwei Linien, die durch Ziele oder Posten verlaufen.

5.8 linear_function

Berechnet die lineare Funktion $f(x) = mx + b$ anhand zweier Punkte (point1, point2). Gibt die Steigung m und den y-Achsenabschnitt b zurück. Bei einer vertikalen Linie, wo $x_1 = x_2$, wird (None, None) zurückgegeben.

5.9 check_intersection

Überprüft, ob sich zwei Liniensegmente, definiert durch die Punkte (p1, p2) und (q1, q2), schneiden. Gibt True und die Koordinaten des Schnittpunkts zurück, falls sie sich schneiden. Wenn es keinen Schnittpunkt gibt, wird False und (-1, -1) zurückgegeben.

5.10 calculate_angle

Berechnet den Winkel, der durch drei Punkte (a, x, b) gebildet wird, wobei x der Scheitelpunkt ist. Gibt den Winkel in Grad zurück, wobei die Werte im Bereich von 0 bis 360 grad liegen.