Aufgabe 4: Krocket

Team-ID: 00219

Team: Team

Bearbeiter/-innen dieser Aufgabe:
Lennart Frank

15. November 2024

Inhalt

Lösungsidee

Interpretation der Aufgabenstellung

Als erstes habe ich die Aufgabenstellung untersucht, um diese in greifbarere Ziele zu unterteilen, dabei habe ich die Annahme getroffen, dass die Tore als Strecken in einem zweidimensionalen Koordinatensystem dargestellt werden können, sowie dass der Ball auf einer weiteren Strecke rollt. Um dessen Radius darzustellen

habe ich den Durchmesser als Zwei um den Radius verschobene Strecken dargestellt.

Rechenansatz

Als Rechenansatz habe ich mir überlegt verschiedene Strecken mit einem Start und Endpunkt auf dem ersten und letzten Tor auszuprobieren und die Anzahl durchkreuzter Tore zu zählen. Dann nimmt man den Besten Wert davon und hat den Besten Schlag im von uns betrachteten Bereich.

Umsetzung

Python

Einlesen der Daten

Als erstes habe ich die Datei geöffnet, und alle Punkte in einen Array geschrieben. Wenn es das Beispiel Krocket 3 war, habe ich noch zusätzlich die Daten sortiert, da sonst mein Algorithmus nicht funktioniert. Außerdem habe ich den Radius zu einem Durchmesser umgerechnet und diese Daten zurückgegeben.

Berechnung des Vektors

Um den Vektor zu berechnen, Starte ich damit eine Geringe Anzahl and Start und Endpunkten basierend auf der ersten und letzten Strecke zu generieren. (Dies dient der Laufzeitoptimierung später mehr dazu)
Anschließend wird Brute Force mäßig für alle Start und Endpunkte kombiniert zu Zählen wie viele Tore gekreuzt werden.
Um dies zu Zählen erstelle ich mir basierend auf Start und Endpunkt zwei parallele Strecken, welche die Breite des Balles darstellen sollen. Diese erweitere ich dann noch, da durch die Verschiebung sonst die Strecken zu kurz sind, um die erste zu kreuzen.
Dann gehe ich einfach alle Tore durch und prüfe, ob die getestete Strecke durch das Tor kreuzen kann, indem ich ihre Orientation prüfe und zusätzlich noch prüfe, ob sich ihre Definitionsbereiche treffen.
Zuletzt muss ich nur noch Prüfen welcher der Vektoren die meisten Schnittpunkte hat. Wenn nun nicht alle Tore gekreuzt wurden, Teste ich mehr Punkte auf der Strecke, bis ich mein Maximum an Punkten erreicht habe. Durch diese Vorgehensweise kann ich die Laufzeit des Programms auf ca. 60 Sekunden für alle Beispiele Zusammen auf meinem Rechner reduzieren. Ohne diese Optimierung würde der Code ca. 93 Sekunden Brauchen.

Darstellung der Ergebnisse

Um die Ergebnisse Anschaulich darzustellen, erstelle ich mithilfe einer Grafikbibliothek .svg Dateien um die Tore sowie den Schlag darzustellen. Außerdem gebe ich noch den Start und Endpunkt sowie eine Gleichung an.

Rust

Optimierung

Da ich mit der Laufzeit des Programmes unzufrieden war, habe ich mich dazu entschieden mein Python Code als Prototypen zu nehmen und diesen in Rust nachzustellen. Dadurch konnte ich nicht nur die Laufzeit drastisch verbessern, sondern auch die Problematik lösen, dass die richtigen Bibliotheken sowie Python Version installiert sein müssen, um das Programm auszuführen.

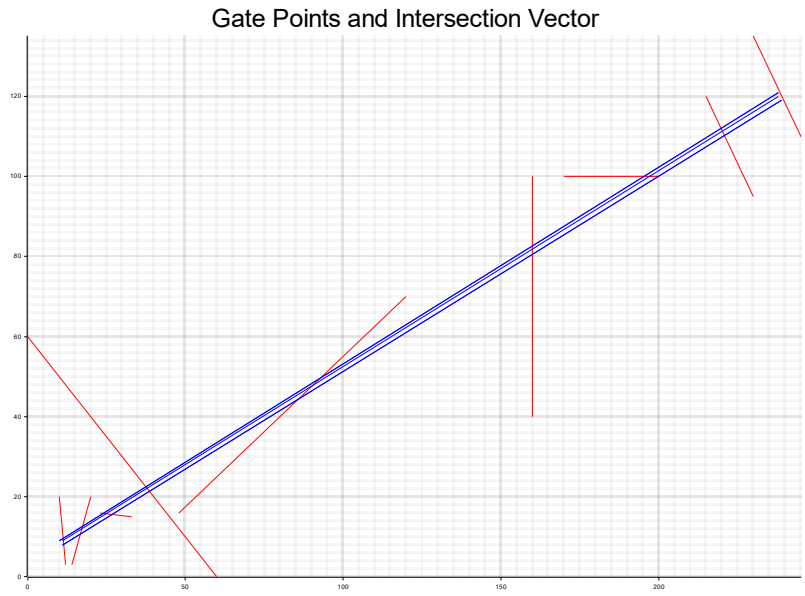
Ergebnisse der Optimierung

Durch die Implementierung in Rust konnte ich die Laufzeit der Berechnung aller Beispiele infolge von ca. 60 Sekunden auf ca. 800ms reduzieren.

Beispiele

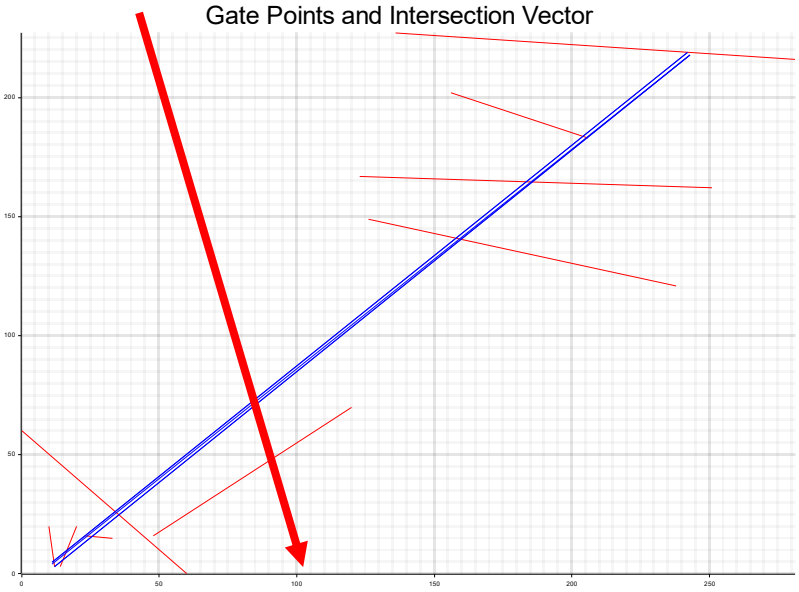
Beispiel 1

Start (11.285714285714286, 9.071428571428571),
End (238.57142857142858, 120.71428571428572)
Crossed Gates: 9 out of 9



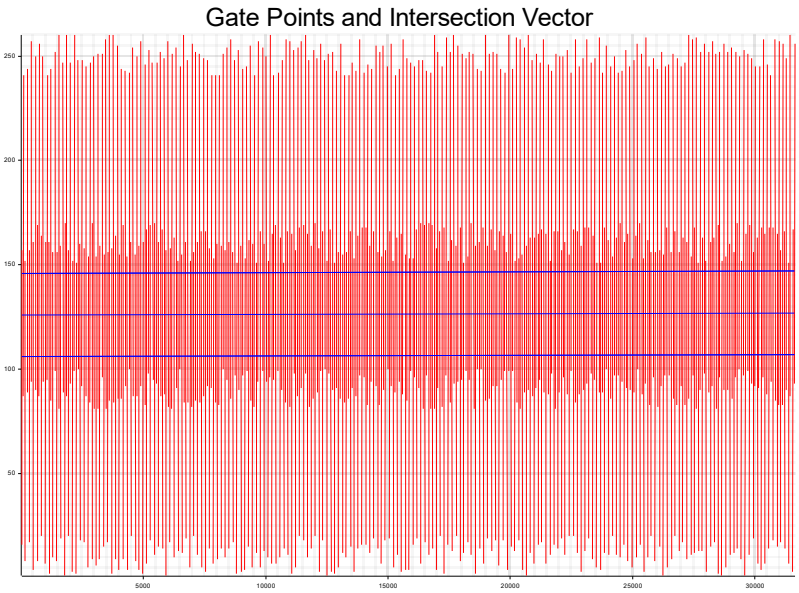
Beispiel 2

Start (11.850746268656717, 4.26865671641791),
End (243.12686567164178, 218.87313432835822)
Crossed Gates: 8 out of 9
⇨ Nicht mit einem (Geraden) Schlag möglich



Beispiel 3

Start (22.714285714285715, 126.78571428571429),
End (31628.5, 127.92857142857144)
Crossed Gates: 396 out of 396



Beispiel 4

Start (6.138297872340425, 181.84042553191466),
End (8946.531914893618, 4672.063829787234)
Crossed Gates: 344 out of 344
Graph kann des Platzes halber im Verzeichnis gefunden werden.

Beispiel 5

Start (1317.0, 14680.702127659575),
End (48839.85106382979, 2761.0638297872338)
Crossed Gates: 406 out of 406
Graph kann des Platzes halber im Verzeichnis gefunden werden.

Quellcode

Ich werde im Folgendem auf den Python Programmcode näher eingehen und am Ende auf einige Unterschiede in der Rust Implementierung, da in Rust der Code sehr ähnlich aufgebaut ist und an einen Mix aus Python und Dart erinnert, aber der Python code vermutlich für die meisten verständlicher ist.

Einzelne Beispiele nacheinander Testen:

```
def main():
    global_start_time = time.time() #Zeitmessung für Laufzeit starten
    file_paths = []

    file_paths_num = {}

    for file_path in file_paths: #Die Verschiedenen Beispiele Berechnen
        print(f"Processing file: {file_path}")
        gatePoints, width, gates2 = read_input(file_path)
        plot_r_gates(gates2)
        print(f"Width: {width}")
        start_time = time.time()
        intersection_vector, paralell_vectors, faulty_gates =
find_intersection_vector(gates2, width)
        end_time = time.time()

        print(f"Time taken to find the intersection vector: {end_time - start_time}
seconds")

        plot_intersection_vector(intersection_vector, width, gates2)
        plot_paralell_vector(paralell_vectors[0])
        plot_paralell_vector(paralell_vectors[1])
        plot_faulty_gates(faulty_gates)
        plt.savefig(f"Runde_1/Beispiele/A4_Krocket/krocket_plot_{file_paths_num.get(file_p
ath)}.svg", format='svg')
        plt.clf()
    global_end_time = time.time()
    print(f"Total time taken: {global_end_time - global_start_time} seconds")
```

Graphische Ausgabe:

```
def plot_r_gates(lines):
    for n in range(len(lines)):
        (x1, y1), (x2, y2) = lines[n]
        if n == 0 or n == len(lines) - 1: plt.plot([x1, x2], [y1, y2], color='red',
marker='o')
        else: plt.plot([x1, x2], [y1, y2], color='gray')
    . . .
def plot_intersection_vector(vector, width, gates):
    (x1, y1), (x2, y2) = vector
    plt.plot([x1, x2], [y1, y2], marker='x', color='red', linestyle='--',
label='Intersection Vector')
    . . .
    # Breite Anzeigen
    polygon_points = [ . . . ]
    polygon = plt.Polygon(polygon_points, color='blue', alpha=0.3, label='Width Area')
    plt.gca().add_patch(polygon)
    plt.legend()
```

Die Beste Strecke finden:

```
def find_intersection_vector(gates, width=1.0, setY = False):
    def extending_length(gates, width):
        . . .
        return length/3
    def extend_vector(start_point, end_point, length):
        . . .
        return extended_start, extended_end

    def count_crossed_gates(start_point, end_point, gates, width):
        def do_lines_intersect(p1, p2, q1, q2):
            def ccw(A, B, C):
                return (C[1] - A[1]) * (B[0] - A[0]) > (B[1] - A[1]) * (C[0] - A[0])
            return ccw(p1, q1, q2) != ccw(p2, q1, q2) and ccw(p1, p2, q1) != ccw(p1, p2,
q2)

        def is_point_on_line(px, py, x1, y1, x2, y2):
            return (x2 - x1) * (py - y1) == (y2 - y1) * (px - x1) and min(x1, x2) <= px <=
max(x1, x2) and min(y1, y2) <= py <= max(y1, y2)

def move_point_by_half_width(point, direction_vector, width):
            . . .
            return (point[0] + perp_dx, point[1] + perp_dy), (point[0] - perp_dx, point[1]
- perp_dy)

count = 0
        direction_vector = (end_point[0] - start_point[0], end_point[1] - start_point[1])
        start_point1, start_point2 = move_point_by_half_width(start_point,
direction_vector, width)
        end_point1, end_point2 = move_point_by_half_width(end_point, direction_vector,
width)
        start_point1, end_point1 = extend_vector(start_point1, end_point1,
extending_length(gates, width))
        start_point2, end_point2 = extend_vector(start_point2, end_point2,
extending_length(gates, width))
        faulty_gates = []
        for gate in gates:
            (x1, y1), (x2, y2) = gate
            if ((is_point_on_line(start_point1[0], start_point1[1], x1, y1, x2, y2) and
is_point_on_line(end_point1[0], end_point1[1], x1, y1, x2, y2) and
is_point_on_line(start_point2[0], start_point2[1], x1, y1, x2, y2) and
_point_on_line(end_point2[0], end_point2[1], x1, y1, x2, y2)) or
(do_lines_intersect(start_point1, end_point1, (x1, y1), (x2, y2)) and
_lines_intersect(start_point2, end_point2, (x1, y1), (x2, y2)))):
                count += 1
            else:
                faulty_gates.append(gate)
        return count, [start_point1, end_point1, start_point2, end_point2], faulty_gates
```

```
best_start_point = None
    best_end_point = None
    start_point1 = None
    end_point1 = None
    start_point2 = None
    end_point2 = None
    max_crossed_gates = 0
    faulty_gates = []

    def generate_points(gate, num_points):
        (x1, y1), (x2, y2) = gate
        return [(x1 + t * (x2 - x1), y1 + t * (y2 - y1)) for t in np.linspace(0, 1,
num_points)]

    num_points = 15
    max_points = 150
    all_gates_crossed = False

    while not all_gates_crossed and num_points <= max_points:
        (x1_start, y1_start), (x2_start, y2_start) = gates[0]
        (x1_end, y1_end), (x2_end, y2_end) = gates[-1]

        start_points = generate_points(gates[0], num_points)
        # print(start_points)
        end_points = generate_points(gates[-1], num_points)

        for start_point in start_points:
            for end_point in end_points:
                crossed_gates, points, faulty_gate = count_crossed_gates(start_point,
end_point, gates, width)
                if crossed_gates > max_crossed_gates:
                    max_crossed_gates = crossed_gates
                    best_start_point = start_point
                    best_end_point = end_point
                    start_point1, end_point1, start_point2, end_point2 = points
                    faulty_gates = faulty_gate
                    if max_crossed_gates == len(gates):
                        all_gates_crossed = True
                        break
            if all_gates_crossed:
                break

        if not all_gates_crossed:
            num_points += 20 # Mehr Punkte prüfen

return extend_vector(best_start_point, best_end_point, extending_length(gates, width)),
[[start_point1, end_point1], [start_point2, end_point2]], faulty_gates
```

Unterschiede in Rust sind unter anderem, dass ich teilweise Definieren muss welche Größe und Typ eine Variable haben wird, da der Code kompiliert wird und Rust versucht möglichst effizient zu sein, nicht nur in Laufzeit, sondern auch in Ressourcennutzung und Fehlervermeidung. Außerdem funktionieren return Statements ein wenig anders und es gibt eine Rudimentäre Typendefinition für Variablen (Unterscheidung zwischen Veränderbaren und Konstanten Variablen). Abgesehen davon halt noch Semikolons und ein paar andere Sachen, aber es steht interessierten natürlich frei die Implementierung sich selbst anzuschauen.