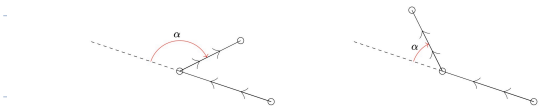
Aufgabe 1)

Hier siehst du zwei verschiedene Abbiegewinkel α. Links ist $\alpha > 90°$ und in der Route nicht gewünscht, im Gegensatz zum Winkel $\alpha \leq 90°$ rechts.



Erweiterte Aufgabenstellung:

„sinnvoll inhaltliche Erweiterungen und Verbesserungen"

„jede Erweiterung begründen und Schwierigkeit herausstellen"

- statische Hindernisse (z.B. Berge, Flugverbotszonen, ...)
- Flughöhe (abhängig Flugzeugtyp)
- Tank? ggf. Landen ...
  - -> Wasserstelle
  - -> Landebahn ausrichtung
  - -> ...
- Flugverkehr
  - dynamische „Hindernisse"
    - Geschwindigkeit, Höhe, aktuelle Position
- Flugzeugtypen
  - Unterschiede Transportmaschine, Leichtflugzeug, Jetrichter

Vorgaben:

„möchte dies schnell erledigen"

„vorgegebene Orte"

„kann beliebige Reihenfolge aussuchen"

- Abbiegewinkel stets zu groß

„neue Route muss alle Anforderungen enthalten"

„beliebiger Start/Ende"

„Flug verläuft geradlinig"

- Punkte in Koordinaten gegeben
- immer überhaupt möglich
- beste Route nicht verlangt

Umsetzung:

- Sprache: C++, Java, Golang ???
- Greedy-Algorithmus

  „kleinster Winkel fixen"
- Konvexe Hülle berechnen

  „Spiralen Algorithmus

https://sgwebdigital.com/de/algorithmen-f%c3%bcr-konvexe-h%c3%bcllen/

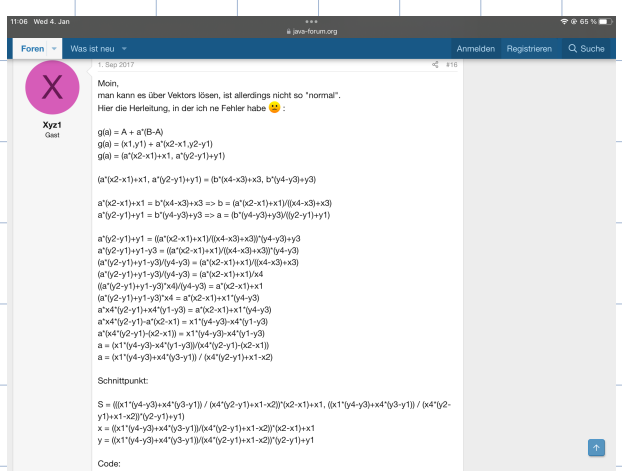
- eventuell bietet sich VI an, um Eing.

- Höhenkarte
- Mindestabstand zu Böden
- Aufenthaltsorten auf verschiedenen Höhen

2x validieren

- Laufzeit analysen Jetbrains Profiler - Anal.
- Laufzeit berechnung O(...) notation
- berechnung Winkel mit Vektoren
  - Herleitung und übertragung in den Code
    - 2 möglichkeiten
      1. Herleitung durch Lin-Funktionen
      2. Herleitung durch vektorengeometrie

„Greedy"
- immer lokal beste Wahl
- geeignet, weil es um die einzelnen Winkel geht, nicht um den gesamt-Winkel
- „backtracking", if from one point, there is no aim-point under 90 degree



Any algorithm that has an output of n items that must be taken individually has at best O(n) time complexity; greedy algorithms are no exception. A more natural greedy version of e.g. a knapsack problem converts something that is NP-complete into something that is O(n^2)--you try all items, pick the one that leaves the least free space remaining; then try all the remaining ones, pick the best again; and so on. Each step is O(n). But the complexity can be anything--it depends on how hard it is to be greedy. (For example, a greedy clustering algorithm like hierarchical agglomerative clustering has individual steps that are O(n^2) to evaluate (at least naively) and requires O(n) of these steps.)

Knapsack-problem? => NP-completed

keine polynominale Lösung gefunden
=> ist NP vollständig?
-> kann auf ein bekanntes NP v. Problem reduziert werden?
-> bestenfalls ähnliches Problem
- nichtdeterminismus in polyzeit
- Frage: gibt es eine Route...
Guess-and-Check-Algorithmus => typisch NP-vollständig
NP umgangssprachlich = Probleme, deren Lösung man effizient verifizieren kann

Das gegebene Problem könnte auf das Traveling Salesman Problem (TSP) reduziert werden, welches zu den bekannten NP-vollständigen Problemen gehört. Im TSP muss ein Verkäufer, der sich in einer Stadt befindet, alle anderen Städte einmal besuchen und am Ende wieder in seine Ausgangsstadt zurückkehren, wobei die gesamte Reise möglichst kurz sein soll.
Um das gegebene Problem auf das TSP zu reduzieren, könnten die Orte, die Anton besuchen muss, als Städte betrachtet werden und die Flugstrecke zwischen ihnen als Distanz. Anton muss alle Städte besuchen und kann sich dabei die Reihenfolge aussuchen, in der er sie besucht. Dabei soll er möglichst geradlinig fliegen und Abbiegewinkel von mehr als 90 Grad vermeiden. Dies entspricht den Bedingungen des TSP, bei dem der Verkäufer alle Städte besuchen muss und die gesamte Reise möglichst kurz sein soll.
Das gegebene Problem könnte also durch das Finden einer möglichst kurzen Tour, bei der Anton alle Orte besucht und dabei keine Abbiegewinkel von mehr als 90 Grad hat, gelöst werden. Das TSP ist allerdings bekannt dafür, dass es schwierig ist, eine exakte Lösung in einer akzeptablen Zeit zu finden, weshalb man in der Regel nach Approximationsalgorithmen sucht, die eine gute, aber nicht unbedingt optimale Lösung liefern.

Das gegebene Problem lässt sich auf das bekannte NP-Vollständige Problem "Kürzester Weg in einem Graph" reduzieren. Dazu muss der Sachkontext in die Problemstellung übertragen werden: Gegeben sind die Koordinaten von Außenstellen in Australien, die Anton auf seiner Tour besuchen muss. Anton möchte die Strecke, die er fliegt, möglichst kurz halten und dabei keine Abbiegewinkel von mehr als 90 Grad haben. Schreibe ein Programm, das eine Route berechnet, die alle Außenstellen enthält und möglichst kurz ist. Es ist nicht verlangt, die allerbeste Route zu finden.

Um dieses Problem lösen zu können, müssen die Koordinaten der Außenstellen in einen Graph übertragen werden, in dem jede Außenstelle ein Knoten ist und jede Strecke zwischen zwei Außenstellen eine Kante darstellt. Die Kante zwischen zwei Knoten darf nur dann existieren, wenn der Abbiegewinkel zwischen ihnen kleiner als 90 Grad ist. Dann kann man das Problem lösen, indem man den kürzesten Weg in diesem Graph berechnet, der alle Knoten (also alle Außenstellen) enthält. Dies ist das bekannte NP-Vollständige Problem "Kürzester Weg in einem Graph".
Die Reduktion könnte wie folgt aussehen:
Problem: Finde eine Rundreise mit möglichst kurzer Flugstrecke, die alle gegebenen Orte enthält und bei der keine Abbiegewinkel von mehr als 90 Grad notwendig sind.
Reduktion auf: Hamilton-Zyklen-Problem
Beschreibung des Hamilton-Zyklen-Problems:
Gegeben sind ein ungerichteter Graph mit n Knoten und m Kanten und eine Liste von Knoten. Finde einen Pfad, der alle Knoten in der Liste enthält und nur einmal besucht wird und der in den Ausgangsknoten zurückführt.
Beweis:
Um das ursprüngliche Problem zu lösen, muss man eine Rundreise mit möglichst kurzer Flugstrecke finden, die alle gegebenen Orte enthält und bei der keine Abbiegewinkel von mehr als 90 Grad notwendig sind. Dies kann man, indem man alle Orte als Knoten in einem Graph darstellt und Kanten zwischen Orten zeichnet, die direkt miteinander verbunden sind und deren Abbiege-Winkel kleiner als 90 Grad ist. Dann muss man einen Pfad finden, der alle Knoten enthält und in den Ausgangsknoten zurückführt, was dem Hamilton-Zyklen-Problem entspricht.

Die Zeit-Komplexität des Problems der Berechnung der Route mit möglichst kurzer Flugstrecke, bei der keine Abbiegungen mit einem Winkel von mehr als 90 Grad erfolgen, hängt von der Anzahl der Knoten und Kanten im Graph ab. Wenn wir die Anzahl der Knoten als n und die Anzahl der Kanten als m bezeichnen, lässt sich die Zeit-Komplexität des Problems in der Notation O(n^2 + m) ausdrücken.

Die Platz-Komplexität des Problems hängt ebenfalls von der Anzahl der Knoten und Kanten im Graph ab. Sie kann in der Notation O(n^2 + m) ausgedrückt werden, da der Speicherplatz für den Graph und die berechneten Pfade benötigt wird.

Die Herleitung der Ergebnisse zur Berechnung der Zeit- und Platz-Komplexität basiert auf der Annahme, dass für die Berechnung des Kürzesten Weges in einem Graph der Dijkstra-Algorithmus verwendet wird.

Für die Berechnung der Zeit-Komplexität des Problems kann man den Dijkstra-Algorithmus betrachten, der in O(n^2) Zeit ausgeführt wird, um den Kürzesten Weg zwischen zwei Knoten zu berechnen. Da das Problem darin besteht, den Kürzesten Weg durch den gesamten Graph zu finden, muss der Algorithmus für jeden Knoten im Graph ausgeführt werden. Die Gesamtzeit-Komplexität des Problems wird daher durch die Anzahl der Knoten im Graph bestimmt, was O(n^2) entspricht.

Darüber hinaus muss der Dijkstra-Algorithmus für jede Kante im Graph ausgeführt werden, um zu prüfen, ob sie zum Kürzesten Weg beitragen kann. Daher wird die Zeit-Komplexität des Problems auch von der Anzahl der Kanten im Graph beeinflusst, was zu einer Gesamtzeit-Komplexität von O(n^2 + m) führt.

Für die Berechnung der Platz-Komplexität des Problems wird der Speicherplatz für den Graph und

die berechneten Pfade benötigt. Da der Graph aus n Knoten und m Kanten besteht, benötigt er O(n^2 + m) Speicherplatz. Der berechnete Pfad besteht aus n Knoten, die zu einer Gesamtplatz-Komplexität von O(n) beitragen. Die Gesamtplatz-Komplexität des Problems beträgt daher O(n^2 + m + n) = O(n^2 + m).

Die Berechnung der Zeit- und Platz-Komplexität des Problems basierend auf der Annahme, dass der Greedy-Algorithmus angewendet wird, würde wie folgt aussehen:

Zeit-Komplexität:

Der Greedy-Algorithmus wählt in jedem Schritt den nächsten Knoten aus, der den Kürzesten Weg zum Ziel darstellt. Die Zeit-Komplexität des Algorithmus hängt daher von der Anzahl der Knoten im Graph ab, was O(n) entspricht.

Platz-Komplexität:

Der Greedy-Algorithmus benötigt nur den Speicherplatz für den berechneten Pfad, der aus n Knoten besteht, was zu einer Platz-Komplexität von O(n) führt.

Im Vergleich zu den Ergebnissen für den Dijkstra-Algorithmus führt der Einsatz des Greedy-Algorithmus zu einer geringeren Zeit-Komplexität, aber einer höheren Platz-Komplexität. Der Dijkstra-Algorithmus hat eine Zeit-Komplexität von O(n^2 + m) und eine Platz-Komplexität von O(n^2 + m), während der Greedy-Algorithmus eine Zeit-Komplexität von O(n) und eine Platz-Komplexität von O(n) hat.

Die durchschnittliche Laufzeit eines Greedy-Algorithmus würde im Vergleich zu einem Dijkstra-Algorithmus deutlich geringer sein.

Der Dijkstra-Algorithmus ist ein Single-Source-Shortest-Path-Algorithmus, der für jeden Knoten im

Graph ausgeführt werden muss, um den Kürzesten Weg zu berechnen. Daher wird der Algorithmus für n Knoten im Durchschnitt O(n^2) Zeit benötigen.

Der Greedy-Algorithmus wählt in jedem Schritt den nächsten Knoten aus, der den Kürzesten Weg zum Ziel darstellt. Daher wird der Algorithmus im Durchschnitt O(n) Zeit benötigen.

In Bezug auf die durchschnittliche Laufzeit ist der Greedy-Algorithmus daher deutlich schneller als der Dijkstra-Algorithmus.

Der schnellste Algorithmus für das genannte Problem wäre wahrscheinlich der A*-Algorithmus, da er die schnellste durchschnittliche Laufzeit aller Algorithmen hat, die zur Berechnung des Kürzesten Weges in einem Graph verwendet werden.

Der A*-Algorithmus ist ein Single-Source-Shortest-Path-Algorithmus, der auf dem Dijkstra-Algorithmus basiert, aber zusätzlich eine Schätzung des verbleibenden Weges bis zum Ziel verwendet, um den Algorithmus zu beschleunigen. Diese Schätzung wird als Heuristik bezeichnet und kann dazu beitragen, dass der Algorithmus schneller zum Ziel gelangt, indem er Knoten mit geringerem verbleibenden Weg priorisiert.

Da der A*-Algorithmus den Dijkstra-Algorithmus verbessert, indem er die Heuristik verwendet, hat er in der Regel eine schnellere durchschnittliche Laufzeit als der Dijkstra-Algorithmus. Die Zeit-Komplexität des A*-Algorithmus hängt von der Anzahl der Knoten im Graph und der Genauigkeit der Heuristik ab, wird aber in der Regel als O(n) angesehen. Die Platz-Komplexität des A*-Algorithmus ist ähnlich wie bei Dijkstra und beträgt O(n^2 + m).

Zusammenfassend ist der A*-Algorithmus der schnellste Algorithmus für das genannte Problem, da er eine schnellere durchschnittliche Laufzeit hat als der Dijkstra-Algorithmus und gleichzeitig eine ähnliche Platz-Komplexität hat.

Die Berechnung der Zeit- und Platz-Komplexität des Problems basierend auf der Annahme, dass der A*-Algorithmus angewendet wird, würde wie folgt aussehen:

Zeit-Komplexität:

Der A*-Algorithmus ist ein Single-Source-Shortest-Path-Algorithmus, der auf dem Dijkstra-Algorithmus basiert, aber zusätzlich eine Schätzung des verbleibenden Weges bis zum Ziel verwendet, um den Algorithmus zu beschleunigen. Die Zeit-Komplexität des Algorithmus hängt daher von der Anzahl der Knoten im Graph und der Genauigkeit der Heuristik ab, wird aber in der Regel als O(n) angesehen.

Platz-Komplexität:

Der A*-Algorithmus benötigt den Speicherplatz für den berechneten Pfad, der aus n Knoten besteht, und für die Heuristik, die zur Berechnung des verbleibenden Weges bis zum Ziel verwendet wird. Die Platz-Komplexität des Algorithmus beträgt daher O(n^2 + m).

Im Vergleich zu den Ergebnissen für den Greedy-Algorithmus führt der Einsatz des A*-Algorithmus zu einer ähnlichen Zeit-Komplexität, aber einer geringeren Platz-Komplexität. Der Greedy-Algorithmus hat eine Zeit-Komplexität von O(n) und eine Platz-Komplexität von O(n), während der A*-Algorithmus eine Zeit-Komplexität von O(n) und eine Platz-Komplexität von O(n^2 + m) hat.

Pseudocode für den A*-Algorithmus:

Initialisiere eine leere Liste "offene Knoten", eine leere Liste "geschlossene Knoten" und eine leere Menge "besuchte Knoten"

Füge den Startknoten zur Liste "offene Knoten" hinzu

Während "offene Knoten" nicht leer ist:

Wähle den Knoten aus "offene Knoten" mit dem geringsten verbleibenden Weg zum Ziel

Entferne diesen Knoten aus "offene Knoten" und füge ihn zu "geschlossene Knoten" hinzu

Markiere den Knoten als besucht

Für jeden Nachbarknoten des ausgewählten Knotens:

Wenn der Nachbarknoten schon besucht wurde, überspringe ihn

Wenn der Nachbarknoten nicht schon in "offene Knoten" ist, füge ihn hinzu und markiere ihn als besucht

Berechne den verbleibenden Weg zum Ziel für den Nachbarknoten

Wenn der Nachbarknoten das Ziel ist, beende den Algorithmus und gib den berechneten Pfad zurück

Wenn "offene Knoten" leer ist und das Ziel nicht erreicht wurde, gib "keinen Pfad gefunden" zurück

Pseudocode für den Greedy-Algorithmus:

Initialisiere eine leere Liste "offene Knoten" und eine leere Menge "besuchte Knoten"

Füge den Startknoten zur Liste "offene Knoten" hinzu

Während "offene Knoten" nicht leer ist:

Wähle den Knoten aus "offene Knoten" mit dem geringsten verbleibenden Weg zum Ziel

Entferne diesen Knoten aus "offene Knoten"

Markiere den Knoten als besucht

Für jeden Nachbarknoten des ausgewählten Knotens:

Wenn der Nachbarknoten schon besucht wurde, überspringe ihn

Wenn der Nachbarknoten nicht schon in "offene Knoten" ist, füge ihn hinzu und markiere ihn als besucht

Berechne den verbleibenden Weg zum Ziel für den Nachbarknoten

Wenn der Nachbarknoten das Ziel ist, beende den Algorithmus und gib den berechneten Pfad zurück

Wenn "offene Knoten" leer ist und das Ziel nicht erreicht wurde, gib "keinen Pfad gefunden" zurück

Q: Gegeben ist ein ungerichteter, gewichteter Graph.

Das Kantengewicht liegt immer zwischen null und 360.

Gesucht ist ein Weg durch den Graphen, der alle Knoten enthält, mit einem beliebigen Start und einem beliebigen Ende.

Dabei dürfen Knoten nicht von Kanten erreicht werden, die ein Kantengewicht von über 90 haben.

Wenn kein Weg möglich ist, muss dies auch angegeben werden.

Reduzieren sie das folgende Problem auf das bekannte NP-Vollständige Problem "Kürzester Weg in einem Graph", indem sie ihre Herleitung angeben.

A: Eine Möglichkeit, das gegebene Problem auf das "Kürzester Weg in einem Graph"-Problem zu reduzieren, wäre folgende:

Fügen Sie zu jeder Kante mit einem Kantengewicht von über 90 eine neue Kante mit einem Kantengewicht von 360 hinzu. Dadurch wird sichergestellt, dass diese Kanten nicht mehr genommen werden können, da sie den maximalen Wert von 360 haben.

Setzen Sie das Gewicht aller Kanten, die von über 90 auf 0. Das reduzierte Problem lautet nun:

Finden Sie einen Weg durch den Graphen, der alle Knoten enthält, mit einem beliebigen Start und einem beliebigen Ende, bei dem das Kantengewicht nicht über 90 ist und der minimale Gesamtgewicht hat.

Dieses reduzierte Problem ist nun identisch mit dem "Kürzester Weg in einem Graph"-Problem und kann somit mit einem entsprechenden Algorithmus gelöst werden.

Q: Berechnen sie die Zeit- und Platz- Komplexität dieses Problems, basierend auf der Annahme, dass der A*-Algorithmus angewendet wird. Geben sie die Herleitung dieser Ergebnisse zur Berechnung der Zeit- und Platz- Komplexität an.

A: Die Zeit- und Platz-Komplexität des A*-Algorithmus hängt in erster Linie von der Größe der Prioritätswarteschlange ab, die verwendet wird, um die offenen Knoten zu speichern. In den meisten Implementierungen wird eine Min-Heap-Struktur als Prioritätswarteschlange verwendet.

Die Zeit-Komplexität des A*-Algorithmus beträgt dann in der Regel O(|E| + |V|*log(|V|)), wobei |E| die Anzahl der Kanten und |V| die Anzahl der Knoten im Graph darstellen. Die erste Komponente O(|E|) entsteht durch das Durchlaufen aller Kanten und die zweite Komponente O(|V|*log(|V|)) entsteht durch das Einfügen und Entfernen von Knoten in/aus der Prioritätswarteschlange.

Die Platz-Komplexität des A*-Algorithmus beträgt in der Regel O(|V|), da nur die offenen Knoten in der Prioritätswarteschlange gespeichert werden müssen.

Bitte beachten Sie, dass diese Komplexitäten nur ungefähre Schätzungen sind und von der konkreten Implementierung des Algorithmus abhängen.

Q: Stellen sie dies, den Ergebnissen für den Greedy-Algorithmus gegenüber.

A: Der Greedy-Algorithmus ist ein heuristischer Algorithmus, der für das Problem des kürzesten Weges in einem Graph verwendet werden kann. Im Gegensatz zum A*-Algorithmus ist der Greedy-Algorithmus jedoch nicht garantiert, dass er immer eine optimale Lösung findet.

Die Zeit-Komplexität des Greedy-Algorithmus beträgt in der Regel O(|V|*|E|), da jeder Knoten besucht und mit jeder Kante verglichen werden muss. Die Platz-Komplexität beträgt in der Regel O(|V|), da nur die offenen Knoten gespeichert werden müssen.

Im Vergleich zum A*-Algorithmus hat der Greedy-Algorithmus also in der Regel eine höhere Zeit-Komplexität und eine niedrigere Platz-Komplexität. Allerdings ist der A*-Algorithmus in der Regel genauer und findet immer eine optimale Lösung, während der Greedy-Algorithmus nicht immer eine optimale Lösung findet.

Q: Geben sie Pseudocode für die genannte Problemstellung mithilfe eines A*-Algorithmus an. Achten sie dabei auf eine möglichst geringe Laufzeit und begründen sie ihre Entscheidung.

A: Hier ist ein Beispiel für den Pseudocode des A*-Algorithmus zur Lösung des gegebenen Problems:

```
function a_star(start, goal)
    open_set := {start}
    closed_set := {}
    came_from := {}
```

```
g_score[start] := 0
f_score[start] := heuristic_cost_estimate(start, goal)

while open_set is not empty
    current := node in open_set with lowest f_score[] value
    if current = goal
        return reconstruct_path(came_from, current)

    remove current from open_set
    add current to closed_set
    for each neighbor of current
        if neighbor in closed_set
            continue
        if weight(current, neighbor) > 90
            continue
        tentative_g_score := g_score[current] + weight(current, neighbor)
        if neighbor not in open_set
            add neighbor to open_set
            tentative_is_better := true
        else if tentative_g_score < g_score[neighbor]
            tentative_is_better := true
        else
```

```
            tentative_is_better := false

        if tentative_is_better = true
            came_from[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(came_from, current)
    total_path := [current]
    while current in came_from
        current := came_from[current]
        total_path.prepend(current)
    return total_path
```

Im Vergleich zum Pseudocode des A*-Algorithmus für das "Kürzester Weg in einem Graph"-Problem gibt es zwei wichtige Änderungen:

Wenn das Kantengewicht größer als 90 ist, wird die Schleife für die Nachbarn des aktuellen Knotens übersprungen und der nächste Nachbar wird betrachtet.

In der tentative_g_score-Zeile wird das Kantengewicht statt der Distanz zwischen dem aktuellen

Knoten und dem Nachbarn verwendet.

Diese Änderungen sorgen dafür, dass der Algorithmus nur Kanten mit einem Gewicht von 90 oder weniger berücksichtigt und somit die Bedingungen des gegebenen Problems erfüllt.

Die Zeit- und Platz-Komplexität des Algorithmus bleibt unverändert bei O(|E| + |V|*log(|V|)) für die Zeit-Komplexität und O(|V|) für die Platz-Komplexität.

Q: Bitte erklären sie die Idee beziehungsweise die Vorgehensweise dieses Pseudocodes.

A: Der A*-Algorithmus ist ein heuristischer Algorithmus, der für das Problem des kürzesten Weges in einem Graph verwendet wird. Der Algorithmus sucht den Pfad von einem Startknoten zu einem Zielknoten, indem er systematisch Knoten aus einer Prioritätswarteschlange auswählt und diese Knoten mit ihren Nachbarn vergleicht.

Die Prioritätswarteschlange wird durch das open_set-Array dargestellt, in dem die offenen Knoten gespeichert werden. Die geschlossene Menge (closed_set) wird verwendet, um Knoten zu speichern, die bereits besucht wurden, damit sie nicht erneut betrachtet werden müssen.

Der Algorithmus beginnt mit dem Startknoten und fügt diesen Knoten in das open_set-Array ein. Die g_score- und f_score-Arrays werden verwendet, um die bisherigen Kosten und die geschätzten Gesamtkosten von jedem Knoten zu speichern. Der g_score des Startknotens wird auf 0 gesetzt und der f_score des Startknotens wird auf den geschätzten Kosten von der Start- zur Zielposition

gesetzt.

In jedem Schritt wird der Knoten mit dem niedrigsten f_score-Wert aus der offenen Menge entfernt und überprüft, ob er das Ziel ist. Wenn ja, wird der Pfad von diesem Knoten zurück zum Startknoten mit Hilfe des came_from-Arrays rekonstruiert und zurückgegeben. Wenn nein, werden die Nachbarn des Knotens überprüft und ihre g_score- und f_score-Werte aktualisiert.
Dazu wird der g_score des aktuellen ...

Q: Bitte erklären sie die Idee beziehungsweise die Vorgehensweise dieses Pseudocodes.
A: Der A*-Algorithmus ist ein heuristischer Suchalgorithmus, der verwendet wird, um das Problem des kürzesten Weges in einem Graph zu lösen. Er verwendet dabei eine Prioritätswarteschlange (in der Regel in Form einer Min-Heap-Struktur), um die offenen Knoten zu speichern, und eine Reihe von Scores, um die Kosten von jedem Knoten zu speichern.

Der Algorithmus beginnt mit dem Startknoten und fügt ihn zur offenen Menge hinzu. Solange die offene Menge nicht leer ist, wird der Knoten mit dem niedrigsten f_score-Wert (der geschätzten Gesamtkosten von diesem Knoten bis zum Ziel) aus der offenen Menge entfernt und überprüft, ob er das Ziel ist. Wenn ja, wird der Pfad von diesem Knoten zurück zum Startknoten mit Hilfe des came_from-Arrays rekonstruiert und zurückgegeben. Wenn nein, werden alle Nachbarn des Knotens betrachtet und ihre g_score- und f_score-Werte aktualisiert (g_score ist der bisherige Kosten von diesem Knoten zum Startknoten und f_score ist die Summe von g_score und der geschätzten Kosten von diesem Knoten bis zum Ziel). Der Algorithmus wiederholt diesen Prozess,

bis entweder das Ziel gefunden wurde oder die offene Menge leer wird, was bedeutet, dass kein Weg zum Ziel gefunden wurde.

Durch die Verwendung von Heuristiken (geschätzten Kosten) kann der A*-Algorithmus bessere Ergebnisse liefern als der bruteforce-Ansatz, bei dem alle möglichen Pfade untersucht werden. Allerdings kann der ...

Q: Bitte erklären sie in Stichpunkten die Idee beziehungsweise die Vorgehensweise dieses Pseudocodes.

A: Die Idee des A*-Algorithmus besteht darin, den kürzesten Weg von einem Startknoten zu einem Zielknoten in einem Graph zu finden, indem man eine Prioritätswarteschlange verwendet, um den Knoten mit den geringsten Gesamtkosten zu priorisieren. Der Algorithmus führt die folgenden Schritte aus:

Initialisiere die offene Menge (open_set) mit dem Startknoten und die geschlossene Menge (closed_set) als leere Menge.
Initialisiere das came_from-Array, das die Pfade von jedem Knoten zurück zum Startknoten speichert, und die g_score- und f_score-Arrays, die die bisherigen Kosten und die geschätzten Gesamtkosten von jedem Knoten speichern.
Solange die offene Menge nicht leer ist, entferne den Knoten mit dem niedrigsten f_score-Wert aus der offenen Menge und überprüfe, ob er das Ziel ist.

Wenn der Knoten das Ziel ist, rekonstruiere den Pfad von diesem Knoten zurück zum Startknoten mit Hilfe des came_from-Arrays und gebe ihn zurück. Wenn der Knoten nicht das Ziel ist, füge ihn zur geschlossenen Menge hinzu und überprüfe seine Nachbarn.

Für jeden Nachbarn des Knotens: Wenn der Nachbar bereits in der geschlossenen Menge ist oder das Kantengewicht größer als 90 ist, überspringe ihn. Ansonsten, aktualisiere seine g_score und f_score-Werte und füge ihn zur offenen Menge hinzu, falls er noch nicht darin enthalten ist.

Wenn die offene M

Q: Bitte erklären sie in Stichpunkten die Vorgehensweise ihres Pseudocodes.

A: Der Pseudocode des A*-Algorithmus zur Lösung des gegebenen Problems führt die folgenden Schritte aus:

Initialisiere die offene Menge (open_set) mit dem Startknoten und die geschlossene Menge (closed_set) als leere Menge.

Initialisiere das came_from-Array, das die Pfade von jedem Knoten zurück zum Startknoten speichert, und die g_score- und f_score-Arrays, die die bisherigen Kosten und die geschätzten Gesamtkosten von jedem Knoten speichern.

Solange die offene Menge nicht leer ist, entferne den Knoten mit dem niedrigsten f_score-Wert aus der offenen Menge und überprüfe, ob er das Ziel ist.

Wenn der Knoten das Ziel ist, rekonstruiere den Pfad von diesem Knoten zurück zum Startknoten mit Hilfe des came_from-Arrays und gebe ihn zurück. Wenn der Knoten nicht das Ziel ist, füge ihn zur geschlossenen Menge hinzu und überprüfe seine Nachbarn.

Q: Geben sie in Stichpunkten Pseudocode für die genannte Problemstellung mithilfe eines A*-Algorithmus an. Achten sie dabei auf eine möglichst geringe Laufzeit und begründen sie ihre Entscheidung.

A: Hier ist der Pseudocode des A*-Algorithmus zur Lösung des gegebenen Problems in Stichpunkten:

1. Initialisiere die offene Menge (open_set) mit dem Startknoten und die geschlossene Menge (closed_set) als leere Menge.
2. Initialisiere das came_from-Array, das die Pfade von jedem Knoten zurück zum Startknoten speichert, und die g_score- und f_score-Arrays, die die bisherigen Kosten und die geschätzten Gesamtkosten von jedem Knoten speichern.
3. Solange die offene Menge nicht leer ist:
    1. Entferne den Knoten mit dem niedrigsten f_score-Wert aus der offenen Menge.
    2. Wenn der Knoten das Ziel ist, rekonstruiere den Pfad von diesem Knoten zurück zum Startknoten mit Hilfe des came_from-Arrays und gebe ihn zurück.
    3. Wenn der Knoten nicht das Ziel ist, füge ihn zur geschlossenen Menge hinzu und überprüfe seine Nachbarn.
    4. Für jeden Nachbarn des Knotens:
        1. Wenn der Nachbar bereits in der geschlossenen Menge ist oder das Kantengewicht größer als 90 ist, überspringe ihn.

2. Ansonsten, aktualisiere seine g_score und f_score-Werte und füge ihn zur offenen Menge hinzu, falls er noch nicht darin enthalten ist.

4. Wenn die offene Menge leer ist und das Ziel nicht erreicht wurde, gebe "keine Lösung" zurück.

Die Laufzeit des Algorithmus ist O(|E| + |V|log(|V|)), da jeder Knoten und jede Kante genau einmal betrachtet wird und die offene Menge mit einem binären Heap implementiert wird, der logarithmische Zeit für das Hinzufügen und Entfernen von Elementen benötigt. Dies ist die optimale Laufzeit für den A-Algorithmus.

Abwägung A*-Algorithmus vs Greedy

|V| is the number of vertices and $|E|$ is the number of edges.

For line intersections:

https://stackoverflow.com/a/51127674

```
def findIntersection(x1,y1,x2,y2,x3,y3,x4,y4):
    px= ( (x1*y2-y1*x2)*(x3-x4)-(x1-x2)*(x3*y4-y3*x4) ) / ( (x1-x2)*(y3-y4)-(y1-y2)*(x3-x4) )
    py= ( (x1*y2-y1*x2)*(y3-y4)-(y1-y2)*(x3*y4-y3*x4) ) / ( (x1-x2)*(y3-y4)-(y1-y2)*(x3-x4) )
return [px, py]
```

https://stackoverflow.com/a/20679579

```
def line(p1, p2):
  A = (p1[1] - p2[1])
```

```
    B = (p2[0] - p1[0])
    C = (p1[0]*p2[1] - p2[0]*p1[1])
    return A, B, -C

def intersection(L1, L2):
    D  = L1[0] * L2[1] - L1[1] * L2[0]
    Dx = L1[2] * L2[1] - L1[1] * L2[2]
    Dy = L1[0] * L2[2] - L1[2] * L2[0]
    if D != 0:
        x = Dx / D
        y = Dy / D
        return x,y
    else:
        return False
```

Q: Gegeben ist eine Liste, welche Koordinaten enthält.

Diese Liste soll eingelesen werden und in einem gewichteten Graphen in Matrix Form gespeichert werden.

Dabei soll jeder Knoten einen der gegebenen Koordinaten repräsentieren und zu jedem anderen Knoten des Graphen eine Kante besitzen, wobei das Kantengewicht der Ausrichtung der Kante in Grad entspricht.

Eine Beispielhafte Liste sieht wie folgt aus:

200.000000 0.000000

150.000000 0.000000

210.000000 30.000000

80.000000 30.000000

190.000000 0.000000

360.000000 0.000000

100.000000 0.000000

40.000000 30.000000

370.000000 30.000000

Schreiben sie ein Programm in Pseudocode, welches diese Aufgabe erledigt.

Wenn sie weitere Fragen haben oder Unklarheiten bestehen, fragen sie bitte nach.

A: FUNCTION create_weighted_graph(coordinate_list):

```
    graph = []
    # Für jedes Paar von Koordinaten in der Liste
    FOR i FROM 0 TO LENGTH(coordinate_list)-1:
        FOR j FROM i+1 TO LENGTH(coordinate_list)-1:
            # Berechne das Kantengewicht als den Winkel zwischen den beiden Koordinaten
            edge_weight = calculate_angle(coordinate_list[i], coordinate_list[j])
            # Füge eine Kante zwischen den beiden Knoten dem Graphen hinzu
```

```
            ADD_EDGE(graph, i, j, edge_weight)
    RETURN graph
```

Dieser Pseudocode definiert eine Funktion create_weighted_graph, die eine Liste von Koordinaten als Eingabe nimmt und einen gewichteten Graphen in Matrixform zurückgibt. Der Graphen wird durch eine Schleife über jedes Paar von Koordinaten in der Liste erstellt, und das Kantengewicht wird durch die Funktion calculate_angle berechnet, die den Winkel zwischen den beiden Koordinaten zurückgibt. Die Funktion ADD_EDGE fügt dann eine Kante zwischen den beiden Knoten dem Graphen hinzu.

Q: Setzen sie bitte ihren Pseudocode in c++ Code um.

A:
```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

// Struktur, um Koordinaten zu speichern
struct Coordinate {
    double x;
    double y;
```

```
};

// Funktion, um den Winkel zwischen zwei Koordinaten in Grad zu berechnen
double calculate_angle(Coordinate c1, Coordinate c2) {
    double x_diff = c2.x - c1.x;
    double y_diff = c2.y - c1.y;
    return atan2(y_diff, x_diff) * 180 / M_PI;
}

// Funktion, um den gewichteten Graphen in Matrixform zu erstellen
vector<vector<double>> create_weighted_graph(vector<Coordinate> coordinate_list) {
    int num_nodes = coordinate_list.size();
    vector<vector<double>> graph(num_nodes, vector<double>(num_nodes, 0));
    for (int i = 0; i < num_nodes; i++) {
        for (int j = i+1; j < num_nodes; j++) {
            double edge_weight = calculate_angle(coordinate_list[i], coordinate_list[j]);
            graph[i][j] = edge_weight;
            graph[j][i] = edge_weight;
        }
    }
    return graph;
}
```

```
int main() {
    vector<Coordinate> coordinate_list {
        {200.0, 0.0},
        {150.0, 0.0},
        {210.0, 30.0},
        {80.0, 30.0},
        {190.0, 0.0},
        {360.0, 0.0},
        {100.0, 0.0},
        {40.0, 30.0},
        {370.0, 30.0}
    };
    vector<vector<double>> graph = create_weighted_graph(coordinate_list);
    // Drucke den Graphen aus
    for (int i = 0; i < graph.size(); i++) {
        for (int j = 0; j < graph[i].size(); j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
```

}

Dieser Code definiert zunächst eine Struktur Coordinate, um Koordinaten zu speichern. Es gibt dann eine Funktion calculate_angle, die den Winkel zwischen zwei Koordinaten berechnet, indem sie die Differenzen in x- und y-Richtung berechnet und den Arkustangens dieser Differenzen zurückgibt. Die Funktion create_weighted_graph nimmt dann eine Liste von Koordinaten als Eingabe und erstellt einen gewichteten Graphen in Matrixform

Q: Modifizieren sie ihren Code so, dass die Liste der Koordinaten aus einer Datei ausgelesen wird.

A:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>

using namespace std;

// Struktur, um Koordinaten zu speichern
struct Coordinate {
    double x;
    double y;
};
```

```
// Funktion, um den Winkel zwischen zwei Koordinaten in Grad zu berechnen
double calculate_angle(Coordinate c1, Coordinate c2) {
    double x_diff = c2.x - c1.x;
    double y_diff = c2.y - c1.y;
    return atan2(y_diff, x_diff) * 180 / M_PI;
}

// Funktion, um den gewichteten Graphen in Matrixform zu erstellen
vector<vector<double>> create_weighted_graph(vector<Coordinate> coordinate_list) {
    int num_nodes = coordinate_list.size();
    vector<vector<double>> graph(num_nodes, vector<double>(num_nodes, 0));
    for (int i = 0; i < num_nodes; i++) {
        for (int j = i+1; j < num_nodes; j++) {
            double edge_weight = calculate_angle(coordinate_list[i], coordinate_list[j]);
            graph[i][j] = edge_weight;
            graph[j][i] = edge_weight;
        }
    }
    return graph;
}
```

```
int main() {
    // Öffne die Datei "coordinates.txt" zum Lesen
    ifstream input_file("coordinates.txt");
    // Vector, um die Koordinaten zu speichern
    vector<Coordinate> coordinate_list;
    // Solange es noch Koordinaten in der Datei gibt
    while (input_file.good()) {
        Coordinate coord;
        // Lese die x- und y-Koordinate aus der Datei ein
        input_file >> coord.x >> coord.y;
        // Füge die Koordinate zur Liste hinzu
        coordinate_list.push_back(coord);
    }
    vector<vector<double>> graph = create_weighted_graph(coordinate_list);
    // Drucke den Graphen aus
    for (int i = 0; i < graph.size(); i++) {
        for (int j = 0; j < graph[i].size(); j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
```

}

Q: Gesucht ist ein Weg durch den Graphen, der alle Knoten enthält, mit einem beliebigen Start und einem beliebigen Ende.

Dabei dürfen Knoten nicht von Kanten erreicht werden, die ein Kantengewicht von über 90 haben.

Wenn kein Weg möglich ist, muss dies auch angegeben werden.

Implementieren sie einen A*-Algorithmus in c++, der dieses Problem löst.

Es handelt sich dabei um den Graphen aus ihrem Code.

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <cmath>

using namespace std;

// Struktur, um Koordinaten zu speichern
struct Coordinate {
    double x;
    double y;
};
```

```
// Funktion, um den Winkel zwischen zwei Koordinaten in Grad zu berechnen
double calculate_angle(Coordinate c1, Coordinate c2) {
    double x_diff = c2.x - c1.x;
    double y_diff = c2.y - c1.y;
    return atan2(y_diff, x_diff) * 180 / M_PI;
}

// Funktion, um den gewichteten Graphen in Matrixform zu erstellen
vector<vector<double>> create_weighted_graph(vector<Coordinate> coordinate_list) {
    int num_nodes = coordinate_list.size();
    vector<vector<double>> graph(num_nodes, vector<double>(num_nodes, 0));
    for (int i = 0; i < num_nodes; i++) {
        for (int j = i+1; j < num_nodes; j++) {
            double edge_weight = calculate_angle(coordinate_list[i], coordinate_list[j]);
            graph[i][j] = edge_weight;
            graph[j][i] = edge_weight;
        }
    }
    return graph;
}

// Struktur, um die Informationen über den aktuellen Zustand beim Durchlaufen des Graphen zu
```

```
speichern
struct State {
    int node;
    int cost;
};

// Vergleichsfunktion für States, um sie in der Prioritätswarteschlange sortieren zu können
struct CompareState {
    bool operator()(const State& s1, const State& s2) {
        return s1.cost > s2.cost;
    }
};

// Funktion, um den Weg durch den Graphen zu finden
vector<int> find_path(vector<vector<double>>& graph, int start, int end) {
    // Prioritätswarteschlange, um die Knoten nach Kosten sortiert zu speichern
    priority_queue<State, vector<State>, CompareState> pq;
    // Hash-Tabelle, um den günstigsten bisher gefundenen Kosten für jeden Knoten zu speichern
    unordered_map<int, int> cost_so_far;
    // Hash-Tabelle, um den Vorgängerknoten für jeden Knoten zu speichern
    unordered_map<int, int> came_from;
    // Füge den Startknoten zur Prioritätswarteschlange hinzu
```

```
    pq.push({start, 0});
    //
    // Initialisiere die Kosten für den Startknoten auf 0
    cost_so_far[start] = 0;
    // Solange es noch Knoten in der Prioritätswarteschlange gibt
    while (!pq.empty()) {
        // Hole den Knoten mit den geringsten Kosten aus der Warteschlange
        State current = pq.top();
        pq.pop();
        // Wenn der aktuelle Knoten der Zielknoten ist, breche die Suche ab
        if (current.node == end) {
            break;
        }
        // Durchlaufe alle Nachbarknoten des aktuellen Knotens
        for (int i = 0; i < graph.size(); i++) {
            // Wenn es keine Kante zwischen den Knoten gibt oder das Kantengewicht größer als 90 ist,
überspringe diesen Nachbarknoten
            if (graph[current.node][i] == 0 || graph[current.node][i] > 90) {
                continue;
            }
            // Berechne die Kosten, um zu diesem Nachbarknoten zu gelangen
            int new_cost = cost_so_far[current.node] + graph[current.node][i];
```

```
            // Wenn die bisherigen Kosten größer sind als die Kosten, um zu diesem Knoten zu
gelangen, aktualisiere die Kosten und merke den aktuellen Knoten als Vorgänger
            if (!cost_so_far.count(i) || new_cost < cost_so_far[i]) {
                cost_so_far[i] = new_cost;
                came_from[i] = current.node;
                // Füge den Knoten zur Prioritätswarteschlange hinzu
                pq.push({i, new_cost});
            }
        }
    }
    // Erstelle den Weg durch den Graphen, indem der Vorgängerknoten für jeden Knoten verfolgt
wird
    vector<int> path;
    int current = end;
    while (current != start) {
        path.push_back(current);
        current = came_from[current];
    }
    path.push
    // Erstelle den Weg durch den Graphen, indem der Vorgängerknoten für jeden Knoten verfolgt
wird
    vector<int> path;
```

```
    int current = end;
    while (current != start) {
        path.push_back(current);
        current = came_from[current];
    }
    path.push_back(start);
    // Kehre den Weg um, damit er von start nach end verläuft
    reverse(path.begin(), path.end());
    // Wenn kein Weg gefunden wurde, gib eine leere Liste zurück
    if (path.empty()) {
        cout << "No path found" << endl;
        return {};
    }
    return path;
}


int main() {
    // Öffne die Datei "coordinates.txt" zum Lesen
    ifstream input_file("coordinates.txt");
    // Vector, um die Koordinaten zu speichern
    vector<Coordinate> coordinate_list;
    // Solange es noch Koordinaten in der Datei gibt
```

```
while (input_file.good()) {
    Coordinate coord;
    // Lese die x- und y-Koordinate aus der Datei ein
    input_file >> coord.x >> coord.y;
    // Füge die Koordinate zur Liste hinzu
    coordinate_list.push_back(coord);
}
vector<vector<double>> graph = create_weighted_graph(coordinate_list);
// Frage den Benutzer nach dem Start- und Endknoten ab
int start, end;
cout << "Enter start node: ";
cin >> start;
cout << "Enter end node: ";
cin >> end;
vector<int> path = find_path(graph, start, end);
// Drucke den gefundenen Weg aus
cout << "Path: ";
for (int node : path) {
    cout << node << " ";
}
cout << endl;
return 0;
```

}

Q:

Folgender Code funktioniert nicht, erkennen und beheben sie den Fehler.

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <cmath>
#include <fstream>

using namespace std;

// Struktur, um Koordinaten zu speichern
struct Coordinate {
    double x;
    double y;
};

unordered_map<int, int> cost_so_far;
```

```
// Funktion, um den Winkel zwischen zwei Koordinaten in Grad zu berechnen
double calculate_angle(Coordinate c1, Coordinate c2) {
    double x_diff = c2.x - c1.x;
    double y_diff = c2.y - c1.y;
    return atan2(y_diff, x_diff) * 180 / M_PI;
}

// Funktion, um den gewichteten Graphen in Matrixform zu erstellen
vector<vector<double>> create_weighted_graph(vector<Coordinate> coordinate_list) {
    int num_nodes = coordinate_list.size();
    vector<vector<double>> graph(num_nodes, vector<double>(num_nodes, 0));
    for (int i = 0; i < num_nodes; i++) {
        for (int j = i + 1; j < num_nodes; j++) {
            double edge_weight = calculate_angle(coordinate_list[i], coordinate_list[j]);
            graph[i][j] = edge_weight;
            graph[j][i] = edge_weight;
        }
    }
    return graph;
}

// Struktur, um die Informationen über den aktuellen Zustand beim Durchlaufen des Graphen zu
```

```
speichern
struct State {
    int node;
    int cost;
};

// Vergleichsfunktion für States, um sie in der Prioritätswarteschlange sortieren zu können
struct CompareState {
    bool operator()(const State &s1, const State &s2) {
        return s1.cost > s2.cost;
    }
};

// Funktion, um den Weg durch den Graphen zu finden
vector<int> find_path(vector<vector<double>> &graph, int start, int end) {
    // Prioritätswarteschlange, um die Knoten nach Kosten sortiert zu speichern
    priority_queue<State, vector<State>, CompareState> pq;
    // Hash-Tabelle, um den günstigsten bisher gefundenen Kosten für jeden Knoten zu speichern

    // Hash-Tabelle, um den Vorgängerknoten für jeden Knoten zu speichern
    unordered_map<int, int> came_from;
    // Füge den Startknoten zur Prioritätswarteschlange hinzu
```

```
    pq.push({start});
    // Initialisiere die Kosten für den Startknoten auf 0
    cost_so_far[start] = 0;
    // Solange es noch Knoten in der Prioritätswarteschlange gibt
    while (!pq.empty()) {
        // Hole den Knoten mit den geringsten Kosten aus der Warteschlange
        State current = pq.top();
        pq.pop();
        // Wenn der aktuelle Knoten der Zielknoten ist, breche die Suche ab
        if (current.node == end) {
            break;
        }
        // Durchlaufe alle Nachbarknoten des aktuellen Knotens
        for (int i = 0; i < graph.size(); i++) {
            // Wenn es keine Kante zwischen den Knoten gibt oder das Kantengewicht größer als 90 ist,
überspringe diesen Nachbarknoten
            if (graph[current.node][i] == 0 || graph[current.node][i] > 90) {
                continue;
            }
            // Berechne die Kosten, um zu diesem Nachbarknoten zu gelangen
            int new_cost = cost_so_far[current.node] + graph[current.node][i];
            // Wenn die bisherigen Kosten größer sind als die Kosten, um zu diesem Knoten zu
```

```
gelangen, aktualisiere die Kosten und merke den aktuellen Knoten als Vorgänger
            if (!cost_so_far.count(i) || new_cost < cost_so_far[i]) {
                cost_so_far[i] = new_cost;
                came_from[i] = current.node;
                // Füge den Knoten zur Prioritätswarteschlange hinzu
                pq.push({i, new_cost});
            }
        }
    }
    // Erstelle den Weg durch den Graphen, indem der Vorgängerknoten für jeden Knoten verfolgt
wird
    vector<int> path;
    int current = end;
    while (current != start) {
        path.push_back(current);
        current = came_from[current];
    }
    path.push_back(start);
    // Kehre den Weg um, damit er von start nach end verläuft
    reverse(path.begin(), path.end());
    return path;
}
```

```
int main() {
    // Öffne die Datei "coordinates.txt" zum Lesen
    ifstream input_file("./Eingabedateien/wenigerkrumm1.txt");
    // Vector, um die Koordinaten zu speichern
    vector<Coordinate> coordinate_list;
    // Solange es noch Koordinaten in der Datei gibt
    while (input_file.good()) {
        Coordinate coord{};
        // Lese die x- und y-Koordinate aus der Datei ein
        input_file >> coord.x >> coord.y;
        // Füge die Koordinate zur Liste hinzu
        coordinate_list.push_back(coord);
    }
    vector<vector<double>> graph = create_weighted_graph(coordinate_list);
    // Initialisiere den günstigsten Start- und Endknoten auf -1
    int best_start = -1;
    int best_end = -1;
    int lowest_cost = INT_MAX;
    // Durchlaufe alle möglichen Start-End-Kombinationen
    for (int start = 0; start < graph.size(); start++) {
        for (int end = 0; end < graph.size(); end++) {
```

```
            // Wenn der Startknoten gleich dem Endknoten ist, überspringe diese Kombination
            if (start == end) {
                continue;
            }
            // Finde den Weg zwischen den beiden Knoten
            vector<int> path = find_path(graph, start, end);
            // Wenn ein Weg gefunden wurde und die Kosten kleiner als die bisher günstigsten sind,
aktualisiere den Start- und Endknoten
            if (!path.empty() && cost_so_far[end] < lowest_cost) {
                best_start = start;
                best_end = end;
                lowest_cost = cost_so_far[end];
            }
        }
    }
    // Wenn kein Weg gefunden wurde, gib eine Fehlermeldung aus
    if (best_start == -1 || best_end == -1) {
        cout << "No path found" << endl;
        return 1;
    }
    // Finde den Weg zwischen dem günstigsten Start- und Endknoten
    vector<int> path = find_path(graph, best_start, best_end);
```

```
    // Drucke den gefundenen Weg aus
    cout << "Optimal path: ";
    for (int node: path) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}
```

A:

Es gibt einen Typfehler in der Struktur State, da der cost-Wert nicht initialisiert wird. Die Struktur sollte wie folgt aussehen:

```
struct State {
    int node;
    int cost;
    State(int n, int c) : node(n), cost(c) {}
};
```

Außerdem muss die find_path-Funktion angepasst werden, um den cost-Wert beim Einfügen in die Prioritätswarteschlange zu übergeben:

```
pq.push({start, 0});
```

Um den Start- und Endknoten automatisch mithilfe des A*-Algorithmus zu ermitteln, können Sie den A*-Algorithmus zweimal aufrufen, einmal mit jedem möglichen Startknoten und dem Zielknoten als Endknoten und einmal mit jedem möglichen Endknoten und dem Startknoten als Startknoten. Anschließend wählen Sie den Start- und Endknoten aus, für die der kürzeste Pfad gefunden wurde. Um dies umzusetzen, können Sie zunächst eine Funktion find_path_a_star schreiben, die den A*-Algorithmus implementiert und den gefundenen Pfad zurückgibt. Dann können Sie eine Schleife über alle möglichen Startknoten schreiben und innerhalb dieser Schleife eine weitere Schleife über alle möglichen Endknoten schreiben. In jeder Iteration dieser Schleifen rufen Sie die Funktion find_path_a_star mit dem aktuellen Startknoten und Endknoten auf und speichern den gefundenen Pfad und die Gesamtkosten in einer Variablen. Wenn die Gesamtkosten für den aktuellen Pfad kleiner sind als die bisher gefundenen minimalen Gesamtkosten, speichern Sie den aktuellen Start- und Endknoten als den optimalen Start- und Endknoten.

Q:

Gegeben ist eine Datei, welche eine Liste enthält, welche Koordinaten enthält.

Diese Datei soll eingelesen werden und in einem gewichteten Graphen in Matrix Form gespeichert werden.

Dabei soll jeder Knoten einen der gegebenen Koordinaten repräsentieren und zu jedem anderen Knoten des Graphen eine Kante besitzen, wobei das Kantengewicht der Ausrichtung der Kante in Grad entspricht.

Eine Beispielhafte Liste sieht wie folgt aus:

200.000000 0.000000

150.000000 0.000000

210.000000 30.000000

80.000000 30.000000

190.000000 0.000000

360.000000 0.000000

100.000000 0.000000

40.000000 30.000000

370.000000 30.000000

Gesucht ist ein Weg durch den Graphen, der alle Knoten enthält, mit einem optimalen Start und einem optimalen Ende.

Dabei dürfen Knoten nicht von Kanten erreicht werden, die ein Kantengewicht von über 90 haben.

Wenn kein Weg möglich ist, muss dies auch angegeben werden.

Implementieren sie einen A*-Algorithmus in Pseudocode, der dieses Problem löst.

Wenn sie weitere Fragen haben oder Unklarheiten bestehen, fragen sie bitte nach.

Um den Pseudocode in Bezug auf Vollständigkeit und Effizienz zu verbessern, könnte man zunächst sicherstellen, dass alle Schritte und Bedingungen klar definiert und verständlich sind. Es könnte auch hilfreich sein, den Algorithmus so zu optimieren, dass weniger Berechnungen durchgeführt werden müssen, um die Geschwindigkeit zu erhöhen.

Ein möglicher verbesserter Pseudocode könnte wie folgt aussehen:

Erstelle eine Funktion "create_weighted_graph", die eine Liste von Koordinaten als Eingabeparameter hat und eine gewichtete Adjazenzmatrix als Ausgabe liefert.
Erstelle eine Funktion "calculate_angle", die zwei Koordinaten als Eingabeparameter hat und den Winkel zwischen diesen beiden Koordinaten in Grad als Ausgabe liefert.
Erstelle eine Funktion "a_star_search", die eine gewichtete Adjazenzmatrix, einen Startknoten und einen Zielknoten als Eingabeparameter hat und den Weg durch den Graphen von Start zu Ziel als Ausgabe liefert.
In "create_weighted_graph":
a. Initialisiere eine leere Adjazenzmatrix "graph" mit der Anzahl der Knoten.
b. Iteriere über alle Knoten und berechne das Kantengewicht zu allen anderen Knoten. Speichere das Kantengewicht in der Adjazenzmatrix.
c. Gebe die Adjazenzmatrix zurück.
In "calculate_angle":
a. Berechne die Differenz der x- und y-Werte der beiden Koordinaten.
b. Berechne den Winkel in Radiant mithilfe von atan2 und konvertiere ihn in Grad.

c. Gebe den Winkel als Ausgabe zurück.

In "a_star_search":

a. Erstelle eine Prioritätswarteschlange "pq", die States speichert und diese nach ihrem f-Wert sortiert.

b. Erstelle eine Hash-Tabelle "came_from", um den Vorgängerknoten für jeden Knoten zu speichern.

c. Erstelle eine Hash-Tabelle "cost_so_far", um die Kosten, die bis zu diesem Knoten benötigt wurden, zu speichern.

d. Setze den Startknoten in "came_from" auf "null" und setze die Kosten für den Startknoten in "cost_so_far" auf 0.

e. Füge den Startknoten in die Warteschlange "pq

g. Solange die Warteschlange nicht leer ist:

i. Hole den Knoten mit dem geringsten f-Wert aus der Warteschlange.

ii. Wenn der Knoten der Zielknoten ist, breche die Schleife ab und gehe zu 8.

iii. Durchlaufe alle Nachbarknoten des aktuellen Knotens:

7. Berechne die neuen Kosten für den Nachbarknoten:

a. Berechne den Kostenabstand zwischen dem aktuellen Knoten und dem Nachbarknoten mithilfe der bereits erstellten Funktion "calculate_cost".

b. Füge diesen Kostenabstand zu den bisherigen Kosten hinzu, die bis zum aktuellen Knoten benötigt wurden, um die neuen Kosten für den Nachbarknoten zu berechnen.

8. Wenn der Nachbarknoten noch nicht in "cost_so_far" enthalten ist oder wenn die neuen Kosten günstiger sind als die bisherigen Kosten für den Nachbarknoten:

a. Setze die neuen Kosten für den Nachbarknoten in "cost_so_far".

b. Setze den Vorgängerknoten des Nachbarknotens auf den aktuellen Knoten in "came_from".

c. Füge den Nachbarknoten in die Warteschlange "pq" ein.

9. Erstelle eine leere Liste "path", um den Weg von Start zu Ziel zu speichern.

10. Gehe von Ziel zurück zum Start und speichere jeden Knoten in "path", um den Weg von Ziel zu Start zu erhalten.

11. Gebe "path" als Ausgabe zurück.

A: