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.

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 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.

ie 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.

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.

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.

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".

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.

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 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.

Type to enter text

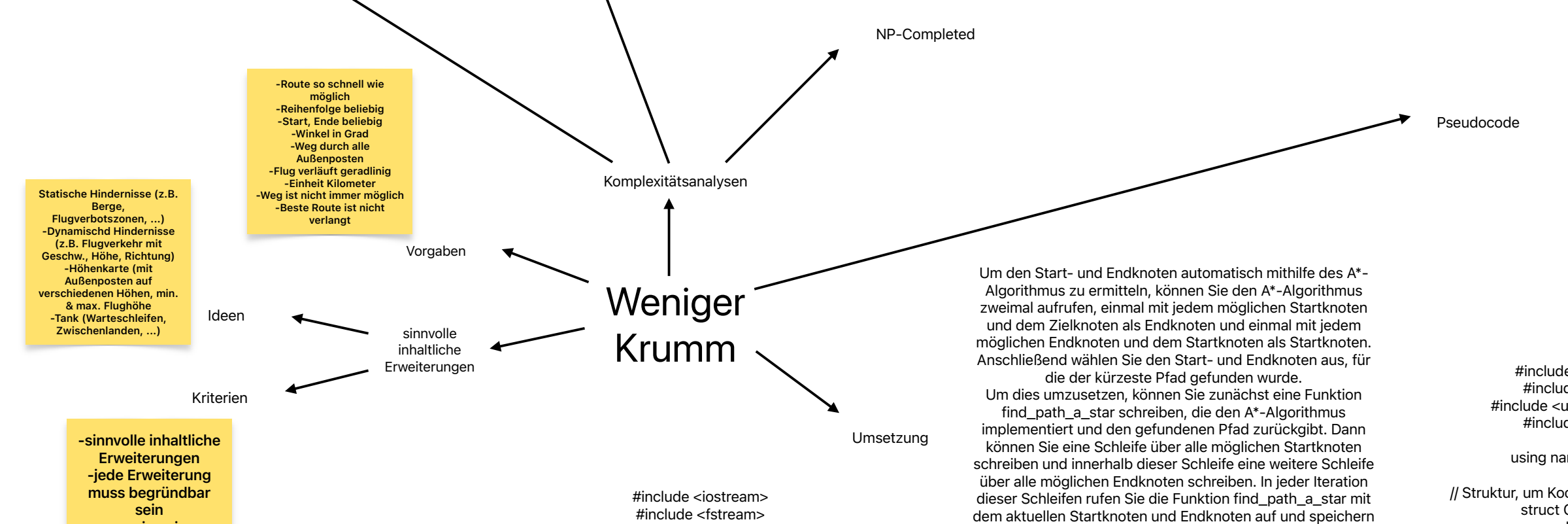
Space-Analysis

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.

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.

Type to enter text

Time-Analysis



NP-Completed

Pseudocode

Komplexitätsanalysen

-Route so schnell wie möglich
-Reihenfolge beliebig
-Start, Ende beliebig
-Winkel in Grad
-Weg durch alle Außenposten
-Flug verläuft geradlinig
-Einheit Kilometer
-Weg ist nicht immer möglich
-Beste Route ist nicht verlangt

Vorgaben

Statische Hindernisse (z.B. Berge, Flugverbotszonen, ...)
-Dynamischd Hindernisse (z.B. Flugverkehr mit Geschw., Höhe, Richtung)
-Höhenkarte (mit Außenposten auf verschiedenen Höhen, min. & max. Flughöhe
-Tank (Warteschleifen, Zwischenlanden, ...)

Ideen

sinnvolle inhaltliche Erweiterungen

Kriterien

-sinnvolle inhaltliche Erweiterungen
-jede Erweiterung muss begründbar sein
-muss eine eigene Schwierigkeit darstellen

Weniger Krumm

Umsetzung

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.

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

struct Node {
double x;
double y;
std::vector<std::pair<Node*, double>> edges;
};

double distance(Node* a, Node* b) {
double dx = a->x - b->x;
double dy = a->y - b->y;
return std::sqrt(dx * dx + dy * dy);
}

int main() {
std::vector<Node> nodes;

// Lies die Datei ein
std::ifstream file("coordinates.txt");
std::string line;
while (std::getline(file, line)) {
std::istringstream iss(line);
Node node;
if (!(iss >> node.x >> node.y)) {
break; // Error beim Parsen der Koordinaten
}
nodes.push_back(node);
}

// Erstelle den Graphen
for (int i = 0; i < nodes.size(); i++) {
for (int j = i + 1; j < nodes.size(); j++) {
double distance = distance(&nodes[i], &nodes[j]);
nodes[i].edges.emplace_back(&nodes[j], distance);
nodes[j].edges.emplace_back(&nodes[i], distance);
}
}

return 0;
}
```

Dieser Code liest die Liste aus einer Datei mit dem Namen "coordinates.txt" ein und speichert jede Zeile als einen Knoten im Graphen. Der Abstand zwischen den Knoten wird dann berechnet und als Gewicht der Kante zwischen ihnen gespeichert.
Der Winkel zwischen zwei Kanten an einem Knoten kann wie folgt berechnet werden:

```
double angle(Node* a, Node* b, Node* c) {
double dx1 = b->x - a->x;
double dy1 = b->y - a->y;
double dx2 = c->x - a->x;
double dy2 = c->y - a->y;
double dot = dx1 * dx2 + dy1 * dy2;
double det = dx1 * dy2 - dx2 * dy1;
return std::atan2(det, dot);
}
```

Sie können diese Funktion aufrufen, um den Winkel zwischen zwei Kanten zu berechnen. Der Winkel wird in Radian zurückgegeben.
Sie können die angle-Funktion verwenden, um den Winkel zwischen zwei Kanten an einem Knoten zu berechnen. Um dies zu tun, müssen Sie jede Kante an dem Knoten durchlaufen und den Winkel zu allen anderen Kanten berechnen.

Hier ist ein Beispiel dafür, wie Sie dies in Ihrem Code tun könnten:

```
for (auto& node : nodes) {
for (auto& edge : node.edges) {
Node* other = edge.first;
double angleSum = 0;
for (auto& otherEdge : other->edges) {
if (otherEdge.first != &node) {
angleSum += angle(&node, other, otherEdge.first);
}
}
edge.second = angleSum;
}
}
```

Dieser Code durchläuft jeden Knoten und jede Kante und berechnet den Winkel zwischen der aktuellen Kante und allen anderen Kanten an dem Knoten, an dem die Kante endet. Das Ergebnis wird dann als Gewicht der Kante gespeichert.

```
#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;
}
```

```
double calculate_angle(pair<double, double> node1,
pair<double, double> node2) {
double x1 = node1.first;
double y1 = node1.second;
double x2 = node2.first;
double y2 = node2.second;
double angle = atan2(y2 - y1, x2 - x1);
return angle;
}
```

Diese Funktion berechnet den Winkel in Radian und gibt ihn als double zurück. Wenn Sie den Winkel in Grad zurückgeben möchten, können Sie ihn einfach mit 180/M_PI (M_PI ist eine Konstante für pi in C++) multiplizieren.

```
#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

```
#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;
}
```

```
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
```

```
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
```

```
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.
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 ..

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 ...

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.

```
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.

```
DEFINIERE Kantengewicht als Funktion(Knoten1, Knoten2):
BERECHNE Abstand zwischen Knoten1 und Knoten2
GEBE Abstand zurück
DEFINIERE Graph als leeres Dictionary
ÖFFNE Datei "coordinates.txt" ZUM Lesen
FÜR jede Zeile IN Datei:
TEILE Zeile in x-Koordinate und y-Koordinate auf
KONVERTIERE x- und y-Koordinate in Gleitkommazahlen
FÜGE Knoten MIT x- und y-Koordinate dem Graph HINZU
FÜR jeden Knoten1 im Graph:
FÜR jeden Knoten2 im Graph:
WENN Knoten1 NICHT gleich Knoten2:
FÜGE Kante ZWISCHEN Knoten1 und Knoten2 MIT Kantengewicht(Knoten1, Knoten2) DEM Graph HINZU
SPEICHERE Winkel ZWISCHEN Knoten1, Knoten2 UND jedem anderen Knoten, der mit Knoten1 verbunden ist, in Graph
```

Dieser Algorithmus liest die Koordinaten aus der Datei ein und erstellt für jedes Paar von Knoten eine Kante mit dem entsprechenden Kantengewicht. Der Winkel zu den anderen Kanten wird ebenfalls gespeichert.

```
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

```
Pseudocode:
def read_coordinate_list(list):
graph = {}
for line in list:
# Split the line into x and y coordinate
x, y = line.split()
x = float(x)
y = float(y)
node = (x, y)
# Initialize empty list of edges for this node
graph[node] = []
# Calculate distances and angles to all other nodes
for other_node in graph:
distance = calculate_distance(node, other_node)
angle = calculate_angle(node, other_node)
# Add an edge to the list of edges for this node
graph[node].append((other_node, distance, angle))
return graph
def calculate_distance(node1, node2):
x1, y1 = node1
x2, y2 = node2
distance = sqrt((x1 - x2)**2 + (y1 - y2)**2)
return distance
def calculate_angle(node1, node2):
x1, y1 = node1
x2, y2 = node2
angle = atan2(y2 - y1, x2 - x1)
return angle
```

Diese Funktion read_coordinate_list nimmt eine Liste von Koordinaten als Eingabe und gibt einen Graphen zurück, der als Dictionary implementiert ist. Jeder Knoten im Graphen wird durch ein Tupel aus x- und y-Koordinate repräsentiert und jeder Eintrag im Dictionary enthält eine Liste von Kanten, die von diesem Knoten ausgehen. Jede Kante besteht aus einem Zielknoten, der Entfernung zwischen den Knoten und dem Winkel, den die Kante zu den anderen Kanten des Knotens hat. Die Funktionen calculate_distance und calculate_angle berechnen entsprechend die Distanz und den Winkel zwischen zwei Knoten.

```
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.
```

```
function A*(graph, start, end):
open_list = []
closed_list = []
current_node = start
g_score = 0
h_score = heuristic(start, end)
f_score = g_score + h_score

open_list.add(current_node, f_score)

while open_list is not empty:
current_node = get node with lowest f_score from open_list
if current_node == end:
return path from start to end

open_list.remove(current_node)
closed_list.add(current_node)

for neighbor in neighbors of current_node:
if neighbor in closed_list:
continue

tentative_g_score = g_score + distance between current_node and neighbor
if neighbor not in open_list:
open_list.add(neighbor)
elseif tentative_g_score >= g_score of neighbor:
continue

update g_score, h_score, and f_score for neighbor
set current_node as predecessor of neighbor

function heuristic(start, end):
return euclidean distance between start and end
```

Der A*-Algorithmus verwendet eine Heuristik, um den optimalen Weg durch den Graph zu finden. In diesem Fall habe ich die euklidische Distanz zwischen dem Start- und Endknoten als Heuristik verwendet, aber andere Heuristiken können auch verwendet werden.

```
function create_graph(coordinate_list):
graph = {}

for i in range(len(coordinate_list)):
graph[i] = []
for j in range(len(coordinate_list)):
if i != j:
x1, y1 = coordinate_list[i]
x2, y2 = coordinate_list[j]
angle = calculate_angle(x1, y1, x2, y2)
if angle <= 90:
graph[i].append((j, angle))

return graph

function calculate_angle(x1, y1, x2, y2):
dx = x2 - x1
dy = y2 - y1
return atan2(dy, dx) * 180 / pi
```

```
function read_and_store_coordinates(coordinate_list):
graph = []
# Iteriere über jede Koordinate in der Liste
for i in range(len(coordinate_list)):
# Splitte die Koordinaten in x und y
x, y = coordinate_list[i].split(" ")
# Wandle x und y in Fließkommazahlen um
x = float(x)
y = float(y)
# Füge die Koordinate als Knoten zum Graphen hinzu
graph.append((x, y))
# Iteriere über jeden Knoten im Graphen
for i in range(len(graph)):
edges = []
# Iteriere über jeden anderen Knoten im Graphen
for j in range(len(graph)):
if i == j:
continue
# Berechne den Winkel zwischen dem aktuellen Knoten und dem anderen Knoten
angle = calculate_angle(graph[i], graph[j])
# Füge eine Kante von dem aktuellen Knoten zu dem anderen Knoten hinzu
edges.append((j, angle))
# Speichere die Kanten für den aktuellen Knoten
graph[i] = (graph[i], edges)
return graph

function calculate_angle(node1, node2):
# Berechne den Winkel zwischen den Knoten node1 und node2
# Hier müssen noch weitere Details implementiert werden, wie z.B. die Berechnung
# des Winkels aus den x- und y-Koordinaten der Knoten
angle = 0
return angle
```

```
DEFINIERE Knoten als STRUKTUR
x_coord: REAL
y_coord: REAL
KANTEN: ARRAY von STRUKTUR
ziel_knoten: Knoten
gewicht: REAL
winkel: REAL
ENDE STRUKTUR

DEFINIERE Graph als STRUKTUR
knoten: ARRAY von Knoten
ENDE STRUKTUR

FUNKTION erstelle_knoten(x_coord: REAL, y_coord: REAL) -> Knoten:
ERSTELLE neuen_knoten: Knoten
SETZE neuen_knoten.x_coord = x_coord
SETZE neuen_knoten.y_coord = y_coord
SETZE neuen_knoten.kanten = LEERE ARRAY
GIB neuen_knoten ZURÜCK
ENDE FUNKTION

FUNKTION berechne_abstand(knoten1: Knoten, knoten2: Knoten) -> REAL:
BERECHNE x_diff: REAL = knoten1.x_coord - knoten2.x_coord
BERECHNE y_diff: REAL = knoten1.y_coord - knoten2.y_coord
BERECHNE abstand: REAL = WURZEL(x_diff^2 + y_diff^2)
GIB abstand ZURÜCK
ENDE FUNKTION

FUNKTION berechne_winkel(knoten1: Knoten, knoten2: Knoten) -> REAL:
BERECHNE x_diff: REAL = knoten2.x_coord - knoten1.x_coord
BERECHNE y_diff: REAL = knoten2.y_coord - knoten1.y_coord
BERECHNE winkel: REAL = TAN(y_diff / x_diff)
GIB winkel ZURÜCK
ENDE FUNKTION

FUNKTION erstelle_kante(von_knoten: Knoten, zu_knoten: Knoten) -> KANTE:
BERECHNE gewicht: REAL = berechne_abstand(von_knoten, zu_knoten)
BERECHNE winkel: REAL = berechne_winkel(von_knoten, zu_knoten)
ERSTELLE neue_kante: KANTE
SETZE neue_kante.ziel_knoten = zu_knoten
SETZE neue_kante.gewicht = gewicht
SETZE neue_kante.winkel = winkel
GIB neue_kante ZURÜCK
ENDE FUNKTION

PROGRAMM:
ERSTELLE graphen: Graph
SETZE graphen.knoten = LEERE ARRAY
FÜR JEDE zeile IN liste:
TEILE zeile IN x_coord, y_coord AUF
ERSTELLE neuer_knoten: Knoten = erstelle_knoten(x_coord, y_coord)
FÜGE neuer_knoten ZU graphen.knoten HINZU
FÜR JEDE knoten IN graphen.knoten:
FÜR JEDE anderer_knoten IN graphen.knoten:
WENN anderer_knoten NICHT gleich knoten:
ERSTELLE neue_kante: KANTE = erstelle_kante(knoten, anderer_knoten)
FÜGE neue_kante ZU knoten.kanten HINZU
GIB graphen ZURÜCK
ENDE PROGRAMM
```

Dieser Pseudocode liest jede Zeile der Eingabeliste ein und erstellt daraus einen Knoten. Dann werden alle möglichen Kanten zwischen den Knoten erstellt und zu den entsprechenden Knoten hinzugefügt. Schließlich wird der vollständige Graphen zurückgegeben.