

Aufgabe 1)

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



Erweiterte Aufgabenstellung:

„sinnvoll inhaltliche Erweiterungen und Verbesserungen“

„jede Erweiterung begründen und Sicherheit herausstellen“

- statische Hindernisse (z.B. Berge, Flugverbotszonen, ...)

- Flughöhe (abhängig Flugzeugtyp)

- Tank? ggf. Landen ...

→ Warteschleife

→ Landebahn ausrichtung

→ ...

- Flugverkehr

- dynamische „Hindernisse“

- Geschwindigkeit, Höhe, aktuelle Position

- Flugzeugtypen

- Unterschiede Transportmaschine, Leichtflugzeug, Motorboot

Vorgaben:

„möchte dies schnell erledigen“

„vorgegebene Orte“

„kann Reihenfolge aussuchen“

- Abbiegewinkel drils zu groß

„neue Route muss alle Anhaltstellen enthalten“

„beliebiger Start/Ende“

„Flug verläuft geradlinig“

- Punkte in Koordinaten gegeben

- immer überhaupt möglich

- beste Route nicht verlangt

Umsetzung:

- Sprache: C++, Java, C#, ...

- Greedy-Algorithmus

„kleinster Winkel first“

- Konvexe Hülle berechnen

„Spiralen Algorithmus“

<https://sgwebdigital.com/de/algorithmen-f%C3%BCr-konvexe-h%C3%BCllen/>

- eventuell bietet sich UI an, um Erg.

- Höhenkarte
- Mindestabstand zu Boden
- Außenposten auf verschiedenen Höhen

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

2v validieren

- Laufzeit analysen Jobbrain Profiler - Anal.
- Laufzeit berechnung $O(\dots)$ notation
- berechnung Winkel mit Vektoren
- Herleitung und Übertragung in den Code
- 2 Möglichkeiten
 1. Herleitung durch Lin-Funktionen
 2. Herleitung durch Vektorgeometrie

The first screenshot shows a forum post by user 'Xyz1' asking for help with finding the intersection of two lines defined by points A, B and C, D. The post includes the equations of the lines and asks for the intersection point S.

The second screenshot shows a reply by user 'DerWissende' explaining that the equations need to be solved for the intersection point S, and that the equations are not in the correct form for solving.

The third screenshot shows the code for the intersection point calculation, which uses the equations of the lines to find the intersection point S.

```

1. Sep 2017
Xyz1
Gast

Moin,
man kann es über Vektoren lösen, ist allerdings nicht so "normal".
Hier die Herleitung, in der ich ne Fehler habe :
:

g(a) = A + a*(B-A)
g(a) = (x1,y1) + a*(x2-x1,y2-y1)
g(a) = (a*(x2-x1)+x1, a*(y2-y1)+y1)

(a*(x2-x1)+x1, a*(y2-y1)+y1) = (b*(x4-x3)+x3, b*(y4-y3)+y3)
(a*(x2-x1)+x1 = b*(x4-x3)+x3 >> b = (a*(x2-x1)+x1)/(x4-x3)+x3
a*(y2-y1)+y1 = b*(y4-y3)+y3 >> a = (b*(y4-y3)+y3)/(y2-y1)+y1

a*(y2-y1)+y1 = ((a*(x2-x1)+x1)/(x4-x3)+x3)/(y4-y3)+y3
a*(y2-y1)+y1-y3 = ((a*(x2-x1)+x1)/(x4-x3)+x3)/(y4-y3)+y3-y3
(a*(y2-y1)+y1-y3)/(y4-y3) = ((a*(x2-x1)+x1)/(x4-x3)+x3)/(y4-y3)
(a*(y2-y1)+y1-y3)/(y4-y3) = (a*(x2-x1)+x1)/(x4-x3)+x3/(y4-y3)
(a*(y2-y1)+y1-y3)/(y4-y3) = a*(x2-x1)/(x4-x3)+x3/(y4-y3)
(a*(y2-y1)+y1-y3)/(y4-y3) = a*(x2-x1)/(x4-x3)+x3/(y4-y3)
a*(y2-y1)+y1-y3 = a*(x2-x1)/(x4-x3)+x3/(y4-y3)
a*(y2-y1)+y1-y3 = a*(x2-x1)/(x4-x3)+x3/(y4-y3)
a = ((x1*(y4-y3)+x4*(y3-y1))/(x4*(y2-y1)+x1-x2))/(y2-y1)+y1

Schnittpunkt:
S = ((x1*(y4-y3)+x4*(y3-y1))/(x4*(y2-y1)+x1-x2))/(y2-y1)+x1, ((x1*(y4-y3)+x4*(y3-y1))/(x4*(y2-y1)+x1-x2))/(y2-y1)+y1

Code:
11:08 - Web 4. Jan
Foren - Was ist neu - Anmelden Registrieren Suche
Xyz1
Gast

1. Sep 2017
DerWissende hat gesagt.:
Man muss a noch in die erste Gleichung einsetzen, nach b bringen und alles in g einsetzen -
Das ist blödsinn.
DerWissende hat gesagt.:
11:08 - Web 4. Jan
Foren - Was ist neu - Anmelden Registrieren Suche
y1+x1-x2)/(y2-y1)+y1
x = (x1*(y4-y3)+x4*(y3-y1))/(x4*(y2-y1)+x1-x2)/(y2-y1)+x1
y = (x1*(y4-y3)+x4*(y3-y1))/(x4*(y2-y1)+x1-x2)/(y2-y1)+y1

Code:
1
import java.awt.geom.Point2D;
2
3
4 // author
5
6 public class GeradenSchnittpunkt {
7
8     public static void main(String[] args) {
9         System.out.println(berechneSchnittpunkt(
10             1.5, 4.5, 4.5, 3.5,
11             1.5, 1.5, 4.5, 4.5));
12
13         // Ausgabe: Point2D.Double[3.0, 3.0]
14         //
15         System.out.println(berechneSchnittpunkt(
16             2.5, 3.5, 4.5, 1.5,
17             3, 1, 5, 3));
18
19         // Ausgabe: Point2D.Double[4.0, 2.0]
20         //
21     }
22
23     private static Point2D.Double berechneSchnittpunkt(
24         double x1, double y1, double x2, double y2,
25         double x3, double y3, double x4, double y4) {
26         return new Point2D.Double(
27             (x1 * (y3 - y4) + x3 * (y4 - y1) + x4 * (y1 - y3)) / ((x1
28             * (y3 - y4) + x3 * (y4 - y1) + x4 * (y1 - y3)) / ((x1
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341

```

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? \Rightarrow NP-Completed

Keine polynomielle Lösung gefunden

\Rightarrow ist NP vollständig?

\rightarrow Kann auf ein bekanntes NP-v. Problem reduziert werden?

\rightarrow bestenfalls ähnliches Problem

- nichtdeterminismus in polyzeit

- Frage: Gibt es eine Route...

knapsack- und Traveling-Salesman-Algorithmus \Rightarrow 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 $O(|E| + |V| \cdot \log(|V|))$, 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 $O(|V| \cdot \log(|V|))$ 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 $O(|E| + |V| \cdot \log(|V|))$ 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: