Aufgabe 2
„Dreiecksbeziehungen“-
Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Mathematische Präzisierung der Aufgabenstellung

Bei der Eingabe handelt es sich um eine Menge $D = \{d_1, ..., d_n\}$ von Dreiecken d_i. Jedes Dreieck ist dabei durch seine drei Eckpunkte vollständig definiert ($d_i = \{p_1, p_2, p_3\}$). Ein Eckpunkt ist dabei wiederum ein Punkt $p_i = (x_i, y_i)$ des $\mathbb{R}^2$.

Die Aufgabenstellung fordert nun, dass eine Abbildung $D' = f(D)$ gefunden werden soll. Diese ordnet der Menge D eine Bildmenge D' zu. Für diese müssen bestimmte Bedingungen gelten:

- Für jedes $d \in D'$ gilt:
$$\forall\ (x, y) \in d : y \geq 0 \wedge x \geq 0 \tag{1}$$
Alle Punkte müssen also über oder auf der x-Achse sowie rechts oder auf der y-Achse liegen.

- Für jedes $d \in D'$ gilt:
$$\exists\ (x, y) \in d : y = 0 \tag{2}$$
Es muss also in jedem Dreieck mindestens einen Punkt geben, der auf der x-Achse liegt. Die Menge aller solchen Punkte eines Dreiecks sei N_i (anschaulich die Menge der Straßenecken).

- Jedes $d'_i \in D'$ muss kongruent zum entsprechenden $d_i \in D$ sein. Genauer gesagt muss d'_i aus d_i durch eine Abfolge von Kongruenzabbildungen, d.h. Translationen, Rotationen und senkrechten Achsenspiegelungen[1] hervorgehen.

- Für jedes $d \in D'$ und jedes $e \in D'$ gilt:
$$d \cap e = \emptyset \tag{3}$$
$d \cap e$ stellt dabei die Schnittfläche der beiden Dreiecke dar. Es dürfen sich also keine zwei Dreiecke überlappen.

  Eine Dreiecksanordnung wird als **erlaubt** bezeichnet, wenn sie diese Bedingungen erfüllt. Die Menge der erlaubten Dreiecksanordnungen sei dabei E.

Nun ist eine Dreiecksanordnung D' gesucht, die **optimal** ist. Eine optimale Dreiecksanordnung sei dabei folgendermaßen definiert:

- D' minimiert den folgenden Wert über alle erlaubten Dreiecksanordnungen E:
$$\max_{d_i \in D'} \min_{d_j \in D'} \min_{n \in N_i} \min_{m \in N_j} |n.x - m.x| \tag{4}$$
Der Minimums-Term bildet dabei den Abstand zwischen zwei Dreiecken als minimalen Abstand der Straßenecken, während der Maximums-Term den maximalen solchen Abstand berechnet.

Die optimale Dreiecksanordnung D' bildet die Ausgabe des Algorithmus, der $f(D)$ möglichst effizient berechnen soll.

[1] und Spiegelungen an einem Punkt, wobei man diese jedoch auch durch Rotationen um 180° erreichen kann. Demzufolge müssen sie nicht betrachtet werden.

1.2 Wahl eines geeigneten Algorithmus

Die Aufgabe ähnelt einem Packproblem aus der algorithmischen Geometrie. Bei diesen muss man Objekte (z.B. Flächen wie Dreiecke) möglichst dicht in gegebene Container (z.B. ebenfalls Flächen) packen, ohne dass sich die Objekte überlappen.[3] In der hier gegebenen Aufgabe hat man jedoch zusätzliche Nebenbedingungen, die im vorherigen Abschnitt schon erläutert worden sind. Außerdem muss nicht die eingenommene Gesamtfläche minimiert werden, sondern ein Abstand auf der x-Achse.

Leider sind jedoch fast alle Packprobleme NP-vollständig, sodass auch hier die Annahme nahe liegt, dass dies der Fall ist. Demzufolge stellt sich die Frage, wie man ein solches Problem möglichst so lösen kann, dass man ein Gleichgewicht zwischen Effizienz (d.h. Laufzeit) des Algorithmus und Optimalität der Lösung einstellt.

Dafür gibt es verschiedene Herangehensweisen:

- **Brute Force und Backtracking:** Bei Brute Force werden einfach alle möglichen Lösungen durchprobiert, während man bei Backtracking eine Lösung schrittweise aufbaut und Schritte wieder zurücknimmt, wenn sie zu keiner zulässigen Gesamtlösung mehr führen können. Beide Ansätze sind in diesem Fall nicht geeignet, da der Lösungsraum extrem groß ist, d.h. es gibt sehr viele mögliche Lösungen. Wenn man Laufzeiten wie $\mathcal{O}(n! \cdot 6^n)$ vermeiden will, die sich durch Beachtung aller Permutationen und Rotationen ergeben, sollte man diese Lösungsansätze also nicht verwenden.

- **Metaheuristiken:** Zu diesen zählt beispielsweise Simulated Annealing, bei dem man die möglichen Lösungen nach einem globalen Maximum bzw. Minimum einer Bewertungsfunktion absucht. Die Bewertungsfunktion wäre in diesem Fall der Gesamtabstand. Außerdem braucht man für Simulated Annealing eine Möglichkeit, aus einer Lösung eine Nachbarlösung zu generieren, was man in diesem Fall durch z.B. Rotationen der Dreiecke erreichen könnte. Da dies jedoch schwierig zu implementieren ist und man schlimmstenfalls genauso viele Lösungen wie bei Brute Force betrachtet, sind solche Heuristiken ebenfalls nicht geeignet. Auch kann man nicht verhindern, mögliche Lösungen doppelt zu betrachten, was für die Laufzeit ebenfalls nicht so gut ist.

- **Dynamic Programming oder Greedy-Ansätze:** DP- und Greedy-Algorithmen sind zwar meistens laufzeiteffizient, jedoch nicht immer optimal. Aus diesem Grund sind sie für eine optimale Lösung dieses Problems nicht geeignet. Beispielsweise könnte die von einem Greedy-Algorithmus getroffene Entscheidung für den besten Folgezustand, also z.B. die Platzierung eines Dreiecks, zu einem nicht optimalen Gesamtergebnis führen. Es könnte dann beispielsweise nicht mehr möglich sein, andere Dreiecke dicht an das aktuelle anzulegen, wodurch der Gesmatabstand erhöht würde.

- **Heuristiken und Approximationsalgorithmen:** Bei Heuristiken versucht man durch intelligentes Raten und zusätzliche Annahmen über die optimale Lösung zu einer guten Lösung zu gelangen. Eine speziell an das Problem angepasste Heuristik ist für dieses Problem das Mittel der Wahl. Dadurch kann man sowohl eine gute (also polynomielle oder pseudopolynomielle) Laufzeit als auch eine Lösung, die relativ nah am Optimum liegt, erreichen. Die heuristische Herangehensweise an dieses Problem wird in den folgenden Abschnitten näher beschrieben.

1.3 Intuitive Beschreibung der Lösungsidee

1.4 Mathematische Präzisierung des Algorithmus

1.5 Laufzeitbetrachtung und NP-Vollständigkeit

Eine Frage, die sich hierbei auch stellt, ist diejenige, ob es für dieses Problem einen in Polynomialzeit terminierenden Algorithmus geben kann. Dies entspricht der Frage, ob das Problem in der Klasse NPC[2] (NP-vollständig bzw. NP-complete) liegt. Ich vermute, dass dies der Fall ist, kann es jedoch nicht beweisen.

Zum Beweis, dass ein Problem in NPC liegt, werden zwei Voraussetzungen benötigt:

1. Eine deterministisch arbeitende Turingmaschine benötigt nur Polynomialzeit, um zu entscheiden, ob eine z.B. von einer Orakel-Turingmaschine vorgeschlagene Lösung tatsächlich eine Lösung des Problems ist. Dies ist hier der Fall, denn wenn eine Lösung vorgeschlagen wird, kann man in Polynomialzeit überprüfen, ob es sich dabei um eine erlaubte Dreiecksanordnung handelt.

   Dazu überprüft man alle vier Bedingungen dafür. Die ersten beiden Bedingungen lassen sich einfach für jedes Dreieck in konstanter Zeit, insgesamt also in $\mathcal{O}(n)$, überprüfen. Für die dritte Bedingung (Kongruenz) ist dies mithilfe von Kongruenzsätzen ebenfalls in linearer Zeit möglich. Bei der vierte Bedingung (keine Überlappung) muss man alle Dreieckspaare, insgesamt also $\mathcal{O}(n^2)$, auf Überlappung überprüfen. Insgesamt erhält man mit $\mathcal{O}(n^2)$ also Polynomialzeit.

2. Das Problem ist NP-schwer. Das bedeutet, dass alle anderen NP-schweren Probleme auf dieses Problem in Polynomialzeit zurückgeführt werden können. Es ist also eine Polynomialzeitreduktion notwendig. Dabei ist ein Problem aus NPC als Ausgangsproblem nötig, wie z.B. 3-Satisfiability. Eine solche Reduktion zu vollziehen, ist mir jedoch nicht möglich.[3]

Literatur

[1] GeeksforGeeks-Artikel zur DP-Lösung von Subset Sum, `https://www.geeksforgeeks.org/subset-sum-problem-dp-25/`

[2] GeeksforGeeks-Artikel zum Backtracen bei der DP-Lösung, `https://www.geeksforgeeks.org/perfect-sum-problem-print-subsets-given-sum/`

[3] Wikipedia-Artikel zu Packproblemen, `https://en.wikipedia.org/wiki/Packing_problems`

[2] Genaugenommen ist diese Klasse nur für Entscheidungsprobleme definiert, daher handelt es sich bei diesem Suchproblem um NP-Äquivalenz.

[3] Auch wenn eine Beziehung zwischen *3*-SAT und *Drei*ecken natürlich naheliegt.

2 Umsetzung

2.1 Allgemeine Hinweise zur Benutzung

Das Programm wurde in C++ implementiert und benötigt bis auf die *Standard Library* (STL) und die beigelegte `argparse`-Library[4],die für die Verarbeitung der Konsolenargumente zuständig ist, keine weiteren Bibliotheken. Es wurde unter Linux kompiliert und getestet; auf anderen Betriebssystemen müsste mit G++ erneut kompiliert werden.

Die Eingabe und Ausgabe des Programms erfolgt in Dateien, die mithilfe der Konsolenparameter frei gewählt werden können. Dafür gibt es folgende Parameter:

```
Usage: ./main --input INPUT --svg SVG --output OUTPUT
```

2.2 Struktur des Programms und Implementierung der Algorithmen

2.2.1 Die Datei `main.cpp`

2.2.2 Die Datei `triangles.cpp`

2.2.3 Die Datei `triangleAlgorithm.cpp`

3 Beispiele

3.1 Beispiel 1

Abbildung 1: Die Dreiecksanordnung für das Beispiel 1

Ausgabe für Beispiel 1

```
1 Gesamtabstand: 0.000 Meter
2 Platzierung der Dreiecke:
3 D1 0.000 0.000 138.000 37.000 37.000 138.000
4 D2 135.000 0.000 273.000 37.000 172.000 138.000
5 D3 270.000 0.000 408.000 37.000 307.000 138.000
6 D4 405.000 0.000 543.000 37.000 442.000 138.000
7 D5 540.000 0.000 678.000 37.000 577.000 138.000
```

[4] `https://github.com/hbristow/argparse`

3.2 Beispiel 2

Abbildung 2: Die Dreiecksanordnung für das Beispiel 2

Ausgabe für Beispiel 2

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 40.000 458.000 170.000 0.000 20.000 0.000
D2 190.000 558.000 320.000 0.000 170.000 0.000
D3 395.000 278.000 470.000 0.000 320.000 0.000
D4 600.000 516.000 620.000 0.000 470.000 0.000
D5 750.000 501.000 770.000 0.000 620.000 0.000
```

3.3 Beispiel 3

Abbildung 3: Die Dreiecksanordnung für das Beispiel 3

Ausgabe für Beispiel 3

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 0.000 0.000 64.000 0.000 0.000 190.000
D2 34.000 110.000 189.000 0.000 51.000 220.000
D3 215.000 0.000 214.000 160.000 159.000 172.000
D4 277.000 266.000 232.000 0.000 329.000 211.000
D5 270.000 0.000 335.000 190.000 335.000 0.000
D6 341.000 118.000 538.000 159.000 499.000 225.000
D7 451.000 0.000 377.000 112.000 483.000 56.000
D8 479.000 14.000 541.000 0.000 554.000 68.000
D9 580.000 0.000 586.000 155.000 656.000 240.000
D10 673.000 102.000 591.000 11.000 655.000 0.000
D11 680.000 0.000 745.000 0.000 679.000 207.000
D12 776.000 294.000 791.000 0.000 727.000 190.000
```

3.4 Beispiel 4

Abbildung 4: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
```

```
D1 11.000 206.000 46.000 171.000 111.000 171.000
D2 111.000 171.000 81.000 206.000 11.000 206.000
D3 10.000 38.000 46.000 171.000 11.000 206.000
D4 10.000 38.000 46.000 0.000 46.000 171.000
D5 46.000 171.000 111.000 171.000 111.000 106.000
D6 146.000 36.000 146.000 71.000 46.000 171.000
D7 111.000 0.000 146.000 36.000 46.000 171.000
D8 46.000 0.000 111.000 0.000 46.000 171.000
D9 177.000 196.000 211.000 0.000 217.000 59.000
D10 211.000 0.000 231.000 190.000 260.000 0.000
D11 260.000 0.000 292.000 196.000 252.000 59.000
D12 352.000 195.000 352.000 0.000 397.000 195.000
D13 397.000 195.000 397.000 0.000 352.000 0.000
D14 466.000 196.000 496.000 0.000 466.000 31.000
D15 496.000 166.000 496.000 0.000 466.000 196.000
D16 596.000 66.000 496.000 66.000 496.000 166.000
D17 650.000 196.000 685.000 0.000 650.000 36.000
D18 650.000 196.000 685.000 166.000 685.000 0.000
D19 780.000 166.000 650.000 196.000 685.000 166.000
D20 685.000 114.000 780.000 84.000 685.000 84.000
D21 561.000 196.000 561.000 101.000 596.000 66.000
D22 561.000 196.000 596.000 161.000 596.000 66.000
D23 596.000 66.000 596.000 0.000 526.000 66.000
```

3.5 Beispiel 5

Abbildung 5: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 450.000 172.000 465.000 64.000 476.000 121.000
D2 0.000 160.000 39.000 61.000 76.000 124.000
D3 122.000 155.000 153.000 71.000 169.000 150.000
D4 574.000 277.000 636.000 281.000 622.000 383.000
```

```
D5 364.000 106.000 411.000 97.000 441.000 155.000
D6 605.000 172.000 654.000 139.000 615.000 229.000
D7 708.000 64.000 749.000 91.000 717.000 175.000
D8 274.000 313.000 285.000 295.000 302.000 346.000
D9 6.000 221.000 60.000 250.000 7.000 296.000
D10 374.000 217.000 398.000 172.000 407.000 277.000
D11 676.000 289.000 657.000 373.000 640.000 322.000
D12 333.000 59.000 365.000 128.000 312.000 148.000
D13 497.000 285.000 522.000 211.000 558.000 289.000
D14 100.000 0.000 170.000 20.000 120.000 70.000
D15 337.000 151.000 373.000 140.000 348.000 214.000
D16 96.000 61.000 104.000 154.000 80.000 165.000
D17 271.000 142.000 297.000 209.000 279.000 236.000
D18 48.000 307.000 33.000 406.000 6.000 340.000
D19 504.000 115.000 530.000 154.000 489.000 205.000
D20 668.000 46.000 666.000 131.000 605.000 122.000
D21 276.000 259.000 336.000 253.000 370.000 321.000
D22 224.000 295.000 158.000 313.000 171.000 232.000
D23 699.000 271.000 743.000 323.000 703.000 376.000
D24 93.000 223.000 136.000 331.000 86.000 334.000
D25 753.000 251.000 770.000 273.000 767.000 332.000
D26 298.000 114.000 335.000 24.000 260.000 42.000
D27 792.000 178.000 766.000 213.000 750.000 191.000
D28 243.000 244.000 254.000 273.000 240.000 303.000
D29 439.000 273.000 467.000 204.000 467.000 316.000
D30 68.000 201.000 97.000 163.000 80.000 232.000
D31 549.000 148.000 584.000 151.000 552.000 225.000
D32 564.000 131.000 544.000 63.000 621.000 79.000
D33 464.000 366.000 403.000 377.000 418.000 313.000
D34 240.000 212.000 213.000 10.000 249.000 89.000
D35 729.000 192.000 714.000 240.000 663.000 184.000
D36 171.000 172.000 188.000 213.000 116.000 223.000
D37 57.000 334.000 70.000 288.000 69.000 408.000
```

3.6 Eigene Beispiele

4 Quellcode

```cpp
#include<bits/stdc++.h>

using namespace std;

class Point{
    public:

    double x;
    double y;

    Point(double _x, double _y){
        x = _x;
        y = _y;
    }

    Point(){
```

```
        x = 0;
        y = 0;
    }
};

class Vektor{
    public:

    double x;
    double y;

    Vektor(double _x, double _y){
        x = _x;
        y = _y;
    }

    Vektor(Point a, Point b){
        x = b.x - a.x;
        y = b.y - a.y;
    }

    Vektor(){
        x = 0;
        y = 0;
    }

    double betrag(){
        return sqrt(x * x + y * y);
    }
};

class Triangle{
    public:

    Point point1;
    Point point2;
    Point point3;
    int id;
    Vektor p1p2;
    Vektor p2p3;
    Vektor p3p1;

    Triangle(Point p1, Point p2, Point p3, int idd){
        point1 = p1;
        point2 = p2;
        point3 = p3;
        id = idd;
        p1p2 = Vektor(point1,point2);
        p2p3 = Vektor(point2,point3);
        p3p1 = Vektor(point3,point1);
    }
};

```

```
70  Point addVektor(Point &p, Vektor &v){
71      p.x += v.x;
72      p.y += v.y;
73      return p;
74  }
75
76  double dotProduct(Vektor &v1, Vektor &v2){
77      return v1.x * v2.x + v1.y + v2.y;
78  }
79
80  double angle(Vektor &v1, Vektor &v2){
81      double cosvalue = dotProduct(v1,v2)/(v1.betrag()*v2.betrag());
82      return acos(abs(cosvalue));
83  }
84
85  pair<Point,double> locateSmallestAngle(Triangle t){
86          double p1angle = angle(t.p1p2,t.p3p1);
87          double p2angle = angle(t.p1p2,t.p2p3);
88          double p3angle = angle(t.p2p3,t.p3p1);
89          Point smallest = t.point1;
90          double smallestAngle = p1angle;
91          if(p2angle < smallestAngle){
92              smallest = t.point2;
93              smallestAngle = p2angle;
94          }
95          if(p3angle < smallestAngle){
96              smallest = t.point3;
97              smallestAngle = p3angle;
98          }
99          return {smallest,smallestAngle};
100     }
```

Quellcode 1: Die ein Dreieck repräsentierende Klasse `Triangle`

```
1   #include<bits/stdc++.h>
2   #include"triangles.cpp"
3
4   using namespace std;
5
6   vector<int> sol;
7
8   bool getSubsetsRec(vector<int> arr, int i, int sum, vector<int>& p,
    ↪   vector<vector<bool>> &dp)
9   {
10      // If we reached end and sum is non-zero. We print
11      // p[] only if arr[0] is equal to sun OR dp[0][sum]
12      // is true.
13      if (i == 0 && sum != 0 && dp[0][sum])
14      {
15          p.push_back(i);
16          sol = p;
17          return true;
18      }
```

```
19
20      // If sum becomes 0
21      if (i == 0 && sum == 0)
22      {
23          sol = p;
24          return true;
25      }
26
27      // If given sum can be achieved after ignoring
28      // current element.
29      if (dp[i-1][sum])
30      {
31          // Create a new vector to store path
32          vector<int> b = p;
33          if(getSubsetsRec(arr, i-1, sum, b,dp)){
34              return true;
35          }
36      }
37
38      // If given sum can be achieved after considering
39      // current element.
40      if (sum >= arr[i] && dp[i-1][sum-arr[i]])
41      {
42          p.push_back(i);
43          if(getSubsetsRec(arr, i-1, sum-arr[i], p,dp)){
44              return true;
45          }
46      }
47      return false;
48  }
49
50  void subsetSum(vector<int> set,int sum){
51      int n = set.size();
52      vector<vector<bool>> dp(n,vector<bool>(sum+1));
53      for (int i=0; i<n; ++i) {
54          dp[i][0] = true;
55      }
56
57      // Sum arr[0] can be achieved with single element
58      if (set[0] <= sum)
59         dp[0][set[0]] = true;
60
61      // Fill rest of the entries in dp[][]
62      for (int i = 1; i < n; ++i)
63          for (int j = 0; j < sum + 1; ++j)
64              dp[i][j] = (set[i] <= j) ? dp[i-1][j] ||
65                                         dp[i-1][j-set[i]]
66                                       : dp[i - 1][j];
67
68      int best = sum;
69      for(;best>=0;best--){
70          if(dp[n-1][best]) break;
71      }
```

```
72      cout << best << endl;
73      vector<int> p;
74      getSubsetsRec(set, n-1, best, p,dp);
75      for(auto x: sol) cout << x << " ";
76      cout << "\n";
77  }
78
79  pair<vector<Triangle>,double> doAlgorithm(vector<Triangle> triangles, bool
    ↪  debug){
80      Point centerPoint = Point(300,0);
81      vector<Point> bestPoint;
82      vector<int> bestAngle;
83      for(size_t i = 0; i<triangles.size(); i++){
84          auto &t = triangles[i];
85          Point p;
86          double angle;
87          tie(p,angle) = locateSmallestAngle(t);
88          bestPoint.push_back(p);
89          bestAngle.push_back((int) ceil(10000*angle));
90          /*Vektor translation = Vektor(p,centerPoint);
91          t.point1 = addVektor(t.point1,translation);
92          t.point2 = addVektor(t.point2,translation);
93          t.point3 = addVektor(t.point3,translation);*/
94          cout << p.x << " " << p.y << " " << angle << "\n";
95      }
96      for(auto x: bestAngle){
97              cout << x << " ";
98          }
99          cout << "\n";
100     subsetSum(bestAngle,(int) floor(10000*M_PI));
101     //TODO rotate -> no overlap
102     //TODO what to do with triangles outside the subset
103     return {triangles,0};
104 }
```

Quellcode 2: Die Datei `triangleAlgorithm`, die alle wesentlichen Bestandteile des Algorithmus enthält