Aufgabe 4
Streichholzrätsel

Tassilo Tanneberger 3.11.2020

Theorie und Lösungsidee

Streichholzgraphen

So eine Streichholzkonstruktion ist mathematisch gesehen ein Streichholzgraph (engl. matchstick graph). Damit ein Graph als Streichholzgraph bezeichnet werden kann muss er in der 2D-Ebene darstellbar sein. Das bedeutet Kanten dürfen sich nicht kreuzen und alle Knoten liegen auf einer Ebene. Das zweite Kriterium ist, dass die Länge der Kanten immer dieselbe ist.

Beschreibung von Graphen

Es gibt drei große Darstellungsmöglichkeiten von Graphen. Diese werde ich hier nacheinander vorstellen und Vor- bzw. Nachteile für die Beschreibung von Matchstick-Graphen erläutern.

Adjazenzmatrix ist eine quadratische Matrix, wobei jeder Eintrag $a_{ij} \in \{0, 1\}$ der Matrix darstellt, ob i mit j verbunden ist, also ob eine Kante existiert. Das Problem mit dieser Beschreibung ist, dass sie keine Aussage trifft, wo sich die Knoten befinden oder wie lang die Kanten sind. Somit kann es auch für eine Adjazenzmatrix mehrere Darstellungen geben. Zudem haben wir viele redundante Daten, weil die Matrix symmetrisch an der Hauptdiagonale ist da die Kanten ungerichtet sind.

Adjazenzliste speichern zu einem Knoten alle Knoten, die mit diesem verbunden sind. Der Vorteil ist hier, dass man zu einem gegebenen Knoten sehr schnell alle anliegenden Kanten ermitteln kann. Das Problem mit den redundanten Daten bleibt aber bestehen, weil, wenn i mit j verbunden ist, i in der Knotenliste von j eingetragen sein muss und auch umgekehrt.

Kantenlisten sind Listen welche Paare (2-Tupel) aus Knoten speichern. Bei dieser datenstrukturellen Repräsentation muss nur sichergestellt werden, dass die Reihenfolge von i und j in dem Tuple unwichtig ist. Dieser Ansatz wird dann besonders mächtig wenn wir für i bzw. j gleich die konkreten Koordinaten des Knotens einsetzen: $i \leftarrow (x, y)$. Wenn wir das tun haben wir zwei Ziele erreicht. Einmal haben wir nur noch eine Datenstruktur, welche Auskunft über den gezeichneten Graphen gibt. Zweitens ist es sehr einfach die Differenz zweier Graphen zu bilden, was ich später noch erläutern werde.

Algorithmus Idee

Vereinfacht gesagt suchen wir als erstes die ideale Ausrichtung der Graphen zueinander, sodass möglichst viele Kanten und Knoten übereinander liegen und dadurch die minimale Anzahl notwendig umzulegender Streichhölzer gefunden werden kann. Nachdem wir nun die Graphen richtig ausgerichtet haben suchen wir alle identischen Kanten aus beiden Graphen und entfernen diese aus den jeweiligen Kantenlisten. Die Reste der beiden Graphen sind die umzulegenden Kanten. Genauer gesagt, die Menge an Kanten des Ursprungsgraphen muss in die Menge der Kanten des Zielgraphen transformiert werden. Nun können wir überprüfen ob die Anzahl umzulegender Kanten die vom Nutzer gebene Anzahl umzulegener Streichhölzer unterschreitet (umformbar) bzw. überschreitet (nicht umformbar).

Formale Definition Gegeben seien die Kantenliste des Ursprungsgraphen G_1 und die des Zielgraphen G_2. Eine Kante ist definiert als $((x_1, y_1), (x_2, y_2)) \in G_1, G_2$, wobei die Tupel (x_1, y_1) und (x_2, y_2) die beiden Knoten repräsentieren, welche mit dieser Kante miteinander verbunden sind.

Finden der idealen Ausrichtung Für die Rotation nehmen wir einfach eine Rotationsmatrix im zweidimensionalen Raum. Für die Rotation haben wir die Variable r, welche als Zählvariable genutzt wird. Bei der Rotation wird einfach ein Vektor-Matrix-Produkt aus der Rotationsmatrix mit den Koordinaten der Knoten gebildet.

$$\varphi = \frac{\pi \cdot (r \ mod\ 12)}{6} \ r \in \mathbb{N}$$
$$\begin{pmatrix} cos(\varphi) & -1 \cdot sin(\varphi) \\ sin(\varphi) & cos(\varphi) \end{pmatrix}$$

Wir haben die Knoten nun um den Koordinatenursprung gedreht. Danach müssen die Kanten nur noch an (0|0) ausgerichtet werden. Dabei werden die kleinsten Koordinaten x_{min} und y_{min} gesucht. Und alle Knoten werden dann einfach um $|x_{min}|$ und $|y_{min}|$ verschoben. Für eine Spiegelung an der Y-Achse multiplizieren wir einfach die X-Koordinate mit -1, für eine Spiegelung an der X-Achse die Y-Koordinate mit -1. Wir haben also $12 \times 2 \times 2$ Möglichkeiten durch die wir durch iterieren müssen: 12 Rotationen (um Vielfache von 30°), jeweils zwei X-symmetrische und zwei Y-symmetrische Anordungen.

Graphendifferenz Nachdem wir nun die ideale Ausrichtung der beiden Graphen zueinander gefunden haben, bilden wir die Differenz der beiden Graphen. Das bedeutet wir entfernen die Kanten, die in beiden Graphen vorkommen. Die restlichen Kanten aus Graph G_1 sind die, die es in die Kanten aus Graph G_2 umzulegen gilt. Wenn es also weniger Kanten in G_1 als in G_2 gibt, kann keine Umformung stattfinden. Wenn beide Graphen die gleiche Anzahl Kanten haben, können wir einfach über beide Mengen iterieren und die Kanten umlegen.

Implementation

Das Program wurde in C++ umgesetzt. In diesem Abschnitt werden ausgewählte Quellcodestücke etwas genauer betrachtet.

Code-Beispiel 1.)

```
void Vertex::rotate_by_degree(unsigned int rot) {
    // Does a vector space transformation with every individual vector stored in this matrix
    //  cos(r) -1 * sin(r)
    //  sin(r)  cos(r)
    // Afterwards it brings every vector into the first quadrant

    float radians = (M_PI * (rot % 12)) / 6;
    float x_0 = std::cos(radians), x_1 = std::sin(radians) , y_0 = -1.0 * std::sin(radians),
    y_1 = std::cos(radians);
    float temporary_x = coordinates_.first;

    coordinates_.first = x_0 * temporary_x + y_0 * coordinates_.second;
    coordinates_.second = x_1 * temporary_x + y_1 * coordinates_.second;
}
```

Dies ist die Funktion, welche einen Knoten um den Koordinatenursprung rotiert.

Code-Beispiel 2.)

```
void Graph::difference(const std::shared_ptr<Graph> &other_graph){
    std::unordered_multiset<Edge> edge_set;
    edge_set.insert(base_graph_.begin(), base_graph_.end());
    edge_set.insert(other_graph->base_graph_.begin(), other_graph->base_graph_.end());

    auto predicate = [&edge_set](const Edge& k){
        return edge_set.count(k) > 1;
    };

    base_graph_.erase(std::remove_if(base_graph_.begin(), base_graph_.end(), predicate),
    base_graph_.end());
    other_graph->base_graph_.erase(std::remove_if(other_graph->base_graph_.begin(),
    other_graph->base_graph_.end(), predicate), other_graph->base_graph_.end());
}
```

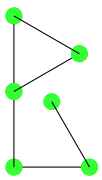
Dies ist die Funktion welche die Differenz zweier Graphen bildet. Dabei nutzen wir `std::unordered_multiset<Edge>`. Dieser Datencontainer zählt, wie oft ein bestimmtes Element in ihm enthalten ist. Das können wir ausnutzen, indem wir alle Kanten der beiden Graphen dem Container hinzufügen und dann aus den Graphen nur die Kanten entfernen, die doppelt in der `std::unordered_multiset` vorhanden sind. Ob ein Element entfernt werden muss, beurteilt die Lambda-Funktion `predicate`.

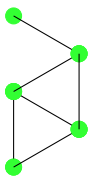
Code-Beispiel 3.)

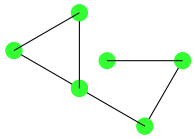
```
unsigned int array[4] = {0, 1, 3, 2};
for (unsigned int i = 0; i < 4; i++) {
        mirage_x = (array[i] & 1) == 1;
        mirage_y = (array[i] & 2) == 2;
        // 0 0  => 0 1 => 1 1 => 1 0 So only one bit flips every iteration

        if (i > 0 and i % 2 == 0) {
            mirror_x(vertices);
        } else if (i > 0 and i % 2 == 1) {
            mirror_y(vertices);
        }

        // rotation stuff and comparisons ...
}
```

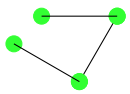
Dieses Code-Stück stellt die Reihenfolge dar, in der die Spiegelungen durchgeführt werden. Dabei wird bei jeder Iteration dieser Schleife nur eine Spiegelung durchgeführt - also ein "bit flip" pro Schleifendurchlauf. Zudem werden die Bits auch nur in der Reihenfolge zurückgetauscht, in der sie getauscht wurden (ähnlich dem FIFO-Prinzip). Wie der Kommentar vermuten lässt, wird dann dort die Rotation durchgeführt und die Anzahl übereinander liegender Kanten gezählt und potenziell als ideale Ausrichtung gespeichert.

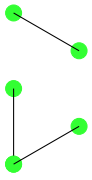
Beispiel-Druchläufe


Ursprungsgraph


Zielgraph


Ideale Ausrichtung keine Spiegelungen aber eine Rotation um 60° entgegen des Uhrzeigersinnes.


G_1 nach der Differenz.


G_2 nach der Differenz.

Im folgenden ist die Eingabe in das Programm und der dazugehörige Konsolen output zusehen. Für eine Visualisierung bietet die Klasse `Graph` auch die Methode `visualize` an. Diese Methode erzeugt Tikz-Code, welcher auch genutzt wurde um die Beispiele von gerade zu erzeugen.

```
Aufgabe 4 des Bundeswettbewerbs Informatik 2020
================================================


Geben sie ein in wie vielen Umformungsschritten der Graph transformiert werden soll
4
Nun geben sie die Knoten der beiden Graphen ein. Diese Knoten bekommen jeweils eine ID die später für die Eingabe der Kanten genutzt wird.
Es reicht eine präzision von 3 Nachkomma Stellen aus.
Kleine Hilfe:
cos(30°) = 0.8661 cos(60°) = 0.5
sin(30°) = 0.5 sin(60°) = 0.8661
Vertex: 0
x:0
y:0
Vertex: 1
x:1
y:0
Vertex: 2
x:0
y:1
Vertex: 3
x:0
y:2
Vertex: 4
x:0.8661
y:0.5
Vertex: 5
x:0.8661
y:1.5
Vertex: 6
x:0.5
y:0.8661
Vertex: 7
x:
Die eingegebenen Knoten:
Vertex: 0 x = 0 y = 0
Vertex: 1 x = 1 y = 0
Vertex: 2 x = 0 y = 1
Vertex: 3 x = 0 y = 2
Vertex: 4 x = 0.8661 y = 0.5
Vertex: 5 x = 0.8661 y = 1.5
Vertex: 6 x = 0.5 y = 0.8661
```

Figure 1: Selbst erdachtes Beispiel Eingabe der Knoten

```
Eingabe der Kanten mit hilfe der IDs die gerade für die Knoten vergeben wurden
Eingabe für Graph1
Wie viele Kanten soll der Graph1 besitzen ?
6
Edge: 0
0
1
Edge: 1
0
2
Edge: 2
2
3
Edge: 3
1
6
Edge: 4
2
5
Edge: 5
3
5
Eingabe für Graph2
Wie viele Kanten soll der Graph2 besitzen ?
6
Edge: 0
0
2
Edge: 1
0
4
Edge: 2
2
4
Edge: 3
2
5
Edge: 4
5
3
Edge: 5
4
5
Eine Spiegelung an der X Achse: 0 Eine Spiegelung an der Y Achse: 0
Rotation um 60° entgegen des Uhrzeigersinn
Übereinanderliegende Kanten: 3
Edge:0 / x_0: 1.73 y_0: 0 x_1: 2.23 y_1: 0.87 => x_0: 0 y_0: 0 x_1: 0 y_1: 1
Edge:1 / x_0: 1.73 y_0: 0 x_1: 0.87 y_1: 0.5 => x_0: 0 y_0: 0 x_1: 0.87 y_1: 0.5
Edge:2 / x_0: 2.23 y_0: 0.87 x_1: 1.23 y_1: 0.87 => x_0: 0.87 y_0: 1.5 x_1: 0 y_1: 2
```

Figure 2: Selbst erdachtes Beispiel Eingabe der Kanten