Aufgabe 1: Superstar

Team-ID: 00922

Team-Name: Zweiundvierzig

Bearbeiter/-innen dieser Aufgabe:
Franz Miltz

26. März 2019

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabe und meine Lösung umfassen mehrere Teilprobleme. Das erste was sich stellt, ist der triviale Fall. Hierzu muss bestimmt werden, in welchem Winkel Lisa laufen muss, wenn sie kein Hindernis in ihrem Weg hat. Dass sie in diesem Fall einer geraden Strecke folgt ist klar, da anderenfalls ein kürzerer Weg zum Schnittpunkt mit der Straße existieren würden.

Um die komplizierteren Fälle zu betrachten werden zwei wesentliche Schritte benötigt. Zunächst müssen alle *relevanten* Punkte berachtet werden, die für die Ermittlung des optimalen Weges eine Rolle spielen. Wenn diese Punkte dann bestimmt sind, müssen in den resultierenden Graphen alle möglichen Kanten (optimalerweise mit Kantelängen) eingezeichnet werden. In diesem Graphen muss nun der Pfad gesucht werden, den Lisa gehen muss, um so spät wie möglich ihr Haus verlassen zu können.

1.1 Freier Weg

Wir betrachten das Dreieck ΔABC wobei die Seite b Lisas Abstand zur y-Achse und die Seite a der Abstand des Schnittpunktes der beiden Wege S in y-Richtung von Lisa ist. Somit ist der Abstand zwischen Lisa und dem Punkt $\sqrt{a^2+b^2}$. Uns ist egal, wo der Bus sich im Moment befindet, weil er zwar die Zeitdifferenz am Punkt S verändert, aber die Änderung in Abhängigkeit des Winkels bzw. des Verhältnisses $\frac{a}{b}$ bleibt gleich, d.h. die Position des Busses ist für die Ableitung irrelevant.
Somit können wir die Zeitdifferenz Δt in Abhängigkeit des Faktors r in $a = rb$ bei bestimmten Geschwindigkeiten v_B und v_L bestimmen:

$$\Delta t = \frac{r \cdot b}{v_B} - \frac{\sqrt{(r^2+1)} \cdot b}{v_L} \tag{1}$$

Für die Ableitung gilt folglich:

$$\frac{d}{dr}\left(\frac{r \cdot b}{v_B} - \frac{\sqrt{r^2+1} \cdot b}{v_L}\right) = \frac{b}{v_B} + \frac{b \cdot r}{v_L\sqrt{r^2+1}} \tag{2}$$

Nun suchen wir Extremstellen:

$$\frac{d}{dr}\Delta t = 0 \Leftrightarrow r = \pm\frac{v_L}{\sqrt{v_B^2 - v_L^2}} \tag{3}$$

Da wir nur an positiven r interessiert sind, vernachlässigen wir die Vorzeichen an dieser Stelle. Somit wissen wir, wie Lisa laufen muss, wenn ihr kein Hindernis im Weg ist. Den tatsächlichen Winkel auszurechnen wäre auf Grund der Modellierung, die ohne Winkel auskommt, unnötig.

1.2 Relevante Punkte

Ich habe bereits erwähnt, dass der Weg bei freier Strecke gradlinig verlaufen muss. Dieser Umstand gilt mehr oder weniger auch, wenn Hindernisse im Weg sind. Denn da Lisa keine Form der Trägheit hat, ist es für sie am effizientesten stets solange geradeaus zu laufen, bis sie ihre Richtung ändern kann, weil der nächste Wegpunkt in sicht ist. Das bedeutet, dass sich Lisas Pfad, abgesehen vom Schnittpunkt mit der Straße, ausschließlich aus Eckpunkten der Hindernisse zusammensetzt. Um dies etwas zu präzisieren, kann sogar gesagt werden, dass der Pfad sogar nur aus kovexen Eckpunkten der Polygone besteht. Dies kann man sich leicht erklären, indem man ein konkaves Polygon betrachtet. Zwischen zwei konvexen Ecken verläuft der kürzeste Weg nie über eine konkave Ecke. Sollte nun ein anderes Polygon diesen direkten Weg blockieren, ist auch hier die kürzeste Strecke über die konvexen Ecken jenes Polygons.
Wenn wir nun also versuchen alle relevanten Punkte zu finden, müssen wir alle konvexen Ecken der Polygone betrachten.

1.3 Kanten

Um herauszufinden, welche Punkte direkt durch eine Kante miteinander verbunden werden können, muss ein Schnittalgorithmus her. Zu die Funktionsweise dieses Algorithmus werde ich stärker in der Umsetzung thematisieren. Relevant ist im Moment, dass er existiert. Wir können also überprüfen, ob zwei Strecken in unserer Ebene sich schneiden. Somit können wir alle diejenigen Wegpunkte miteinander verbinden, zwischen denen die Strecke mit keiner Kante eines Polygons kollidiert.
Die Länge dieser Kante herauszufinden ist nun trivial, weil sie mit dem Satz des Pythagoras und den Koordinaten der beiden Punkte beschrieben werden kann.

1.4 Optimaler Weg

Es gibt viele Algorithmen, um einen optimalen Weg zwischen zwei Punkten in einem Graphen zu finden. Leider bringen uns diese Algorithmen nur etwas, wenn wir die potentiellen Punkte auf der Straße markieren, die erreichbar sind. Darauf möchte ich jedoch verzichten, weil es einen besseren Ansatz gibt. Statt die Punkte zu suchen, die wir auf der Straße optimalerweise erreichen können, schauen wir nur, welche Punkte des Graphen unter dem optimalen Winkel einen freien Weg zur Straße haben und markieren diese.
Beginnend bei dem Startknoten, Lisas Haus, starten wir nun unsere Suche nach dem optimalen Weg. Dabei fügen wir in jedem Schritt einer Warteschlange all jene Pfade hinzu, die aus der aktuellen Instanz entstehen können. Wir fügen dem bisher gelaufenen Pfad also alle unbesuchten Nachbarn einzeln hinzu. Die Warteschlange sortiert hierbei alle Teilpfade nach der minimalen Zeitdifferenz, für die dieser Pfad sorgen kann.
Diese minimale Differenz ist hierbei eine tatsächliche Untergrenze. Sie berechnet sich aus der Länge der bereits gelaufenen Strecke sowie der Länge des Weges, den Lisa wählen sollte, wenn es ab dem aktuellen Knoten kein Hindernis mehr gäbe. Damit wird klar, dass der optimale Weg genau dann gefunden wurde, wenn der letzte besuchte Knoten des obersten Teilpfades auf der Warteschlange einen freien, optimalen Weg zur Straße hat. Denn in diesem Fall wird die Untergrenze zur tatsächlichen Zeitdifferenz.
Wenn diese Folge von Knoten in unserem Graphen gefunden wurde, wird es mehr oder weniger trivial den Zeitpunkt zu bestimmen, an dem Lisa spätestens loslaufen muss, um den Bus zu bekommen. Dieser Zeitpunkt ist die Abfahrtzeit des Busses plus die Zeitdifferenz (die aller Wahrscheinlichkeit nach negativ ist).

2 Umsetzung

Ich habe das Programm auf einem Linux-System in der Programmiersprache Go implementiert. Die Wahl der Sprache hat mehrere Gründe. Einerseits ist die Sprache sehr leicht verständlich und ist daher in meinen

Augen ähnlich wie Python für den BwInf sehr geeignet und andererseits ist die Sprache schnell. Zum einen lassen sich Projekte sehr schnell kompilieren und zum anderen sind sie ähnlich schnell wie solche, die in Sprachen wie C/C++ geschrieben wurden. Ein weiterer Vorteil, den ich unter Umständen ausnutzen werde ist die Tatsache, dass Go sehr gute Unterstützung für Nebenläufigkeit bereitstellt. Hinzu kommt der lobenswerte Umstand, dass Go sehr einheitlich ist. Der Quellcode wird automatisch formatiert und die Sprache zwingt mich meinen Code zu kommentieren. All diese Eigenschaften machen Go zu einer der geeignetsten Sprache für die Lösung der Probleme im Rahmen des Bundeswettbewerbs Informatik.
Das entstandene Programm ist eine Applikation, die aus der Command Line einen Server startet, auf den dann im Browser zugegriffen werden kann. Da es für die Lösung der Aufgabe nicht relevant ist, wie ich die Dateien einlese und wie ich meinen Output ausgebe, werde ich diese Schnittstellen nicht weiter thematisieren.
Was hingegen wichtig ist, ist sind die Datenstrukturen, die genutzt werden, um den Graphen zu erstellen. Hierzu gibt es vier geometrische Objekte, die im Quellcode abstrahiert werden. Diese Objekte befinden sich im `internal`-Paket des Projektes. Das kleinste Element ist ein einfacher Punkt. Dieser besteht aus zwei zwei Gleikommazahlen, den Koordinaten X und Y.

```
type Point struct {
  X, Y float64
}
```

Hinzu kommen Geraden, die in der *abc*-Form dargestellt werden:

```
type Line struct {
  A, B, C float64
}
```

Es gilt also:

$$ax + by = c \tag{4}$$

Da das Problem nahezu ausschließlich Strecken umfasst, wird auch hierfür eine Datenstruktur benötigt:

```
type LineSegment struct {
  L    *Line
  A, B *Point
}
```

Wie man sieht setzt sich eine Strecke einerseits aus den beiden Punkten zusammen, die ihr Ende bilden und andererseits aus der Geradengleichung, die den Verlauf der Strecke beschreibt.
Wenn man nun Polygone beschreiben will, kann man sich entscheiden, ob diese als geordnete Menge von Punkten oder von Strecken beschrieben werden. Ich habe mich für Punkte entschieden, weil die Inputdaten ebenfalls in diesem Format vorhanden sind und Strecken im Wesentlichen auch nur Paare von Punkten sind. Um die spätere Berechnungen einfacher zu machen, nehmen wir eine weitere Abstraktion vor. Die Ecken der Polygone sind nun nicht nur einfache Punkte, sondern kennen weiterhin ihre Nachbarn sowie das Polygon dem sie angehören.

```
type Corner struct {
  *Point
  Plygn *Polygon
  N1    *Corner
  N2    *Corner
}
```

Damit lässt sich ein Polygon folgendermaßen beschreiben:

```
type Polygon struct {
  Corners []*Corner
  N       int
}
```

Dabei ist `Polygon.N` die Anzahl der Ecken des Polygons.
Nun lesen wir die Eingabedaten und konstruieren das Gelände, in dem Lisa sich bewegen muss. Dieses Gelände wird als `internal/terrain.Terrain` abstrahiert.

```
type Terrain struct {
  PolCnt int
  Plygns []*internal.Polygon
  Start  *internal.Point
}
```

Dabei ist `Start` der Punkt an dem Lisa zu laufen beginnt. Um aus diesem Gelände jetzt einen Graphen zu generieren, benötigen wir folgende Schritte.
Zunächst müssen wir nacheinander alle Ecken der Polygone ablaufen und entscheiden, ob der Punkt relevant ist. Wenn er das ist, wird er der Liste der relevanten Punkte hinzugefügt.

```
pt2indx := make(map[*internal.Point]int)
indx2pt := make(map[int]*internal.Point)
pt2indx[t.Start] = 0
indx2pt[0] = t.Start
count := 1
for _, plygn := range t.Plygns {
  for _, c := range plygn.Corners {
    if !c.IsConcave() {
      indx2pt[count] = c.Point
      pt2indx[c.Point] = count
      count++
    }
  }
}
```

Wie man sieht erstellen wir zwei Maps (im Wesentlichen Hashtabellen), die einerseits Indizes auf Punkte und andererseits Punkte auf Indizes abbilden. Diesen Maps fügen wir nun zunächst Lisas Zuhause und anschlieSSend alle konvexen Ecken der Polygone hinzu.
Wie finden wir heraus, ob eine Ecke konvex ist? Ganz einfach:

```
func (crnr *Corner) IsConcave() bool {
  a := crnr.N1.Point
  b := crnr.Point
  c := crnr.N2.Point
  return (b.X-a.X)*(c.Y-b.Y)-(b.Y-a.Y)*(c.X-b.X) < 0
}
```