35. Bundeswettbewerb Informatik

Runde 1

01.09.2016 - 28.11.2016

Aufgabe 5

Buhnenrennen

23. November 2016

Eingereicht von: *Wouldn't IT be nice...* (Team-ID: 00007)

Tim Hollmann (6753)
ich@tim-hollmann.de

Anike Heikrodt (6841)
anikeheikrodt@online.de

Wir versichern hiermit, die vorliegende Arbeit ohne unerlaubte fremde Hilfe entsprechend der Wettbewerbsregeln des Bundeswettbewerb Informatik angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Tim Hollmann & *Anike Heikrodt*, den 23. November 2016

0.1 Einleitendes - Minimax-Algorithmus

Bei dem vorliegenden Problem handelt es sich um ein endliches zwei-Personen Nullsummenspiel mit perfekter Information. Unserer Ansicht nach ist der Minimax-Algorithmus hier jedoch nicht anwendbar, da die Spieler nicht rundenbasiert abwechselnd, sondern gleichzeitig ziehen.

1 Lösungsidee

Es gilt einen Weg zu finden, auf dem Max Minnie an keinem Punkt einholen könnte. Daher muss für jede von Minnie auf einem *sicheren* Minnieweg durchquerte Lücke gelten: Minnie erreicht sie, bevor Max sie frühestens erreichen kann. (Denn sonst gäbe es einen Weg, wie Maxi sie fangen könnte - durch Warten - und der Weg wäre nicht mehr sicher). (Um auszuschließen, dass Minnie auf einem Minnieweg nicht von Max eingeholt werden kann, reicht es aus, lediglich die Lücken zu betrachten; könnte Max Minnie zwischen den Buhnen erreichen, gäbe es auch auf jeden Fall eine Möglichkeit für ihn, die nächste Buhne auf Minnies Weg vor ihr zu erreichen; denn würden sich beide vom gemeinsamen Treffpunkt zwischen den Buhnen weiter zur Lücke bewegen, könnte Max sie aufgrund seiner höheren Geschwindigkeit immer früher als Minnie erreichen.)

1.1 Schritt 1: Kürzeste Wege für Max

Wir ermitteln im ersten Schritt zunächst für alle Lücken, zu welchem Zeitpunkt Max diese *frühestens* erreichen kann. Wir ermitteln die kürzesten Wege von Max's Startpunkt zu jeder Lücke und teilen die Wegstrecke durch seine Geschwindigkeit; dadurch ist jeder Lücke l ein ein Zeitwert $t_{Max}(l)$ zuweisbar. t_{Max}

Dazu interpretieren wir die Buhnenlandschaft als Graphen, bei dem die Lücken den Knoten entsprechen und die Abstände der Lücken zweier benachbarter Buhnen den Gewichten der Kanten zwischen ihnen. Jede Lücke besitzt daher ausgehende Kanten zu jeder anderen Lücke auf der dahinter gelegenen Buhne. (Abbildung **1**.)

Um nun die kürzesten Wege von Max's Startpunkt zu jedem anderen Knoten in diesem kantengewichteten Graphen zu ermitteln, bietet sich ein modifizierter Dijkstra single-source shortest paths (SSSP)-Algorithmus an. Dieser ist in der Hinsicht modifiziert, dass er die von „Minilücken-Knoten" ausgehenden Kanten nicht beachtet, da Max sich auf diesen ja nicht bewegen darf; er kann eine Minnielücke zwar erreichen, diese jedoch nicht durchqueren, sprich darf er deren ausgehende Kanten nicht verfolgen.) Der Dijkstra SSSP Algorithmus ist ein „normaler" Dijkstra-Algirithmus, bei dem es keinen Zielknoten und keine Abbruchbedingung beim Erreichen dessen gibt. Daher terminiert der Algorithmus erst, wenn er die kürzesten Wege aller (mit dem Startpunkt zusamenhängenden) Knoten des Graphen ermittelt hat.[1] (Algorithmus **1**)

[1] Zu weiterführenden Informationen zur Funktionsweise des Dijkstra-Algorithmus siehe `https://de.wikipedia.org/wiki/Dijkstra-Algorithmus`.

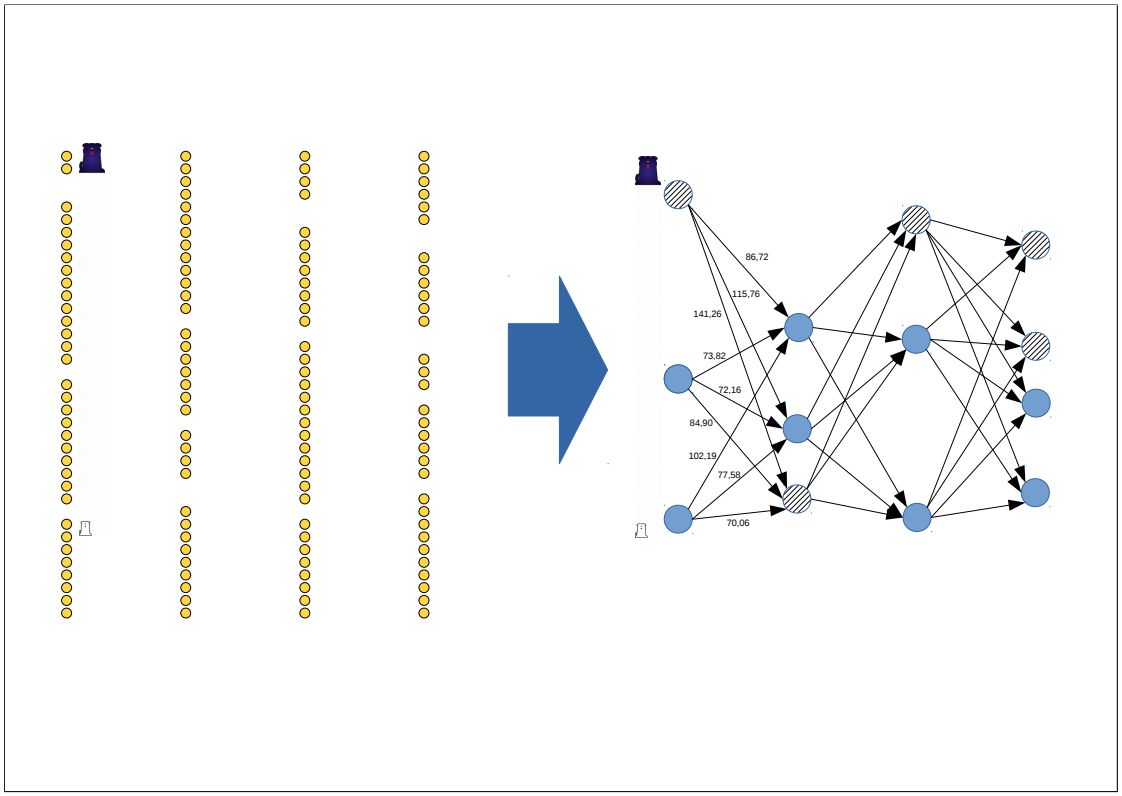

Abbildung 1: Umwandlung der Buhnenlandschaft in einen Graphen. Die Knoten repräsentieren die Lücken und die Kantengewichte die Entfernungen; Maxilücken schraffiert dargestellt und Kantengewichte der ersten Buhnenreihe eingetragen.

Die resultierenden Wegstrecken in Metern für alle Knoten werden durch Maxs Geschwindigkeit (30 km/h = 8,333 m/s) geteilt; jetzt wissen wir für jeden Knoten l, zu welchem Zeitpunkt $t_{Max}(l)$ in Sekunden Max diesen *frühestens* erreichen kann.

1.2 Schritt 2: Suche eines Weges für Minnie

Ein Minnieweg ist formal ein (endlicher) Pfad $P = \{S_1, a_1, S_2, a_2, S_3, \ldots, S_n\}$, wobei S einem Knoten, a einem überschrittenen Kantengewicht, S_1 dem Startpunkt von Minnie und S_n einem Knoten der letzten Buhnenreihe („Zielknoten“) entspricht. Wie bereits festgestellt, muss für einen *sicheren* Minnieweg zusätzlich gelten

$$\left(\sum_{i=1}^{x-1} a_i \right) \; / \; 5.55 < t_{Max}(S_x) \quad \forall \, S_x \in P$$

also die Summe aller bisher überschrittenen Kantengewichte ($\rightarrow$ zurückgelegter Weg von Minnie) in Metern geteilt durch Minnies Geschwindigkeit[2] ($\Rightarrow$ Zeitpunkt der Ankunft von

[2] 20 km/h = 5,55 m/s

Algorithmus 1 : Modifizierter Dijkstra SSSP-Algorithmus

Eingabe : Graph $G = (V, E)$, Startknoten $v_0 \in V$.
Ausgabe : dist : $V \to \mathbb{R}$.

```
1  dist(v) ← ∞   ∀ v ∈ V\{v_0}
2  dist(v_0) ← 0
3  besucht = ∅, unbesucht = V
4  solange [ min_{x ∈ unbesucht} dist(x) ] ≠ ∞   // noch zusammenhäng. Elem. in unbesucht
5   tue
6      current ← arg min_{x ∈ unbesucht} dist(x)
7      unbesucht ← unbesucht − {current}
8      besucht ← besucht ∪ {current}
9      wenn current ist Minnielücke dann
10         continue     // ignorieren der von Minnielücken ausgehenden Kanten.
11     für jedes x ∈ {(current, target, weight) ∈ E : target ∈ unbesucht} tue
12         dist(target) ← min ( dist(current) + weight,   dist(target) )
           /* Sonst: Update der Kosten für die Nachbarn von current        */
```

Minnie in Sekunden) muss für jeden Knoten des Pfades kleiner sein als Max's frühester Ankunfts-Zeitpunkt für eben diesen Knoten ($t_{Max}(S_x)$).
Um zu überprüfen, ob ein solcher Pfad existiert, verwenden wir eine Tiefensuche, die von Minnies Startpunkt ausgehend immer nur diejenigen Knoten verfolgt, für die die obrige Regel erfüllt ist.[3] Wird dadurch ein Knoten erreicht, der eine Lücke in der letzten Buhnenreihe repräsentiert, ist ein Minnieweg gefunden; ist die Tiefensuche nicht erfolgreich, ist folglich auch die Existenz eines Minnieweges ausgeschlossen, da Max Minnie auf jedem Weg erreichen kann.
Dafür ist ein standard Tiefensuche-Algorithmus ausreichend; Minnie kann bzw. darf sich auf allen Kanten des Graphen bewegen. Der Tiefensuche-Algorithmus wird als bekannt vorausgesetzt.

1.3 Laufzeitkomplexität

Die Laufzeit nimmt in Abhängigkeit der Eingabegröße (in diesem Fall: Anzahl der Lücken) maximal polynomiell zu. Teilweise durch die nicht geordneten Eingabedateien bedingt, geschieht das Einlesen des Graphs in $\mathcal{O}\left(n^2\right)$ (für jeden Knoten wird für jeden anderen Knoten überprüft, ob dieser sich 70 Meter weiter rechts befindet und daher eine Kante generiert werden muss; zum Aufbau des Grahpen siehe **2.2**). Der Dijkstra single-source shortest paths Algorithmus besitzt in der beschriebenen Form eine Laufzeitkomplexität,

[3] Da es ausreicht, „irgendeine" mögliche Lösung zu finden und nicht etwa die mit der kürzesten Wegstrecke o.Ä. (spezielle, „optimale" Lösung), können wir auf eine Breitensuche verzichten.

die von $\mathcal{O}(|E| + |V|)$ beschränkt wird. Die Tiefensuche besitzt ebenfalls bekannter Weise eine worst-case-Laufzeit von $\mathcal{O}(|E| + |V|)$.
Wir nehmen die Laufzeitkomplexität also als durch $\mathcal{O}(n^2)$ beschränkt an. Wie die Beispiele zeigen (siehe **3**), bearbeitet der Algorithmus selbst die größten gegebenen Beispiele in Sekundenbruchteilen und ist daher völlig ausreichend.[4]

2 Umsetzung

Die Umsetzung erfolgte in C++ 11 als Konsolenanwendung.

2.1 Programminterne Repräsentation der Lücken/Knoten

Programmintern wird ein Knoten als Instanz der Klasse `node` repräsentiert; diese besitzt die Eigeschaften

1. x- und y-Koordinate als `double`,
2. die boolesche Eigenschaft `isMini`, die die Größe der Lücke angibt (`true` für eine Minnielücke)
3. eine Menge von Kanten, die die Knoten Beschreiben, die von dem jeweiligen Knoten aus erreichbar sind; im Endeffekt also eine Art Adjazenzliste. Gespeichert ist ein Tupel aus dem Index des Zielknotens und dem Kantengewicht als `double`.

2.2 Einlesen der Knoten und Konstruktion des Graphen

Die Lücken werden zeilenweise aus der Datendatei gelesen und für jede eine Instanz der Klasse `node` angelegt und in einer Menge `nodes` gespeichert; die daraus resultierende Indexnummer wird im Folgenden zur Verschlüsselung und Identifikation der Knoten in den Kanten-Tupeln verwendet: Anschließend werden für jedes Knoten-Objekt diejenigen anderen Knoten-Objekte ermittelt, deren x-Abstand 70 Meter beträgt, sich also in der nächsten Buhnenreihe befinden; zu diesen Lücken wird eine gerichtete Kante erstellt.

Bewegungsrichtung
An dieser Stelle ist es sinnvoll, zu überlegen, zu welchen Lücken Kanten erzeugt werden sollen; dass sich Minnie in Bewegungsrichtung weiter nach rechts bewegen soll, ist zunächst einleuchtend. Es ist fraglich, ob es vorteilhaft sein kann, sich zurück auf die vorhergegangene Buhnenreihe oder zu einer anderen Lücke der selben Buhnenreihe zu bewegen. Da wir uns nicht vorstellen können, dass dies vorteilhaft sein kann und wir uns auch keine Situation konstruieren konnten, bei der eine solche Bewegung für einen sicheren Minnieweg notwendig wäre, haben wir das Erstellen derartiger Kanten nicht implementiert. Derartige Bewegungen können nur

[4] Eventuell hätte durch Verwendung eines Minimax-Algorithmus (siehe **0.1**) eine noch schnellere Laufzeit erreicht werden können.

für unnötig verlängerte Wegstrecken sorgen, die sowohl für Minnie als auch für Max nachteilhaft sind.

Ist der Graph konstruiert, wird der Dijkstra-Algorithmus (siehe Algorithmus **1**) gestartet und anschließend die Tiefensuche nach einem Lösungsweg (nach **1.2**) initiiert.

2.3 Funktionen und ihre Aufgaben

2.3.1 `int main(int argc, char* argv[])`

Hauptfunktion; nimmt den Kommandozeilenparameter (Pfad zur Datendatei) entgegen, liest die Datei aus, erstellt die `node`-Objekte, initiiert den Dijkstra-Algorithmus (`getShortestPathsMax`) und die Tiefensuche (`dfs`). Anschließend gibt sie (sofern gefunden) den Minnieweg aus.

2.3.2 `vector<double> getShortestPathsMax (...)`

Implementierung des modifizierten Dijkstra SSSP-Algorithmus aus **1**. Ermittelt die kürzesten Wege von Max's Startpunkt zu jedem anderen erreichbaren Knoten im Graphen.

2.3.3 `bool dfs(...)`

Implementierung der Tiefensuche aus **1.2**; ermittelt rekursiv, ob ein Pfad existiert, der Minnies Startpunkt mit einem Knoten der letzten Buhnenreihe verbindet und dabei für jeden Knoten die Bedingung aus **1.2** erfüllt. Dazu werden von Minnies Startpunkt ausgehend rekursiv nur diejenigen Knoten verfolgt, für die die Bedingung erfüllt ist (auf denen Minnie also nicht von Max geschnappt werden kann); siehe **1.2**.

2.4 Kompilat und Kommandozeilenargumente

Der Quelltext wurde für Linux in verschiedenen Optimierungsstufen unter 64 Bit kompiliert; Die Executables sind unter `5‣bin‣linux_64_o*.out` zu finden.

Kommandozeilenparabeter

```
Executable <filename>
filename: (relativer) Pfad zur Datendatei.
```

3 Beispiele

Hinweis: Ausgabedateien
Die Ausgaben des Programmes für alle eigenen und vorgegebenen Beispiele finden Sie auch unter `5‣data‣out‣*.txt` .

Datei	**Laufzeit**	**Lösung; Minnieweg**
data/buhnenrennen1.txt	0.00011 Sek.	(4 Lücken) (0,42.95), (70,45.9), (140,42.2), (210,49.4)
data/buhnenrennen2.txt	0.000161 Sek.	(8 Lücken) (0,42.1), (70,14.1), (140,28), (210,67.9), (280,64.5), (350,56.55), (420,50.3), (490,97.85)
data/buhnenrennen3.txt	0.000246 Sek.	(8 Lücken) (0,57.1), (70,55.85), (140,76.8), (210,100.95), (280,167.65), (350,89.55), (420,101.65), (490,145.1)
data/buhnenrennen4.txt	0.000105 Sek.	(8 Lücken) (0,41.8), (70,15.65), (140,16.55), (210,25.55), (280,34.4), (350,40.05), (420,58.3), (490,53.5)
data/buhnenrennen5.txt	0.000152 Sek.	(8 Lücken) (0,25.25), (70,28.25), (140,63.2), (210,42.05), (280,34.95), (350,12.85), (420,28.1), (490,27.5)
data/buhnenrennen6.txt	0.000247 Sek.	(18 Lücken) (0,135.45), (70,119.4), (140,119.25), (210,121.45), (280,123.45), (350,106.95), (420,141.45), (490,118.9), (560,118.1), (630,84.6), (700,116.5), (770,126.05), (840,123),(910,131.1), (980,91.1), (1050,71.55), (1120,11.45), (1190,26.1)
data/buhnenrennen7.txt	0.00018 Sek.	(21 Lücken) (0,180.95), (70,173.5), (140,149.4), (210,191.9), (280,187.7), (350,256.5), (420,269.55), (490,175.25), (560,184.5), (630,152.3), (700,91.4), (770,71), (840,111.75), (910,137.95), (980,131.55), (1050,162.85), (1120,173.95), (1190,146.6), (1260,120.5), (1330,108.05), (1400,86.3)
data/buhnenrennen8.txt	0.000317 Sek.	(24 Lücken) (0,243.4), (70,254), (140,260), (210,225.3), (280,209), (350,189.8), (420,245.05), (490,190.6), (560,142.85), (630,186.3), (700,184.9), (770,174.6), (840,156.95), (910,179.75), (980,206.5),(1050,221.35), (1120,305.7), (1190,126.55), (1260,103.75), (1330,129.55), (1400,142.75), (1470,121.75), (1540,244.4), (1610,175.6)
data/buhnenrennen9.txt	0.004837 Sek.	(27 Lücken) (0,134.9), (70,206.3), (140,123.05), (210,137.15), (280,181.1), (350,190.75), (420,233.45), (490,249.2), (560,277.45), (630,310.25), (700,321.15), (770,317.05), (840,302.45), (910,297.5), (980,277.45), (1050,288.2), (1120,289.25), (1190,305.6), (1260,299.55), (1330,308.1), (1400,315.2),

		(1470,285.8), (1540,242.1), (1610,192.95), (1680,194.25), (1750,285.65), (1820,347.85)
data/buhnenrennen10.txt	0.000397 Sek.	(30 Lücken) (0,264.75), (70,287.65), (140,317.4), (210,318.15), (280,268.25), (350,206.5), (420,165.25), (490,282.25), (560,270.75), (630,247.15), (700,151.05), (770,264.65), (840,248), (910,166.45), (980,34.2), (1050,30.15), (1120,64.4), (1190,109.35), (1260,79.2), (1330,57.4), (1400,157.85), (1470,60.7), (1540,79.25), (1610,121.25), (1680,121.8), (1750,184.7), (1820,121.45), (1890,70.65), (1960,112.3), (2030,50.4)
data/buhnenrennen11.txt	0.001391 Sek.	(33 Lücken) (0,40.95), (70,66.35), (140,71.9), (210,91.2), (280,85.95), (350,141.25), (420,105.85), (490,240.3), (560,237), (630,268.6), (700,269.75), (770,273.9), (840,304.95), (910,272.05), (980,283.15), (1050,303.95), (1120,291.25), (1190,304.15), (1260,230.05), (1330,227.1), (1400,116.7), (1470,130.75), (1540,108.75), (1610,70.85), (1680,109.15), (1750,78.55), (1820,43.8), (1890,53.95), (1960,46.15), (2030,54.65), (2100,120.7), (2170,128.85), (2240,101.1)
data/buhnenrennen12.txt	0.000694 Sek.	(36 Lücken) (0,229.6), (70,158.45), (140,178.4), (210,173.15), (280,383.5), (350,437.6), (420,327), (490,197.5), (560,155.8), (630,81.55), (700,160.2), (770,49.85), (840,94.2), (910,71), (980,61.05), (1050,12.95), (1120,21.15), (1190,19.1), (1260,75.15), (1330,90.5), (1400,129.7), (1470,257.45), (1540,294.95), (1610,198.6), (1680,143.75), (1750,192.1), (1820,206.6), (1890,213.85), (1960,150.5), (2030,116.65), (2100,99.95), (2170,108.8), (2240,186.6), (2310,183.8), (2380,200.2), (2450,249.3)
data/buhnenrennen13.txt	0.000486 Sek.	(39 Lücken) (0,24.9), (70,55.4), (140,83.9), (210,12.75), (280,133.65), (350,105.2), (420,174.85), (490,358.85), (560,344.35), (630,320.65), (700,343.35), (770,375.9), (840,442.7), (910,455.4), (980,415.3), (1050,395), (1120,391.4), (1190,422), (1260,431.5), (1330,415.85), (1400,469.45), (1470,462), (1540,443.75), (1610,461.15), (1680,473.2), (1750,328), (1820,380.35), (1890,369.15), (1960,257.1), (2030,241.55), (2100,233), (2170,349.65), (2240,338.25), (2310,347.35), (2380,382.6), (2450,335.15), (2520,349.05), (2590,339), (2660,266.75)
data/buhnenrennen14.txt	0.030429 Sek.	(42 Lücken) (0,257), (70,217.6), (140,128.85), (210,12.6), (280,44.6), (350,137.05), (420,458.05), (490,438.15), (560,361.05), (630,386.8), (700,476.7), (770,476.7),

		(840,451.85), (910,404.55), (980,437), (1050,447.35), (1120,415.4), (1190,405.9), (1260,400.75), (1330,416.6), (1400,359.05), (1470,227.75), (1540,154.75), (1610,176.85), (1680,173.65), (1750,151.1), (1820,131.75), (1890,147.45), (1960,280.25), (2030,226.4), (2100,236.15), (2170,236.65), (2240,227.9), (2310,243.5), (2380,102.85), (2450,78.6), (2520,105.8), (2590,80.7), (2660,92.5), (2730,107.4), (2800,137.15), (2870,39.55)
data/eigen0.txt	6.7e-05 Sek.	(2 Lücken) (0,10.1), (70,9.9) *(BwInf-Beispiel von Webseite)*
data/eigen1.txt	6.7e-05 Sek.	Kein Minnieweg gefunden

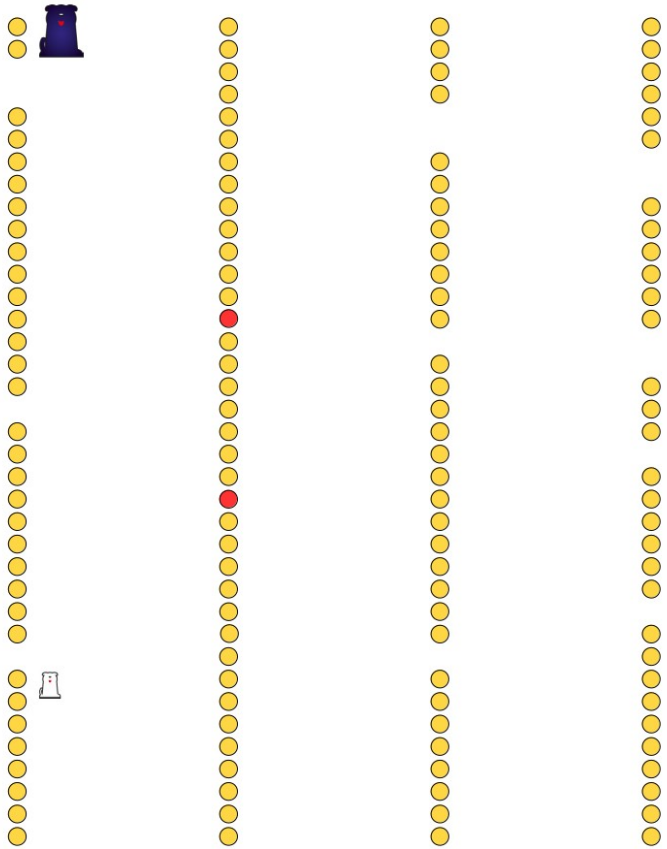

Abbildung zu Eigen1.txt; Veränderte Variante von buhnenrennen1.txt; die unterste Lücke in der zweiten Buhnenreihe wurde entfernt. Alle rot markierten Lücken können vom Max früher erreicht werden als von Mininie.
Da alle Lücken der zweiten Reihe rot sind, gibt es keinen sicheren Minnieweg.

4 Quelltext

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <tuple>
#include <set>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <algorithm>
#include <limits>
using namespace std;

const double INFTY = std::numeric_limits<double>::max();
const double buhnenBreite = 70;
const double speedMax = (double) 30 / 3.6 ;      // 30 km/h ^= 30/3.6 m/s
const double speedMinnie = ( double)20 / 3.6;    // 20 km/h ^= 20/3.6 m/s

typedef pair<int,double> edge; // (target, weight)

struct node
{
   double x,y; // Koordinaten der Lücke
   set<edge> edges; // Ausgehende Kanten
   bool isMini; // ist Minnielücke?

   node( double _x, double _y, set<edge> _edges, bool _isMini )
      : x(_x), y(_y), edges(_edges), isMini(_isMini) {}
};

vector<double> getShortestPathsMax ( const vector<node>& nodes, const int startNode )
{ // modifizierte Variante von Dijkstra's algorithmus für single-source shortest paths
   vector<double> dists (nodes.size(), INFTY );
   dists[startNode] = 0;

   set<pair<double, int>> priority_queue; // [(cost, node)]
   priority_queue.insert( { 0, startNode } );

   while( !priority_queue.empty() )
   {
      int current = priority_queue.begin()->second;
      priority_queue.erase( priority_queue.begin() ); // ^= poll

      if ( nodes[current].isMini )
         continue; // Ausgehende Kanten von Minnnielücken nicht beachten

      for( auto edge : nodes[current].edges )
         if ( dists[edge.first] > dists[current] + edge.second )
```

```
49          { // Durch den aktuellen Knoten wird ein Knoten [edge.first] kürzer erreicht als bisher bekannt
50              priority_queue.erase( { dists[edge.first], edge.first } ); // alten Eintrag aus Queue entfernen, sofern bereits drinnen
51              dists[edge.first] = dists[current] + edge.second;
52              priority_queue.insert( { dists[edge.first], edge.first} );
53          }
54      }
55
56      return dists;
57  }
58
59  double dist ( const node& a, const node& b )
60  { return sqrt( pow(a.x-b.x, 2) + pow(a.y-b.y,2) ); }
61
62  bool dfs(
63      const vector<int> path, // Menge aller bereits besuchten Knoten
64      const int weightSum, // bisherige Summe der Kantengewichte auf diesem Pfad
65      const int lastRow,      // X-Koordinate der letzten Buhnenreihe
66      const vector<node>& nodes, // [Knoten des Graphen
67      const vector<double>& shortestPaths_max, // Kürzeste Wege von Max zu jedem Knoten
68      vector<int>& solutionPath )      // Lösungspfad, in den bei Erfolg geschrieben wird
69  { // rekursive Tiefensuche
70      int current = path.back();
71
72      if ( nodes[current].x == lastRow )
73      { // letzte Buhnenreihe erreicht; Erfolg
74          solutionPath = path;
75          return true;
76      }
77
78      for( auto e : nodes[current].edges )
79          if ( ((weightSum + e.second) / speedMinnie) < (shortestPaths_max[e.first] / speedMax) )
80          { // Minnie kann [e.first] in kürzerer Zeit erreichen als Max
81              vector<int> tempPath = path; tempPath.push_back(e.first);
82              if (
83                  std::find(path.begin(), path.end(), e.first) == path.end() // Zur Zyklenvermeidung. [Undone, da nicht möglich?]
84                  && dfs( // Rekursiver Aufruf
85                      tempPath,
86                      weightSum + e.second,
87                      lastRow,
88                      nodes,
89                      shortestPaths_max,
90                      solutionPath
91                  )
92              ) return true;
93          }
```

```
94
95      return false;
96  }
97
98  int main(int argc, char* argv[])
99  {
100
101     if ( argc == 0 || argc == 1 )
102     {
103         cerr << endl << "Fehler - Kein Dateiname übergeben. 1 Parameter: ↩
                Dateiname.";
104         cout << endl << help_string;
105         return EXIT_FAILURE;
106     }
107
108     string filename = argv[1];
109
110     vector<node> nodes; // Knoten
111
112     int start_minnie, start_max; // IDs der Startknoten der Hunde
113     double mostRightColumn = 0; // X-Koordinate der letzten Buhnenreihe ↩
            bzw. -spalte
114
115     // *** Einlesen der Knoten/Lücken *** //
116     ifstream fin( filename );
117
118     cout << endl << "Lade Datei '" << filename << "'" << endl;
119
120     if (!fin.good())
121         { cerr << "Fehler beim Einlesen der Datendatei '" << filename << ↩
                "'" << endl; exit(0); }
122
123     while( true )
124     {
125         char c; double x,y;
126         fin >> c >> x >> y;
127         if ( fin.eof() ) break;
128
129         if ( c == 'X' ) start_max = nodes.size();
130         if ( c == 'M') start_minnie = nodes.size();
131         mostRightColumn = max( mostRightColumn, x);
132
133         nodes.push_back( node( x, y, set<edge>(), (c == 'm' || c == 'M') ↩
                ) );
134     }
135
136     if (start_max == 0 && start_minnie == 0)
137         cerr << endl << "Warnung - Beide Startpositionen 0; fehler in ↩
                Eingabedatei?";
138
139     if ( mostRightColumn == 0 )
140         cerr << endl << "Warnung - Letzte Buhnenreihe hat X-Koordinate 0 ↩
                (???)";
```

```
141
142     double start_t = clock(); // Laufzeit stoppen
143
144     // *** Erstellen der Kanten *** //
145     for( auto &i : nodes )
146        for( int j = 0; j < nodes.size(); j++ )
147           if ( (i.x + buhnenBreite == nodes[j].x) ) // nächste Buhne ↩
                  rechts oder
148              i.edges.insert( make_pair(j, dist(i, nodes[j]) ) );
149
150     // Ausgeben der Knoten
151     cout << endl << "Startpunkt Minnie:\t(" << nodes[start_minnie].x << ↩
           "," << nodes[start_minnie].y << ")";
152     cout << endl << "Startpunkt Max: \t(" << nodes[start_max].x << "," ↩
           << nodes[start_max].y << ")";
153     cout << endl << "Letzte Buhnenreihe (X):\t" << mostRightColumn;
154
155     cout << endl << "Lücken: " << nodes.size();
156     // outputNodes(nodes);
157
158     // *** Kürzeste Pfade für Max ermitteln *** //
159     vector<double> shortestPaths_max = getShortestPathsMax (nodes, ↩
           start_max);
160
161     cout << endl << endl << "Suche Minnieweg, bitte warten...";
162
163     // *** Tiefensuche nach Pfad für Minnie *** //
164     vector<int> minnieweg; // gefundener Lösungsweg / "Minnieweg"
165
166     dfs( vector<int>(1,start_minnie), 0, mostRightColumn, nodes, ↩
           shortestPaths_max, minnieweg );
167
168     double end_t = clock();
169     cout << ".fertig; " << (double) (end_t - start_t) / CLOCKS_PER_SEC ↩
           << " Sek." << endl;
170
171     // *** Ausgabe des Minnieweges (sofern gefunden) *** //
172     if ( minnieweg.size() != 0 ) {
173        cout << endl << "Minnieweg gefunden: (" << minnieweg.size() << " ↩
              Lücken)" << endl;
174        for( int i = 0; i < minnieweg.size(); i++ )
175           cout << ( i == 0 ? "" : ", " ) << "(" << ↩
                 nodes[minnieweg[i]].x << "," << nodes[minnieweg[i]].y << ")";
176     }else{
177        cout << endl << "Kein Minnieweg gefunden..";
178     }
179     cout << endl;
180     return EXIT_SUCCESS;
181 }
```

Listing 1: 🗀 5▸src▸main.cpp - Quelltext der main.cpp