Mathematisches Praktikum — Sommersemester 2018
Prof. Dr. Arnold Reusken — Sven Gross, Hauke Sass, Johannes Stemick

igpm Institut für Geometrie und Praktische Mathematik | RWTH AACHEN UNIVERSITY

Aufgabe 5

Bearbeitungszeit: drei Wochen (bis Montag, den 25. Juni 2018)

Mathematischer Hintergrund: Graphen, diskrete Optimierung, Dijkstra-Algorithmus, A^*-Algorithmus

Informatischer Hintergrund: dynamische Programmierung, *greedy*-Algorithmen

Elemente von C++: Vererbung, virtuelle Funktionen, abstrakte Basisklassen

Aufgabenstellung

Implementieren Sie den Algorithmus von Dijkstra und den A^*-Algorithmus, um in einem gegebenen Distanzgraphen von einem Knoten aus die (im Sinne einer Kostenfunktion) kürzesten Wege zu den übrigen Knoten (Dijkstra), bzw den kürzesten Weg zu einem beliebigen anderen Knoten (A^*) zu berechnen. Eine abstrakte Basisklasse zur Repräsentation eines Distanzgraphen wird Ihnen zur Verfügung gestellt. Implementieren Sie eine sinnvolle Hierarchie von davon abgeleiteten Klassen, sodass ihre Algorithmen für alle Testbeispiele funktionieren.

Gerichtete Graphen

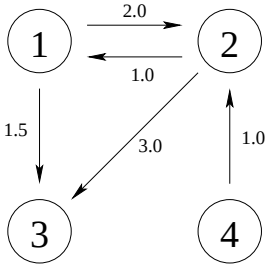
Sei $V = \{1, \dots, N\}$ eine Menge von Knoten ("vertices") und $E \subset V \times V$ eine Menge von (gerichteten) Kanten ("edges"). Das Tupel $G = (V, E)$ heißt *gerichteter Graph*. Eine Kante $e = (v_1, v_2) \in E$ verläuft dabei von Knoten $v_1 \in V$ zu Knoten $v_2 \in V$. Gibt es zu jeder Kante $e = (v_1, v_2) \in E$ auch die entgegengesetzte Kante $\bar{e} = (v_2, v_1) \in E$, so spricht man von einem *ungerichteten Graphen*. Ist ein gerichteter Graph (V, E) mit $E \cap \{(v, v) : v \in V\} = \emptyset$ zusammen mit einer Abbildung $\beta : E \to \mathbb{R}^+$ (Bewertung der Kanten) gegeben, so heißt das Tripel $G = (V, E, \beta)$ ein *Distanzgraph*. Normalerweise wird β auf ganz $V \times V$ fortgesetzt und die Fortsetzung mit $c : V \times V \to \mathbb{R}^+$ ("costs") bezeichnet:

$$c((v_1, v_2)) := \begin{cases} \beta((v_1, v_2)), & \text{falls } (v_1, v_2) \in E, \\ 0, & \text{falls } v_1 = v_2, \\ \infty, & \text{sonst.} \end{cases}$$

Beispiel: Der Distanzgraph $G = (V, E, \beta)$ mit $|V| = 4$ Knoten, der Kantenmenge $E = \{(1,2), (1,3), (2,1), (2,3), (4,2)\}$ und

$$\beta((v_1, v_2)) := \frac{v_2}{|v_1 - v_2|}\ , \qquad (v_1, v_2) \in E$$

kann wie nebenstehend veranschaulicht werden.



Ein Distanzgraph G kann z.B. als Modell für ein Fluglinniennetz dienen: Die Knoten $v \in V$ stehen dann für verschiedene Flughäfen, eine Kante $e \in E$ entspricht einer Fluglinie zwischen den betreffenden Flughäfen, wobei die Bewertung $\beta(e)$ beispielsweise als die durch den Flug entstehenden Kosten oder aber die Entfernung zwischen den Flughäfen interpretiert werden kann. Darüberhinaus sind viele weitere Anwendungsfelder denkbar.

Eine interessante Problemstellung im Zusammenhang mit dem Fluglinienmodell ergibt sich aus der Frage nach den preiswertesten (bzw. kürzesten) Verbindungen von einem bestimmten Flughafen aus. Dies wird auch als "single-source shortest path problem" bezeichnet und soll im Folgenden in einem graphentheoretischen Rahmen definiert werden.

Eine Folge $p = (v_0, v_1, \ldots, v_n) \in V^{n+1}$ wird als *Weg* in G bezeichnet, wenn jeweils $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq n$ gilt (bzw. $v_0 \in V$ im Falle $n = 0$). Die Kosten eines Weges p ergeben sich aus der Summe der Kosten der Kanten, d.h. $c(p) = \sum_{i=1}^{n} c((v_{i-1}, v_i))$. Für zwei Knoten $v, v' \in V$ ist die *Distanz* $d(v, v')$ zwischen v und v' definiert als

$$d(v, v') := \begin{cases} \infty, & \text{falls } P_G(v, v') = \emptyset, \\ \min\{c(p) : p \in P_G(v, v')\}, & \text{sonst,} \end{cases}$$

wobei $P_G(v, v')$ die Menge der Wege von v nach v' in G bezeichne. Das *single-source shortest path problem* lautet nun: *Gegeben ein Knoten v_0 eines Distanzgraphen, wie lauten die Distanzen $d(v_0, v)$ für alle $v \in V$?* Eine Antwort darauf liefert der Algorithmus von Dijkstra. Der A^*-Algorithmus eignet sich eher, wenn der Zielknoten $v_f \in V$ schon bekannt ist und nur der kürzeste Weg von v_0 nach v_f gesucht ist.

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra [2] beruht auf dem Prinzip des greedy-Vorgehens, d.h. wenn im Verlauf des Algorithmus zwischen Alternativen gewählt werden muss, so entscheidet man sich für die zu diesem Zeitpunkt günstigste. Da diese Entscheidung im Folgenden nicht mehr revidiert wird, muss sichergestellt sein, dass diese lokale Optimalitätsbedingung am Ende zu einem globalen Optimum führt. Dies ist für den Algorithmus von Dijkstra tatsächlich der Fall, da es keine Kanten mit negativen Kosten gibt.

Zu jedem Zeitpunkt des Algorithmus hat man bereits eine Menge S von Knoten, für die ein kürzester Weg und damit die Distanz bereits bekannt ist (am Anfang ist dies $S = \{v_0\}$ und $d(v_0, v_0) = 0$). Nun wählt man unter den restlichen Knoten in $R := V \setminus S$ einen aus, dessen vorläufige Distanz minimal ist unter allen Knoten in R (greedy-Auswahl) und fügt ihn zu S hinzu. Anschließend werden die vorläufigen Distanzen für R aufdatiert. Dies wird solange wiederholt, bis $S = V$ ist. Diese Technik, für die Lösung des großen Problems auf die zwischengespeicherten Lösungen von kleineren Teilproblemen zurückzugreifen, wird als *dynamische Programmierung* bezeichnet.

Algorithmus 1 Algorithmus von Dijkstra zur Berechnung von $D[v] = d(v_0, v)$ für alle $v \in V$

```
1:  S := {v_0};
2:  D[v_0] := 0;
3:  for v ∈ V \ S do
4:      D[v] := c((v_0, v));                          ▷ Initialisierung von D
5:  end for
6:  while V \ S ≠ ∅ do
7:      wähle Knoten v_1 in V \ S, für den D[v_1] minimal ist;
8:      S := S ∪ {v_1};
9:      for v ∈ V \ S do
10:         D[v] := min{D[v], D[v_1] + c((v_1, v))};   ▷ Anpassen von D
11:     end for
12: end while
```

Folgende Tabelle gibt den Verlauf des Algorithmus für den im vorigen Beispiel skizzierten Distanzgraphen für $v_0 = 4$ wieder:

Iteration	S	v_1	$D[1]$	$D[2]$	$D[3]$	$D[4]$
0	$\{4\}$	–	∞	1.0	∞	0.0
1	$\{2, 4\}$	2	2.0	1.0	4.0	0.0
2	$\{1, 2, 4\}$	1	2.0	1.0	3.5	0.0
3	$\{1, 2, 3, 4\}$	3	2.0	1.0	3.5	0.0

Für die tatsächliche Implementierung des Algorithmus empfiehlt es sich, mit R statt mit S zu arbeiten, da die for-Schleifen beide über R laufen. Der Algorithmus kann leicht so ergänzt werden, dass nicht nur die Distanzen bestimmt werden, sondern auch der jeweils zugehörige kürzeste Weg mitberechnet wird. Dazu bestimmt man zu jedem Knoten v (außer v_0) den zugehörigen Vorgängerknoten v_{pre} im kürzesten Weg $(v_0, \ldots, v_{\text{pre}}, v)$ von v_0 nach v.

A^*-Algorithmus

Im Gegensatz zum Dijkstra wird beim A^*-Algorithmus [1] der kürzeste Weg zwischen zwei bestimmten Knoten $v_0, v_f \in V$ im Graph gesucht (und nicht wie bei Dijkstra die kürzesten Wege von v_0 zu *allen* Knoten im Graph). Die Idee des Algorithmus besteht darin, immer die Knoten zuerst zu untersuchen, die *vermutlich* am schnellsten zum Zielknoten führen. Dazu wird zusätzliche Information in Form einer Heuristik h herangezogen, die die Kosten zwischen zwei Knoten schätzt. Der A^*-Algorithmus gehört daher zur Klasse der *informierten Suchalgorithmen*.

Im Laufe des Algorithmus wird allen bekannten Knoten v ein Wert $f(v) = g(v) + h(v)$ zugeordnet. Dieser Wert $f(v)$ gibt an, wie lang der Weg von v_0 nach v_f über v im besten Fall ist. Hierbei sind $g(v)$ die bisherigen Kosten vom Start v_0 zu v und $h(v)$ die geschätzten Kosten von v zum Ziel v_f. Wichtig ist, dass die geschätzten Kosten nie die tatsächlichen übersteigen, also stets $h(v) \leq c(v, v_f)$ gilt. Eine solche Heuristik h nennt man *zulässig*. Die Luftlinie zwischen zwei Städten ist z.B. eine geeignete Schätzung für die kürzeste Autostrecke, da diese nicht kürzer als die Luftlinie sein kann. Überschätzt man die tatsächlichen Kosten, läuft der Algorithmus zwar schneller, aber er findet nicht unbedingt eine optimale Lösung.

Ausgehend vom Startknoten wird immer der nächste Knoten mit kleinstem f-Wert betrachtet. Jeder Knoten ist zu jedem Zeitpunkt in eine der drei folgenden Kategorien unterteilt.

- unbekannter Knoten: Knoten wurde noch nie betrachtet; es ist kein Weg bekannt. Außer dem Startknoten sind am Anfang alle unbekannt.
- bekannter Knoten: Knoten wurde schon betrachtet; es ist ein Weg zu ihm bekannt, dieser muss aber nicht optimal sein. Alle bekannten Knoten werden in einer *priority queue* gespeichert. Aus dieser Liste wird immer der Knoten mit kleinstem f-Wert als nächstes ausgewählt.
- abschließend untersuchter Knoten: Kürzester Weg zu diesem Knoten ist bekannt. Nachbarknoten von abschließend untersuchten Knoten werden in die *priority queue* aufgenommen und sind damit bekannt. Ist dieser Nachbarknoten bereits enthalten, werden die geschätzten Kosten dieses Knotens aktualisiert. Alle abschließend untersuchten Knoten müssen auch gespeichert werden, damit sie nicht wieder in die *priority queue* aufgenommen werden.

Damit am Ende der kürzeste Weg zurückverfolgt werden kann, muss zu jedem bekannten und jedem abschließend untersuchten Knoten sein Vorgängerknoten gespeichert werden. Soll ein Nachbar eines abschließend untersuchten Knotens der *priority queue* hinzugefügt werden, existiert dort aber schon, muss dementsprechend sein Vorgängerknoten aktualisiert werden.

Wird der Zielknoten abschließend untersucht, terminiert der Algorithmus, und der optimale Weg kann mittels Vorgängerknoten rekonstruiert werden. Wenn es keine Elemente in der *priority queue* mehr gibt, gibt es keinen Weg von v_0 nach v_f und der Algorithmus bricht ebenfalls ab.

Zur Priority Queue: Eine *priority queue* ist eine Datenstruktur, aus der Sie effizient[1] das Element höchster Priorität (bei uns: mit kleinstem f-Wert) ausgeben und entfernen können, und beliebige Elemente einfügen können. Überlegen Sie, warum die Klasse `std::priority_queue` der Standardbibliothek sich nicht für unsere Zwecke eignet und überlegen Sie sich, wie Sie das Problem umgehen können (z.B. mittels `std::make_heap`).

[1] Laufzeit $\mathcal{O}(\log(n))$ für die beschriebenen Operationen, wobei n die Anzahl der Elemente in der *priority queue* bezeichnet

Datenstruktur

Ihre Algorithmen erhalten als Eingabeparameter einen Graphen der abstrakten Basisklasse `DistanceGraph`, den wir Ihnen in der `unit.h` vorgegeben haben. Sie sollen nun davon abgeleitete Klassen implementieren, die Graphen für verschiedene Anwendungsszenarien repäsentieren können:

- Berechnung von Routen: Die Knoten entsprechen beispielsweise Städten, die Kanten Straßenverbindungen zwischen diesen Städten, wobei die Kantenbewertung die Streckenlänge oder Fahrtzeit angibt.
- Finden von Wegen durch ein Labyrinth: Jede Zelle eines Labyrinth-Gitters wird als Knoten interpretiert, der entweder passierbar ist (bis zu vier Nachbarn, die über Kanten mit dem Knoten verbunden sind) oder ein Stück Mauer darstellt (isoliert, keine Nachbarn). Der Einfachheit halber setzen wir $\beta(e) = 1$ für alle Kanten $e \in E$, alle Wege sind also gleich teuer.

Genaue Beschreibungen der einzelnen Beispiele finden Sie in der `README`-Datei. Überlegen Sie sich an Hand dessen geeignete Heuristiken (Methode `estimatedDistance`) und eine sinnvolle Klassenhierarchie!

Die vorgegebenen rein virtuellen Methoden der abstrakten Basisklasse `DistanceGraph` müssen von Ihnen in den jeweiligen abgeleiteten Klassen implementiert werden. Je nachdem, welcher Art von Graph an Ihre Algorithmen übergeben wird, werden dann die entsprechenden Methoden der jeweiligen abgeleiteten Klasse verwendet. Sie können den abgeleiteten Klassen selber noch Methoden hinzufügen, nur können diese nicht explizit in der Implementierung von A^*/Dijkstra auftauchen, da diese nicht zur Schnittstelle der abstrakten Basisklasse gehören. Vergessen Sie nicht, den Eingabeoperator `operator>>` für die abgeleiteten Graphklassen zu implementieren. Beachten Sie dazu die Dateiformate der Eingabedaten im nächsten Abschnitt.

Schnittstellen

Das Dateiformat der Routengraphen (`Graph1.dat` bis `Graph4.dat`) ist wie folgt festgelegt: Als erstes kommen die Anzahl der Knoten N und die Anzahl der Kanten $|E|$, anschließend folgen für jede Kante e des Graphen der Start- und der Endknoten (also zwei ganze Zahlen zwischen 0 und $N-1$) sowie die Kosten $\beta(e)$. Darunter finden Sie für alle Knoten die geographischen Koordinaten (Längen- und Breitengrad als Dezimalzahl).

Im Falle der Labyrinthgraphen (`Maze1.dat` bis `Maze5.dat`) werden zunächst Höhe h und Breite b des Labyrinths angegeben. Danach folgt das Labyrinth als $h \times b$-Matrix ('`#`' = Wand, '`.`' = passierbar). Die Nummerierung der Knoten erfolgt zeilenweise, d.h., die erste Zeile besteht aus den Knoten $0, 1, \ldots, b-1$, dann folgt die zweite Zeile mit den Knoten $b, b+1, \ldots, 2b-1$, etc.

Mithilfe der Funktion

```
std::vector<CellType> ErzeugeLabyrinth( int breite, int hoehe, unsigned int seed );
```

kann ein zufälliges Labyrinth der Größe `hoehe`×`breite` erzeugt werden. `seed` ist dabei ein beliebiger Wert, um den Zufallszahlengenerator geeignet zu initialisieren, wobei eine feste Wahl von `seed` immer wieder dasselbe zufällige Labyrinth liefert. Der zurückgegebene Vektor der Länge `breite`·`hoehe` enthält das generierte Labyrinth, wobei die ersten `breite` Einträge für die erste Zeile im Labyrinth stehen, die zweiten `breite` Einträge für die zweite Zeile, etc. Die Vektoreinträge sind vom Typ `CellType`, der als `enum class` in der `unit.h` definiert ist und die Enumeratoren `Wall` (Mauer), `Ground` (passierbar), `Start` (passierbarer Startknoten) und `Destination` (passierbarer Zielknoten) enthält. `Start` und `Destination` kommen in einem solchen Labyrinth genau einmal vor.

In der Header-Datei `unit.h` sind die Datentypen `VertexT`, `EdgeT` und `CostT` definiert. Die Konstante `infty` enthält eine (im Vergleich zu den Kosten) sehr große `double`-Zahl und kann daher in Ihren Algorithmen stellvertretend für ∞ verwendet werden. Die Konstante `undefinedVertex` kann als Platzhalter für noch unbekannte `VertexT`-Werte verwendet werden, z.B. falls der Vorgängerknoten noch nicht bekannt ist.

Um Ihre Ergebnisse auf Korrektheit zu überprüfen, stellen wir Ihnen verschiedende Funktionen zur Verfügung. Mithilfe von `PruefeHeuristik` können Sie die für einen bestimmten Graphen implementierte Heuristik

auf Zulässigkeit testen. Die Funktion `PruefeDijkstra` überprüft, ob die durch den Dijkstra-Algorithmus berechneten Distanzen zu allen Knoten im Graphen korrekt sind. Die Funktion `PruefeWeg` stellt fest, ob ein übergebener Weg der kürzeste im Graphen ist. Genauere Beschreibungen zur Syntax und zur Nutzung der Prüfroutinen finden Sie in der Datei `unit.h`.

Kompilation Beachten Sie, dass die Datei `unit.o` mit den Compilerflags `-std=c++11` und `-D_GLIBCXX_USE_CXX11_ABI=0` erstellt wurde. Letzterer Flag stellt die Kompatibilität der Standardbibliotheken unter den g++-Versionen 4.8 und 5.2 unter C++11 her. Nutzen Sie in Ihrer Makefile ebenfalls die entsprechenden Flags, um problemlos mit g++-5.2 zu kompilieren (Dies ist insbesondere im Hinblick auf Aufgabenteil 5b wichtig!).

Testat Am Anfang des Hauptprogrammes soll vom Benutzer eine Beispielnummer zwischen 1 und 10 eingelesen werden. Um das Testat zu bestehen, muss Ihr Programm Folgendes leisten:

- Für die Beispiele 1–4 (`Graph1.dat` bis `Graph4.dat`) sollen mittels Dijkstra von *jedem* Startknoten v_0 die kürzesten Distanzen zu allen Knoten im Graphen korrekt berechnet werden. Sie müssen also in einer Schleife über alle Knoten v_0 variieren und `PruefeDijkstra` aufrufen, ohne dass es zu einer Fehlermeldung kommt.

  Außerdem soll Ihr A^*-Algorithmus für *jede* mögliche Kombination von Start- und Zielknoten korrekt den kürzesten Weg berechnen. Sofern ein Weg existiert, soll jeweils `PruefeWeg` aufgerufen werden und die Korrektheit bestätigen.

- Für die Beispiele 5–9 (`Maze1.dat` bis `Maze5.dat`) soll Ihr A^*-Algorithmus zu bestimmten, von uns vorgegebenen Start- und Zielpunkten den kürzesten Weg finden, d.h. es muss jeweils fehlerfrei `PruefeWeg` aufgerufen werden. Sie erhalten die zu benutzenden Start-Ziel-Paare in einem `std::vector` von der Funktion `StartZielPaare`, siehe `unit.h`. Die Anzahl der Start-Ziel-Paare variiert je nach Beispielnummer.

- Für das Beispiel 10 soll Ihr A^*-Algorithmus in einem zufällig generierten 256×256-Labyrinth den kürzesten Weg finden. Dieser soll wieder mit `PruefeWeg` auf Korrektheit hin überprüft werden.

Visualisierung und Ausblick auf Aufgabe 5b

Im zweiten, noch kommenden Teil der Aufgabe sollen die Arbeitsschritte der im ersten Teil implementierten Graphalgorithmen graphisch visualisiert werden. Dafür stellen wir Ihnen jetzt schon in der Datei `unit.h` eine abstrakte Basisklasse `GraphVisualizer` zur Verfügung. Beim Aufruf des Dijkstra- oder A^*-Algorithmus wird dann zusätzlich ein `GraphVisualizer`-Objekt übergeben, das die Visualisierung übernimmt.

Sie können bereits bei der Bearbeitung des ersten Teils die abgeleitete Klasse `TextVisualizer` nutzen, die Sie in der Header-Datei `text_visualizer.h` finden und die die Aktionen der Graphalgorithmen auf der Konsole kommentiert. Das könnte im Bedarfsfall bei der Fehlersuche recht hilfreich sein.

Literatur

[1] *Wikipedia-Artikel zum A^*-Algorithmus.* `https://en.wikipedia.org/wiki/A*_search_algorithm`.

[2] Cormen, T. H., C. E. Leiserson und R. L. Rivest: *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1994.