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

Aufgabe 5, Teil b)

Bearbeitungszeit: 2.5 Wochen (bis Montag, den 13. Juli 2018)

Mathematischer Hintergrund: Graphen, Koordinatensysteme

Informatischer Hintergrund: Grafikprogrammierung, Arbeit mit Bibliotheken

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

---

Aufgabenstellung

Implementieren Sie Klassen zur Visualisierung des A*-Algorithmus sowohl für die knotenbasierten Routengraphen (`Graph1.dat - Graph4.dat`) als auch für die Labyrinthe (`Maze1.dat - Maze5.dat`, sowie zufällig erzeugte Labyrinthe). Visualisieren Sie Ihre Algorithmen mit Hilfe der SFML-Bibliothek [1], sodass einzelne Schritte des Algorithmus graphisch nachvollzogen werden können. Auch hier steht Ihnen eine Basisklasse zur Verfügung, von der Sie zwei abgeleitete Klassen zur Visualisierung der verschiedenen Graph-Typen implementieren sollen. Benutzen Sie ihre Visualisierung, um diverse Heuristiken auszuprobieren und beobachten Sie das Verhalten des A*-Algorithmus im Hinblick auf die Anzahl sowie die Verteilung der untersuchten Knoten.

Programmbibliotheken

So wie Bibliotheken Sammlungen von Literatur sind, sind Programmbibliotheken Sammlungen von Funktionen beziehungsweise Funktionalitäten, die Ihr Programm nutzen kann. Ein Ihnen bekanntes Beispiel ist die C++-Standardbibliothek, die vom Compiler automatisch eingebunden wird und Ihnen zum Beispiel die Implementationen für `std::string`, mathematische Funktionen und die Standardbiliotheks-Algorithmen bereitstellt. Programmbibliotheken werden ähnlich zu normalen Programmen erstellt (es gibt ebenfalls Kompilations- und Linkerschritte), am Ende des Linkens erstellt der Compiler aber keine ausführbare Datei, sondern eine Bibliotheksdatei, eine so genannte *library*. Deren Inhalt sind alle Funktionen, die im kompilierten Code definiert wurden. Die Bibliotheksdateien enden unter Linux auf `.so` oder `.a`, je nach Art der Bibliothek. Typische `C++`-Libraries bestehen aus einer oder mehrerer solcher Bibliotheksdateien und einer Reihe von zugehörigen Header-Dateien, in denen die von der Bibliothek bereitgestellten Datentypen und Funktionen deklariert werden, so dass Sie diese nutzen können.

Die Simple and Fast Multimedia Library (SFML)

Die `C++`-Library `SFML` ist eine eigentlich auf Spieleprogrammierung ausgerichtete betriebssystemunabhängige Bibliothek, die ihrem Benutzer zum Beispiel Funktionen zur Erstellung und Handhabung von Fenstern, Interaktionen mit dem Betriebssystem, Wiedergabe von Audio, Verarbeitung von Tastatur- und Mauseingaben und hardwarebeschleunigte zweidimensionale grafische Darstellung ermöglicht. Auf der Projektwebseite [1] finden Sie unter *Download* → *SFML-2.5.0* → *Linux* eine bereits kompilierte Version der Bibliothek, die mit den `g++`-Versionen im Rechnerraum kompatibel ist. Entpacken Sie diese am besten in einen Unterordner Ihres Programms — der de-facto Standard unter Linux ist ein Unterordner mit dem Namen `lib`. Instruktionen zum weiteren Setup und zur Benutzung von `SFML` finden Sie unter *Learn* → *Tutorials* → *Getting Started: SFML and Linux*. Bringen Sie als Erstes den von uns zur Verfügung gestellten Beispielcode `sfml_test.cpp` ans Laufen, bevor Sie mit der weiteren Bearbeitung der Aufgabe fortfahren. Weitergehende Informationen finden Sie

in den Tutorials (z.B. *Learn* → *Tutorials* → *Graphics module: Shapes*) sowie in den Klassenbeschreibungen (*Learn* → *API Documentation* → *Classes*).

Arbeiten mit Programmbibliotheken

Um die zugehörigen Header-Dateien einer Bibliothek in Ihrem Programm mittels `#include` einbinden zu können, müssen Sie dem Compiler den entsprechenden Include-Pfad mithilfe der Compileroption `-I`*Pfad* mitteilen. Außerdem müssen beim Linken die benutzten Bibliotheken hinzugefügt werden (Compileroption `-l`*Bibliothek*) sowie der zugehörige Linker-Pfad mitgeteilt werden (Compileroption `-L`*Pfad*), der die Bibliotheksdateien enthält. Im Falle der SFML (hier z.B. im Unterordner `lib/SFML-2.5.0`) sieht das in etwa so aus:

```
g++ -Ilib/SFML-2.5.0/include/ -c a_stern.cpp
g++ a_stern.o unit.o -o prog -Llib/SFML-2.5.0/lib/ -lsfml-graphics -lsfml-window -lsfml-system
```

Um Ihnen das Übersetzen zu erleichtern, haben wir Ihnen ein entsprechendes Makefile zur Verfügung gestellt, das Sie evtl. auf Ihren Bibliothekspfad anpassen müssen. In diesem Makefile befindet sich auch eine Regel `run`, die Ihnen das Ausführen Ihres Programmes mittels `make run` ermöglicht, indem vor Programmaufruf die Umgebungsvariable `LD_LIBRARY_PATH` geeignet gesetzt wird. Das hängt damit zusammen, dass die SFML-Bibliotheksdateien (mit Endung `.so`) dynamisch (d.h., erst zur Laufzeit) gelinkt werden.

Datenstruktur

In der Datei `unit.h` wurde eine abstrakte Basisklasse `GraphVisualizer` zur Verfügung gestellt, von der Sie zwei Klassen zur Visualisierung von Routengraphen bzw. Labyrinthen ableiten und implementieren sollen. Beim Aufruf des A^*-Algorithmus wird dann zusätzlich ein `GraphVisualizer`-Objekt übergeben, das die Visualisierung übernimmt. Dazu ergänzen Sie Ihren A^*-Algorithmus um entsprechende Methodenaufrufe des `GraphVisualizer`-Objekts. Mögliche Visualisierungen sind in Abb. 1 und 2 (Routengraph) bzw. Abb. 3 (Labyrinth) dargestellt, an denen Sie sich orientieren können. Ihrer eigenen Kreativität bei noch schöneren Visualisierungen sind aber natürlich keine Grenzen gesetzt!

Visualisierung

Ihre Visualisierung soll Folgendes leisten:

- annähernde Darstellung der Knoten entsprechendend ihrer Koordinaten (Routengraphen) bzw. Darstellung als Weg oder Wand (Labyrinth),
- unterschiedliche Farbgebung von unbekannten, bekannten und abschließend untersuchten Knoten,
- Markierung des gerade untersuchten Knotens sowie des untersuchten Nachbarknotens aus der Queue,
- Beschriftung der Knoten mit ihren aktuellen g- und h-Werten (bisherige Kosten und geschätzte Kosten des restlichen Weges); diese Beschriftung sollte man auf Wunsch auch komplett ausblenden können (z.B. für große Labyrinthe),
- Darstellung des von Ihrem A*-Algorithmus gefundenen Weges.
- *Zusätzlich* für die Routengraphen (Beispiele 1–4):
  - Darstellung der gerichteten Kanten und Kantenbewertungen,
  - unterschiedliche Farbgebung von unbekannten, besuchten und aktiven Kanten (die das gerade untersuchte Knoten-/Nachbarknoten-Paar verbinden) sowie optimalen Kanten, die auf dem gefundenen Weg liegen.

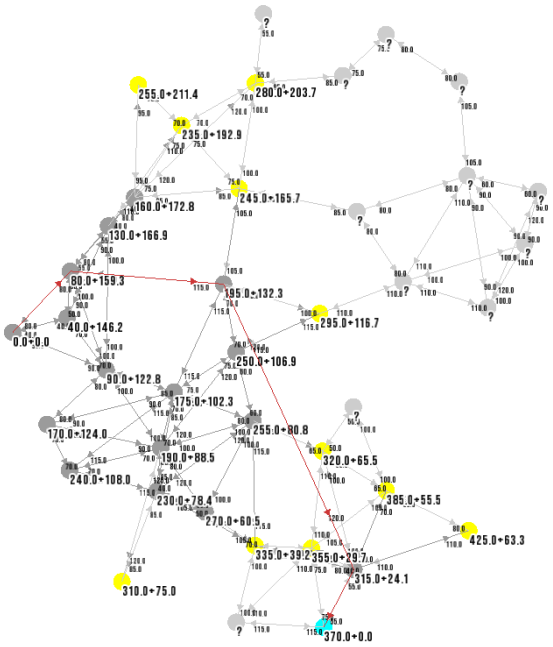

Abbildung 1: Visualisierung des kürzesten Weges von Aachen nach Garmisch-Partenkirchen.

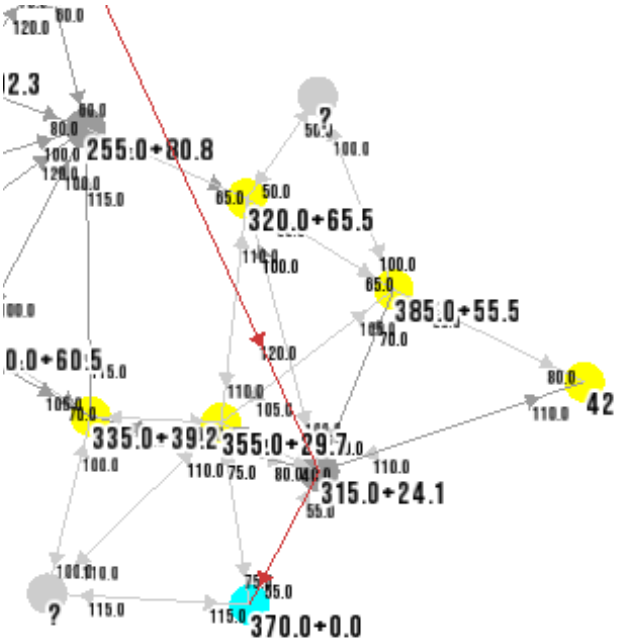

Abbildung 2: Ausschnitt desselben Beispiels (Bayern).

Benutzen Sie Ihre Visualisierung, um verschiedene Heuristiken h im Falle der zufällig erzeugten Labyrinthe auszuprobieren. Untersuchen Sie das Verhalten des A*-Algorithmus für

- $h(v) \equiv 0$,
- $h(v) = \|v_f - v\|_1$,
- $h(v) = \|v_f - v\|_2$,
- $h(v) = C\,\|v_f - v\|_2$ mit $C > 1$.

Welche der Heuristiken sind zulässig und welche nicht? Welches Verhalten des A^*-Algorithmus beobachten Sie im Falle nicht zulässiger Heuristiken?

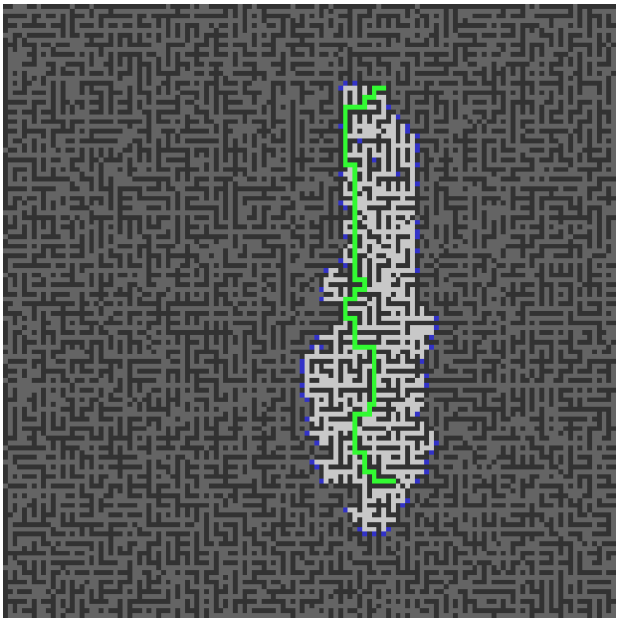

Abbildung 3: Visualisierung des kürzesten Weges in einem zufällig erzeugten Labyrinth.

Hinweise zur Implementierung der Visualisierungsklassen

Verwenden Sie den zur Verfügung gestellten Beispielcode `sfml_test.cpp` mit `Makefile`, um sich mit der SFML-Bibliothek ein wenig vertraut zu machen. Starten Sie dann zunächst mit der Visualisierung für Labyrinthe. Wenn diese erfolgreich funktioniert, wenden Sie sich als Nächstes der (schwierigeren) Visualisierung der Routengraphen zu, bei der auch die Kanten visualisiert werden sollen.

Die abgeleiteten Visualisierungsklassen sollten als Attribute neben einem Grafikfenster-Objekt vom Typ `sf::RenderWindow` und einer Referenz auf den zu visualisierenden Graphen auch den aktuellen Zustand der Knoten und evtl. Kanten speichern. Relevante Zustandsinformationen zu einem Knoten v sind beispielsweise der Status (Enumeration `VertexStatus`) sowie die Werte $g(v)$ und $h(v)$. Für Routengraphen, bei denen auch die Kanten visualisiert werden sollen, sind zu einer Kante e ihr Status (Enumeration `EdgeStatus`) sowie die Kantenbewertung $\beta(e)$ von Belang.

Den Konstruktoren der abgeleiteten Visualisierungsklassen sollen die Breite und Höhe des angeforderten Grafikfensters übergeben werden.[1] Abhängig davon müssen Sie sich dann eine sinnvolle Abbildung der Knotenkoordinaten des Graphen auf die Grafikkoordinaten des Fensters (Ursprung links oben, x nach rechts, y nach unten) überlegen bzw. im Falle der Labyrinthe eine sinnvolle Blockgröße für die Darstellung der Wand-/Weg-Knoten finden.

Die Methoden `markVertex`, `markEdge`, `updateVertex` sollten die jeweiligen Zustände anpassen und evtl. zum Schluss die Methode `draw` aufrufen, um den Graphen mit den modifizierten Zuständen zu zeichnen.

Die Methode `draw` zeichnet den Graphen entsprechend seines aktuellen Zustands. Ihr Code ähnelt daher in gewisser Weise dem des einführenden SFML-Beispiels, da nach einer Event-Schleife die Grafikbefehle zum Zeichnen der Knoten und Kanten folgen, die zusammen von den `sf::RenderWindow`-Methodenaufrufen `clear` und `display` eingeschlossen sind. Zum Zeichnen der Knoten bieten sich die SFML-Klassen `CircleShape` oder `RectangleShape` an. Weitere Hinweise, die auch zum Zeichnen der Kanten hilfreich sind, finden sich im entsprechenden Tutorial (*Graphics module: Shapes*). Um Beschriftungen hinzuzufügen, beachten Sie bitte die Hinweise im Tutorial *Graphics module: Text and fonts*. Wir haben eine freie Schriftart als Fontdatei hinzugefügt, die Sie hierzu verwenden können.

Sollte die Visualisierung zu schnell ablaufen, können Sie bei jedem Aufruf der `draw`-Routine die Ausführung etwas verzögern, indem Sie die Funktion `sf::sleep` verwenden (siehe *Learn* → *API Documentation* → *Modules* → *System module*). Soll umgekehrt die Visualisierung beschleunigt werden (z.B. für sehr große Graphen), können Sie nur in jedem n-ten Aufruf der Methode `draw` tatsächlich eine Grafikausgabe machen, wobei $n \in \mathbb{N}$ entsprechend gewählt wird.

Literatur

[1] *SFML-Library.* `https://www.sfml-dev.org/`.

[1] Für Beispiel 3 (Europa) können Sie ein Fenster der Größe 1300×800 (Breite × Höhe), für Beispiel 4 (Deutschland) ein Fenster der Größe 700×800 verwenden. Für alle anderen Beispiele sind quadratische Fenster, z.B. 1000×1000, sinnvoll.