Hochschule Niederrhein	PE2 / SE2	Dozenten
Fachbereich 03	Routenplanung	Rethmann / Brandt
Bachelor Informatik	SoSe 2021	Stockmanns / Quix

Routenplanung in C++

Im Rahmen dieser zusätzlichen Übung kann sukzessive eine etwas umfangreichere Software implementiert werden. Jede Aufgabe definiert einen kleinen Teil der Software, sodass am Ende daraus ein echtes Produkt zusammengesetzt werden kann. Jeder Teil wird für sich getestet (Komponententest). Die Funktionalität wächst also im Laufe der Zeit, dieses Vorgehen nennt man auch inkrementelle Software-Entwicklung.

Bei diesem Projekt soll eine Karte mittels eines Graphen modelliert werden. Die Knoten repräsentieren die Städte (oder Straßenkreuzungen) und die Kanten die Strecken dazwischen. Weitere Funktionalitäten wie das Finden der kürzesten Wegstrecke, effiziente Möglichkeiten der Integration von Wegänderungen bei Stau und Sperrungen, sowie Streckenplanung mit mehreren Zielpunkten können zusätzlich implementiert werden.

Informationen zu Graphen und Algorithmen zur Routenplanung finden Sie im Modul *Algorithmen und Datenstrukturen* sowie im Web. Wir geben an vielen Stellen in den Aufgaben entsprechende Hinweise.

Aufgabe R1: (Priority-Queue, Vorrangwarteschlange[1])

Erstellen Sie eine Datenstruktur `pqueue_t`, die beliebig viele Zeichenketten speichern kann. Zur internen Kapselung der Zeichenketten und Prioritätswerte, erstellen Sie eine Datenstruktur `pqentry_t`. Verwenden Sie für die Priorität den Datentyp `float`.

Die Datenstruktur soll beliebig viele Datensätze aufnehmen können, d.h. sie muss bei Bedarf automatisch vergrößert werden. Zudem soll die Implementierung einen Zugriff auf die interne Repräsentation verhindern (Datenkapselung).

Implementieren Sie für die Datenstruktur `pqueue_t` die folgenden Operationen:

- `pqueue_t* pq_create();`
  erstellt eine neue, leere Vorrangwarteschlange.
- `void pq_insert(pqueue_t *pq, char *value, float p);`
  fügt den Wert `value` mit der Priorität `p` in die Vorrangwarteschlange `pq` ein.
- `char* pq_extractMin(pqueue_t *pq);`
  liefert den Wert aus der Vorrangwarteschlange `pq` mit höchster Priorität, also mit kleinstem nummerischen Wert der Komponente `priority` und entfernt den Eintrag aus `pq`.

[1] `https://de.wikipedia.org/wiki/Vorrangwarteschlange`

- `void pq_decreaseKey(pqueue_t *pq, char* value, float p);`
  ändert die Priorität des Wertes `value` in der Vorrangwarteschlange `pq` auf den neuen Wert `p`.
- `void pq_remove(pqueue_t *pq, char* value);`
  löscht den Wert `value` aus der Vorrangwarteschlange `pq`.
- `void pq_destroy(pqueue_t *pq);`
  zerstört die Vorrangwarteschlange `pq` und gibt den belegten Speicher wieder frei.

Diese Datenstruktur wird für die Berechnung kürzester Wege mittels des Algorithmus von Dijkstra benötigt. Oft wird ein binärer Heap verwendet, um eine Vorrangwarteschlange zu realisieren.

Aufgabe R2: (Klassen, Exceptions, generische Programmierung)

Implementieren Sie die obige Vorrangwarteschlange aus Aufgabe R1 als Klasse in C++.

- Die Fehlervariable soll durch ein Exception-Handling ersetzt werden.
- Verwenden Sie außerdem die Technik der Templates, damit der Heap Werte von beliebigen Datentypen speichern kann.
- Ersetzen Sie die Funktion `pq_create` durch einen Konstruktor und die Funktion `pq_destroy` durch einen Destruktor.
- Realisieren Sie die Speicherverwaltung mit `new` und `delete`.

Aufgabe R3: (Vererbung, Graphen)

Graphen[2] bestehen aus Knoten und Kanten. Knoten sind als Kreise und Kanten als Pfeile dargestellt. Erstellen Sie hierzu die Klasse `DiGraph` als Hauptklasse für Graphen, sowie die Klassen `Node` und `Edge` für die Knoten und Kanten.

Da ein Graph aus einer beliebigen Anzahl von Knoten und Kanten bestehen kann, muss eine geeignete Datenstruktur zur Speicherung der Objekte gewählt werden. Dünne Graphen, also solche, die nur wenige Kanten enthalten, werden mittels Adjazenzlisten[3] gespeichert, wohingegen dichte Graphen mittels einer Adjazenzmatrix[4] gespeichert werden können. Implementieren Sie zwei von der Klasse `DiGraph` abgeleitete Klassen `SparseGraph` und `DenseGraph` mit der jeweiligen Repräsentation als Adjazenzliste bzw. als Adjazenzmatrix.

Eine Datenstruktur, die sowohl dünne als auch dichte Graphen effizient speichern kann, ist das Adjazenz-Array. Wenn Sie diese Art der Speicherung nutzen, müssen keine von `DiGraph` abgeleiteten Klassen erstellt werden.

[2] `https://de.wikipedia.org/wiki/Graph_(Graphentheorie)`
[3] `https://de.wikipedia.org/wiki/Adjazenzliste`
[4] `https://de.wikipedia.org/wiki/Adjazenzmatrix`

Auf dem GitLab-Server zur Vorlesung finden Sie ein Testprogramm, mit dem Sie Ihre Implementierung testen können. Außerdem finden Sie dort ein Programm zur Ausgabe von Graphen mittels SFML.

Aufgabe R4: (Algorithmus von Dijkstra)

Erweitern Sie Ihr Programm aus Aufgabe R3 um eine Methode, die für einen gegebenen Start- und Endknoten den kürzesten Weg mittels des Algorithmus von Dijkstra[5] berechnet und diesen Weg als Liste von Kanten zurück gibt. Der Prototyp soll wie folgt aussehen:

```
std::list<Edge> dijkstra(std::string start, std::string end)
```

Verwenden Sie als Datenstruktur die in Aufgabe R2 entwickelte Vorrangwarteschlange `pqueue` mit dem Datentyp `Node` als Template.

Ergänzen Sie das Programm um die Möglichkeit eine Reiseroute zu planen, bei der mehrere Way-Points `wayp` auf dem Weg liegen, die in der gegebenen Reihenfolge besucht werden sollen. Implementieren Sie hierzu eine Klasse `Navigator`, die eine Methode `planeTour` bereit stellt, dessen Prototyp wie folgt aussieht:

```
std::list<Edge> planeTour(std::vector<std::string> wayp)
```

Visualisieren Sie die Strecke ähnlich zu Aufgabe R3.

Aufgabe R5: (STL)

Nutzen Sie für Ihre Implementierung des Graphen und des Algorithmus von Dijkstra Klassen der STL wie `std::vector` oder `std::priority_queue`.

[5] `https://de.wikipedia.org/wiki/Dijkstra-Algorithmus`