Praktikum Simulative Untersuchung von Protokollfunktionen des Internet

Aufgabenblatt 5 [1]

Vorbereitung

Aufgabe 1: Sets
Lesen Sie sich zur Vorbereitung bitte den folgenden Text durch. Wenn Sie Fragen haben, stellen Sie diese bitte gleich am Anfang des praktischen Termins, da Sie die beschriebene Datenstruktur zur Lösung der Aufgabe benötigen:

Nachdem Sie schon die Datenstruktur `map` und die Benutzung von Iteratoren zum Traversieren beliebiger Container kennengelernt haben, folgt nun eine weitere Datenstruktur, nämlich das `set`. Ein `set` ist eine (sortierte) Menge von Datenelementen, in der jedes Element nur einmal vorkommt.

Wie auf jeden anderen Container auch, kann man auf ein `set` mit Hilfe von Iteratoren zugreifen, wobei (anders als bei `maps`) der Iterator direkt auf ein Element zeigt und nicht auf ein Paar von Schlüssel und Wert.

Folgende Methoden von `set` dürften für Sie nützlich sein:

- `insert(Element)` fügt ein Element ein, sofern es noch nicht vorhanden ist.
- `erase(Element)` entfernt das übergebene Element aus dem `set`.
- `find(Element)` sucht das übergebene Element im `set` und gibt einen Iterator auf das gefundene Element zurück. Ist das Element nicht vorhanden, gibt die Methode `end()` zurück.
- `begin()` gibt einen Iterator auf das erste Element zurück.
- `end()` gibt einen Iterator zurück, der **hinter** das letzte Element zeigt.
- `count()` gibt die Anzahl der Elemente im `set` zurück.
- `empty()` prüft, ob das `set` leer ist.

Aufgabe 2: Vorrangwarteschlangen
Bei der Bearbeitung der heutigen Aufgaben werden sie außerdem auf die Datenstruktur 'Vorrangwarteschlange' stoßen. Diese haben wir in Form eines Binär-Heaps bereits für sie implementiert. Eine solche Datenstruktur verwaltet Einträge, welche mit Prioritäten (numerischen Werten) versehen sind. Ihre wichtigsten Operationen sind:

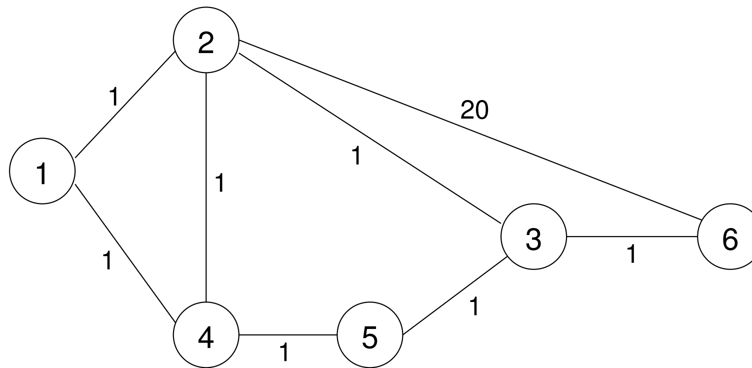
- `insert(Element,Priorität)` fügt ein Element mit gegebener Priorität ein.
- `extractMin()` entfernt das Element mit *geringstem* Prioritätswert und gibt ihn zurück.
- `decreaseKey(Element,neue Priorität)` setzt die Priorität eines bereits verwalteten Elements auf das Minimum des neuen und des alten Prioritätswerts

Zusätzlich dürften auch die folgenden Operationen nützlich sein:

- `contains(Element)` prüft ob das übergebene Element bereits eingefügt wurde.
- `empty()` prüft, ob die Vorrangwarteschlange leer ist.
- `size()` gibt die Anzahl der verwalteten Elemente zurück.

[1] Stand: 12. Oktober 2023

Aufgabe 3: Dijkstra
Führen Sie für das schon bekannte Netzwerk den Dijkstra-Algorithmus für den Quellknoten 1 aus. Notieren Sie für jeden Schritt die Menge der schon behandelten Knoten sowie Kosten und Last Hop für jeden Zielknoten.



Praktischer Teil

Denken Sie bitte – wie immer bei den praktischen Aufgaben – an aussagekräftige Debug-Ausgaben. Die Sourcen für diese Aufgabe befinden sich im Verzeichnis `protsim05`.

Aufgabe 4: Dijkstra
Aufgabe dieses Termins ist es, Link State Routing zu implementieren. Kern dieses Routing-Verfahrens ist der Dijkstra-Algorithmus. Dieser benötigt globales Wissen über das Netzwerk, nämlich die Knoten und die Kosten aller Kanten im Netz.

Für diese erste Aufgabe erkundet der `FakedRoutingDaemon` die Topologie des Netzes auf "magische Weise" ohne den Austausch von Nachrichten.

Zur Erzeugung, Aufbewahrung und Erkundung dieser Topologie dient die Klasse `Topology` und ihre Hilfsklassen `TopologyNode` und `TopologyLink`. Deren Benutzung können Sie sich aus der API-Dokumentation erschließen.

Sobald der Routing Daemon die Topologie erzeugt hat, ermittelt er die günstigsten Routen mit der Funktion `doDijkstra()`. Ihre Aufgabe ist es nun, diese Funktion in der Datei `routing/Dijkstra.cc` zu komplettieren.

Mengen von Knoten (im Programm `Topology::NodeSet`) sind als `set` von Zeigern auf `TopologyNode` implementiert. (Da der Elementtyp ein Zeiger ist, ist das `set` nach Adressen im Speicher sortiert, die bei jedem Durchlauf anders sein können. Da wir aber die Sortierung in unserem Fall nicht benötigen und die Adressen sich ja während des Ablaufs nicht ändern, stellt dies kein Problem da.)

Die Distanzvektoren in dieser Aufgabe sind als `map` mit Schlüsseln vom Typ `TopologyNode*` und Datenelementen vom Typ `double` (nämlich den Kosten) definiert.

Die resultierenden Routen werden in einer `PathMap` gespeichert. Diese ist eine `map` mit Schlüssel `TopologyNode*` und Wert `TopologyLink*`. Sie soll für jeden Zielknoten eine zu wählende Kante des Ausgangsknotens zurückliefern.

Weitere Hinweise stehen wie immer im Quelltext.

Aufgabe 5: Dijkstra
Führen Sie dasselbe Experiment wie beim letzten Mal aus, d.h. erhöhen Sie nach einiger Zeit die Link-Kosten zwischen Knoten 3 und 6 auf 1000. Was ist der wesentliche Unterschied zu Distance Vector Routing und wie kommt es zu diesem Unterschied?

Aufgabe 6: (optional) Link State Routing
Im wahren Leben müssen die Informationen über die Topologie des Netzwerkes natürlich erst durch das Senden und Empfangen von Nachrichten gewonnen werden. Dies geschieht durch das Fluten von Link State Advertisements von jedem Knoten im Netz.

Der `LinkStateRoutingDaemon` ist für das Senden eigener Link State Advertisements und die Erzeugung der Netzwerk-Topologie aus den empfangenen Nachrichten verantwortlich. Letzteres sollen Sie nun implementieren. Ergänzen Sie hierfür bitte die Methode `calculateRoutes()` in der Datei `LinkStateRoutingDaemon.cc`. Die nötigen Link State Advertisements befinden sich in der Link State Advertisement Database `LSAdb`, die eine `map` mit Schlüssel `NodeAddressT` und Wert `LinkStateAdvertisement` ist. Aus letzterem können Sie jeweils die Links eines Knoten mit Kosten und Peer-Knoten erfahren. Auch hier hilft Ihnen wieder die API-Dokumentation.

Wählen Sie Run2 beim Starten von Protsim aus, dabei wird automatisch der LinkStateRoutingDaemon ausgewählt (führen Sie auch ruhig das Experiment aus der vorherigen Aufgabe nochmal aus).