Übung 5 - Flüge Buchen

Nachdem wir Ihnen in der letzten Übung eine beträchtliche Menge neues Material präsentiert und Ihnen damit auch eine große Arbeitsbelastung auferlegt haben, haben wir beschlossen, für diese Übung ein entsprechendes geringeres Lernpensum festzulegen, um Ihnen einen angemessenen Ausgleich zu bieten.
Die Übung behandelt eine bekannte Erweiterung des Dijkstra-Algorithmus, den A*-Algorithmus, der in leicht abgewandelter Form noch immer zur Routenberechnung in Navigationssystemen verwendet wird.
Wir erweitern den Dijkstra Algorithmus um die Möglichkeit "Heuristiken" zusätzlich zu den Kantengewichten zu Nutzen um die Wichtigkeit eines Knotens für die (erwartete) Route zu bestimmen.
Die Intuition hinter diesen Algorithmus besteht darin, dass er zusätzliche Informationen verwendet, um die Suche nach dem optimalen Pfad zu beschleunigen.
Solche zusätzlichen Informationen können zum Beispiel die Richtung zum Ziel sein, wodurch wir Knoten die uns vom Ziel entfernen als unwichtiger Einstufen können.

> *Wenn Sie von Berlin nach Düsseldorf wollen fahren Sie höchstwahrscheinlich nicht über Dresden sondern Hannover, auch wenn der Weg nach Dresden (2h10) kürzer ist als bis Hannover (3h13).*

Eine Heuristik ist eine Schätzung oder Annäherung an die tatsächlichen Kosten oder Entfernungen zwischen den Knoten im Netzwerk. Anstatt alle möglichen Wege zu berechnen und zu vergleichen, verwenden Routing-Algorithmen mit Heuristiken diese Schätzungen, um den Suchraum zu reduzieren und den Aufwand der Routenberechnung zu verringern.
Der Mehrwert von Heuristiken liegt darin, dass sie es ermöglichen, schnellere Entscheidungen zu treffen und den Algorithmus effizienter zu gestalten. Indem sie die Suche auf vielversprechende Wege lenken, können Heuristiken den Algorithmus dazu bringen, sich auf diejenigen Routen zu konzentrieren, die wahrscheinlich die besten Ergebnisse liefern.
Dies führt zu einer erheblichen Reduzierung der Berechnungszeit und ermöglicht es, Routen in Echtzeit zu berechnen, selbst in großen Netzwerken.
Darüber hinaus bieten Heuristiken die Flexibilität, verschiedene Faktoren zu berücksichtigen, die für die Routenberechnung relevant sein können. So können Sie Verkehrsbedingungen, Straßentypen oder andere Einschränkungen in Betracht ziehen, um die beste Route basierend auf den spezifischen Anforderungen des Benutzers zu finden.
Oftmals möchte man das die Laufzeit einer Heuristik in $O(1)$ liegt damit Sie in der Gesamtanalyse nicht ins Gewicht fällt.
Die Daten die Sie in den Folgeaufgaben verarbeiten, enstprechend dem weltweiten Flugnetz, also allen Flughäfen und deren Verbindungen untereinander.

Wir haben Ihnen relevante Klassen aus der VL in `vlib` hinterlegt - sozusagen Ihre im Laufe der VL eigens konstruierte `stdlib` - nutzen und erweitern Sie diese gerne wo immer sinnvoll.

Gutes gelingen!

Aufgabe 1: Datenverarbeitung

> *Dateien die in dieser Aufgabe editiert werden sollen:*
>
> - *weighted_struct_graph.h*
> - *load_airports.cpp*

In dieser Aufgabe erweitern Sie die Symbol-Digraphen Implementierung, welche Sie in der VL kennen gelernt haben, um die Fähigkeit in den Nodes beliebige Attribute, in beliebiger aber zur Compilezeit fester Form, zu speichern.
Zusätzlich erweitern wir den Symbol-DiGraph um Kantengewichte. Die Nodes sollen über Structs implementiert werden die der jeweilige Verwender des Graphen definieren muss. Diese erweiterte Implementierung nutzen Sie dann um die Daten aus `data/airports.csv` und `data/routes.csv` als Graph einzulesen.

Der `WeightedStructDigraph` wird als constructor einen `std::vector` von Structs entgegennehmen, welche genutzt werden um die Knoten des Graphen zu initialisieren.

Kanten werden nach construction über eine `add_edge` Funktion realisiert welche die entsprechende Funktion auf dem inneren Graphen aufruft. `add_edge` bekommt als Argumente die beiden Knoten, sowie das Gewicht der Kante zwischen den beiden.
Wir werden auch hier wieder `template`s nutzen um die gewünschte Generalität zu erreichen.
Lassen Sie sich bei der Implementierung stark vom Symbol-DiGraphen inspirieren und bieten Sie die selben Operatoren an. Weiterhin möchten wir gerne die aus der VL bekannten `Edge` Klasse für unsere gewichteten Kanten wiederverwenden. Dazu sollen Sie den `EdgeWeightedDigraph` wiederverwenden.

> *Hinweis: Ihre Structs müssen für einen RBTree die Operatoren `<,>,=` implementieren.*

Aufgabe 1.1 (2p): Implementieren sie in `weighted_struct_graph.h` den neuen Graphtyp.

Aufgabe 1.2 (4p): Lesen Sie die Daten aus `data/airports.csv` als Knoten ihres Graphen ein, speichern Sie relevante Attribute im Knoten. Lesen Sie weiterhin die Daten aus `data/routes.csv` als Kanten Ihres Graphen ein, nutzen Sie die Spalte `Time` als Kantengewicht.
Implementieren Sie das Laden des Graphen in `load_airports.cpp::load_data`. Dabei bekommt die Funktion als Argumente die Dateipfade für die Flughäfen (`airports.csv`) und die Routen (`routes.csv`) als jeweils erstes und zweites Argument.

> *Hinweis: Die Struktur der Daten ist in `DATASOURCES.pdf` beschrieben.*

Aufgabe 2: A* Algorithmus

> *Dateien die in dieser Aufgabe editiert werden sollen:*
>
> - *a_star.cpp*
> - *distances.cpp*
> - *routing_cli.cpp*

In dieser Aufgabe werden Sie den *A*-Algorithmus* implementieren, der Ihre Implementierung des `WeightedStructDiGraph` verarbeiten wird.

Die Idee hinter dem A*-Algorithmus besteht darin, dass wir oft mehr Informationen über den zu durchsuchenden Graphen haben als nur seine Nachbarschaft.
In unserem Fall wissen wir zum Beispiel, dass Flughäfen auf der Erde einen festen Ort haben und dass ein Flug von Berlin nach London höchstwahrscheinlich nicht über Dubai führen wird, da Dubai in einer komplett anderen Himmelsrichtung liegt.
Der A*-Algorithmus nutzt diese zusätzlichen Informationen in Form einer Heuristik, um die Kosten für den Gesamtpfad vom Startknoten zum Zielknoten in jedem Schritt abzuschätzen.
Dabei erweitert er den aus der Vorlesung bekannten Dijkstra-Algorithmus, indem er bei der Prioritätswarteschlange nicht nur die Kosten des Pfades zu den enthaltenen Knoten berücksichtigt, sondern auch die Abschätzung der Distanz zum Endknoten für jeden Knoten hinzufügt.

Er kombiniert dadurch die Vorzüge des Dijkstra-Algorithmus für Pfadsuche, nämlich das immer der optimale Pfad gefunden wird, mit den Vorzügen der Greedy Beste-Zuerst Suche, welche deutlich schneller ist, aber falsche Ergebnisse liefern kann.
Wenn die genutzte Heuristik einige Anforderungen erfüllt, ist die Optimalität der Pfade garantiert:

- Die Heuristik muss *zulässig* sein: Sie darf die Kosten nie Überschätzen, ansonsten leidet die Laufzeit massiv.
- Die Heuristik soll *konsistent* sein: Für jeden Knoten v und jeden Nachfolgeknoten v' von v gilt $h(v) \leq C(v, v') + h(v')$. Hierbei bezeichnet $C(v, v')$ die tatsächlichen Kosten, um von v nach v' zu kommen

> *Es ist möglich eine zulässige aber nicht konsistente Heuristik zu nutzen, aber unsere angestrebte Implementierung würde in diesem Falle inkorrekte Ergebnisse liefern.*
> *Im Austausch gegen Laufzeit könnte man allerdings auch hier einen korrekten Algorithmus verfassen.*

Ablauf Des A*-Agorithmus

Eingabe:

- ein gerichteter gewichteter Graph mit ausschließlich positiven Kantengewichten $G = (V, E)$
- ein Startknoten $v_S \in V$
- ein Zielknoten $v_Z \in V$
- eine Heuristik $h(v) : V \to \mathbb{R}$

Ausgabe:

- den kürzesten Pfad $P_s = (v_s, \ldots, v_z)$ von v_s nach v_z als Liste an Knoten, wenn Pfad existiert
- ein leerer Pfad $P_s = ()$, sonst

Ablauf:

1. Initialisiere eine Menge V_O aller *offenen Knoten*, mit dem Startknoten als einziges Element
2. Initialisiere eine leere Menge V_C aller *geschlossener Knoten*
3. Initialisiere eine Map $C : V \to \mathbb{R}$, welche jedem Knoten eine Zahl zuordnet die den Kosten des Knotens entspricht.
   - setze die Kosten aller Knoten auf $+\infty$, außer für den Startknoten dieser hat Kosten $C(v_s) = 0$
4. Initialisiere eine Map $P : V \to V$, in welcher gespeichert wird welcher Knoten der Elternknoten eines Knoten ist.
   - Hinweis: eine Map zwischen den Knoten-Indizes
5. Solange: $V_O \neq \emptyset$
   1. v_i sei der *offene* Knoten mit den geringsten Kosten in $C(\_) + h(\_)$
      - also der Knoten mit der geringsten Summe aus Kosten um den Knoten zu erreichen und erwartete Kosten von dem Knoten zu v_z
      - Hinweis: Sie können eine IndexMinPQ nutzen um die Selektion effizient zu gestalten.
      - Falls v_i der Zielknoten v_z ist beende und gebe den Pfad von v_s nach v_z aus P zurück.
   2. füge v_i in V_C ein
   3. für jeden Nachbar n_k von v_i:
      - wenn $n_k \in V_C$, dann fahre mit der Schleife fort
      - berechne die Kosten für den Pfad $(v_s, \ldots, v_i, n_k)$
      - wenn die Kosten geringer sind als die in C hinterlegten Kosten:
        - update die Kosten für n_k in C
        - setze $P(n_k) = v_i$, also v_i als Elternknoten von n_k
        - füge n_k in V_O ein
6. Falls v_z *nicht* erreicht wurde gebe einen leeren Pfad zurück

Aufgabe 2.1 (6p): Implementieren sie den A*-Algorithmus entsprechend dem beschriebenen Ablauf in `a_star.cpp`.
Nutzen Sie als Heuristik in dieser Aufgabe die euklidische Distanz die sie in `distances.cpp::euclidean` implementieren.
Beachten Sie die Hinweise zu Heuristiken um sicherzustellen das der Algorithmus korrekt terminiert.

> *Hinweis: Die Kantenlängen sind in h angegeben. Die Örtlichkeit und Distanz zwischen Flughäfen wäre aber in km wie können Sie die notwendigen Eigenschaften herstellen?*

Aufgabe 2.2 (4p): Schreiben Sie in `routing_cli.cpp` ein CLI welches als Argumente zwei IATA-Codes von Flughäfen bekommt und die schnellste Verbindung zwischen den Flughäfen in unseren Daten findet.

- die erste Zeile ist der Pfad als Kette von IATA-Codes, getrennt durch `-`.
  - falls der Pfad nicht existiert soll die Ausgabe ein `-` in dieser Zeile sein.
- die Zweite Zeile ist die Reisedauer in `h`.
  - falls der Pfad nicht existiert soll die Ausgabe `inf` für die Reisedauer sein

Beispiel (CLI):

```
$ ./out/routing LAX TXL                                    language-console
```

Beispiel (Pfad existiert):

```
LAX-LDN-TXL                                                    language-console
12.50
```

Beispiel (Pfad existiert nicht):

```
-                                                              language-console
inf
```

> *Hinweis: Sie könne eine Map nutzen um IATA Codes den Vertex Indices zuzuweisen, oder auch die IATA codes anstatt der OpenflightsID als Schlüssel im RBTree nutzen.*

Aufgabe 3: Andere Distanzen & Heuristiken

> *Dateien die in dieser Aufgabe editiert werden sollen:*
>
> - *distances.cpp*
> - *A3.md*

In dieser Aufgabe wollen wir unsere Implementierung von A* dadurch ausbauen,
dass wir andere Heuristiken hinzunehmen und diese dann untereinander Vergleichen.
Das euklidische Distanzmaß ist eine einfache und effektive Heuristik für unseren Anwendungsfall,
hat aber den Nachteil das sie annimmt das alle Flughäfen auf einer Ebene liegen.
Wie wir wissen ist die Erde nicht flach, sondern ein Ellipsoid (eine gedehnte Kugel) und es ist auch nicht möglich die Oberfläche einer Kugel auf die flache Ebene ohne Verzerrung zu projezieren.
Ein vermeintlich besseres Distanzmaß ist die sogenannte Geodesische Distanz, welche die Distanz als Pfad über die Oberfläche eines Ellipsoids berechnet, diese sollen Sie über die Haversine Formel berechnen.

Dieses vermeintlich bessere Distanzmaß wollen wir mit zwei weiteren Vergleichen:
Der Manhattan Distanz, auch Taxifahrerdistanz oder L1-Norm. Sowie einer Heuristik die Knoten anhand Ihres out-degrees bevorzugt denn je höher der out-degree desto besser, da wir so zu zentralen Hubs fliegen können die mehr mögliche Verbindungen abbilden.

Aufgabe 3.1 (2p): Implementieren Sie die verschiedenen Distanzmaße wie beschrieben in den entsprechenden Funktionen in `distances.cpp`.

Aufgabe 3.2 (2p): Vergleichen Sie die Laufzeit und Korrektheit von verschiedenen Heuristiken basierend auf den Distanzmaßen.
Vergleichen Sie jeweils die Laufzeit für verschiedene existierende Pfade und Pfade die im Graphen nicht existieren.
Versuchen Sie Beispiele für Pfade zu finden in denen die Heuristiken falsche Ergebnisse liefern.
Notieren Sie ihre Ergebnisse in `A3.md`, im Stammverzeichnis ihres Repositories.

> *Hinweis: Sie können sich mit Hilfe ihres CLI eine Sammlung von interessanten Pfaden zusammenstellen.*

> *Hinweis: Denken Sie daran das die Heuristiken zulässig sein sollen.*

> *Hinweis: Die out-degree Distanz ist nicht konsistent, können Sie einen Fall generieren wo der gefundene Weg nicht der Kürzeste ist?*

> *Hinweis: Die Zeit können sie auf Linux / MacOS mit Hilfe des Kommandozeilenbefehls `time` messen, oder innerhalb ihres Codes mit Hilfe von `std::chrono`.*