**DAP2 Praktikum für ETIT und IKT, SoSe 2020, Langaufgabe L5**

Fällig am 13.07. um 14:00

Es gelten üblichen Programmierregeln in der gewohnten Härte. Hässlicher Code wird bestraft. `assertions` können Sie weglassen.

- Implementieren Sie den Bellman–Ford-Algorithmus, um den kürzesten Pfad von einem Punkt zu allen anderen Punkten in einem Graph zu finden.
- Implementieren Sie ebenfalls den Floyd–Warshall-Algoritmus, um alle kürzesten Pfade zwischen den Knoten eines Graphen zu finden.
- Implementieren Sie den Dijkstra-Algorithmus um den kürzesten Pfad von Start zum Ziel durch den Graphen zu finden.

Alle diese von Ihnen zu implementierenden Funktionen liefern nicht den Pfad selbst zurück, sondern nur die minimalen Kosten (eben den kürzesten Pfad) durch den Graphen. Dadurch wird es für Sie einfacher.

**Beachten Sie folgende Anmerkungen:**

- Verwenden Sie das Gerüst `FromTo.cpp` . In diesem Gerüst sind bereits alle Vorbereitungen zum Laden und Handhaben eines Graphen vorhanden. Außerdem wird die Kommandozeile untersucht, der Graph geladen, die entsprechende Funktion aufgerufen (Bellman-Ford, Floyd-Warshall, Dijkstra), alle Ausgaben gemacht, sowie die Exceptions vom Typ `const char*` abgefangen und ausgegeben.

- Ihre zu Implementierenden Funktionen dürfen keine Ausgaben machen und sie dürfen das Programm nicht beenden. Sie können im Fehlerfall eine Exception vom Typ `const char*` erzeugen.

- Alle `Nodes` und `Edges` des Graphen sind in einem Objekt vom Typ `Graph` abgelegt. Der Graph wird für Sie vom Programmgerüst automatisch als gerichteter Graph geladen und an die zu implementierenden Funktionen als Referenz übergeben.

- Folgende Funktionen des Graphen sind von Interesse:

  `int NumberOfNodes()`
  Liefert Anzahl der Knoten im Graph.

  `int NumberOfEdges()`
  Liefert Anzahl der Kanten im Graph.

  `Edge & GiveEdge(const size_t EdgeIndex)`
  Liefert eine Referenz auf eine Kante zurück. Es wird die Kante mit der Nummer `EdgeIndex` zurückgeliefert. `EdgeIndex` darf zwischen 0 und `NumberOfEdges()-1` liegen.

  `Node & GiveNode(const size_t NodeIndex)`
  Liefert eine Referenz auf einen Knoten zurück. Es wird der Knoten mit der Nummer

`NodeIndex` zurückgeliefert. `NodeIndex` darf zwischen 0 und `NumberOfNodes()-1` liegen.

- Knoten sind Objekte vom Typ `Node`. Beim Programmieren können Sie keine `Nodes` selber erzeugen. Sie sind nur über die Referenzen des Graphen zugänglich. Über einen `Node` stehen Ihnen folgende Operationen zur Verfügung:

  `size_t NumberEdgesOut()`
  : Liefert Ihnen die Anzahl der ausgehenden Kanten dieses Knotens zurück.

  `size_t NumberEdgesIn()`
  : Liefert Ihnen die Anzahl der hereinkommenden Kanten dieses Knotens zurück.

  `size_t IndexEdgeOut( size_t const WhichOne )`
  : Liefert Ihnen den Index der ausgehenden Kante in der Kantenliste des Graphen zurück. `WhichOne` bezieht sich auf die Nummer der Kante des betrachteten Knotens und darf zwischen 0 und `NumberEdgesOut()-1` des Knotens liegen.

  `size_t IndexEdgeIn( size_t const WhichOne )`
  : Dies ist die selbe Funktion für die eingehenden Kanten des Knotens.

- Knoten sind durch Kanten vom Type `Edge` verbunden. Beim Programmieren können Sie keine `Egdes` selber erzeugen. Sie sind nur über die Referenzen des Graphen zugänglich. Über eine `Edge` stehen Ihnen folgende Funktionen zur Verfügung:

  `size_t Source()`
  : Index des Startknotens der Kante in der globalen Knotelis-te des Graphen Source kann zwischen 0 und `NumberOfNodes()-1` des Graphen liegen.

  `size_t Target()`
  : Die selbe Funktion für den Endknoten der Kante.

  `double Cost()`
  : Liefert die Kosten der Kante zurück.

- Bei der Implementierung von Dijkstra müssen die Kosten der Abstände der Knoten in einer Min-Heap gespeichert werden. Eine solche (`class PriorityQueue`) ist bereits vorgefertigt. Sie können gerne Ihrer eigene Min-Heap implementieren, oder eine andere verwenden. Die Verwendung einer geeigneten Min-Heap im Dijkstra ist aber Pflicht. Folgende Funktionen stellt die vorhandene Min-Heap zur Verfügung:

  `PriorityQueue()`
  : erzeugt Ihnen die Min-Heap.

  `bool IsEmpty()`
  : zeigt an, ob die Min-Heap leer ist.

  `void Insert(double Priority)`
  : fügt einen Eintrag mit der gegebenen Priority ein. Diese Einträge sind später über die Reihenfolge des Einfügens indizierbar. Das erste Eintrag hat den Index 0, der zweite

1, und so weiter. Es macht Sinn, die zu den Knoten des Graphen gehörigen Prioritätswerte in der gleichen Reihenfolge wie die Indices der Nodes im Graphen in die Min-Heap einzufügen. Auf diese Weise kann man in beiden Strukturen den gleichen Index verwenden.

`double GetPriority(size_t Entry)`

liefert Ihnen für den Index `Entry` die aktuelle Priority zurück. Der `Entry` entspricht der Reihenfolge nachdem die Einträge in den Min-Heap erfolgt sind. Wenn dieser Eintrag schon über `ExtractMinimum` aus dem Min-Heap herausgelesen wurde, ist das Ergebnis nicht mehr gültig.

`void DecreasePriority(size_t Entry, double Priority)`

Diese Funktion kann die `Priority` des Eintrages `Entry` verkleinern. Vergrößern ist nicht erlaubt.

`void ExtractMinimum(double &Priority, size_t &Entry)`

liest und entfernt den Eintrag `Entry` mit der kleinsten `Priority`. Es wird die `Priority` und der `Entry` des Eintrages zurückgeliefert. Nach der ersten Anwendung dieser Funktion können keine weiteren Einträge in den Min-Heap mehr vorgenommen werden.

- Sie dürfen keine neuen Objekte vom Typ `Node`, `Edge` oder gar `Graph` erzeugen, sondern im Bedarfsfall nur die übergebenen Referenzen auf die bestehenden Objekte verwenden.
- Verwenden die Konstante `infinity` als unendliche Entfernung zu einem Knoten.
- Da der Graph als gerichteter Graph abgelegt wird, ist die Möglichkeit vorhanden, schon beim Einladen des Graphen aus der Datei aus jedem gerichteten Graphen mit der Option `-u` einen ungerichteten Graphen zu erzeugen. In diesem Fall wird einfach jede Kante beim Einladen verdoppelt und in jede der beiden Richtungen eingefügt. Damit ist der Graph dann wieder ungerichtet. Ob dies vorgenommen wird, kann im Framework über die Optionen `-d` und `-u` gesteuert werden.
- Wenn Sie einen Graphen laden, der sowohl Distanzinformation als auch Reisezeiten beinhaltet, werden Ihre Algorithmen zweimal hintereinander aufgerufen. Die Funktion `Cost()` wird dabei ohne Ihr Zutun beim ersten Aufruf Distanzen liefern, beim zweiten Aufruf Reisezeiten. Dies ist für Ihren Algorithmus unerheblich. Ignorieren Sie das einfach, und freuen Sie sich, dass Ihr Algorithmus doppelt so häufig benutzt wird.
- Das Gerüst `FromTo` gibt eine Bedienungsanleitung beim Aufruf ohne Parameter aus.
- Es sind verschiedene Graphen im TIGER-Format als Roadmap vorhanden. Diese werden Ihnen automatisch vom Gerüst geladen und als Graph übergeben. Der Namen der Roadmap muss daher als Aufrufparameter übergeben werden. Das Format ist für Sie von keinem Interesse.
- Graphen, die als Namensbestandteil „`Ge`“ beinhalten sollten mit der Option `-d` aufgerufen werden, um tatsächlich als gerichteter Graph interpretiert zu werden.
- Dateien mit dem Namensbestandteil „`Un`“ beinhalten ungerichtete Graphen. Diese sollten mit `-u` geladen werden.

- Die Datei `BspGraphGeKleinSplit.tiger` beinhaltet einen gerichteten Graph, der in zwei Teile zerfällt.
- Dateien mit „`Neg`“ im Namensbestandteil beinhalten negative Zyklen.
- Die Datei `AK.tiger` ist möglicherweise zu groß für Ihren Speicher.

**Beschreibung der zu implementierenden Funktionen**

- Implementieren Sie folgende Funktionen:

```
void BellmanFord( size_t Start, Graph &TheGraph,
                  vector<double> &Result,
                  bool CheckNegativeCycles)

void APSP(  const Graph &TheGraph, vec
            vector< vector<double> > &Result)

double Dijkstra(  size_t Start, size_t Destination,
                  Graph &TheGraph)
```

- `Start` ist der Zeiger auf den Startknoten des zu suchenden Pfades. Destination ist der Endknoten.
- In `TheGraph` sind alle Knoten und Kanten gespeichert.
- `Result` beinhaltet die von Ihnen berechneten Pfadlängen. Dies ist bei `BellmanFord` ein eindimensionaler `vector`. Speicher in passender Größe ist bereits vorhanden. Bei `APSP` ist dies eine quadratische Matrix aus `vector`. Auch hier ist Speicher in passender Größe bereits vorhanden.
- `CheckNegativeCycles` bewirkt, dass sie bei `true` den Graphen auf negative Zyklen überprüfen müssen. Ist das der Fall, erzeugen Sie eine `Exception` vom Typ `const char *`.

- `BellmanFord` berechnet alle Pfadkosten vom Startknoten zu allen anderen Knoten und legt die Berechnung Knotenweise nach den Knotenindices sortiert in `Result` ab.
- `APSP` berechnet alle Pfadkosten von allen Knoten zu allen anderen Knoten und legt die Berechnung knotenweise nach den Knotenindices sortiert der Matrix `Result` ab. Verwenden Sie den Floyd-Warshall-Algorithmus.
- `Dijkstra` berechnet die minimalen Pfadkosten vom Start zum Ziel durch den Graphen.