1 Lernziele

Allgemein Es soll eine Problemstellung aus dem Bereich der effizienten Algorithmen eigenständig in einer Gruppe bearbeitet werden. Dazu ist eine Einarbeitung in das Themengebiet und das Erstellen eines Projektplans notwendig. Die Algorithmen und Datenstrukturen sollen implementiert und in einem konkreten Anwendungsfall analysiert werden. Dabei ist das Problem mit Fachwissen unter Anwendung geeigneter Methoden und Verfahren entsprechend des aktuellen Stands der Technik zu lösen, das Projekt zu planen, durchzuführen, abzuschließen und zu dokumentieren.

Offene Bibliothek Die Boost-Graph-Library[1] stellt Container bereit, um Graphen zu definieren, und stellt Algorithmen bereit, um Probleme wie *kürzeste Wege* oder *minimaler Spannbaum* sehr effizient zu lösen. Wir wollen in diesem Projekt eine stark vereinfachte Bibliothek entwerfen, die in Modulen wie ALD oder PE2 des Bachelor-Studiengangs Informatik eingesetzt werden kann, um grundlegende Kenntnisse der Programmierung in C++ und auch Wissen über Graphalgorithmen zu vermitteln. Außerdem sollen die Algorithmen visualisiert werden, es soll also möglich sein, Graphen aus einer Datei zu lesen, einen Algorithmus auszuwählen und schrittweise anzeigen zu lassen. Ein schönes Beispiel dafür ist das Projekt ViGrAl[2].

2 Graphalgorithmen

Im Vorfeld wurde bereits eine Bibliothek zum Speichern von Graphen erstellt, im Listing 1 ist die Schnittstelle `IGraph` dargestellt, die alle konkreten Graph-Klassen implementieren müssen. Es sind Algorithmen wie Shortest-Path nach Dijkstra, Minimum-Spanning-Tree nach Kruskal, Max-Flow nach Goldberg-Tarjan (push relabel) und Depth-First-Search implementiert.

Listing 1: Interface IGraph zum Speichern von Graphen

```
template <class K, template <class> class Edge>
class IGraph {
public:
    virtual void addNode(K node) = 0;
    virtual void addEdge(Edge<K> edg) = 0;
    virtual size_t noOfNodes() = 0;
    virtual size_t noOfEdges() = 0;
    virtual size_t noEdgesOf(K node) = 0;
    virtual bool contains(Edge<K> e) = 0;
    virtual Edge<K> getEdge(K n1, K n2) = 0;
    virtual std::vector<K> getNodes() = 0;
    virtual std::vector<Edge<K>> getEdges() = 0;
    virtual bool adjacent(K n1, K n2) = 0;
    virtual std::vector<K> getNeighbors(K node) = 0;
    virtual std::vector<Edge<K>> getEdgesOf(K node) = 0;
};
```

[1] https://www.boost.org/doc/libs/1_77_0/libs/graph/doc/index.html
[2] https://github.com/sischi/vigral

Um Graphen speichern und einlesen zu können, haben wir eine Schnittstelle `IGraphSerializer` definiert, die konkrete Serializer-Klassen implementieren müssen, siehe Listing 2. Dabei ist ein sehr grundlegendes Problem zu lösen: Knoten sind in unseren Graphen einfach nur Zahlen (fortlaufend) und Kanten sind einfach nur Tupel aus zwei Zahlen. In einer Anwendung haben Knoten und Kanten aber in der Regel weitere Attribute wie Name, Farbe oder Form. Unsere Lösung nutzt zwei Klassen `NodeProps` und `EdgeProps` als Parameter der Schnittstelle `IGraphSerializer`. Jede Node- und Edge-Property-Klasse stellt eine Methode `getProps` und einen speziellen Konstruktor zur Verfügung. Die Methode `getProps` liefert eine `map`, in der zu jedem Attribut-Namen der Wert, kodiert als `string`, enthält. Der Konstruktor nimmt eine solche `map` entgegen und weist den jeweiligen Attributen den korrekten Wert zu.

Listing 2: Interface IGraphSerializer zum (De-)Serialisieren von Graphen

```
template <class K, template<class> class Edge, class NodeProps,
          class EdgeProps>
struct IGraphSerializer {
    virtual void save(IGraph<K, Edge> *g,
          std::unordered_map<K, NodeProps> *np,
          std::unordered_map<Edge<K>, EdgeProps,
                  typename Edge<K>::Hasher,
                  typename Edge<K>::EqComp> *ep) = 0;
    virtual void read(IGraph<K, Edge> *g,
          std::unordered_map<K, NodeProps> *np,
          std::unordered_map<Edge<K>, EdgeProps,
                  typename Edge<K>::Hasher,
                  typename Edge<K>::EqComp> *ep) = 0;
};
```

Die Schnittstelle `IGraphVisualizer` definiert nur zwei Methoden, eine, um den Graphen zu zeichnen, eine, um einen Weg im Graphen hervorzuheben. Für SFML und FLTK sind konkrete Klassen zur Darstellung von Graphen auf dem Bildschirm vorhanden.

3 Aufgabe

Zunächst ist ein Refactoring der bestehenden Bibliothek durchzuführen. Dabei soll auch untersucht werden, um welche Methoden die Schnittstellen erweitert werden müssen, um eine Visualisierung der Graphalgorithmen zu ermöglichen. Außerdem soll untersucht werden, welche Entwurfsmuster genutzt werden können.

Dann sollen weitere Algorithmen wie Tourenplanung (Traveling Salesperson Problem), Max-Flow nach Dinic, Minimum-Cost-Matching oder Connected-Components implementiert werden. Für die in EAL behandelten speziellen Graphklassen sollen Algorithmen implementiert werden. Um auch andere Standard-Formate zu unterstützen, sollen (De-)Serialisierer implementiert werden, die GraphViz[3] und GraphML[4] realisieren.

Dieses Projekt sollte von mindestens zwei Studierenden bearbeitet werden.

[3] `http://graphviz.org/`
[4] `http://graphml.graphdrawing.org/`