Shortest-Path – Programmierbeispiel 3

1. File Input

```
while(!readFile.eof()){
    std::string input;
    //Get line-information from input file as a single line (delimited by \n)
    getline(readFile, input);

    //Split string using different delimiters to get line data in vector
    std::vector<std::string> lineData;
    auto startIndex = 0;
    auto endIndex = 0;
    auto counter = -1;

    std::string delimiter;
    while(endIndex < input.size()){
        //Check which delimiter to use based on current position in the string
        if(counter == 0) delimiter = ":";
        if(counter>0) delimiter = " \"";
        if(counter %2 == 0 && counter != 0) delimiter = "\" ";
        endIndex = input.find(delimiter, startIndex);

        //Add data to line-vector
        std::string data = input.substr(startIndex, endIndex - startIndex);
        lineData.push_back(data);

        startIndex = endIndex + delimiter.size();
        counter++;
    }

    //Delete " from last entry, which is not removed by delimiters
    lineData.back().pop_back();

    //Delete empty vector entries
    lineData.erase(lineData.begin()+0);
    lineData.erase(lineData.begin()+1);

    this->createAdjacencyList(lineData);
}
```

Die Laufzeit der äußeren While-Schleife ist linear O(L), mit L…Anzahl der Linien im Verkehrsnetz/Zeilen im Inputfile

Das Einlesen der Daten war im Beispiel relativ komplex, da die einzelnen Stationen und Distanzen nicht durch ein einheitliches Trennsymbol getrennt werden, sondern durch unterschiedliche Strings von Zeichen. *getline(readFile, input)* liefert die Daten einer Verkehrsmittellinie, die danach in der folgenden while-Schleife bearbeitet und aufgeteilt wird.

```
while(endIndex < input.size()){
    //Check which delimiter to use based on current position in the string
    if(counter == 0) delimiter = ":";
    if(counter>0) delimiter = " \"";
    if(counter %2 == 0 && counter != 0) delimiter = "\" ";
    endIndex = input.find(delimiter, startIndex);

    //Add data to line-vector
    std::string data = input.substr(startIndex, endIndex - startIndex);
    lineData.push_back(data);

    startIndex = endIndex + delimiter.size();
    counter++;
}
```

Die Laufzeit der inneren While-Schleife ist linear O(S), mit S…Anzahl der Stationen pro Linie.

Die Linienbezeichnung wird zu Beginn durch einen Doppelpunkt abgespalten und danach werden abwechselnd , “‘ und ,“ ‘ als Trennzeichenketten verwendet, da manche Stationen auch Leerzeichen beinhalten. Die Daten werden in einen Vektoren gespeichert, mit dem dann eine Adjacency-List erstellt wird.

Daraus folgt für die File-Input-Funktion eine Laufzeit von O(L) * O(S) = O(L*S)

2. Graph generieren

lineData beinhaltet S Strings, die die Stationsdaten beschreiben und kreiert für jeden zweiten String (Station) eine neue Station und platziert sie in der Map. Daraus folgt eine Laufzeit von O(S).

```
void Graph::createAdjacencyList(std::vector<std::string> lineData){
    auto lineName = lineData.at(0);
    //Add station-information to adjacency-list
    for(auto i = 1; i < (int)lineData.size(); i+=2){
        auto alreadyExists = false;

        auto current = this->createNewStation(lineData.at(i), lineName, 0);

        //Add adjacent stations, if they exist
        if(i<=(int)lineData.size()-2){
            auto next = this->createNewStation(lineData.at(i+2), lineName, std::stoi(lineData.at(i+1)));
            current->adjacentStations.push_back(next);
        }
        if(i>=3){
            auto previous = this->createNewStation(lineData.at(i-2), lineName, std::stoi(lineData.at(i-1)));
            current->adjacentStations.push_back(previous);
        }

        //Adjacency list is empty
        if((int)this->stations.size() == 0){
            this->stations[current->name] = current;
        }else{
            //Check if current station has an entry in adjacency-list
            if(this->stations[current->name] != nullptr){
                //Add adjacent stations from current station to entry in adjacency-list
                for(const auto& station : current->adjacentStations){
                    this->stations[current->name]->adjacentStations.push_back(station);
                }
                alreadyExists = true;
            }
            if(!alreadyExists){
                //Create new entry in adjacency-list for current station
                this->stations[current->name] = current;
            }
        }
    }
}
```

current beschreibt einen Haupteintrag in der Adjacency-List, wobei weiters überprüft wird, ob es bei der Station einen Vorgänger und/oder einen Nachfolger gibt, d.h. ob die Station die erste oder letzte Station einer Linie ist oder nicht. Ist dies der Fall werden diese Stationen zum Listeneintrag in einem Vektor angehängt. Es wird anschließend überprüft, ob es bereits einen Eintrag für die Station in der Adjacency-Liste gibt. Wenn ja, dann werden die anliegenden Stationen auf diesen übertragen, ansonsten wird ein neuer erstellt.

3. Dijkstra-Algorithmus

Der von uns implementierte Dijkstra-Algorithmus verwendet folgende Maps für das Finden des kürzesten Weges zwischen zwei Stationen:

```
//Maps for Dijkstra-Algorithm
std::unordered_map<std::string, int> costs;
std::unordered_map<std::string, bool> visited;
std::unordered_map<std::string, Station*> path;
```

- costs: Speichert für jede Station die kürzeste Distanz zu einer angegebenen Startstation ab
- visited: Speichert ab, ob für eine Station bereits der kürzeste Weg gefunden wurde
- path: Speichert für jede Station seinen Vorgänger im kürzesten Weg ab (notwendig für Ausgabe)

Die Stationen, in der Queue werden in einem set gespeichert, da dadurch die kürzeste Distanz direkt am Anfang des sets liegt und leicht darauf zugegriffen werden kann.

```
std::set<std::pair<int, std::string>> checkNext;
```

Die Startstation wird in die Kosten-Map mit Kosten 0, sowie ins Priority-Queue-Set eingefügt. Der Path zeigt bei der Startstation auf die Station selbst, was später als Abbruchbedingung für die Backtracking-Rekursion gilt.

```
//Insert start-node to
this->costs[start] = 0;
checkNext.insert(std::make_pair(0, start));
this->path[start] = this->stations[start];
```

Die While-Schleife läuft maximal für jede Station einmal durch, daraus ergibt sich nur für die While-Schleife alleine eine Laufzeit von O(S), wobei S für die Stationen im set steht.

```
while(checkNext.begin()->second != dest){
    auto distance = checkNext.begin()->first;
    auto stationName = checkNext.begin()->second;

    ///setColor(3); std::cout << stationName << " via " << this->stations[stationName]->line << ": " << this->costs[sta

    for(auto& adj : this->stations[stationName]->adjacentStations){
        //Check if Station has already been visited
        if(visited[adj->name]) continue;

        //Cost has not been initialised yet
        if(this->costs[adj->name] == 0 && adj->name != start){
            this->costs[adj->name] = distance + adj->cost;
            checkNext.insert(std::make_pair(this->costs[adj->name], adj->name));
            //Insert to path
            this->path[adj->name] = this->createNewStation(stationName, adj->line, adj->cost);
        }

        //Update Distance if necessary
        if((distance + adj->cost) < costs[adj->name]){
            ///setColor(5); std::cout << "   [UPDATE] "; setColor(7); std::cout << adj->name << " from " << this->costs
            auto toUpdate = checkNext.find(std::make_pair(this->costs[adj->name], adj->name));

            //Delete old entry from set
            if(toUpdate != checkNext.end()){
                checkNext.erase(toUpdate);
            }

            //Insert new entry with updated Cost
            this->costs[adj->name] = distance + adj->cost;
            checkNext.insert(std::make_pair(this->costs[adj->name], adj->name));
            //Update path entry
            this->path[adj->name] = this->createNewStation(stationName, adj->line, adj->cost);
        }
        counter++;
        ///std::cout << "   checking " << adj->name << " via " << adj->line << ": " << adj->cost << std::endl;
    }
    //Remove pair with lowest distance from set and add it to visited
    checkNext.erase(checkNext.begin());
}
```

Die For -Schleife läuft für jede Station die anliegenden Nachbarstationen durch. Da eine Station nicht mit jeder anderen Station verbunden ist benennen wir diese Größe mit A für anliegende Stationen. Es ergibt sich eine Laufzeit von O(A) für die For-Schleife

Die Laufzeit für find und insert in einem std::set ist logarithmisch, also O(log(S))

Insgesamt ergibt sich für unseren Dijkstra-Algorithmus somit eine Laufzeit von:

O(S) * (O(A) * O(log(S)) = O(A*S*log(S));

Solange unsere Zielstation nun nicht die Station mit dem aktuell kürzesten Pfad ist werden für jede Station alle anliegenden Stationen überprüft. Wenn die Distanz zu einer Station noch nicht bekannt ist, oder ein kürzerer Weg gefunden wird diese upgedatet und in der Kosten-Map gespeichert. Wenn alle anliegenden Stationen einer Station gecheckt wurden, wird die Station als visited markiert und aus dem set entfernt. Der kürzeste Weg zu dieser Station wurde gefunden.

4. Path durch Backtracking

```
std::string Graph::getPath(std::string pathString, std::string station){
    //Check if start station is reached, since path(start) == start
    if(station != this->path[station]->name){
        //Concatenate Stations to output-string
        pathString = getPath(pathString, this->path[station]->name)
                   + " =[ " + std::to_string(this->path[station]->cost) + " | ( " + this->path[station]->line + " ) ]=> [ "
                   + station + " ]\n";
    }else{
        //Add first station
        pathString = "[ " + station + " ]\n" + pathString;
    }
    return pathString;
}
```

getPath ist eine rekursive Funktion, die aus der Path-Map einen korrekten Output-String des kürzesten Paths zwischen zwei Stationen generiert. Dabei wird die Rekursion immer vorne an den Output-String angehängt.

```
=========================
Start: Hauptbahnhof
Destination: Schwedenplatz
Total Cost: 12
[ Hauptbahnhof ]
 =[ 1 | ( 18 ) ]=> [ Blechturmgasse ]
 =[ 1 | ( 18 ) ]=> [ Kliebergasse ]
 =[ 1 | ( 1 ) ]=> [ Laurenzgasse ]
 =[ 1 | ( 62 ) ]=> [ Johann-Strauss-Gasse ]
 =[ 1 | ( 1 ) ]=> [ Mayerhofgasse ]
 =[ 2 | ( 1 ) ]=> [ Paulanergasse ]
 =[ 1 | ( 1 ) ]=> [ Resselgasse ]
 =[ 1 | ( 1 ) ]=> [ Karlsplatz ]
 =[ 2 | ( U1 ) ]=> [ Stephansplatz ]
 =[ 1 | ( U1 ) ]=> [ Schwedenplatz ]

=========================
Checked Connections: 154
Elapsed Time: 0 micro-seconds
```

Beispiel:

Start der Rekursion bei Zielstation „Schwedenplatz“, path[„Schwedenplatz“] zeigt auf die Station „Stephansplatz“ mit der Linie „U1“, path[„Stephansplatz“] zeigt wiederum auf die Station „Karlsplatz“ mit der Linie „1“, usw. bis die Startstation „Hauptbahnhof“ erreicht wurde.

Überarbeitung:

Die Funktion *getPath* wurde durch die ebenfalls rekursive Funktion *printPath* ausgetauscht, die die Stationen direkt ausgibt und somit um einiges lesbarer und eleganter ist. Die Laufzeit dieser Funktion ist wie bei *getPath* ebenfalls O(N).

```
void Graph::printPath(std::string station){
    if(station == this->path[station]->name){
        std::cout << "[ " << station << " ]" << std::endl;
        return;
    }

    printPath(this->path[station]->name);
    std::cout << "  ==[ "; setColor(6); std::cout << this->path[station]->cost; setColor(8);
    std::cout << " | ( "; setColor(7); std::cout << this->path[station]->line; setColor(8);
    std::cout << " ) ]==>  [ " << station << " ]" << std::endl;
}
```

printPath wird maximal für alle Stationen in der path-Map aufgerufen. Daraus folgt eine Laufzeit von O(N), mit N…Anzahl der Einträge in der path-Map

5. Messungen

- Leopoldau -> Reumannplatz: 994µs, 338 Verbindungen überprüft
- Suessenbrunner Str/Oberfeldgasse -> Quellenstrasse/Favoritenstrasse: 1021µs, 373 Verbindungen überprüft
- Absberggasse -> Hoechstaedtplatz: 1046 µs, 403 Verbindungen überprüft
- Strozzigasse -> Litfassstrasse: 1039µs, 489 Verbindungen überprüft
- Wenzgasse -> Fickeysstrasse: 1985µs, 555 Verbindungen überprüft
- Leopoldau -> Hietzing, Kennedybruecke: 2004µs, 570 Verbindungen überprüft