**Dokumentation Algorithmus**

**Datenstruktur:**

// List of visited nodes  
std::vector<std::string> visited;  
// Heap  
std::priority\_queue<HeapNode, std::vector<HeapNode>, CompareHeapNode> heap;  
// Map for storing last connection for every node  
std::map<std::string,Connection\*> lastcons;

Wir benutzen für unseren Algorithmus 3 verschiedenen Datenstrukturen.

**Vektor**: Ein Vektor von Strings wird zur Speicherung von bereits besuchten Knotenpunkte verwendet.

**Priority\_Queue**: Der Heap speichert alle verfügbaren Verbindung von den bereits besuchten Stationen geordnet nach dem Gewicht der Verbindungsstrecke.

**Map**: Zur Speicherung der letzten Verbindungen zu jedem Knotenpunkt um nachvollziehen zu können, woher man kam.

**Der Dijkstra Algorithmus:**

Wir verwenden den Dijkstra und gehen Knoten für Knoten in sämtliche Richtungen durch unsere vorgegebenen Verbindungen. Bei jedem Schritt wird der Knoten mit dem kürzesten Gewicht zu unserem Startpunkt gewählt von welchen aus wieder weitergegangen wird. Dieser Vorgang wird solange wiederholt, bis das Ziel erreicht wurde beziehungsweise bis festgestellt wurde, dass keine Route zwischen den Stationen verfügbar ist.

Wenn das gewünschte Ziel als Knotenpunkt gefunden wird, wird die kürzeste Route ausgegeben, welche im Heap gespeichert wird. Falls keine Route zwischen den Punkten verfügbar ist, wird eine Meldung ausgegeben und der Programmverlauf wird beendet.

Gesucht wird vom eingegebenen Zielknoten zum Startknoten.

**const void** ConnectionPlan::getShortestPath(std::string start\_station, std::string end\_station) {  
 *// List of visited nodes* std::vector<std::string> visited;  
 *// Heap* std::priority\_queue<HeapNode, std::vector<HeapNode>, CompareHeapNode> heap;  
 *// Map for storing last connection for every node* std::map<std::string,Connection\*> lastcons;  
  
 *// Init variables* **int** mintime=0;  
 *// Start search from destination* std::string currentnode = end\_station;  
 Connection\* lastconnection = **nullptr**;  
  
 **do** {  
 *// Set current node visited* visited.push\_back(currentnode);  
 *// Iterate through each neighbour of current node* **for**(**unsigned int** i = 0; i < stations[currentnode].size(); i++){  
 *// Push neighbour to heap if not already visited* **if** (std::find(visited.begin(), visited.end(), stations[currentnode].at(i)->getDestination()) == visited.end()) {  
 *// Check if line changes* **if**(lastconnection == **nullptr** || lastconnection->getLine() == stations[currentnode].at(i)->getLine()) {  
 heap.push(HeapNode(stations[currentnode].at(i),  
 mintime + stations[currentnode].at(i)->getTraveltime(),  
 lastconnection));  
 }**else**{  
 *// If line changes, add time for changing* heap.push(HeapNode(stations[currentnode].at(i),  
 mintime + stations[currentnode].at(i)->getTraveltime() + changingtime,  
 lastconnection));  
 }  
 }  
 }  
 **do** {  
 **if**(heap.empty()){  
 *// Heap empty, problem occured* std::cout << **"No path could be found! :("** << std::endl;  
 **return**;  
 }  
 *// Pop node with least weight from heap and update currentnode and mintime* HeapNode node = heap.top();  
 currentnode = node.connection->getDestination();  
 mintime = node.weight;  
 lastconnection = node.connection;  
 heap.pop();  
 lastcons.insert({currentnode, node.lastconnection});  
 *// Ignore nodes which are already visited, go to next one* }**while** (std::find(visited.begin(), visited.end(), currentnode) != visited.end());  
 *// Do as long as current node is not start station* } **while** (currentnode!=start\_station);  
  
 *// Print found path* printPath(lastcons, start\_station, end\_station);  
}

**Testfälle:**







