MOALIM MAKARAN IBRAHIM & VURAL ZELIHA & ZAHKAROV ALEKSANDR

ÜBUNG 3 – GRAPHEN

Ausgewählte Programmiersprache: C++

Verwendete Algorithmen: Dijkstra's und A* Algorithmen

❖ Main

Hier wird einen Graphen aus der Datei "paths.txt" eingelesen. Die Knoten werden als Kommandozeilenargumente übergeben. Die Algorithmen sind "Dijkstra's Algorithmus" und "A* Algorithmus". Der Code erzeugt also zwei mögliche Pfade und gibt sie auf der Konsole aus.

Die Funktion "find_path_dijkstra" und "find_path_astar" sind jeweils die Implementierung der Algorithmen und gibt den gefundenen Pfad sowie den Gesamtkosten als Ergebnis zurück. Die Funktion "pretty_print" wird verwendet, um den Pfad und die Gesamtkosten auf der Konsole auszugeben.

Die Anweisungen "SetConsoleTextAttribute" dienen nur dazu die Farbe der Textausgabe in der Konsole zu ändern.

Anschließend werden Testfälle ausgeführt, um die Laufzeiten von Dijkstra und A* Algorithmen bei der Suche nach kürzesten Wegen zwischen verschiedenen Knotenpaaren zu vergleichen. Gemessen wird mit der chrono Bibliothek und die Ergebnisse werden auf der Konsole ausgegeben.

❖ Funktionen

```
struct Edge {
    std::string station;
    int time;
    std::string line_name;
};

struct Node {
    std::string station;
    int distance;
    int heuristic;
};

using Graph = std::map<std::string, std::vector<Edge>>;
```

- struct Edge: Repräsentiert eine Kante im Graphen. Enthält den Namen der Zielstation, die Zeit, die benötigt wird, um diese Station zu erreichen, und den Namen der Verbindungslinie.
- struct Node: Repräsentiert einen Knoten im Graphen. Enthält den Namen der Station, die Entfernung vom Startknoten und die heuristische Schätzung der verbleibenden Entfernung zum Zielknoten.
- Graph: Ein Alias für std::map<std::string, std::vector<Edge>>, der den Graphen repräsentiert. Der Schlüssel ist der Name der Station und der Wert ist eine Liste von Kanten, die von dieser Station ausgehen.

```
bool operator<(const Node &a, const Node &b)
```

- operator<: Überlädt den <-Operator für Node-Objekte, um sie in einer Prioritätswarteschlange (std::priority_queue) zu verwenden. Die Knoten werden nach der Summe von Entfernung und heuristischer Schätzung sortiert.

```
std::vector<std::string> splitAndStrip(const std::string &line, char
delimiter) {
```

- splitAndStrip: Eine Hilfsfunktion zum Aufteilen einer Zeichenkette in Teile und Entfernen von Leerzeichen aus den Teilen.
- Laufzeitkomplexität: O(n) → Hierbei steht 'n' für die Länge der Eingabezeichenkette `line`.

```
Graph read_file(const std::string &filename) {
```

- Die Funktion `read_file()` liest eine Datei mit dem gegebenen Dateinamen ein, die eine ähnliche Streckenführung in Form von Zeilen enthält, die durch ein bestimmtes Trennzeichen (`"`) getrennt sind, und gibt einen Graphen von Stationen mit Informationen über ihre Nachbarstationen und Entfernungen zwischen ihnen zurück.
- Laufzeitkomplexität: O(n * m) → Hierbei steht 'n' für die Anzahl der Kanten im Graphen und 'm' für die durchschnittliche Länge der Zeilen in der Eingabedatei.

```
std::pair<std::deque<std::pair<std::string, std::string>>, int>
find_path_dijkstra(const Graph &graph, const std::string &start, const std::string
&end) {
```

- find_path_dijkstra: Implementiert den Dijkstra-Algorithmus zur Suche des kürzesten Pfads im Graphen.
    - Initialisiert Entfernungen für alle Stationen auf unendlich (std::numeric_limits<int>::max()) und die Entfernung für die Startstation auf 0.
    - Verwendet eine Prioritätswarteschlange (std::priority_queue) für die zu verarbeitenden Knoten.
    - Aktualisiert die Entfernungen für benachbarte Knoten, wenn ein kürzerer Pfad gefunden wird.
    - Speichert die vorherigen Knoten und Verbindungslinien für jeden Knoten, um den Pfad zurückzuverfolgen.
    - Gibt den kürzesten Pfad und die Gesamtkosten (Zeit) zurück.

    - Laufzeitkomplexität: O((n + m) * log n) → Hierbei steht 'n' für die Anzahl der Knoten und 'm' für die Anzahl der Kanten im Graphen. Die Laufzeit des Dijkstra-Algorithmus beträgt O((n + m) * log n), wobei 'log n' aufgrund der Prioritätswarteschlange entsteht.

```
std::pair<std::deque<std::pair<std::string, std::string>>, int>
find_path_astar(const Graph &graph, const std::string &start, const
std::string &end) {
```

- find_path_astar: Implementiert den A*-Algorithmus zur Suche des kürzesten Pfads im Graphen.

- Verwendet die gleiche Initialisierung und Datenstrukturen wie der Dijkstra-Algorithmus.
- Berechnet die heuristische Schätzung für jeden Knoten mit der Funktion calculate_heuristic. In diesem Fall ist die Heuristik einfach der absolute Unterschied in der Anzahl der Zeichen zwischen den Stationsnamen.
- Aktualisiert die Entfernungen für benachbarte Knoten wie im Dijkstra-Algorithmus, aber fügt auch die heuristische Schätzung der verbleibenden Entfernung zum Zielknoten hinzu. Dies hilft, die Suche auf vielversprechendere Pfade zu konzentrieren und die Laufzeit des Algorithmus zu reduzieren en.wikipedia.org.

    - Speichert die vorherigen Knoten und Verbindungslinien für jeden Knoten, um den Pfad zurückzuverfolgen.
    - Gibt den kürzesten Pfad und die Gesamtkosten (Zeit) zurück.

- Laufzeitkomplexität: O((n + m) * log n) → Die Laufzeit des A*-Algorithmus ist ähnlich wie die des Dijkstra-Algorithmus, da sie sich auf dem Dijkstra-Algorithmus aufbaut. Daher beträgt die Laufzeit auch hier O((n + m) * log n).

```
void pretty_print(const std::deque<std::pair<std::string, std::string>>
&path, int total_cost) {
```

- Die Funktion `pretty_print()` gibt den gefundenen Pfad zwischen zwei Knoten sowie die Gesamtkosten und die Anzahl der Stationen aus. Wenn keine Route gefunden wird, wird "Kein Pfad gefunden" ausgegeben. Die Funktion verwendet eine Schleife, um durch die Stationen im Pfad zu iterieren, und gibt für jede Station den Namen und die Linie aus, auf der sie liegt. Wenn der Wechsel der Linie erforderlich ist, wird eine entsprechende Nachricht ausgegeben. Schließlich werden die Gesamtkosten und die Anzahl der Stationen ausgegeben.
- Laufzeitkomplexität: O(n) → Hierbei steht 'n' für die Anzahl der Stationen im Pfad.

Algorithmen

Das Programm hat zwei Algorithmen zur Suche des kürzesten Weges zwischen zwei Stationen in einem Graphen: den Dijkstra-Algorithmus und den A*-Algorithmus. Der Graph stellt ein Verkehrsnetz dar, in dem die Knoten Stationen und die Kanten Verbindungen zwischen den Stationen sind.

Der Dijkstra-Algorithmus findet den kürzesten Weg, indem er eine Prioritätswarteschlange verwendet, um diejenigen Knoten auszuwählen, die am nächsten am Startknoten liegen und deren Entfernung vom Startknoten bekannt ist.

Der A*-Algorithmus verwendet eine geschätzte Entfernung zum Zielknoten als zusätzliche Information, um die Reihenfolge der Knoten in der Prioritätswarteschlange zu bestimmen.

Die Funktionen find_path_dijkstra und find_path_astar implementieren den Dijkstra-Algorithmus bzw. den A*-Algorithmus. Beide Funktionen geben ein Paar zurück, das aus einer Liste von Stationen und den Gesamtkosten besteht. Die Liste von Stationen enthält die Stationen auf dem kürzesten Weg von der Startstation zur Endstation, und die Gesamtkosten sind die Summe der Zeiten, die benötigt werden, um von einer Station zur nächsten zu gelangen.

Der Hauptunterschied zwischen den beiden Algorithmen besteht darin, dass der A*-Algorithmus eine heuristische Schätzung verwendet, um die Suche zu beschleunigen, während der Dijkstra-Algorithmus dies nicht tut. In diesem speziellen Fall ist die Heuristik einfach der absolute Unterschied in der Anzahl der Zeichen zwischen den Stationsnamen, was möglicherweise nicht die beste Schätzung für die tatsächliche verbleibende Entfernung ist. Allerdings dient es als einfaches Beispiel für die Verwendung von Heuristiken in einem A*-Algorithmus.

❖ O-Notation & Laufzeitvergleich & Speicherbedarf

„calculate_heuristic“:

- Laufzeitkomplexität: O(1) → Diese Funktion hat eine konstante Laufzeit, da sie nur einfache arithmetische Operationen ausführt.

Main:

Die Laufzeit des Hauptprogramms (Main-Funktion) hängt von der Anzahl der Testfälle ab, die in der Schleife durchlaufen werden. Die Ausführung der Algorithmen für jeden Testfall hat eine Laufzeitkomplexität von O((n + m) * log n). Daher ergibt sich insgesamt eine Laufzeitkomplexität von O(T * (n + m) * log n), wobei 'T' die Anzahl der Testfälle ist.

Dijkstra-Algorithmus:

Der Dijkstra-Algorithmus verwendet eine Prioritätswarteschlange, um die Knoten mit der kürzesten Distanz zu verarbeiten. Die Laufzeit des Dijkstra-Algorithmus hängt von der Anzahl der Knoten (V) und der Anzahl der Kanten (E) im Graphen ab. Die Laufzeitkomplexität des Algorithmus beträgt $O(V^2)$ für einfache Implementierungen, kann jedoch auf O(V + E * log(V)) reduziert werden, wenn eine Prioritätswarteschlange verwendet wird.

A*-Algorithmus:

Der A*-Algorithmus ist eine Erweiterung des Dijkstra-Algorithmus, die eine heuristische Funktion verwendet, um die Suche zu beschleunigen. Die Laufzeit des A*-Algorithmus hängt von der Heuristik und der Anzahl der Knoten (V) und Kanten (E) im Graphen ab. Im schlimmsten Fall beträgt die Laufzeitkomplexität O(V + E * log(V)), ähnlich wie beim Dijkstra-Algorithmus.

Laufzeiten für beide Algorithmen vergleichen:

Um die Laufzeit der Algorithmen zu messen, haben wir die std::chrono-Bibliothek verwendet und folgende Schritte haben wir umgesetzt:

1. <chrono>-Bibliothek wurde im Code hinzugefügt
2. Die Startzeit vor dem Aufruf des Algorithmus (Dijkstra oder A*) wurde gemessen
3. Algorithmus ausgeführt
4. Die Endzeit nach dem Abschluss des Algorithmus gemessen
5. Die Differenz zwischen der Start- und Endzeit, um die Laufzeit des Algorithmus zu erhalten, wurde berechnet

6. Die gemessene Laufzeit ausgegeben

Um die Messungen für mehrere Testfälle und beide Algorithmen (Dijkstra und A*) durchzuführen, haben wir die oben genannten Schritte in einer Schleife implementiert.

Speicherbedarf

Der Speicherbedarf der beiden Algorithmen hängt von der Anzahl der Knoten (V) und der Anzahl der Kanten (E) im Graphen ab. Die Speicherkomplexität beider Algorithmen ist ähnlich, da sie ähnliche Datenstrukturen verwenden, um den Graphen und die Informationen über besuchte Knoten und Kanten zu speichern.

Dijkstra-Algorithmus: Die Speicherkomplexität des Algorithmus beträgt O(V + E) .

A*-Algorithmus: Die Speicherkomplexität des Algorithmus beträgt ebenfalls O(V + E).

Insgesamt sind die Speicheranforderungen für beide Algorithmen ähnlich und hängen von der Größe des Eingabegraphen ab. Es ist wichtig zu beachten, dass der tatsächliche Speicherbedarf von der Implementierung der verwendeten Datenstrukturen und der Größe der Eingabedaten abhängt.