Projekt Navigation

Fach: Software Engineering (SEL)
Dozent: Bradley Richards
Studenten: Lehkdup Shöntsang und Jakob Koller
GitHub Projekt: https://github.com/1jakob2/NavigationProject

Daten Erfassung

Zur Prüfung der Suchalgorithmen Best-First, A* etc. haben wir ein umfangreichen Datensatz von Knoten und Kanten aus Bern erstellt, welche wir im Späteren porjektverlauf mit Daten von anderen Gruppen ergänzt haben. Diese Informationen wurden in einer CSV Datei abgespeichert. Das Trennzeichen war das Semikolon. Jeder Knoten enthielt den Namen des Knotens, die Ost- und Nordkoordinaten, in jeder Kante standen Anfang und Ende des Knotens sowie die tatsächliche Entfernung, die zwischen ihnen besteht. Wir nutzten das vorgegebene Tool «maps.geo.admin.ch» um die Daten Erfassung erstens zu visualisieren und die Koordinaten als auch die Distanzen zu ermitteln.
Die Informationen wurden in zwei Dateien (csv) übertragen, eine war «knodes», und die andere war «edges». Wir mussten die Daten etwas umformatieren damit es das Script «MappData» korrekt lesen konnte.

Phase zwei

Die Algorithmen erklärt

Depth-First-Suchalgorithmus

Vorteile:
Der Depth-First verbraucht wenig Speicher, da er nur den aktuellen Pfad speichert. Besonders im Vergleich zu Algorithmen die alle Knoten durchsuchen wie die Breitensuche. Der rekursive Aufbau von Depth-First ist einfach zu implementieren und zu verstehen. Wegen der Natur von Depth-First, welcher immer bis zum Ende eines Pfades geht, bevor er zurückgeht und einen anderen Pfad nimmt, eignet er sich für tiefe aber schmale Datensätze.

Nachteile:
Bei Flachen Datensätzen ist der Depth-First ineffizient. Es bietet auch keine Möglichkeit den Algorithmus mit Heuristiken zu optimieren.

Datenstruktur:
Eine «Adjazenzliste» (Im Code «adjList») wird verwendet, um ein Netzwerk von Knoten darzustellen. Sie besteht aus einer HashMap, die jedem Knoten eine Liste von benachbarten Knoten zuordnet.

Der Suchvorgang startet bei einem vorgegebenen Startknoten und bewegt sich entlang der Verbindungen, bis der vorgegebene Zielknoten erreicht ist.

Die Wichtigsten Funktionen:

- depthFirst(): Diese Methode startet den Suchprozess, indem sie eine Pfadliste erstellt und die rekursive Funktion aufruft.

- depthFirstRecursive(): Die rekursive Funktion, die jeden Knoten besucht, prüft, ob das Ziel erreicht wurde, und fügt den aktuellen Knoten dem Pfad hinzu.
- haveBeenThere(): Überprüft, ob ein bestimmter Knoten bereits im aktuellen Pfad vorhanden ist, um Schleifen zu vermeiden.
- printPath(): Gibt den finalen Pfad nach Abschluss der Suche aus.

Breadth First Suchalgorithmus -

Aufbau:

1. **Datenstruktur**:
    - Der Algorithmus nutzt eine Adjazenzliste (adjList), wie beim Depth First.
    - Der Suchprozess verwendet eine Liste (paths), die potenzielle Pfade speichert. Jeder Pfad ist eine Liste von Knoten, die auf der Suche nach dem Zielknoten (end) entsteht.
2. **Hauptfunktionen**:
    - breadthFirst(): Diese Methode führt die Breitensuche durch. Sie startet vom Knoten start, breitet sich über alle benachbarten Knoten aus und erweitert sukzessive die Pfade, bis der Zielknoten erreicht ist. Der erste gefundene Pfad zum Ziel wird zurückgegeben.
    - printPath(): Gibt den gefundenen Pfad aus, nachdem die Suche abgeschlossen ist.
3. **Algorithmuslogik**:
    - Zu Beginn wird ein Pfad mit dem Startknoten erstellt und der Liste von Pfaden hinzugefügt.
    - In jeder Iteration des Hauptschleife wird der erste Pfad aus der Liste entfernt und durch alle möglichen Erweiterungen (die direkt verbundenen Knoten) verlängert.
    - Sobald das Ziel erreicht wird, wird die Schleife abgebrochen und der finale Pfad ausgegeben.

Best First Suchalgorithmus

Vorteile:
Die Breitensuche garantiert Korrektheit und Vollständigkeit, da sie den kürzesten Pfad zum Zielknoten findet, indem sie systematisch die Knoten auf jeder Ebene durchsucht, bevor sie tiefer geht. Ausserdem vermeidet sie Schleifen, da jede Ebene vollständig durchsucht wird, bevor die nächste Ebene betrachtet wird. Ein weiterer Vorteil ist die einfache Implementierung, da nur eine Warteschlange benötigt wird, um die möglichen Pfade zu speichern.

Nachteile:
Die Breitensuche kann bei grossen Graphen sehr speicherintensiv werden, da alle Pfade auf jeder Ebene in der Warteschlange gehalten werden müssen, was den Speicherbedarf exponentiell wachsen lässt. Zudem ist sie ineffizient bei tiefen Lösungen, da alle Knoten

auf einer Ebene durchsucht werden, bevor die Suche tiefer geht, was zu unnötigen Untersuchungen führt. Für gewichtete Graphen ist die Breitensuche ungeeignet, da sie keine Kosten zwischen Knoten berücksichtigt und nur den kürzesten Pfad in Bezug auf die Anzahl der Knoten, nicht aber auf die tatsächlichen Kosten, findet.

Aufbau:

1. **Datenstruktur**:
    - Der Algorithmus verwendet eine Adjazenzliste (adjList), um die Verbindungen zwischen Knoten zu speichern, und eine Liste von GPS-Koordinaten (nodeList), um die Positionen der Knoten zu verwalten.
    - Der Kern der Suchstrategie liegt in der Priorisierung von Pfaden, deren Endknoten am nächsten zum Zielknoten liegen, basierend auf einer Heuristik (in diesem Fall die quadratische Entfernung zwischen Knoten).
2. **Hauptfunktionen**:
    - bestFirst(): Diese Methode führt den Best-First-Algorithmus aus. Sie startet mit einem Pfad, der nur den Startknoten enthält, und erweitert diesen, indem sie an jeden Knoten die Nachbarknoten anhängt. Dabei wird immer der Pfad weiterverfolgt, dessen Endknoten der Zielposition am nächsten ist.
    - bestPath(): Wählt den besten Pfad aus der Liste der möglichen Pfade basierend auf der kleinsten Heuristik, also dem kürzesten Abstand zum Ziel.
    - distanceBetween(): Berechnet die quadratische Entfernung zwischen einem Knoten und dem Zielknoten. Diese wird als Heuristik verwendet, um zu bestimmen, wie "nah" ein Knoten am Ziel liegt.
    - printPath(): Gibt den finalen Pfad nach Abschluss der Suche aus.
3. **Algorithmuslogik**:
    - Der Algorithmus startet mit einem Pfad, der nur den Startknoten enthält, und berechnet die Entfernung zum Ziel. Während des Suchprozesses wird der Pfad, dessen Endknoten der Zielposition am nächsten liegt, weiter ausgebaut.
    - Wenn ein Zielknoten erreicht wird, endet die Suche, und der gefundene Pfad wird zurückgegeben.

A* Suchalgorithmus

Vorteile:

A* garantiert eine optimale Lösung, sofern die verwendete Heuristik zulässig ist, d.h., die tatsächlichen Kosten nie unterschätzt werden. Im vorliegenden Fall wird die euklidische Distanz verwendet, die in vielen geografischen Anwendungen eine gute Wahl darstellt. Zudem ist A* effizienter als Breitensuche oder Tiefensuche, da er sowohl die zurückgelegte Distanz als auch eine Heuristik für die verbleibende Distanz nutzt, wodurch weniger Knoten untersucht werden müssen. Ein weiterer Vorteil ist die Flexibilität des Algorithmus, da die Heuristik angepasst werden kann, um für verschiedene Probleme optimiert zu werden, wie etwa durch Berücksichtigung von Verkehrsdaten bei Navigationsaufgaben.

Nachteile:
Ein Nachteil von A* ist der hohe Speicherverbrauch, da nicht nur die Knoten in der Warteschlange, sondern auch Informationen zu allen besuchten Knoten gespeichert werden müssen, was bei grossen Graphen schnell zu erheblichen Speicheranforderungen führen kann. Ausserdem ist die Performance stark von der Qualität der Heuristik abhängig. Eine schlechte Heuristik kann dazu führen, dass unnötig viele Knoten durchsucht werden und der Algorithmus an Effizienz verliert. Schliesslich eignet sich A* nicht ideal für dynamische Umgebungen, da der Algorithmus bei Änderungen ständig neu berechnet werden muss, was ineffizient sein kann.

Aufbau:

1. **Datenstruktur**:
    - Eine Adjazenzliste (adjList) wird verwendet, um die Verbindungen zwischen Knoten zu speichern. Die nodeList enthält GPS-Koordinaten der Knoten, die zur Berechnung der heuristischen Distanz verwendet werden.
    - Der Algorithmus nutzt eine HashMap (searchInfo), um Informationen über jeden Knoten zu speichern, z. B. die bisherige Distanz (distanceSoFar), die geschätzte Distanz zum Ziel (distanceToGoal), und den vorherigen Knoten im Pfad (previousNode).
    - Eine Warteschlange (queue) enthält die Knoten, die noch untersucht werden müssen, und wird basierend auf den Kosten des Pfads aktualisiert.
2. **Hauptfunktionen**:
    - aStar(): Diese Methode implementiert den A*-Algorithmus. Sie durchsucht den Graphen, indem sie die Gesamtkosten eines Pfades als Summe aus der bisher zurückgelegten Distanz und der geschätzten Entfernung zum Ziel bewertet.
    - findNodeLowestCost(): Diese Funktion durchsucht die Warteschlange und findet den Knoten mit den niedrigsten Gesamtkosten (bisherige + geschätzte Distanz). Dieser Knoten wird als nächster untersucht.
    - reconstructPath(): Rekonstruiert den Pfad vom Start- zum Zielknoten, sobald das Ziel erreicht ist, indem es die Informationen aus searchInfo verwendet.
    - distanceBetween(): Berechnet die euklidische Distanz zwischen zwei Knoten basierend auf deren GPS-Koordinaten. Diese Distanz dient als Heuristik.

Dijkstra Algorithmus

Dijkstra Implementierung
Das Grundgerüst der Daten (adjacency list und node list) ist gleich wie bei den anderen Algorithmen. In der vorliegenden Implementierung wird mit einer PriorityQueue gearbeitet, um immer den nächsten Konten mit der geringsten Distanz zu verarbeiten. Am Anfang werden alle Knoten mit einer Distanz von «Unendlich» initialisiert. Nur der Startknoten wird auf 0 gesetzt. Anschliessend wird iterativ ein Knoten nach dem anderen aus der PriorityQueue gelöst. Für jeden Knoten werden alle Nachbarn analysiert und die Distanz

berechnet. Sollte die neue Distanz kleiner sein als die bisherige Distanz wird sie aktualisiert und der Knoten wird der PriorityQueue hinzugefügt. Zudem wird der Knoten auch in der previousNode-Map festgehalten, um den Pfad nachvollziehen zu können. Sobald der Zielknoten erreicht wird, wird der kürzeste Pfad durch die Methode reconstructPath gebaut und ausgegeben. Wird kein Ziel erreicht, gibt der Algorithmus eine leere Liste zurück.

Dijkstra vs. A*
Der Dijkstra Algorithmus ist eine vereinfachte Version des A* Algorithmus. Im Vergleich zu A* verwendet Dijkstra keine Heuristik. Für beide Algorithmen sind die tatsächlichen Kosten von einem Knoten zum nächsten relevant. Jedoch sucht der A* zielgerichteter, da er die geschätzten Kosten vom aktuellen Knoten zum Zielknoten einbezieht. Dijkstra ignoriert zukünftige Kosten oder eine Schätzung davon. Dies macht Dijkstra attraktiv um die kürzesten Wege zu allen Knoten ohne Heuristik zu finden. A* ist eine verbesserte Version von Dijkstra, wenn es darum geht, einen spezifischen Zielknoten zu erreichen.

Performance Vergleich der Algorithmen

In der nachstehenden Tabelle werden die verschiedenen Algorithmen miteinander verglichen. Damit genügend Daten gesammelt werden können und vergleichbare Resultate erzielt werden, muss jeder Algorithmus 20 Aufgaben lösen. Die 20 Aufgaben sind für alle Algorithmen die gleichen. Alle Aufgaben haben den gleichen Zielknoten, jedoch ist der Startpunkt jedes Mal anders.
Die Algorithmen werden nach Distanz, Anzahl Schleifen und Durchlaufzeit verglichen. Im Distanz-Test werden alle Pfade gemessen und summiert. Der Algorithmus, der die kürzeste Distanz liefert, wird als bester gewertet. Die Tests mit der Anzahl Schleifen und Durchlaufzeit messen, wie effizient und performant die Algorithmen sind. Eine kleine Anzahl Schleifen deutet auf eine hohe Effizienz des Algorithmus hin. Eine kurze Durchlaufzeit verspricht eine gute Performance des Algorithmus. Eine weitere Metrik, welche eine Aussage zur Effizienz und zum Ressourcenverbrauch macht, ist der Speicherverbrauch. Sowohl der Speicherverbrauch in Bytes als auch die Auslastung in Prozent des maximalen Speichers der Java Virtual Machine wird gemessen.

Test	DepthFirst	BreadthFirst	BestFirst	A*	Dijkstra
Distanz (m)	11’675’122	--	1’441’236	1’442’118	1’438’790
Schleifen	8’507’898	--	9’870’636	273’264	331’510
Zeit (Sek.)	0.0984	--	0.1422	0.1028	0.1084
Speicher Bytes	13’456’456	--	32’951’560	1’524’336	1’841’952
Speicher (%)	0. 361	--	0.765	0.114	0.121

Erkenntnisse aus den Tests

Insgesamt schneiden A* und Dijkstra am besten ab. Betrachtet man jedoch das Kriterium Distanz, ist BestFirst in einigen Fällen leicht besser als A*, da es stärker auf die Heuristik setzt. Allerdings ist die Speicherauslastung bei BestFirst mit Abstand am höchsten, da der Algorithmus oft einen grösseren Suchraum berücksichtigt. A* und Dijkstra sind in dieser Hinsicht effizienter, da sie den Suchraum gezielter eingrenzen. Für BreadthFirst liegen leider keine konkreten Testdaten vor, da der Algorithmus in unseren Testläufen viel zu lange benötigt. Nach fünfzehn Minuten Rechenzeit haben wir den Test für BreadthFirst abgebrochen.

Der BreadthFirst-Algorithmus verursacht eine sehr hohe Speicherauslastung, wie die 1,6 GB belegten `java.lang.Object[]`-Arrays und die 662 MB `java.util.ArrayList` zeigen. Dies deutet auf eine breite Suche hin, bei der viele Knoten und Verbindungen zwischengespeichert werden. Die exponentielle Zunahme der zu speichernden Knoten mit jeder weiteren Ebene verdeutlicht, dass BreadthFirst für grosse Suchräume speicherintensiv und ineffizient ist.

Class	Count	Shallow
java.lang.**Object[]**	27,608,326	1.6 GB
java.util.**ArrayList**	27,605,486	662.53 MB
int[]	1,703	1.32 MB
byte[]	20,632	1.14 MB
java.lang.**String**	19,677	472.25 kB
java.lang.**Class**	2,654	191.09 kB
java.util.concurrent.**ConcurrentHashMap$Node**	4,265	136.48 kB
java.util.**HashMap$Node**	3,797	121.5 kB
java.lang.reflect.**Method**	1,352	118.98 kB
char[]	34	70.33 kB
java.util.**HashMap$Node[]**	491	65.74 kB
java.util.**LinkedHashMap$Entry**	1,129	45.16 kB
java.util.concurrent.**ConcurrentHashMap$Node[]**	104	44.16 kB
MapData.**MapData$Destination**	1,652	39.65 kB
java.lang.**Class[]**	1,466	39.4 kB
java.lang.invoke.**MemberName**	909	36.36 kB
java.lang.**String[]**	1,042	35.05 kB

Der DepthFirst Algorithmus konnte zuerst bei einfachen Suchabfragen mit den anderen Algorithmen gut mithalten. Sobald die Suche jedoch komplexer wurde, hat der Algorithmus bei den Resultaten nachgelassen. Dies sieht man bei der Distanz, bei welcher DepthFirst eine ca. 10-fach längere Strecke vorschlägt.