# Graphen: Dijkstras-Algorithmus

Inhaltsverzeichnis

* 1 Edsger W. Dijkstra (ca. ½ Seite)
* 2 Graphen (ca. 1/2 Seiten)
  + 2.1 Undirected
  + 2.2 Directed
  + 2.3 Weighted
* 3 Der Dijktstra-Algorithmus (ca. 3 Seiten)
  + 3.1 Problemstellung und Voraussetzungen
  + 3.2 Beschreibung des Dijkstra-Algorithmus
  + 3.2 Lazy-Implementierung des Dijkstra-Algorithmus
  + 3.3 Eager-Implementierung des Dijkstra-Algorithmus
* 4 Erweiterung des Dijkstra-Algorithmus (ca. 1 1/2 Seite)
  + 4.1 A\*-Search
  + 4.2 Weitere
* 5. Visualizer (1 Seite)
  + Funktionsweise
  + Algorithmen Implementation
* 6 Anwendungsmöglichkeiten (ca. 1 1/2 Seiten)
  + 5.1 Navigation
  + 5.2 Routing

Ehrenwörtliche Erklärung

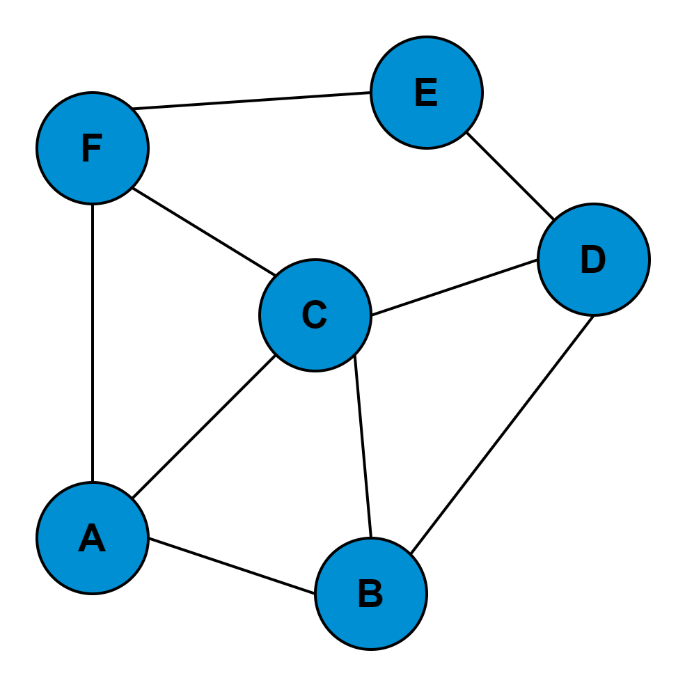
Wir versicheren hiermit, dass wir die vorliegend Arbeit mit dem Titel Graphen Dijkstra-Algorithmus selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

Graphen

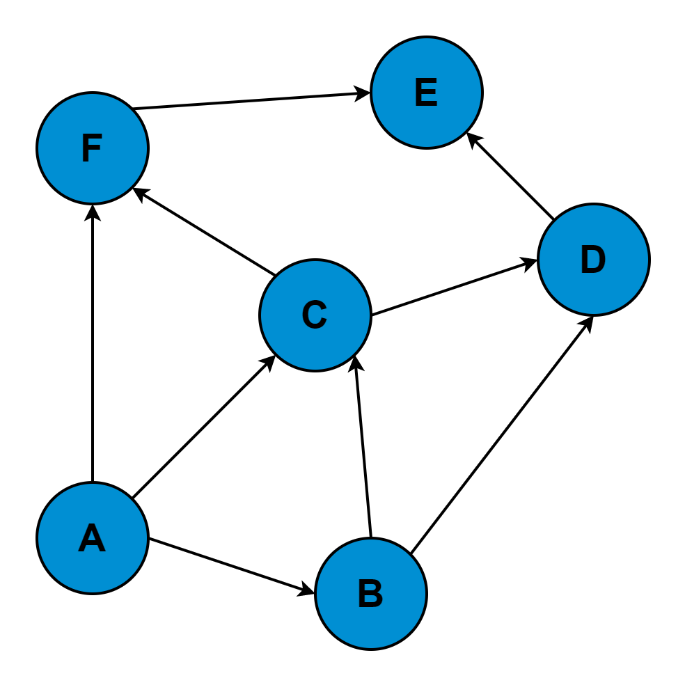
Ungerichteter Graph

In einem ungerichteten Graph haben die Kanten keine Orientierung oder Richtung. Dies bedeutet, dass die Kante (u, v) identisch ist mit der Kante (v, u). In so einem Graph könnten Knoten beispielsweise Städte darstellen und eine Kante könnte eine bidirektionale Straße repräsentieren.



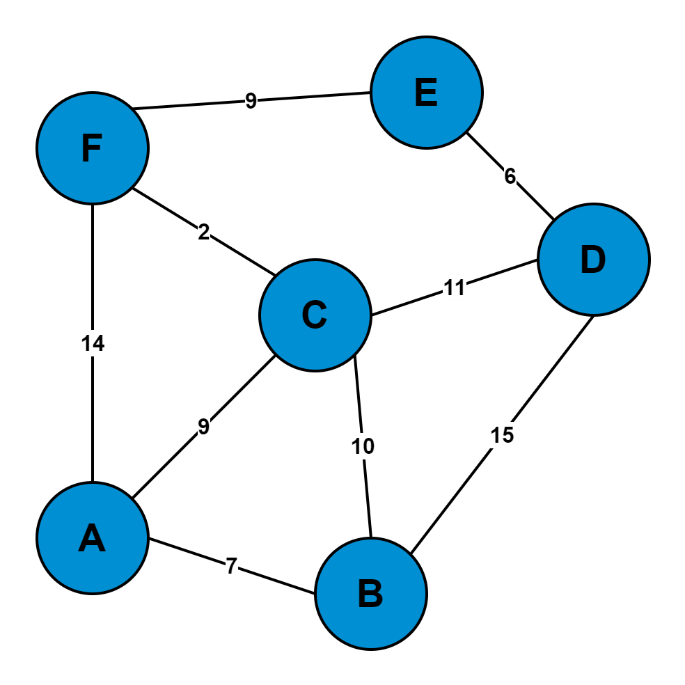
Gerichteter Graph

Im Gegensatz zum ungerichteten Graph haben alle Kanten in einem gerichteten Graph eine Orientierung oder Richtung. Deshalb beschreibt eine Kante (u, v) den Weg von dem Knoten u zum Knoten v. Die Knoten könnten Menschen darstellen und eine Kante könnte eine Person u sein, die der Person v ein Geschenk gekauft hat.



Gewichteter Graph

In einem gewichteten Graph enthalten die Kanten einen beliebigen Wert wie z.B. Kosten, eine Entfernung oder eine Menge. Gewichtete Kanten werden als Triplet (u, v, w) dargestellt und können in gerichteten oder ungerichteten Graphen vorkommen.



3 Beschreibung des Dijkstra-Algorithmus

3.1 Problemstellung und Voraussetzungen

Der Dijkstra-Algorithmus wird verwendet, um das Problem des kürzesten Pfades auf einem gewichteten Graphen zu lösen. Das Gesamtgewicht eines Pfades ergibt sich aus der Summe der Gewichte aller Kanten entlang des Pfades. Theoretisch ließe sich dies zwar umsetzen, indem einfach alle Pfade berechnet und denjenigen mit den geringsten Pfadkosten bzw. dem geringsten Gewicht auswählt, aufgrund der sehr hohen Anzahl und Einbeziehung vieler Pfade, deren Betrachtung nicht zielführend ist, da sie beispielsweise unnötige Umwege machen oder zyklische Segmente enthalten.  
Eine Sonderform und Vereinfachung des kürzeste-Pfade-Problems stellt das kürzeste-Pfade-Problem mit Startknoten dar, zu dessen Lösung der Dijkstra-Algorithmus verwendet werden kann. Dabei kann sowohl der kürzeste Pfad zu einem bestimmten Zielknoten, als auch der kürzeste Pfad zu jedem Knoten berechnet werden. Dabei beruhen Algorithmen, die zur Lösung dieses Problems verwendet werden, üblicherweise auf der Eigenschaft, dass ein kürzester Pfad zwischen zwei nicht benachbarten Knoten auch weitere kürzeste Pfade enthält. Diese Eigenschaft bezeichnet man als optimale-Teilstruktur-Eigenschaft, und sie stellt eine wichtige Voraussetzung für die Anwendbarkeit von Greedy-Algorithmen, zu denen auch Dijkstras Algorithmus zählt, dar.  
Der Dijkstra-Algorithmus setzt weiterhin voraus, dass alle Gewichte des zu untersuchenden Graphen nicht negativ sind, da dies sonst zu zyklischen Pfaden führen könnte. Ob der Graph gerichtet oder ungerichtet ist spielt keine Rolle.

3.2 Beschreibung des Dijkstra-Algorithmus

Dijkstras Algorithmus nimmt zunächst ein unendliche Kosten zu allen Knoten außer dem Startknoten an, dieser bekommt Kosten von null zugewiesen und er wird als aktuell aktiver Knoten gesetzt. Ausgehend vom aktiven Knoten werden anschließend die vorhandenen Kosten aller benachbarten Knoten (zu denen eine Kante führt) mit der Summe aus den dem aktiven Knoten zugewiesenen Kosten und den Kosten der Kante, die den aktiven mit dem benachbarten Knoten verbindet, verglichen. Dem benachbarten Knoten wird nun der geringere der beiden Werte als neue Kosten zugewiesen. Sind alle benachbarten Knoten besucht, so wird der der aktuelle Knoten als besucht markiert sowie der unbesuchte Knoten, dem aktuell die geringsten Kosten zugewiesen sind, als aktiver Knoten gesetzt und das Vorgehen wiederholt.  
Falls alle Knoten besucht wurden oder die geringsten Kosten zu einem unbesuchten Knoten unendlich sind (dies bedeutet, dass kein Pfad vom Startknoten zu diesem Knoten existiert), ist Dijkstras Algorithmus beendet.   
Ist alternativ nur ein bestimmter Knoten als Zielknoten bekannt und ein Pfad ist nur zu diesem gewünscht, so kann der Dijkstra-Algorithmus bereits beendet werden, sobald dieser als aktiver Knoten gesetzt wird, da zu diesem Zeitpunkt bereits der kürzeste Pfad zum Zielknoten gefunden wurde. Dies wird als Early Stopping bezeichnet und stellt eine Optimierung des Algorithmus dar, da so in vielen Fällen deutlich weniger Knoten besucht und Pfade überprüft werden müssen.

3.3 Lazy-Implementierung des Dijkstra-Algorithmus

Pseudocode:

function dijkstra(graph, n, start, destination):

visited = [false, false, …, false] # size n

prev = [null, null, …, null] # size n

distance = [∞, ∞, …, ∞, ∞] # size n

distance[start] = 0

pq = empty priority queue

pq.insert((start, 0))

while pq.size() != 0:

index, min\_value = pq.poll()

visited[index] = true

for edge in g[index]:

if visited[edge.to]: continue

new\_distance = distance[index] + edge.cost

if new\_distance < distance[edge.to]:

prev[edge.to] = index

distance[edge.to] = new\_distance

pq.insert((edge.to, new\_distance))

if index == destination:

return distance[destination]

return ∞

3.4 Eager-Implementierung des Dijkstra-Algorithmus

Pseudocode:

function dijkstra(g, n, start, destination):

visited = [false, false, …, false] # size n

prev = [null, null, …, null] # size n

distance = [∞, ∞, …, ∞, ∞] # size n

distance[start] = 0

ipq = empty indexed priority queue

ipq.insert((start, 0))

while pq.size() != 0:

index, min\_value = pq.poll()

visited[index] = true

for edge in g[index]:

if visited[edge.to]: continue

new\_distance = distance[index] + edge.cost

if new\_distance < distance[edge.to]:

prev[edge.to] = index

distance[edge.to] = new\_distance

if edge.to not in ipq:

ipq.insert(edge.to, new\_distance)

else:

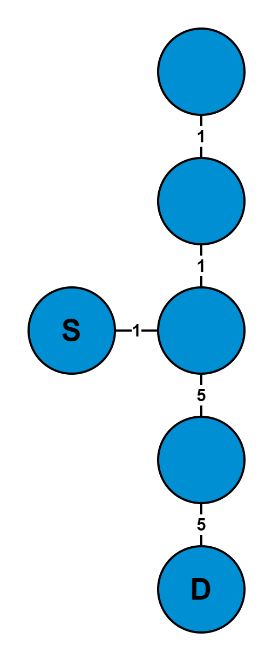
ipq.descreaseKey(edge.to, new\_distance)

if index == destination:

return distance[destination]

return ∞

A\* Search Algorithm



A\* Search ist ein Algorithmus, der bei der Wegfindung und Durchquerung von Graphen weit verbreitet ist. Der Algorithmus zeichnet effizient einen begehbaren Pfad zwischen mehreren Knoten oder Punkten auf dem Graphen. Auf einer Karte mit vielen Hindernissen kann die Wegfindung von den Punkten AA bis BB schwierig sein. Der A\* Search Algorithmus führt jedoch eine Heuristik in einen regulären Algorithmus ein, der im Wesentlichen bei jedem Schritt vorausschaut, damit eine bessere Entscheidung getroffen wird. Deshalb kann A\* Search als eine Erweiterung des Dijkstra-Algorithmus mit einigen Merkmalen der Breitensuche (BFS) betrachtet werden. Wie Dijkstra arbeitet A\*, indem es einen kostengünstigsten Pfad vom Startknoten zum Zielknoten erstellt. Was der A\* Search für viele Suchen anders und besser macht, ist, dass der Algorithmus für jeden Knoten eine Funktion f(n) verwendet, die eine Schätzung der Gesamtkosten eines Pfades unter Verwendung dieses Knotens liefert. Daher ist A\* Search eine heuristische Funktion, die sich von einem Algorithmus dadurch unterscheidet, dass eine Heuristik eher eine Schätzung ist, die nicht zwangsläufig korrekt sein muss.

A\* erweitert Pfade, die bereits kostengünstiger sind, indem diese Funktion verwendet wird:

f(n) = geschätzte Gesamtkosten des Pfades durch den Knoten n

g(n) = Kosten bis zum Erreichen von Knoten n

h(n) = geschätzte Kosten von n bis zum Ziel.

Die Verwendung einer guten Heuristik ist wichtig, um die Leistung von A\* Search zu bestimmen. Der Wert von h(n) würde idealerweise den genauen Kosten für das Erreichen des Ziels entsprechen. Dies ist jedoch nicht möglich, da wir den Weg nicht kennen. Wir können jedoch eine Methode wählen, die uns manchmal den genauen Wert liefert, z.B. wenn Sie geradeaus ohne Hindernisse fahren. Dies führt zu einer perfekten Leistung von A\* in einem solchen Fall.

Wir wollen in der Lage sein, eine Funktion h(n) auszuwählen, die geringer ist als die Kosten, um unser Ziel zu erreichen. Dadurch kann h genau arbeiten. Wenn wir einen höheren Wert für h wählen, führt dies zu einer schnelleren, aber weniger genauen Leistung. Daher ist es üblich ein h(n) zu wählen, das geringer ist als die realen Kosten. Der A\* Search implementiert einer der beiden Heuristiken: Manhatten Distanz, Euklidische Distanz.

Manhatten Distanz:

Diese Methode zur Berechnung von h(n) wird Manhattan-Methode bezeichnet, weil sie die Gesamtzahl der Quadrate berechnet, die horizontal und vertikal bewegt wurden, um das Zielquadrat vom aktuellen Quadrat aus zu erreichen. In diesem Fall ignorieren wir diagonale Bewegungen und eventuelle Hindernisse.

Diese Heuristik ist immer dann genau, wenn unser Weg einer geraden Linie folgt. Das heißt, A\* Search findet Pfade, die Kombinationen von geradlinigen Bewegungen sind. Manchmal bevorzugen wir vielleicht einen Weg, der einer geraden Linie direkt zu unserem Ziel folgt.

Euklidische Distanz:

Diese Heuristik ist etwas genauer als ihr Gegenstück. Wenn wir versuchen, beide gleichzeitig im selben Labyrinth zu laufen, bevorzugt die euklidische Heuristik einen Pfad entlang einer geraden Linie. Dies ist genauer, aber auch langsamer, da ein größeres Gebiet erkundet werden muss, um den Weg zu finden.



Der Hauptnachteil des A\* Algorithmus und in der Tat jeder Best-First-Suche ist sein Speicherbedarf. Da zumindest die gesamte offene Liste gespeichert werden muss, ist der A\*-Algorithmus in der Praxis stark räumlich begrenzt und nicht praktischer als der Best-First-Suchalgorithmus auf aktuellen Maschinen.

Die Zeitkomplexität von A\* hängt von der Heuristik ab. Im schlimmsten Fall ist die Anzahl der expandierten Knoten exponentiell in der Länge der Lösung (der kürzeste Weg), aber polynomiell, wenn der Suchraum ein Baum ist.

5. Visualizer

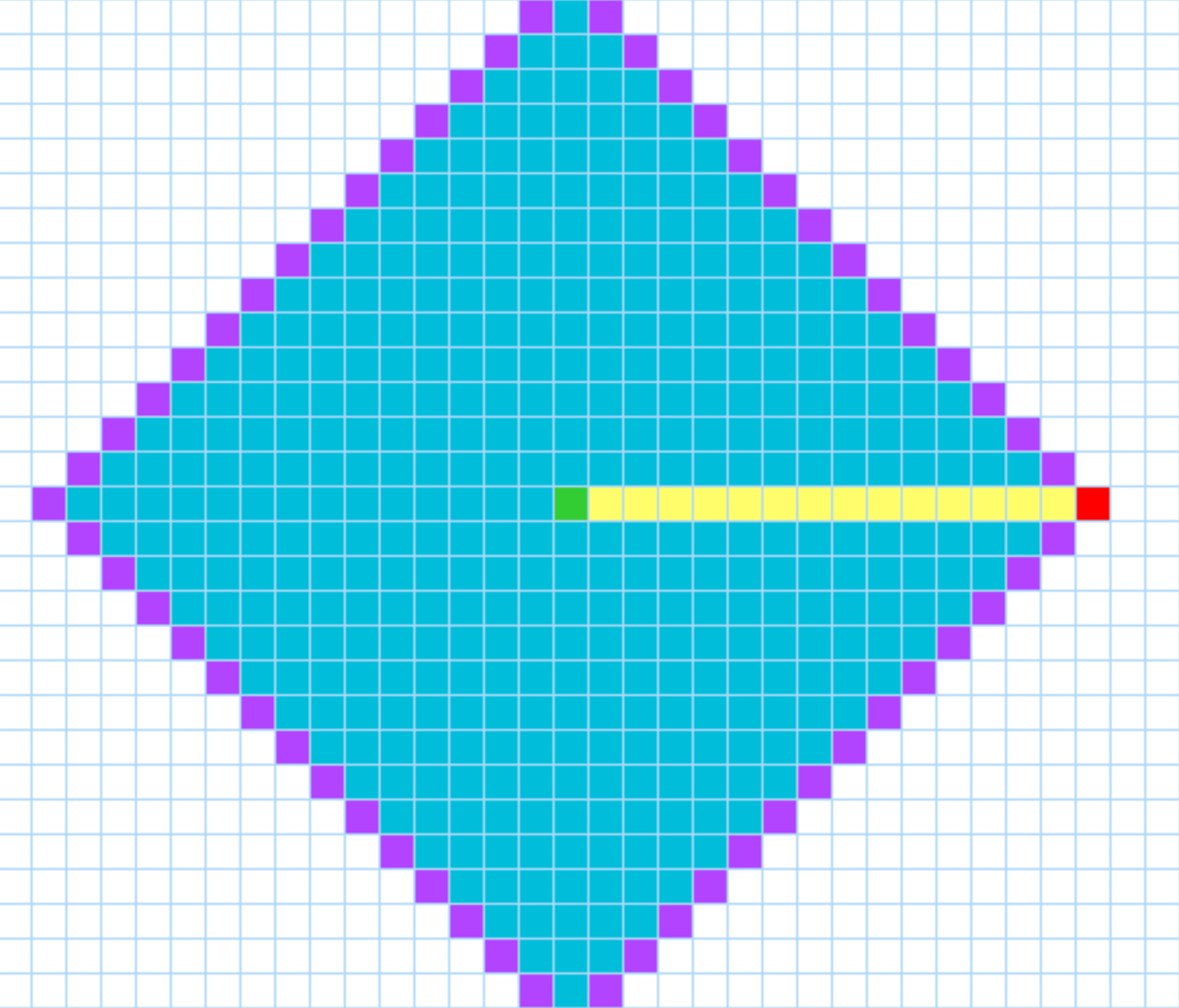
Der Pathfinding Visualizer basiert auf dem Modul Pygame und implementiert sowohl den Dijkstra-Algorithmus mit einer Prioritätswarteschlange als auch den A\* Search Algorithmus mit der Manhatten Distanz als Heuristikfunktion auf einem Graphen, der aus x, y Koordinaten besteht. Anhand eines selbst ausgewählten Start- und Zielknoten können Sie beliebig oft die Algorithmen visualisieren und zudem auch selbst gewählte Hindernisse dem Graphen hinzufügen oder sich ein Labyrinth aus Hindernissen generieren lassen.

Im Folgenden werden die Bedeutungen der Farben im Visualizer definiert:

* Startknoten: grün
* Zielknoten: rot
* Hindernis: schwarz
* Kürzester Pfad: gelb
* Besuchte Knoten: blau
* Zu besuchende Knoten: lila
* Unbesuchte Knoten: weiß

Im Folgenden wird die Funktionsweise des Visualizers erklärt:

* Drücken Sie die linke Maustaste um Start-, Zielknoten sowie Hindernisse festzulegen
* Drücken Sie die rechte Maustaste um Start-, Zielknoten sowie Hindernisse zu entfernen
* Drücken Sie c um all Knoten zurückzusetzen
* Drücken Sie a um den A\* Search Algorithmus zu visualisieren
* Drücken Sie d um den Dijkstra-Algorithmus zu visualisieren
* Drücken Sie m um ein zufälliges Labyrinth aus Hindernisse zu generieren



6. Anwendungsmöglichkeiten

Wir haben oft versucht, in Google Maps oder Apple Karten die Entfernung von einer Stadt zur anderen oder von Ihrem Standort zum nächsten gewünschten Standort zu ermitteln. Dort trifft man auf den Dijkstra-Algorithmus, da es verschiedene Routen bzw. Pfade zum Ziel gibt. Betrachten Sie Deutschland als Graph und stellen Sie eine Stadt oder einen Ort mit einem Scheitelpunkt und die Route zwischen zwei Städten oder Orten als Kante dar. Dann können Sie Dijkstras Algorithmus verwenden, um die kürzeste Route zwischen zwei beliebigen Städten oder Orten zu berechnen.

Auch bei der Erstellung eines Flugplans für die Kunden kommt Dijkstra zum Einsatz. Der Agent hat Zugriff auf eine Datenbank mit allen Flughäfen und Flügen. Neben Flugnummer, Abflughafen und Zielort haben die Flüge Abflug- und Ankunftszeit. Insbesondere möchte der Agent die früheste Ankunftszeit für das Ziel bei gegebenem Startflughafen und Startzeit bestimmen.

Dijkstras Algorithmus wird auch im Internet als Routing-Algorithmus im Open-Shortest-Path-First-Protokoll (OSPF) eingesetzt, das verwendet wird, um den besten Pfad zwischen dem Quell- und dem Zielrouter mithilfe seines eigenen Shortest Path First zu finden. Dijkstras Algorithmus wird häufig in den Routing-Protokollen verwendet, die von den Routern benötigt werden, um ihre Weiterleitungstabelle zu aktualisieren. Der Algorithmus stellt dabei den kürzesten Kostenpfad vom Quellrouter zu anderen Routern im Netzwerk bereit.

Um einen Dateiserver in einem LAN zu bestimmen, kann der Dijkstra Algorithmus ebenfalls verwendet werden. Bedenken Sie, dass für die Übertragung von Dateien von einem Computer auf einen anderen Computer unendlich viel Zeit benötigt wird. Um die Anzahl der „Hops“ vom Dateiserver zu jedem anderen Computer im Netzwerk zu minimieren, besteht die Idee darin, den kürzesten Pfad zwischen den Netzwerken zu minimieren, was zu einer minimalen Anzahl von Hops führt.