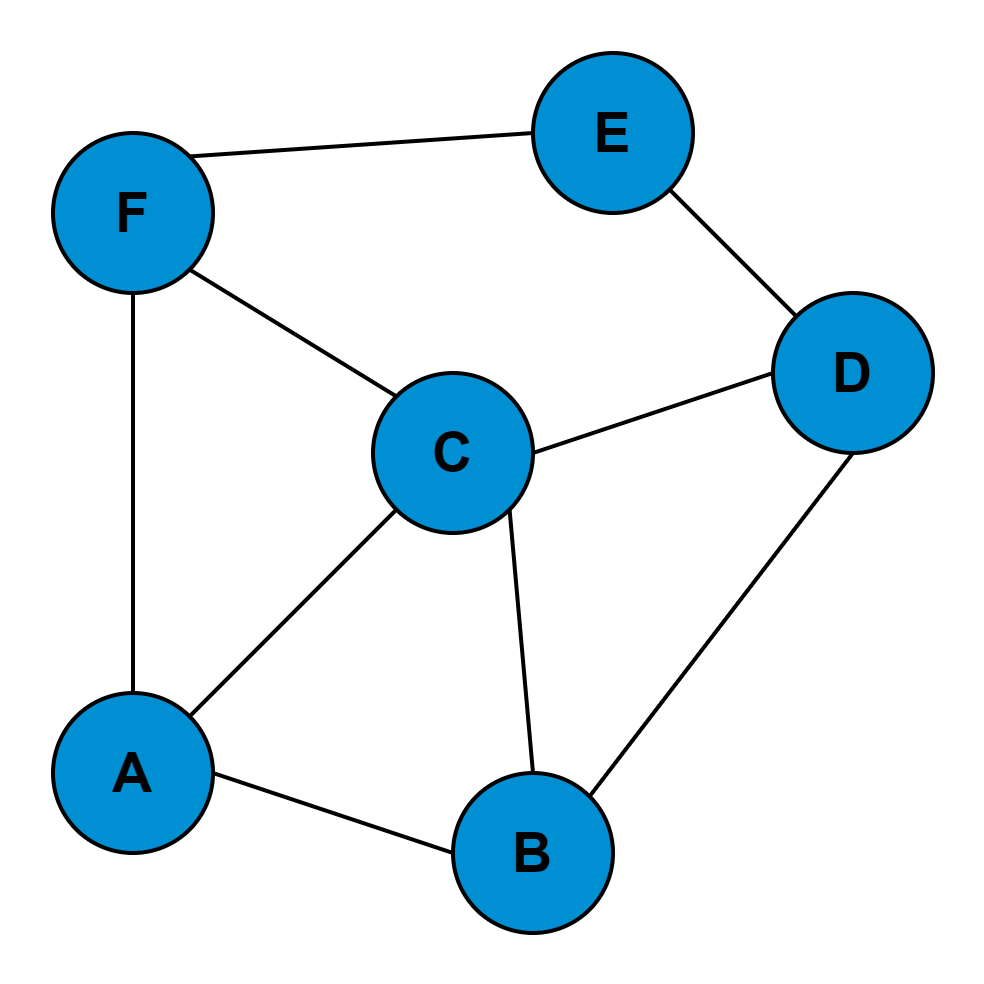
Inhaltsverzeichnis

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

Graphen

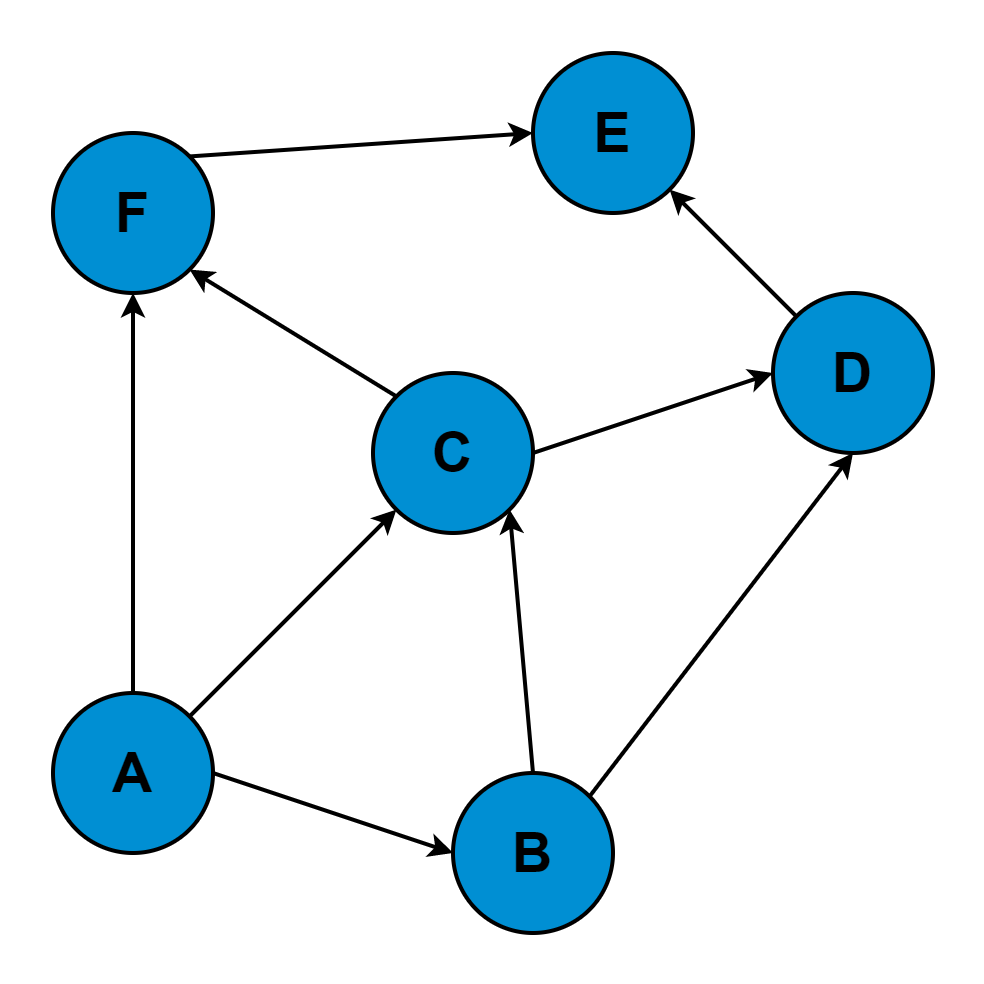
Ungerichteter Graph

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 representieren.



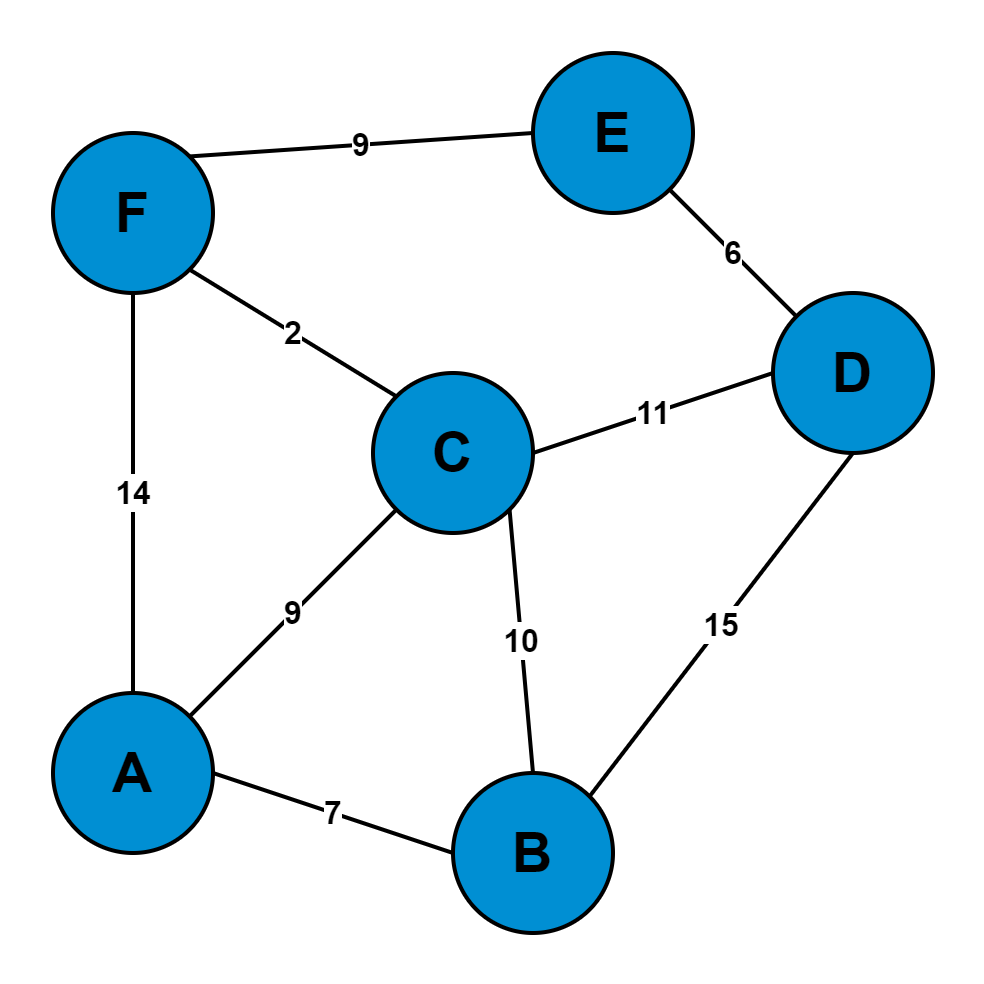
Gerichteter Graph

In einem gerichteten Graph haben alle Kanten eine Orientierung oder Richtung. Eine Kante (u, v) beschreibt den Weg von dem Knoten u zum Knoten v. Die Knoten könnten Menschen darstellen und eine Kante könnte eine Person u sein, die 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.



Dijkstra Lazy

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 ∞

Dijkstra Eager

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:

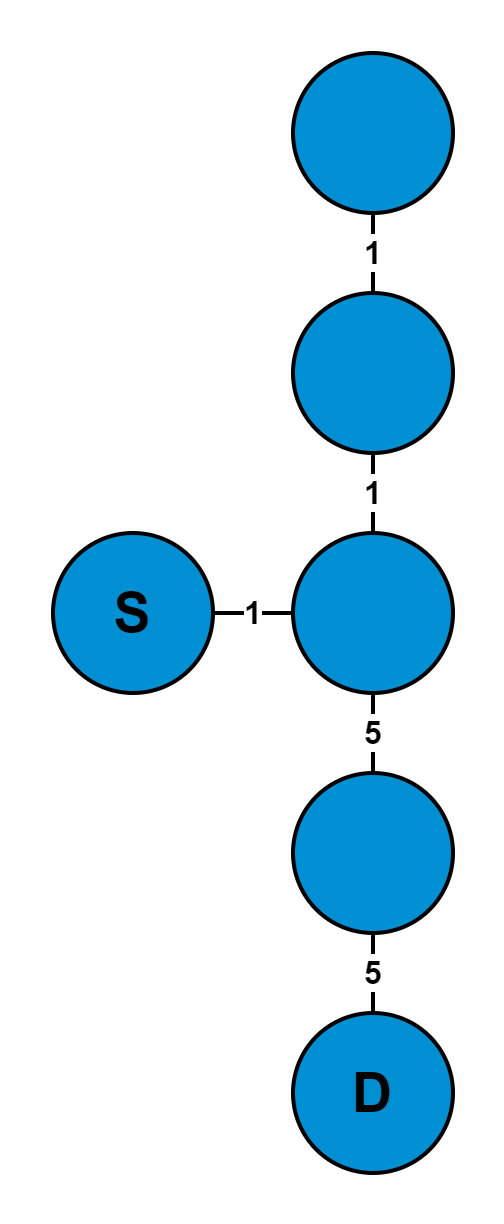
ipq.descreaseKey(edge.to, new\_distance)

if index == destination:

return distance[destination]

return ∞

A\* Search Algorithm



A\* Search ist ein Algorithmus, der bei der Wegfindung und Graphendurchquerung 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. Ein Roboter zum Beispiel fährt in eine Richtung weiter, bis er auf ein Hindernis stößt.

Der A\*-Algorithmus führt jedoch eine Heuristik in einen regulären Graphensuchalgorithmus ein, der im Wesentlichen bei jedem Schritt vorausschaut, damit eine optimalere Entscheidung getroffen wird. Mit A\* würde ein Roboter stattdessen einen Weg ähnlich dem Diagramm rechts unten finden.

A\* ist eine Erweiterung des Dijkstra-Algorithmus mit einigen Merkmalen der Breitensuche (BFS).

Wie Dijkstra arbeitet A\*, indem es einen kostengünstigsten Pfadbaum vom Startknoten zum Zielknoten erstellt. Was A\* für viele Suchen anders und besser macht, ist, dass A\* für jeden Knoten eine Funktion f(n)f(n) verwendet, die eine Schätzung der Gesamtkosten eines Pfads unter Verwendung dieses Knotens liefert. Daher ist A\* eine heuristische Funktion, die sich von einem Algorithmus dadurch unterscheidet, dass eine Heuristik eher eine Schätzung ist und nicht unbedingt beweisbar korrekt ist.

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

f(n)=g(n)+h(n),

f(n)=g(n)+h(n),

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

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

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

Dies ist der heuristische Teil der Kostenfunktion, also wie eine Vermutung.



Die Verwendung einer guten Heuristik ist wichtig, um die Leistung von A\* 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 nicht einmal den Weg 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 normalerweise der Fall, dass wir ein h(n) wählen, das geringer ist als die realen Kosten.

Manhatten Distanz:

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

Diese Heuristik ist immer dann genau, wenn unser Weg einer geraden Linie folgt. Das heißt, A\* 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 in Manhattan. Wenn wir versuchen, beide gleichzeitig im selben Labyrinth zu laufen, bevorzugt der euklidische Pfadfinder 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.