1 Kürzeste Pfade

Das Problem, einen kürzesten Pfad zwischen zwei Knoten eines Digraphen zu finden, hat vielfältige Anwendungen, zum Beispiel in Informations- und Verkehrsnetzwerken. In den folgenden Abschnitten werden effiziente Algorithmen zur Lösung des Kürzesten-Pfade-Problems erarbeitet. Wir wollen aber zunächst das Problem exakt definieren.

Es bezeichne $G = (V, E)$ einen Digraphen mit Knotenmenge V und Kantenmenge E. Wir unterscheiden die folgenden Problemvarianten:

Single Source Shortest Path: Gegeben ein Digraph $G = (V, E)$ mit nicht-negativen Kantenkosten und einem ausgezeichneten Startknoten s, finde für jeden Knoten $i \in V$ einen kürzesten Pfad $P(i)$ von s nach i sowie seine Länge $d(i)$. OBdA setzen wir voraus, dass s ein *Wurzelknoten* in G ist, d.h., jeder Knoten $i \in V$ ist von s aus durch einen Pfad in G erreichbar.[1]

All Pair Shortest Path: Gegeben ein Digraph $G = (V, E)$ mit nicht-negativen Kantenkosten, finde für jedes Knotenpaar $i, j \in V$ einen kürzesten Pfad $P(i, j)$ von i nach j sowie seine Länge $d(i, j)$.

Im Folgenden sei stets

$$n = |V| \text{ und } m = |E|. \tag{1}$$

Oberflächlich betrachtet benötigt man i.A. Laufzeit und Speicherplatz $\Omega(n^2)$ zur Ausgabe eines Pfadsystems $(P(i))_{i \in V}$, da G eventuell $\Omega(n)$ Pfade einer Länge $\Omega(n)$ enthält. Glücklicherweise kann das Pfadsystem viel platzeffizienter abgespeichert werden.

Zentrale Beobachtung Wenn P ein kürzester Pfad von s nach j und (i, j) die letzte Kante in P ist, dann ist der um diese Kante verkürzte Pfad ein kürzester Pfad von s nach i.

Definition 1.1 *Eine Kollektion $(P(i))_{i \in V}$ kürzester Pfade heißt* baumartig*, falls für alle $j \in V$ folgendes gilt: wenn (i, j) die letzte Kante von Pfad $P(j)$ ist, dann ergibt sich $P(i)$ aus $P(j)$ durch die Wegnahme der Kante (i, j).*

Aus der zentralen Beobachtung folgt, dass für jeden Digraphen G eine baumartige Kollektion kürzester Pfade existiert. Sei $(P(i))_{i \in V}$ eine baumartige Kollektion kürzester Pfade. Die Tatsache, dass (i, j) die letzte Kante auf dem Pfad $P(j)$ ist, kann mit Hilfe eines Pointers $Z(j) = i$ gespeichert werden. Auf diese Weise erhalten wir eine Pointerkollektion $Z = (Z(i))_{i \in V}$ (wobei $Z(s) =$ **nil**) mit folgender Eigenschaft: die von j ausgehende Pointerkette durchläuft den Pfad $P(j)$ in umgekehrter Orientierung (also von j nach s). Offensichtlich repräsentiert die Pointerkollektion Z einen Inbaum mit Wurzel s, wobei $Z(j) = i$ angibt, dass i der Elterknoten zu j (also j Kind von i) ist.

Definition 1.2 *Der Ausbaum $T_{SP} = (V, E_{SP})$ mit*

$$E_{SP} := \{(Z(j), j) : \ j \in V \setminus \{s\}\}$$

mit Wurzel s heißt Kürzester-Pfade-Baum *für G.*

[1] Notfalls fügen wir Hilfskanten mit unendlichen Kantenkosten ein.

Nach der vorangegangenen Diskussion sollte klar sein, dass $P(i)$ identisch ist zu dem eindeutigen Pfad von s nach i in T_{SP}. Der Ausbaum T_{SP} erlaubt also, die Kollektion $(P(i))_{i \in V}$ kürzester Pfade platzeffizient (Platzkomplexität $\theta(n)$) darzustellen. Es ergibt sich somit die folgende Version des allgemeinen „Single Shortest Path"-Problems:

Single Source Shortest Path (Version 2) Gegeben ist ein Digraph $G = (V, E)$ mit nichtnegativen Kantenkosten $c(i, j)$ für jede Kante $(i, j) \in E$ und einem Startknoten $s \in V$ (oBdA ein Wurzelknoten). Gesucht ist ein Kürzester-Pfade-Baum für G sowie die zugehörigen Distanzwerte $d(i)$ für alle $i \in V$.

Man überlegt sich leicht, dass sich der Kürzeste-Pfade-Baum T_{SP} in $O(n)$ Schritten aus der Pointerkollektion Z, im Folgenden *Pfad-Pointer* genannt, berechnen läßt. Die zentrale Aufgabe ist demnach die Berechnung der Pfad-Pointer.

1.1 Kürzeste Pfade in azyklischen Digraphen (DAGs)

Wir starten in diesen Abschnitt mit zwei zentralen Definitionen.

Definition 1.3 *Sei $G = (V, E)$ ein DAG und $a, b \in V$. a heißt* Vorgänger *von b in G, falls es in G einen Pfad von a nach b gibt. Wenn zusätzlich $a \neq b$ gefordert wird, sprechen wir von einem* echten Vorgänger. *a heißt* direkter Vorgänger *von b in G, falls $(a, b) \in E$.*

Definition 1.4 *Es sei $L = (v_1, \ldots, v_n)$ eine Auflistung aller Elemente der Knotenmenge V eines DAGs $G = (V, E)$. L heißt* topologisch sortiert*, falls ein Knoten v in L erst dann auftaucht, wenn alle seine echten Vorgänger zuvor schon aufgelistet wurden.*

Das folgende Resultat ist leicht zu beweisen:

Satz 1.5 *Aus einem in Adjazenzlistendarstellung gegebenen DAG $G = (V, E)$ mit n Knoten (darunter mindestens ein Wurzelknoten) und m Kanten kann man in $O(m)$ Schritten eine topologisch sortierte Liste seiner Knoten erstellen.*[2]

Algorithmus zur Berechnung der kürzesten Pfade eines DAG Sei $G = (V, E)$ ein DAG mit Wurzelknoten s und Kantenkosten[3] $c(u, v)$ für alle $(u, v) \in E$. Wir können die Länge der kürzesten Pfade von s nach v (für alle $v \in V$) nach folgenden Schema berechnen: Durchlaufe die Knoten von G in topologischer Ordnung (also s zuerst). Setze $d(s) \leftarrow 0$. Für alle $v \in V$ mit direkten Vorgängern $v_1, \ldots, v_k$ setze

$$d(v) \leftarrow \min_{1 \leq i \leq k} \{d(v_i) + c(v_i, v)\} \ . \tag{2}$$

Satz 1.6 *Der skizzierte Algorithmus berechnet für alle $v \in V$ in $O(m)$ Schritten die Länge $d(v)$ eines kürzesten Pfades von s nach v.*

[2] Falls G keinen Wurzelknoten enthält, so braucht man $O(n + m)$ Schritte zur topologischen Sortierung. Die Existenz eines Wurzelknotens impliziert, dass $m \geq n - 1$ und somit $O(n + m) = O(m)$.

[3] Diesmal sind sogar negative Kantenkosten zulässig.

Beweis Dass $O(m)$ Schritte zur Berechnung von $d(v)$ (für alle $v \in V$) ausreichen, ergibt sich leicht aus Satz 1.5 und der Tatsache, dass die Zuweisung (2) für jeden Knoten v im Wesentlichen $O(\text{outdeg}(v))$ Schritte beansprucht. Hierbei bezeichnet outdeg(v) — genannt der *Ausgangsgrad* von v — die Anzahl der von v ausgehenden Kanten. Die Korrektheit des Verfahrens ergibt sich induktiv. Offensichtlich ist die Wertzuweisung $d(s) \leftarrow 0$ korrekt, da der nur aus s bestehende Punktpfad (der Länge 0) der einzige Pfad von s nach s ist. Da die Knoten von V in topologischer Ordnung durchlaufen werden, können wir bei der Wertzuweisung (2) induktiv voraussetzen, dass $d(v_i)$ die Länge eines kürzesten Pfades von s nach v_i angibt. Offensichtich ist der $d(v)$ zugewiesene Wert dann die Länge eines kürzesten Pfades von s nach v. **qed.**

Wie muss der skizzierte Algorithmus erweitert werden, damit wir nicht nur die Länge der kürzesten Pfade erhalten sondern auch den Kürzesten-Pfade-Baum? Zwei Möglichkeiten:

On-line Wenn bei der Wertzuweisung (2) das Minimum bei Index $j \in \{1, \ldots, k\}$ angenommen wird[4], dann setzen wir einen Pfad-Pointer von v nach v_j. Nach Ablauf des Algorithmus repräsentieren die Pfad-Pointer einen Kürzesten-Pfade-Baum.

Off-line Es ist nicht schwer, geeignete Pfad-Pointer nachträglich mit Hilfe des Array $d(v)$ zu berechnen.

1.2 Der Algorithmus von Dijkstra

Wir beschreiben in diesem Abschnit den Algorithmus von Dijkstra zum Lösen des „Single Source Shortest Path"-Problems. Die Eingabe besteht aus einem Digraph $G = (V, E)$ mit einem ausgezeichneten Startknoten s und nicht-negativen Kantenkosten $c(v, w)$. Für jeden Knoten $v \in V$ soll die Länge $d(v)$ eines kürzesten Pfades in G von s nach v berechnet werden.[5] Dijkstras Algorithmus verwendet dynamisches Programmieren. Zu jedem Zeitpunkt gibt es zwei Knotenmengen T und $S = V \setminus T$ mit folgenden Invarianzeigenschaften:

1. Für alle $v \in S$: $d(v)$ enthält bereits den korrekten Wert und der kürzeste Pfad von s nach v (unter allen möglichen Pfaden) kann so realisiert werden, dass er nur Zwischenknoten aus S verwendet.

2. Für alle $w \in T$: $d(w)$ enthält die Länge des kürzesten Pfades von s nach w, der nur Zwischenknoten aus S benutzt.

Anfangs gilt

$$S = \{s\}, T = V \setminus \{s\}, d(v) = c(s, v) \text{ für alle } v \in V$$

mit der Konvention $c(i, j) = \infty$, falls $(i, j) \notin E$. Der Algorithmus arbeitet in $n-1$ Iterationen und vergrößert S in jeder Iteration um einen Knoten. Grundlage für den Algorithmus ist folgendes

[4]Mehrdeutigkeiten können willkürlich aufgelöst werden.

[5]Die Erweiterung des Algorithmus zur Berechnung des Kürzesten-Pfade-Baums kann wieder durch Setzen von Pointern an geeigneter Stelle erfolgen.

Lemma 1.7 *Sei w_0 der Knoten aus T mit minimalem Wert von $d(w)$, d.h.,*

$$w_0 = \arg\min\{d(w) : \ w \in T\} \ .$$

Dann ist $d(w_0)$ die Länge eines kürzesten Pfades von s nach w_0.

Beweis Wegen der Invarianzeigenschaft für T genügt es einzusehen, dass ein kürzester Pfad von s nach w_0 existiert, der nur Zwischenknoten in S verwendet. Wir machen die scheinheilige Annahme es existiere ein Pfad P mit Zwischenknoten außerhalb S, dessen Länge $d(w_0)$ unterschreitet. Sei u der früheste Knoten außerhalb S auf P. Dann gilt: $d(u) < d(w_0)$. Dies ist ein Widerspruch zur Wahl von w_0. **qed.**

Es ist daher möglich w_0 von T nach S zu transportieren. Es ist leicht einzusehen, dass die Invarianzbedingungen für T durch die Aktualisierung

$$\forall w \in T : \ d(w) = \min\{d(w), d(w_0) + c(w_0, w)\}$$

erhalten bleiben.
Diese Ideen können nun zu Dijkstras Algorithmus zusammengefasst werden:

```
begin
    T ← V \ {s}; d(s) ← 0
    for each w ∈ T do d(w) ← c(s, w);
    while T ≠ ∅ do
      begin
        w_0 ← arg min_{w∈T} d(w);
        remove w_0 from T;
        for each w ∈ T do d(w) ← min{d(w), d(w_0) + c(w_0, w)}
      end
end
```

Die Aktualisierung von $d(w)$ in der „for each“-Schleife kann natürlich auf die direkten Nachfolger von w_0 beschränkt werden.

Nach den vorangegangenen Diskussionen ist klar, dass Dijkstras Algorithmus das „Single Source Shortest Path“-Problem korrekt löst. Die Laufzeit hängt von der gewählten Implementierung ab:

Kostenmatrix-Implementierung Die Laufzeit wird durch die while-Schleife dominiert. Wenn der Digraph G und seine Kantenkosten durch eine Kostenmatrix $C[v, w]$ gegeben sind, dann kann jede der $n-1$ Iterationen auf die offensichtliche Weise in $O(n)$ Schritten ausgeführt werden. Dies führt zu einer Gesamtlaufzeit der Größenordnung $O(n^2)$.

Adjazenzlisten-Implementierung Diesmal wird G durch um Kantenkosten erweiterte Adjazenzlisten dargestellt. Zur Beschleunigung der arg min-Operation werden die Knoten $w \in T$ mit Schlüsselwert $d(w)$ in einer PRIORITY QUEUE Q verwaltet. Eine

PRIORITY QUEUE unterstützt die Operationen INSERT, MIN, DELETE-MIN und DECREASE-KEY. Wie wir wissen ist eine PRIORITY QUEUE als HEAP implementierbar, so daß jede der vier Grundoperationen in $O(\log n)$ Schritten ausführbar ist.[6] Anfangs werden alle Knoten außer s in Q aufgenommen. Danach wird kein INSERT mehr benötigt. Der Knoten w_0, der von T nach S wandern soll, wird mit Operation MIN gefunden und mit DELETE-MIN aus Q entfernt. Bisher haben wir nur $O(n)$ Grundoperationen eingesetzt. Leider machen die Aktualisierungen der Schlüsselwerte für alle direkten Nachfolger von w_0 Anwendungen der Operation DECREASE-KEY notwendig. Jede Kante von G löst potenziell eine solche Operation aus. Es ist nicht schwer sich zu überlegen, dass die (maximal) m Rebalancierungs-Operationen die Laufzeit dominieren. Dies führt zu einer Gesamtlaufzeit der Größenordnung $O(m \log n)$.

Verbesserte Adjazenzlisten-Implementierung Es gibt eine (Ihnen vermutlich unbekannte) verbesserte Implementierung einer PRIORITY QUEUE in Form eines sogenannten FIBONACCI-HEAP's. Mit einem FIBONACCI-HEAP ist eine Anwendung der Grundoperationen INSERT, MIN und DECREASE-KEY in konstanter *amortisierter* Laufzeit[7] ausführbar. Die Operation DELETE-MIN benötigt amortisiert $O(\log n)$ Schritte.[8] In Dijkstra's Algorithmus werden die Operationen INSERT, MIN und DELETE-MIN höchstens n-mal und die Operation DECREASE-KEY höchstens m-mal ausgeführt. Die gesamte Anzahl der Rechenschritte ist nun durch $O(m + n \log n)$ beschränkt.

Ein Vergleich der Laufzeitschranken der ersten beiden Implementierungen zeigt, dass für „dichte Digraphen“ (viele Kanten) die Kostenmatrix-Implementierung und für „dünne Digraphen“ die Adjazenzlisten-Implementierung (wenige Kanten) vorzuziehen ist. Die Grenze zwischen „dicht“ und „dünn“ liegt bei $m = \theta(n^2/\log n)$. Die verbesserte Adjazenzlisten-Implementierung hat (zumindest asymptotisch gesehen[9]) die beste Laufzeit.[10] Wir fassen zusammen:

Satz 1.8 *Dijkstras Algorithmus löst das „Single Source Shortest Path“-Problem in Digraphen mit nicht-negativen Kantenkosten bei der Kostenmatrix-Implementierung in $O(n^2)$ Schritten, bei der Adjazenzlisten-Implementierung mit einer HEAP-Implementierung der*

[6] Hierbei bezeichnet n die maximale Anzahl der in Q gespeicherten Objekte, was in unserer Anwendung mit der Knotenanzahl n des Digraphen übereinstimmt.

[7] Jede einzelne Operation kann teurer sein, aber eine Sequenz von s Operationen kostet insgesamt nur $O(s)$ Schritte. Die amortisierte Laufzeitanalyse, auf die wir zum gegenwärtigen Zeitpunkt nicht näher eingehen, nutzt aus, dass eine teure Operation sich *amortisiert*, indem anschließend eine Weile lang nur billige Operationen auftreten können.

[8] Eine Alternative zu FIBONACCI-HEAP's sind die sogenannten RELAXED-HEAP's. Bei diesen gilt die Schranke $O(\log n)$ für DELETE-MIN sogar für jede einzelne Operation (also nicht nur amortisiert).

[9] Für kleine Digraphen wäre allerdings der Overhead zur Bereitstellung der FIBONACCI-HEAP's zu groß.

[10] Da wir uns die m Kanten des Digraphen zumindest einmal anschauen müssen und eine Sortierung von n Schlüsselwerten bei der Lösung des Problems schwer zu vermeiden sein dürfte, ist die Laufzeit $O(m+n \log n)$ vermutlich nicht zu schlagen.

PRIORITY QUEUE in $O(m \log n)$ Schritten und bei der verbesserten Adjazenzlisten-Implementierung, wobei die PRIORITY QUEUE durch einen FIBONACCI-HEAP implementiert wird, in $O(m + n \log n)$ Schritten.

1.3 Der Algorithmus von Floyd

Wir setzen voraus, dass G ein Digraph mit nicht-negativen Kantenkosten $c(i, j)$ ist. Floyds Algorithmus zum Lösen des „All Pair Shortest Path"-Problems benutzt Dynamisches Programmieren. Seien $V = \{1, \ldots, n\}$ und C_{ij}^k die Länge eines kürzesten Pfades von i nach j, der nur Zwischenknoten in $\{1, \ldots, k\}$ verwendet, wobei $0 \leq k \leq n$ und $1 \leq i, j \leq n$. Offensichtlich gilt:

(1) $C_{ij}^0 = c(i, j)$ % keine Zwischenknoten %

(2) $C_{ij}^k = min\{C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1}\}$.

Der erste Term in der Min-Bildung in (2) entspricht dem Fall, daß der kürzeste Pfad den neu erlaubten Zwischenknoten gar nicht benutzt; der zweite Term entspricht dem Fall, daß k als Zwischenknoten auftaucht. (OBdA taucht k nur einmal auf, da Kreise von k nach k keine negativen Kosten haben und daher eliminiert werden können.) Schließlich sind die Werte von C_{ij}^n die korrekten Endergebnisse (alle Zwischenknoten erlaubt). Diese Ideen können zu Floyds Algorithmus zusammengefaßt werden:

begin
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do** $C_{ij}^0 \leftarrow c(i, j)$ % $c(i, i) = 0$;
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do** $C_{ij}^k \leftarrow \min\{C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1}\}$
end

In einer realen Implementierung würde man anstelle eines 3-dimensionalen Arrays C_{ij}^k nur ein 2-dimensionales Array C_{ij} verwenden. Dann muss man allerdings sauber argumentieren, dass das Verfahren immer noch korrekt ist.

Die Initialisierung kostet $O(n^2)$ Schritte. Die dreifach geschachtelte Schleife mit den Laufvariablen k, i, j kostet $O(n^3)$ Schritte.

Satz 1.9 *Floyds Algorithmus löst das „All Pair Shortest Path"-Problem in $O(n^3)$ Schritten.*

Denkbare Erweiterungen von Floyds Algorithmus: Um auf Anfrage i, j einen kürzesten Pfad von i nach j schnell ausgeben zu können, könnte Floyds Algorithmus um Pfad-Pointer $Z_i(j)$ erweitert werden, die (nach Terminieren des Algorithmus) den kürzesten Pfad relativ zum Startknoten i repräsentieren (insgesamt n^2 Pointer).

1.4 Der Algorithmus von Warshall

Der Algorithmus von Warshall iat ein enger Verwandter von Floyds Algorithmus. Er berechnet den reflexiven–transitiven Hülle $G^* = (V, E^*)$ eines gegebenen Digraphen $G = (V, E)$. Hierbei wird jede Pfadverbindung von v nach w in G durch eine Kante in E^* repräsentiert, d.h.:

$$E^* = \{(v, w) | \text{ In } G \text{ existiert ein Pfad von } v \text{ nach } w\}$$

Punktpfade werden auch als Pfade aufgefasst, so dass E^* insbesondere alle Paare der Form (v, v) mit $v \in V$ enthält.

Warshalls Algorithmus benutzt Dynamisches Programmieren. Seien $V = \{1, \ldots, n\}$ und sei $G = (V, E)$ gegeben durch seine Adjazenzmatrix A. Das Array $B_{ij}^k \in \{0, 1\}$ sei 1 genau dann, wenn es in G einen Pfad von i nach j gibt, welcher als Zwischenstationen nur Knoten aus $\{1, \ldots, k\}$ verwendet. Offensichtlich gilt:

(1) $B_{ij}^0 = A_{ij}$ für $i \neq j$ und $B_{i,i} = 1$.

(2) $B_{ij}^k = B_{ij}^{k-1} \vee (B_{ik}^{k-1} \wedge B_{kj}^{k-1})$.

Der erste Term in der Ver-Oder-ung in (2) entspricht dem Fall, daß ein Pfad von i nach j den neu erlaubten Zwischenknoten k gar nicht benutzt; der zweite Term entspricht dem Fall, daß k als Zwischenknoten auftaucht. Schließlich sind die Werte von B_{ij}^n die korrekten Endergebnisse (alle Zwischenknoten erlaubt). Diese Ideen können zu Warshalls Algorithmus zusammengefaßt werden:

begin
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do** $B_{ij}^0 \leftarrow \begin{cases} A_{ij} & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$;
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do** $B_{ij}^k \leftarrow B_{ij}^{k-1} \vee (B_{ik}^{k-1} \wedge B_{kj}^{k-1})$
end

In einer realen Implementierung würde man anstelle eines 3-dimensionalen Arrays B_{ij}^k nur ein 2-dimensionales Array B_{ij} verwenden. Dann muss man allerdings sauber argumentieren, dass das Verfahren immer noch korrekt ist.

Die Initialisierung kostet $O(n^2)$ Schritte. Die dreifach geschachtelte Schleife mit den Laufvariablen k, i, j kostet $O(n^3)$ Schritte.

Satz 1.10 *Warshalls Algorithmus löst berechnet den reflexiven–transitiven Hülle eines gegebenen Digraphen $O(n^3)$ Schritten.*

Historische Notizen Wir beginnen mit Anmerkungen zu Dijkstra's Algorithmus zur Lösung des „Single Source Shortest Path"-Problems bei nicht-negativen Kantenkosten. Der ursprüngliche Algorithmus von Dijkstra aus dem Jahre 1959 benutzt die Kostenmatrix-Implementierung. Dieser Algorithmus wurde etwa zur gleichen Zeit unabhängig auch noch von Dantzig sowie von Whiting und Hillier entdeckt. Williams präsentierte im Jahre 1964 einen auf HEAP's aufbauenden Sortieralgorithmus (HEAPSORT) und machte auf die HEAP-Implementierung von PRIORITY QUEUE's und die Adjazenzlisten-Implementierung von Dijkstra's Algorithmus aufmerksam. Fredman und Tarjan erfanden 1987 die FIBONACCI-HEAP's und gelangten zur verbesserten Adjazenzlisten-Implementierung von Dijkstra's Algorithms. Die RELAXED-HEAP's wurden 1988 als Alternative zu FIBONACCI-HEAP's von Driscoll, Gabow, Shrairman und Tarjan vorgeschlagen.

Der auf dynamischem Programmieren basierende Algorithmus zur Lösung des „Single Source Shortest Path"-Problems auf DAGs mit beliebigen Kantenkosten scheint „Folklore" zu sein.

Der Algorithmus von Floyd zur Lösung des „All Pair Shortest Path"-Problems stammt aus dem Jahre 1962 und baut auf dem im gleichen Jahr publizierten Algorithmus von Warshall zur Berechnung des reflexiven–transitiven Hülle eines Digraphen auf.

Für weitergehende Resultate und Informationen sei auf die Kapitel 4 und 5 des Buches „Network Flows" von Ahuja, Magnanti und Orlin verwiesen.