
Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: 19. September 2011

Hausaufgabe 1

Implementieren Sie in der Klasse `SSSP` den Algorithmus von Dijkstra zur Berechnung der kürzesten Wege von einem Knoten s zu allen anderen Knoten in dem ungerichteten Graphen G .

Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `SSSP` aber auf keinen Fall deren Interface `ISSSP`.

Hinweis: Verwenden Sie grundlegende Datenstrukturen, wie sie Java bereitstellt, und implementieren Sie diese *nicht* neu.

Hausaufgabe 2

Implementieren Sie in der Klasse `MST` den Algorithmus von Jarnik und Prim zur Berechnung eines Minimum Spanning Tree beginnend von einem Knoten s von einem ungerichteten Graphen G .

Der Algorithmus von Jarnik und Prim wählt zu einer Menge V' den Knoten der über eine Kante mit dem geringsten Gewicht erreichbar ist und fügt ihn der Menge V' hinzu. Gestartet wird der Algorithmus mit $V' = s$.

Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `MST` aber auf keinen Fall deren Interface `IMST`.

Hinweis: Verwenden Sie grundlegende Datenstrukturen, wie sie Java bereitstellt, und implementieren Sie diese *nicht* neu.

Aufgabe 1

Bestimmen Sie die Laufzeit des BFS-Algorithmus. Betrachten Sie zwei Implementierungen: Zum einen basierend auf einer Adjazenzmatrix und zum anderen basierend auf einer Adjazenzliste.

Lösungsvorschlag

- Adjazenzmatrix:

Um alle Nachbarn eines Knoten v in die Queue einzufügen müssen alle möglichen Nachbarn ($n - 1$) überprüft werden. Da dies n mal geschieht erhalten wir also eine Laufzeit von $\mathcal{O}(n^2)$.

- Adjazenzliste:

Hier haben wir direkten Zugriff auf die Nachbarn. Jeder der Nachbarn muss überprüft werden. Damit fallen pro Knoten $\mathcal{O}(\deg(v))$ Rechenoperationen an.

Da $\sum_{v \in V} \deg(v) = |E|$ gilt, wird also jede Kante nur konstant oft angeschaut und wir erhalten eine Laufzeit von $\mathcal{O}(m + n)$

Aufgabe 2

Gegeben Sei die Adjazenzmatrix $A = (a_{ij})_{1 \leq i, j \leq n}$ eines gerichteten, einfachen Graphen G mit n Knoten. Geben Sie einen Algorithmus an, der eine Senke in $\mathcal{O}(n)$ Schritten findet. Eine Senke sei (abweichend von der üblichen Definition) ein Knoten v , der Eingrad $n - 1$ und keine ausgehenden Kanten hat.

Lösungsvorschlag

Da wir hier mit einfachen Graphen arbeiten sei im Folgenden immer $i \neq j$. Wir betrachten die Bedeutung die wir aus einer Abfrage an die Adjazenzmatrix folgern können. Ist $a_{ij} = 1$ so geht eine Kante von Knoten i zu Knoten j . Daraus folgt, dass Knoten i keine Senke sein kann (da er Ausgrad größer 0 hat). Knoten j kann weiterhin eine Senke sein, sein Eingrad ist mindestens 1.

Auf der anderen Seite gilt: Ist $a_{ij} = 0$ so kann Knoten j keine Senke sein, da der Eingrad nicht mehr $n - 1$ sein kann, da wir einfache Graphen betrachten. Knoten i kann eine Senke sein, da sein Ausgrad zumindestens nicht von der möglichen Kante von i nach j abhängt. Betrachten wir nun folgende Vorgehensweise:

- Wir stellen $\lfloor \frac{n}{2} \rfloor$ Anfragen für die Kanten $(2i + 1, 2i + 2)$ mit $i \in \{0, \dots, \lfloor \frac{n}{2} \rfloor - 1\}$ an A . Daraus merken wir uns jeweils die noch möglichen Knoten B für eine Senke.

Für B und einem eventuell noch nicht getesteten Knoten wiederholen wir das Verfahren bis schlussendlich nur noch ein Knoten x übrig ist.

Wir können nun anhand der Adjazenzmatrix überprüfen ob der Knoten x tatsächlich eine Senke ist ($n - 1$ weitere Tests).

- Alternatives Verfahren: Wir beginnen mit der Knotenmenge V , der Kandidaten für die Senke. Wir vergleichen zwei beliebige Knoten aus der Menge V und entfernen den Knoten, der aus den Folgerungen oben definitiv keine Senke sein kann.

Das ganze wiederholen wir, bis nur noch ein Kandidat für die Senke übrig ist, welchen wir dann Explizit überprüfen. (Wie in der ersten Möglichkeit am Ende). Da hier $n - 1$ Vergleiche bis zur einelementigen Kandidatenmenge ausgeführt werden und nocheinmal $n - 1$ Vergleiche für die Bestätigung/Ablehnung der Senke, benötigt der Algorithmus also $\mathcal{O}(n)$ Zeit.

Aufgabe 3

In dieser Aufgabe wollen wir ein alternatives Verfahren zur topologischen Sortierung auf einem DAG G betrachten. Dazu nehmen wir an, dass uns weder die üblichen Graphoperationen zum Traversieren einer Kante oder Abfragen der Nachbarschaft eines Knotens, noch irgendwelche „höheren Datenstrukturen“ wie z.B. Queues, Listen, PriorityQueues oder Stacks zur Verfügung stehen.

Die einzige Operation, die wir benutzen können ist eine Tiefensuche (DFS), die Arrays `dfs_num` und `finish_num` zurückgibt, die zu den jeweiligen Knoten die entsprechenden Werte beeinhalt. Außerdem stehen uns `for`-Schleifen und die übliche Addition bzw. Subtraktion zur Verfügung.

- a) Geben Sie einen Algorithmus (Pseudocode reicht) an, der dieselbe asymptotische Laufzeit erreicht, wie der in der Vorlesung vorgestellte Algorithmus.
- b) Begründen Sie die Laufzeit Ihres Algorithmus kurz.
- c) Beweisen Sie die Korrektheit Ihres Algorithmus.

Lösungsvorschlag

1. Wir führen eine DFS auf dem Graphen G aus. Daraus erhalten wir ein Array mit der Finish-Reihenfolge. In einer For-Schleife setzen wir für alle Knoten i die Ausgabe als $n - \text{finish_num}[i]$.
2. Laut Vorlesung benötigt die DFS gerade $\mathcal{O}(n + m)$ Zeit. Dann wird für jeden Knoten noch eine Rechenoperation ausgeführt ($\mathcal{O}(n)$). Also erhalten wir als Laufzeit insgesamt $\mathcal{O}(n + m)$.
3. Da die Eingabe ein DAG ist (Voraussetzung), wissen wir aus der Vorlesung: Wenn $(v, w) \in E$: dann gilt: $\text{finish_num}[v] > \text{finish_num}[w]$. Also gilt: $(v, w) \in E \Rightarrow n - \text{finish_num}[v] < n - \text{finish_num}[w]$ wobei $n - \text{finish_num}[x]$ gerade die oben definierte Nummer in der topologischen Sortierung ist. Damit gilt $\text{topo}[v] < \text{topo}[w]$ wenn $(v, w) \in E$ ist. Dies ist aber gerade die Bedingung an eine topologische Sortierung und unser Algorithmus ist damit korrekt.

Aufgabe 4

Argumentieren Sie, warum es unklug ist, bei der Berechnung eines kürzesten Weges von s nach t in einem Graphen mit echt positiven Gewichten alle möglichen (einfachen) Wege aufzuzählen und deren Distanz zu berechnen.

Benutzen Sie dazu exemplarisch ein Gitter der Größe n . Die n^2 Knoten werden durch die Tupel (x, y) mit $x, y \in [n]$ identifiziert.

Die Kanten $((x, y), (x + 1, y))$ seien gerichtet während die Kanten $((x, y), (x, y + 1))$ bidirektional verlaufen. Berechnen Sie die Anzahl aller Pfade von $s = (1, 1)$ zu $t = (n, n)$.

Lösungsvorschlag

Die Anzahl der Pfade in einem Gitter der Größe n ist exponentiell in n . Daher benötigt jeder Algorithmus, der alle Pfade auflistet also auch exponentiell viel Zeit. (Dies entspricht nicht der üblichen Vorstellung von effizienter Berechnung (polynomiell in der Größe der Eingabe)).

Die Anzahl der Pfade ergibt sich wie folgt:

In jeder Zeile, gibt es n Möglichkeiten von Zeile x in die Zeile $x + 1$ zu kommen (An der Positionen 1 bis n). Eine solche Auswahl kann in $n - 1$ Zeilen getroffen werden, da das Ziel in der letzten Zeile schon vorgegeben ist. So ergeben sich n^{n-1} verschiedene, einfache Wege vom Knoten $(1, 1)$ zu Knoten (n, n)

Aufgabe 5

Geben Sie einen möglichst effizienten Algorithmus an, der die *Anzahl* der kürzesten Wege von einem Knoten s zu allen anderen Knoten in einem ungerichteten Graphen G mit echt positiven Gewichten berechnet.

Lösungsvorschlag

Wir modifizieren den schon bekannten Dijkstra-Algorithmus. Für jeden Knoten merken wir uns in einem Array N die Anzahl der kürzesten Wege zu den jeweiligen Knoten. Zu dem Startknoten s existiert genau ein Weg.

Wird ein neuer kürzester Weg zu einem Knoten v von dem Knoten w aus entdeckt, so wird die Anzahl der kürzesten Wege $N[v]$ zu Knoten v mit der Anzahl der kürzesten Wege von w abgespeichert ($N[v] := N[w]$). Wird der Knoten v erneut von einem anderen Knoten x mit der gleichen Distanz zu s „entdeckt“ so wird der werden die Anzahl der Wege, die über x zu v führen zu $N[v]$ dazugerechnet ($N[v] += N[x]$).