Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 11

Algorithmen I – Sommersemester 2022

Abgabe im ILIAS bis 13.07.2022, 14:00 Uhr

Bitte beschrifte Deine Abgabe gut sichtbar mit Deinem Namen und Deiner Matrikelnummer. Achte insbesondere bei handschriftlichen Abgaben auf Lesbarkeit und genügend Platz für Korrektur-Anmerkungen. Die Abgabe erfolgt über das Übungsmodul in der Gruppe Deines Tutoriums im ILIAS. Gib Deine Ausarbeitungen in *einer* PDF-Datei ab. Achte darauf, effiziente Algorithmen zu formulieren, also solche mit möglichst geringer asymptotischer Laufzeit!
Wenn du die Korrektheit eines Algorithmus begründen oder dessen Laufzeit analysieren sollst, tue dies getrennt von der Beschreibung des Algorithmus.
Wenn nicht anders spezifiziert oder aus dem Kontext ersichtlich, bezeichnen wir mit Graph einen einfachen ungerichteten Graphen.

Aufgabe 1 - Kann ja mal passieren... (10 Punkte)

Während seiner Undercover-Aktion auf der Mitarbeiterversammlung ist Dr. Meta aufgefallen, dass er bei der Ausdünnung ein relativ wichtiges Detail übersehen hatte: Beim Modellieren des Kontakt-Graphen war ihm entgangen, dass dieser eigentlich gerichtet hätte sein müssen. Infolgedessen sind gerade die Mitarbeiter verloren gegangen, auf die viel eingeredet wurde, die sich aber dennoch nicht an der Verschwörung beteiligt haben. Upps. Dieser Fehler wird nicht noch einmal passieren. Mit dem Wissen, dass der Kontakt-Graph ein gerichteter und kreisfreier Graph (DAG) ist, will Dr. Meta nun die Quelle allen Übels finden.

1. Beweise, dass ein DAG eine Quelle hat. (1 Punkt)

2. Gib einen Algorithmus im Pseudocode an, der einen DAG mit n Knoten und m Kanten als Eingabe erhält und in Zeit $O(n+m)$ alle Quellen ausgibt. Verwende dabei die folgende Signatur: (2 Punkte)

---

listSources($G = (V, E)$: Graph): List⟨Node⟩

---

3. Angenommen, ein Algorithmus löscht iterativ Quellen aus einem DAG. Beschreibe welche Datenstruktur geeignet ist, um die Quellen zu verwalten, sodass eine Quelle v in Zeit $O(\deg(v))$ gelöscht werden kann. (2 Punkte)
   Hinweis: Beim Löschen einer Quelle in einem DAG können neue Quellen entstehen.

Beim Nachdenken über DAGs hat Dr. Meta einen Einfall. Er vermutet, dass ein gerichteter Graph $G = (V, E)$ mit n Knoten genau dann azyklisch ist, wenn es eine topologische Sortierung gibt, also eine Reihenfolge $v_1, \ldots, v_n$ der Knoten, sodass für jede Kante $(v_i, v_j) \in E$ gilt, dass $i < j$.

4. Beweise Dr. Meta's Vermutung. (2 Punkte)
   Hinweis: Achte darauf, dass hier zwei Richtungen gezeigt werden sollen.

Der Kontakt-Graph soll nun erneut ausgedünnt werden. Diesmal ist der Plan jedoch, die k Knoten zu erwischen, die den Verrat ins Rollen gebracht haben.

5. Gib einen Algorithmus in Pseudocode an, der als Eingabe einen DAG G mit n Knoten und m Kanten erhält und in Zeit $O(n + m)$ eine topologische Sortierung von G ausgibt. Verwende dabei die folgende Signatur. Das ausgegebene Array soll die Knoten in aufsteigender topologischer Sortierung enthalten. (3 Punkte)

TOPOSORT($G = (V, E)$: Graph) : [Node; n]

Hinweis: Verwende das Löschen eines Knotens v (in $O(\deg(v))$) als Subroutine.

Lösung 1

1. In einem DAG $G = (V, E)$ lässt sich stets eine Quelle wie folgt bestimmen: Wir wählen einen beliebigen Knoten $v \in V$. Ist $\deg_{\text{in}}(v) = 0$, ist v die gesuchte Quelle. Ansonsten wählen wir eine beliebige eingehende Kante von v und wiederholen das Vorgehen für den Knoten, von dem diese ausgeht. Da G azyklisch ist, betrachten wir damit jeden Knoten höchstens ein Mal. Da zudem G eine endliche Anzahl von Knoten und Kanten hat, finden wir in einer endlichen Anzahl Schritte eine Quelle.

2. Unter Verwendung der Notation aus Übung 4:

```
listSources(G = (V, E): Graph): List⟨Node⟩
    sources: List⟨Node⟩ = ⟨⟩
    for v ∈ V do
        if deg_in(v) = 0 then
            sources.pushBack(v)
        end
    end
    return sources
```

3. Wir können zur Verwaltung der Quellen eine modifizierte Adjazenzliste wie in Übung 4 vorgestellt verwenden: Diese enthält für jeden Knoten eine Liste seiner Nachbarn. Zwischen diesen Listen werden zusätzliche Zeiger gespeichert: Für jede Kante $(u, v) \in E$ besteht je ein Zeiger zwischen dem Eintrag zu u in der Liste zu v und dem Eintrag zu v in der Liste zu u. Zusätzlich können wir uns für jeden Knoten seinen Eingangsgrad speichern und immer dann aktualisieren, wenn einer seiner Vorgänger gelöscht wird. Das Löschen einer Quelle $s \in V$ funktioniert wie in Übung 4 beschrieben in $O(\deg(s))$, indem der Array-Eintrag von s an das Ende des Arrays getauscht wird und die zu s inzidenten Kanten entfernt werden. Um die Quellen selbst zu verwalten, kann eine Liste verwendet werden, die Referenzen auf die entsprechenden Knoten speichert. Damit ist es möglich, in konstanter Zeit neu gefundene Quellen einzufügen sowie bereits gefundene Quellen zu finden.

4. Wir zeigen:

$$G = (V, E) \text{ ist ein DAG } \Leftrightarrow G \text{ hat eine topologische Sortierung}$$

$\Rightarrow$ Wir führen eine Induktion über n durch:

IA: $n = 1$ Für einen DAG mit nur einem Knoten ist die Sortierung klar.

IV: Für ein beliebiges aber festes $n \in \mathbb{N}$ habe ein DAG mit n Knoten eine topologische Sortierung.

IS: $n \rightsquigarrow n + 1$
Sei G ein DAG mit $n + 1$ Knoten. Dann hat G nach Teilaufgabe 1 eine Quelle s. Der Graph $G' = (V', E')$, der durch Entfernen von s aus G entsteht, ist ebenfalls ein DAG. Also gibt es nach **IV** eine topologische Sortierung $v'_1, \ldots, v'_n$ auf G'. Da s in G eine Quelle ist, kann sie in einer topologischen Sortierung auf G am Anfang stehen. Wir erhalten damit eine topologische Sortierung auf G: $v_1 = s$ und für alle $i \in \{2, \ldots, n+1\}$: $v_i = v'_{i-1}$.

$\Leftarrow$ Existiert auf einem Graphen gerichteten Graphen G eine topologische Sortierung $v_1, \ldots, v_n$, so muss er azyklisch sein, denn sonst gäbe es einen Kreis $(v_{i_1}, v_{i_2}, \ldots, v_{i_k} = v_{i_1})$ mit $i_1, \ldots, i_k \in \{1, \ldots, n\}$ und $v_{i_j} < v_{i_{i+j}}$ für alle $j \in \{1, \ldots, k-1\}$. Dann wäre aber $i_1 < i_1$. $\lightning$

5.
```
topoSort(G = (V, E) : Graph) : [Node; n]
    sources: List⟨Node⟩ = listSources(G)
    topo: [Node; n]
    cur_idx: ℕ = 0
    cur_node: Node
    while V ≠ ∅ do
        cur_node = sources.popBack()
        deleteSource(G, sources, cur_node)
        topo[cur_idx] = cur_node
        cur_idx = cur_idx + 1
    end
    return topo
```

Aufgabe 2 - xkcd#1323 (9+3 Punkte)

Alice besitzt n verschließbare, goldene Kisten und m goldene Schlüssel. Jede Kiste ist mit einem Schloss gesichert in das mindestens ein Schlüssel passt. Jeder Schlüssel passt in genau ein Schloss. Eine verschlossene Kiste kann nur auf zwei Arten geöffnet werden: entweder mit einem passenden Schlüssel oder destruktiv mit einem Hammer. Malory hat nun einige Schlüssel gestohlen und alle anderen in den Kisten verschlossen! Alice kennt die Zuordnung von Schlüsseln zu Schlössern und konnte immerhin beobachten welche Schlüssel in welcher Kiste gelandet sind. Da sowohl Schlüssel als auch Kisten wertvoll sind, möchte Alice möglichst wenige Kisten zerstören, um an alle Schlüssel zu gelangen.

1. Wie kann die obige Situation mithilfe eines gerichteten Graphen modelliert werden? (2 Punkte)

2. Gib einen Algorithmus im Pseudocode an, der deinen Graphen als Eingabe erhält und in Zeit $O(n + m)$ entweder einen Kreis ausgibt oder erkennt, dass der Graph keine Kreise enthält. (3 Punkte)

Wir nehmen nun an, dass die Schlüssel so in den Kisten verschlossen wurden, dass der resultierende Graph keine Kreise hat.

3. Alice möchte für eine gegebene Kiste wissen, ob sie alle Schlüssel bekommen kann, indem sie genau diese Kiste gewaltsam öffnet. Beschreibe wie sich Alice' Problem in deine Formulierung als Graphenproblem übersetzt und beschreibe einen Algorithmus, der das Problem in Linearzeit löst. (2 Punkte)

4. Eventuell genügt es nicht, nur eine Kiste mit dem Hammer zu öffnen. Alice fragt sich nun was die kleinste Anzahl Kisten ist, die zerschlagen werden müssen um an alle Schlüssel zu gelangen. Übersetze auch diese Fragestellung in die Modellierung als Graph. Wie kann die Anzahl in Zeit $O(n + m)$ bestimmt werden? (2 Punkte)

Im Folgenden nehmen wir nun nicht mehr an, dass der Graph kreisfrei ist.

*. Beschreibe einen Algorithmus der nun bestimmt, wie viele Kisten mindestens zerschlagen werden müssen, um an alle Schlüssel zu gelangen. (3 Punkte)

Lösung 2

1. Wir modellieren das Problem als gerichteten Graphen. Jede Kiste korrespondiert zu genau einem Knoten, jeder Schlüssel zu genau einer Kante. Da jeder Schlüssel in genau einer Kiste eingeschlossen ist und genau ein Schloss öffnet, richten wir die Kanten wie folgt: öffnet ein Schlüssel, der in der Kiste k steckt, ein Schloss an der Kiste l, so ist die zugehörige Kante (k, l).

2. Idee: Wir suchen Rückkanten via Tiefensuche, um gerichtete Kreise zu finden:

```
findCycle(G = (V, E) : Graph) : List⟨Node⟩
    finished: [Bool; n] = ⟨ false , ... , false ⟩
    parent: [Node; n] = ⟨⊥, ..., ⊥⟩
    for v ∈ V do
        if parent[v] = ⊥ then
            parent[v] = v
            back_edge: Edge = backEdge(G, v, finished, parent)
            if back_edge ≠ ⊥ then
                return collectCycle(G, back_edge)
            end
        end
    end
    return ⊥
```

```
backEdge(G = (V, E) : Graph, v ∈ V, fin : [Bool; n], p : [Node; n]) : Edge
    for w ∈ N(v) do
        if p[w] = ⊥ then
            p[w] = v
            back_edge: Edge = backEdge(G, w, fin, p)
            if back_edge ≠ ⊥ then
                return back_edge
            end
        else if ¬fin[w] then                          ▷ Rückkante gefunden
            return (v, w)
        end
    end
    fin[v] = true
    return ⊥
```

```
collectCycle(G = (V, E) : Graph, back_edge ∈ E) : List⟨Vertex⟩
    cycle: List⟨Node⟩ = ⟨back_edge.snd⟩
    cur_node: Node = back_edge.fst
    while parent[cur_node] ≠ back_edge.snd do
        cycle.pushBack(cur_node)
        cur_node = parent[cur_node]
    end
    cycle.pushBack(cur_node)
    return cycle
```

3. Sei $G = (V, E)$ der in Teilaufgabe 1 modellierte Graph und sei $v \in V$ der Knoten, der zur gegebenen Kiste korrespondiert. Alice möchte wissen, ob jeder Knoten in G mit einem Ausgangsgrad größer 0 von v aus erreichbar ist. Dazu können wir eine Tiefensuche von v aus durchführen und testen, ob diese jeden Knoten in G, der keine Senke ist, gefunden hat.

4. Wir wollen alle Quellen in G finden, von denen mindestens eine Kante ausgeht. Dafür können wir zunächst alle Knoten, bei denen sowohl Eingangs- als auch Ausgangsgrad 0 sind, aus G entfernen. Anschließend verwenden wir den Algorithmus, zum Finden von Quellen in $O(n + m)$, den wir in Aufgabe 1 entwickelt haben.

*. Wir betrachten weiterhin den in Teilaufgabe 1 modellierten Graphen G. Wir müssen nicht allerdings nun nicht nur alle Quellen (mit Ausgangsgrad größer 0) in G finden, sondern auch aus jedem Kreis in G einen (beliebigen) Knoten auswählen. Deswegen gehen wir wie folgt vor:
   Zunächst entfernen wir wieder alle Knoten, die weder eingehende noch ausgehende Kanten haben. Solange der gegebene Graph nicht azyklisch ist, suchen wir einen Zyklus C und wählen einen beliebigen Knoten auf diesem aus. Anschließend kontrahieren wir die Knoten auf C zu einem Knoten zusammen, d.h. wir entfernen sämtliche Knoten in C und fügen einen neuen Knoten ein, welcher genau dann eine ausgehende bzw. eingehende Kante zu einem anderen Knoten v in G hat, wenn einer der gelöschten Knoten über eine solche Kante mit v verbunden war. Im resultierenden Graph genügt das zerschlagen von k Kisten genau dann, wenn es vorher genügt hat.
   Auf dem entstandenen azyklischen Graphen bestimmen wir nun wie in Teilaufgabe 3 alle Quellen.

Hier noch Werbung der Fachschaft:

Eulenfest am 14. Juli 2022

Nächste Woche Donnerstag findet das erste Eulenfest seit 2020 statt! Los geht es um 19 Uhr, im und um den Infobau. Dieses Jahr gibt es Livemusik von den Bands *Fancity* und *Sonnenblumen of Death*. Natürlich haben wir auch wieder DJs. Außerdem werdet ihr mit Getränken und Essen versorgt. Weitere Infos zum Fest findet ihr unter eulenfest-karlsruhe.de.

Gerne könnt ihr die Fachschaft mit einer Helferschicht unterstützen. Tragt euch dafür einfach selbst in das Helfersystem unter redseat.de/eulenfest22 ein.

Die Fachschaft sucht auch noch tatkräftige Unterstützung für unser Sicherheitsteam. Hierfür brauchst du keine besonderen Qualifikationen oder viel Erfahrung, es wird aber vor dem Fest eine ausführliche Einweisung geben. Falls du Interesse hast, melde dich bei Peter (peter.maucher@fsmi.uni-karlsruhe.de) per Mail. Ebenfalls suchen wir Personen mit Sanitätsausbildung, hierfür kannst du dich gerne bei Patrick (patrick.schneider@fsmi.uni-karlsruhe.de) melden.

Mehr Infos

Helfersystem