Karlsruher Institut für Technologie

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

Übungsblatt 13

Algorithmen I – Sommersemester 2022

Abgabe im ILIAS bis 27.07.2022, 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 - KANNHIERMALJEMANDSPLITTEN? (8 Punkte)

Sei W eine Menge von Wörtern. Ein String S der Länge n ist *splittable*, wenn es Trennstellen $0 = s_0 < \cdots < s_k = n$ gibt, sodass $S[s_{i-1}, s_i) \in W$ für alle $i \in [1, k]$.

Wir betrachten nun das Problem MinSplits, wobei ein gegebener String in möglichst wenige Wörter gesplittet werden soll. Die Lösung einer MinSplit-Instanz besteht dann aus der Menge der zugehörigen Trennstellen. Wenn ein String nicht splittable ist, ist diese Menge leer.

1. Sei eine MinSplits-Instanz gegeben durch $S = \texttt{PRIMALGOALPHASEARCHER}$ und

$$W = \{\texttt{ALGO}, \texttt{ALPHA}, \texttt{ARC}, \texttt{ARCH}, \texttt{AS}, \texttt{EAR}, \texttt{GO}, \texttt{GOAL}, \texttt{HAS}, \texttt{HER}, \texttt{PHASE}, \texttt{PRIM}, \texttt{PRIMAL}, \texttt{RIM}, \texttt{SEA}, \texttt{SEARCH}, \texttt{SEARCHER}\}$$

   Gib die Lösung für diese Instanz an. (1 Punkt)

Nun wollen wir ein dynamisches Programm formulieren, was MinSplits löst.

2. Auf welche Teilprobleme kann das Problem reduziert werden und wie sehen Teillösungen für diese aus? (2 Punkte)

3. Gib die Rekurrenz an, mit der Teillösungen aus Teilaufgabe 2 zu einer neuen Teillösung kombiniert werden. (1 Punkt)

4. Beschreibe einen Algorithmus, der die Rekurrenz aus Teilaufgabe 3 verwendet, um eine Instanz von MinSplits zu lösen. Gib dazu an, wie die Teillösungen verwaltet werden, in welcher Reihenfolge die Teillösungen berechnet werden und wie man schließlich die tatsächliche Lösung erhält. Begründe warum dein Algorithmus unter der Annahme, dass das längste Wort in W aus maximal 17 Buchstaben besteht, eine asymptotische Laufzeit von $O(n \cdot |W|)$ nicht überschreitet. (3 Punkte)

5. Was ändert sich in deinem Algorithmus, wenn wir stattdessen das Problem MaxSplits lösen wollen, bei dem ein String in möglichst viele Wörter gesplittet werden soll? (1 Punkt)

Lösung 1

1. S kann in minimal 4 Wörter gesplittet werden, z.B. mit den Trennstellen $(s_0, s_1, s_2, s_3, s_4) = (0, 4, 8, 13, 21)$

2. Teilprobleme sind gegeben durch Teilstrings $S[0, i)$ mit $i \in [1, n]$ von S und der gleichen Menge W von Wörtern. Eine Lösung eines solchen Teilproblems ist dann die minimale Menge von Trennstellen $\text{minSplitters}(i)$, die $S[0, i)$ in Wörter aus W splitten.

3. Wir bezeichnen die Länge eines Wortes $w \in W$ mit $|w|$. Zur Vereinfachung der Notation verwenden wir die Funktion `setmin`, welche aus einer Menge von Mengen diejenige mit der kleinsten Kardinalität liefert, die nicht die leere Menge ist.
   Die gesuchte Rekurrenz ist dann

$$\text{minSplitters}(0) = \{0\}$$

$$\text{minSplitters}(i) = \begin{cases} \emptyset & \mid \neg\text{splittable}(i) \\ \texttt{setmin}(\{\text{minSplitters}(l) \mid S[l, i) \in W\}) \cup \{i\} & \mid \text{sonst} \end{cases}$$

4. Um minSplits für S zu lösen, gehen wir wie folt vor: Zuächst iterieren wir über S und berechnen anhand der Rekurrenz aus Teilaufgabe 3 für jeden Index $i \in [1, n]$ $\text{minSplitters}(i)$. Dabei speichern wir für i den Index $\texttt{prev}(i)$ mit $\text{minSplitters}(i) = \text{minSplitters}(\texttt{prev}(i)) \cup \{i\}$, wenn eine Lösung für $S[0, i)$ existiert, ansonsten $\emptyset$.
   Wenn sie existiert, ist gesuchte Lösung dann $\{n, \texttt{prev}(n), \texttt{prev}(\texttt{prev}(n)), \ldots, 0\}$.
   Eine Teillösung für einen Teilstring $S[0, i)$ wird in Zeit $\Theta(|W|)$ berechnet, denn es muss für jedes Wort $w \in W$ überprüft werden, $S[i - |w|, i) = w$ und ob $S[0, i - |w|)$ splittable ist. Ob $S[i - |w|, i) = w$, kann in $\Theta(|w|)$ bestimmt werden. Da wir wissen, dass die Längen der Wörter in W nach oben durch die Konstante 17 beschränkt sind, benötigen wir also pro Wort Zeit in $\Theta(1)$. Insgesamt können wir alle

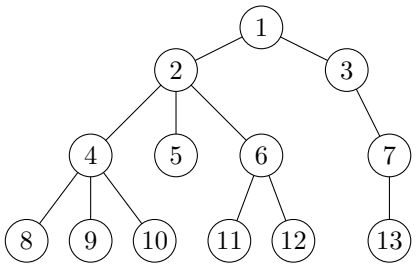
Teillösungen in Zeit $\Theta(n \cdot |W|)$ bestimmen. Zur Rekonstruktion der Lösung muss ein Mal über die Menge der `prev`-Werte iteriert werden, was Zeit in $\Theta(n)$ benötigt. Damit liegt die Gesamtlaufzeit unseres Algorithmus in $\Theta(n \cdot |W|)$.

5. Wir können den Algorithmus beinahe unverändert wiederverwenden und passen nur die Auswahl der Trennstellen bei der Konstruktion von Teillösungen an: Da wir nun die maximale Anzahl an Trennstellen suchen, wählen wir nicht mehr die kleinsten Teillösungen aus, sondern die größten.

Aufgabe 2 - Boston Tree Party (8 Punkte)

Sei $T = (V, E)$ ein Baum. Wir interessieren uns nun für das Problem MaximumIndependentSet, bei dem eine möglichst große Teilmenge $S \subseteq V$ gesucht wird, sodass für jede Kante $\{u, v\} \in E$ gilt: $u \notin S$ oder $v \notin S$.

1. Gib eine Lösung für folgende MaximumIndependentSet-Instanz an: (1 Punkt)



Nun wollen wir ein dynamisches Programm formulieren, was MaximumIndependentSet löst.

2. Auf welche Teilprobleme kann das Problem reduziert werden und wie sehen Teillösungen für diese aus? (2 Punkte)

3. Gib die Rekurrenz an, mit der Teillösungen aus Teilaufgabe 2 zu einer neuen Teillösung kombiniert werden. (1 Punkt)

4. Beschreibe einen Algorithmus, der die Rekurrenz aus Teilaufgabe 3 verwendet, um eine Instanz von MaximumIndependentSet zu lösen. Gib dazu an, wie die Teillösungen verwaltet werden, in welcher Reihenfolge die Teillösungen berechnet werden und wie man schließlich die tatsächliche Lösung erhält. Begründe warum dein Algorithmus eine asymptotische Laufzeit von $O(n)$ nicht überschreitet. (4 Punkte)

Lösung 2

1. $S = \{2, 3, 8, 9, 10, 11, 12, 13\}$

Zum einfacheren Verständnis geben wir im folgenden ein dynamisches Programm an, was die Größe der maximalen Lösung bestimmt. Die zugehörige Menge auszugeben, funktioniert ganz analog. In einer Implementierung würde man ähnlich vorgehen: Um sich teure Mengenvereinigungen zu sparen, wird zunächst die Größe der Lösung bestimmt und währenddessen Pointer gesetzt, die es erlauben die tatsächliche Lösung durch weniger Vereinigungen zu rekonstruieren.

2. Wir definieren einen beliebigen Knoten als Wurzel von T. Dann bezeichnen wir für einen Knoten v den Teilbaum unter v mit T_v. Wir spezifizieren zwei Typen von Lösungen für v

   (1) die Größe der maximalen unabhängigen Knotenmenge in T_v

   (2) die Größe der maximalen unabhängigen Knotenmenge in T_v die v nicht enthält.

3. Wir definieren zwei Funktionen $\text{MIS}_1(v)$ und $\text{MIS}_2(v)$ mit dem Ziel, dass $\text{MIS}_i(v)$ die Teillösung vom Typ (i) für v ist. Falls v ein Blatt ist, gilt $\text{MIS}_1(v) = 1$ und $\text{MIS}_2(v) = 0$. Seien nun $u_1, \dots u_k$ die Kinder von v. Um die beste Lösung für T_v zu erhalten, die v selbst nicht enthält, kombinieren wir jeweils die besten Lösungen der Kindbäume $T_{u_1}, \dots, T_{u_k}$, wobei hier die Kinder selbst ausgewählt werden dürfen, weswegen wir die Typ1-Lösungen der Kinder nutzen:

   $$\text{MIS}_2(v) = \sum_{i \in [k]} \text{MIS}_1(u_i).$$

   Die beste Lösung für T_v kann v enthalten oder nicht. Falls sie v nicht enthält, dann ist sie gleich $\text{MIS}_2(v)$. Wenn sie v enthält, dann dürfen wir die Kinder von v nicht auswählen und betrachten daher für die Kinder Lösungen von Typ2:

   $$\text{MIS}_1(v) = \max \left\{ \text{MIS}_2(v), 1 + \sum_{i \in [k]} \text{MIS}_2(u_i) \right\}.$$

   Die Lösung für den gesamten Baum T mit Wurzel w ist dann $\text{MIS}_1(w)$.

4. Um MaximumIndependentSet für einen Baum T mit Wurzel w zu lösen verwenden wir zwei Arrays A_1 und A_2, welche an Index v jeweils $\text{MIS}_1(v)$ und $\text{MIS}_2(v)$ enthalten werden. Initial sind alle Einträge in A_1 mit 1 initialisiert und alle Einträge in A_2 mit 0.

   Anschließend verfährt der Algorithmus in Schritten. In jedem Schritt werden die Eltern p aller Knoten aus dem vorherigen Schritt betrachtet (im ersten Schritt sind das die Eltern aller Blätter) und die Arrayeinträge $A_1[p]$ und $A_2[p]$ anhand der Rekurrenzen für $\text{MIS}_1(p)$ und $\text{MIS}_2(p)$ aus Teilaufgabe 3 berechnet. Der Algorithmus terminiert, wenn die Einträge $A_1[w]$ und $A_2[w]$ der Wurzel w berechnet wurden. Dann wird $A_1[w]$ ausgegeben.

   Die Initialisierung von A_1 und A_2 geschieht in Linearzeit. Anschließend wird jeder Knoten v des Baums zwei mal betrachtet. Einmal um die Werte $A_1[v]$ und $A_2[v]$ zu

berechnen, einmal um für den Elter p von v die Wert $A_1[p]$ und $A_2[p]$ zu berechnen. Die für diese Berechnungen benötigten Werte können dank wahlfreiem Zugriff in konstanter Zeit aus den Arrays abgelesen werden. Somit können alle Werte in Linearzeit berechnet werden.

Aufgabe 3 - Kruskal Recall (4 Punkte)

Wir haben in der Vorlesung die Union-Find Datenstruktur kennengelernt, um Kruskal's Algorithmus effizient umsetzen zu können. Gib Kruskal's Algorithmus unter Verwendung der Union-Datenstruktur in Pseudocode an. Nenne und begründe das asymptotische Laufzeitverhalten. Verwende die folgende Signatur: (4 Punkte)

KRUSKAL($G = (V, E)$: Graph) : List⟨Edge⟩

Lösung 3

```
1:  KRUSKAL(G = (V, E) : Graph) : List⟨Edge⟩
2:      mst: List⟨Egde⟩
3:      U = UnionFind(V)
4:      SORT(E)                                   // in ascending order
5:      for e = (u, v) ∈ E do
6:          if U.find(u) ≠ U.find(v) then
7:              U.union(u, v)
8:              mst.pushBack(e)
9:          end
10:     end
11:     return mst
```

Es wird bis zu $2m$ Mal `find` und genau $n-1$ Mal `union` durchgeführt. Dies benötigt, wie in der Vorlesung gezeigt, Zeit in $\Theta((n+m)\log^*(n))$.
Wir können zudem davon ausgehen, dass G zusammenhängend ist. Ob dies der Fall ist, lässt sich z.B. mit einer Tiefensuche in Linearzeit bestimmen, ggf. wenden wir Kruskal's Algorithmus auf die Zusammenhangskomponenten von G an. Damit gilt also $n \leq m+1$, womit sich unsere Laufzeitabschätzung vereinfacht zu $\Theta(m\log^*(m))$.
Ohne Zusicherungen über die Werte der Kantengewichte benötigt das Sortieren der Kanten Zeit in $\Theta(m\log(m))$. Also liegt die Gesamtlaufzeit in $\Theta(m\log(m)+(n+m)\log^*(n))$ bzw. mit der obigen Überlegung $\Theta(m\log(m)+m\log^*(m)) = \Theta(m\log(m))$.