Übungsblatt 10

Abgabe via Moodle.
Deadline Fr. 14ter July

Aufgabe 1 (*Tiefensuche iterativ*, 6 *Punkte*)

Entwerfen Sie eine nicht rekursive Tiefensuche ausgehend von einem Knoten s und geben Sie **Pseudocode** an! Die Laufzeit $\mathrm{O}(m+n)$ darf nicht überschritten werden.

Lösung:
Die folgende nicht rekursive Implementierung der Tiefensuche beruht auf der Verwendung eines Stacks. Die Idee ist es, jeweils die Nachbarn des derzeit bearbeiteten Knotens auf einen Stack zulegen. Man initialisiert den Stack S mit dem Startknoten s. Solange der Stack S nicht leer ist wird das vorderste Element vom Stack heruntergenommen, als gesichtet markiert und anschließend alle noch nicht gesichteten Nachbarn vorne auf den Stack gelegt. So ist sichergestellt, das man zuerst in die Tiefe geht.

```
1: procedure DFS(NodeId s, Graph G)
2:    Stack S = ⟨s⟩
3:    visited = ⟨false, ..., false⟩ Array of Boolean
4:    visited[s] = true
5:    while S ≠ ∅ do
6:       NodeId u = S.pop()
7:       forall (u, v) ∈ E do
8:          if !visited[v] do
9:             visited[v] = true
10:            S.push(v)
11: return
```

Aufgabe 2 (*Finden von Kreisen in Graphen*, $2+2+2$ *Punkte*)

Mittels einer modifizierten Tiefensuche kann man in $\mathrm{O}(n)$ Zeit bestimmen, ob ein **ungerichteter** Graph mit n Knoten einen Kreis enthält oder nicht. Gehen Sie davon aus, dass der Graph als Adjazenzfeld mit bigerichteten Kanten dargestellt wird.

1. Geben Sie an, wie Sie die Tiefensuche modifizieren würden, um einen ensprechenden Algorithmus zu erhalten.
2. Zeigen Sie, dass Ihr Algorithmus tatsächlich das gewünschte Laufzeitverhalten aufweist.
3. Liefert Ihr Algorithmus auch für **gerichtete** Graphen ein korrektes Ergebnis? Wenn nein, wie muss der Algorithmus modifiziert werden? Welches asymptotische Laufzeitverhalten hat der (möglicherweise modifizierte Algorithmus) für **gerichteten** Graphen schlimmstenfalls? Begründen Sie jeweils kurz.

Lösung:

1. Im Graph befindet sich genau dann ein Kreis, wenn bei der Tiefensuche *traverseNonTreeEdge* (siehe Vorlesungsfolien) aufgerufen wird. In Folge dessen wird *traverseNonTreeEdge* folgendermaßen instanziiert:
   1: *traverseNonTreeEdge*$(u, v : \mathit{NodeID})$

2: **print** "'Graph enthält einen Kreis."'
3: **exit**

Ansonten wird ganz am Ende des Algorithmus die Anweisung **print** "'Graph enthält keinen Kreis."' eingefügt. Enthält der Graph einen Kreis, wird diese Anweisung natürlich nie erreicht.

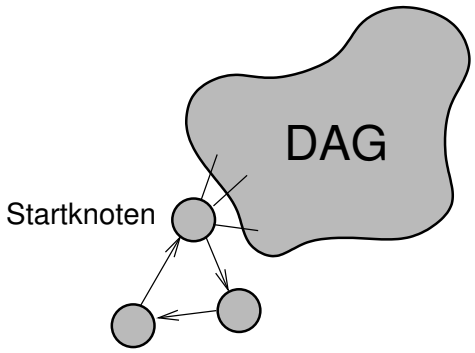
2. Immer wenn die Tiefensuche einen noch nicht markierten Knoten betrachtet, wird dieser markiert. D.h., spätestens beim n-ten Knoten trifft die Suche aber auf einen bereits markierten Knoten und wird abgebrochen. Zudem wird für jeden betrachteten Knoten höchstens eine Kante betrachtet, also nicht mehr als n Kanten insgesamt.

   Das Betrachten eines Knoten braucht aber nur konstante Zeit. Das selbe gilt für Kanten, da man sich in Adjazenzfeldern die nächste Kante eines Knoten stets in konstanter Zeit besorgen kann. Beim Backtracking geht man für jeden Knoten höchstens einmal zu seinem Vorgänger zurück. Für jeden besuchten Knoten wird also auch durch das Backtracking nur konstante Zeit verursacht. Insgesamt dauert die modifizierte Tiefensuche also nicht mehr als $O(n)$ Zeit.

3. Der Algorithmus liefert für gerichtete Graphen so kein korrektes Ergebnis. In einem DAG kann es z.B. passieren, dass ein bereits markierter Knoten gefunden wird, obwohl keine gerichteten Kreise im Graph vorhanden sind. Allerdings kann man dieses Problem beheben, indem man *traverseNonTreeEdge* nochmal ein wenig modifiziert:

   1: *traverseNonTreeEdge*$(u, v : \mathit{NodeID})$
   2: **if** (u, v) ist keine *backward* Kante **return**
   3: **print** "'Graph enthält einen Kreis."'
   4: **exit**

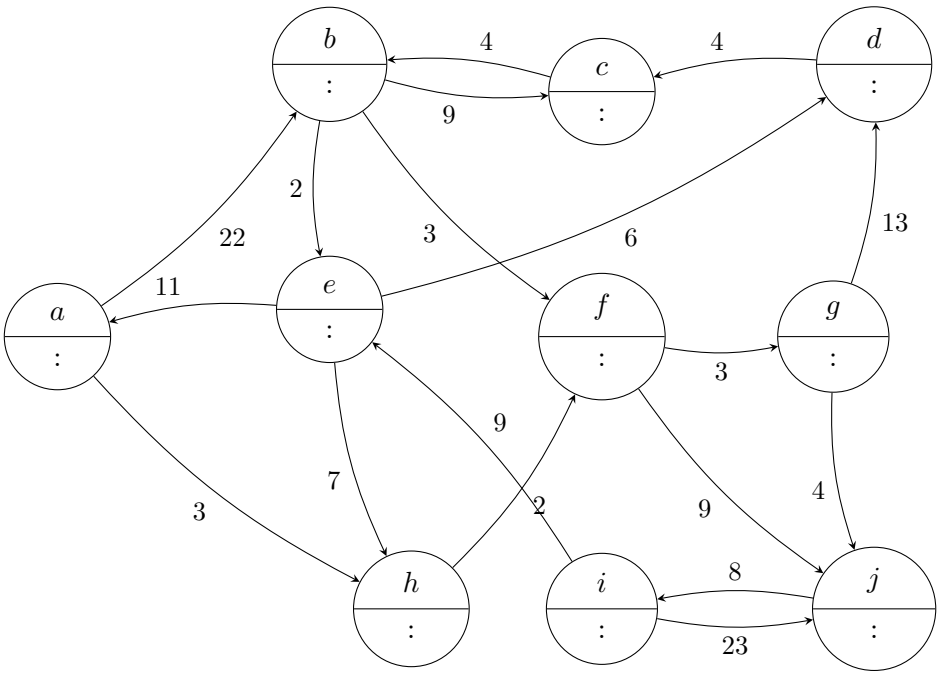
   Der Zeitaufwand liegt im schlimmsten Fall nun in $\Omega(n^2)$. Dies tritt z.B. auf, wenn man einen DAG mit $\Omega(n^2)$ Kanten hat (das gibt es, überlegen), an dem zusätzlich ein gerichteter Kreis hängt. Im schlimmsten Fall wird der gerichtete Kreis von der Tiefensuche als letztes betrachtet (dazu hänge der gerichtete Kreis am Startknoten der Suche).



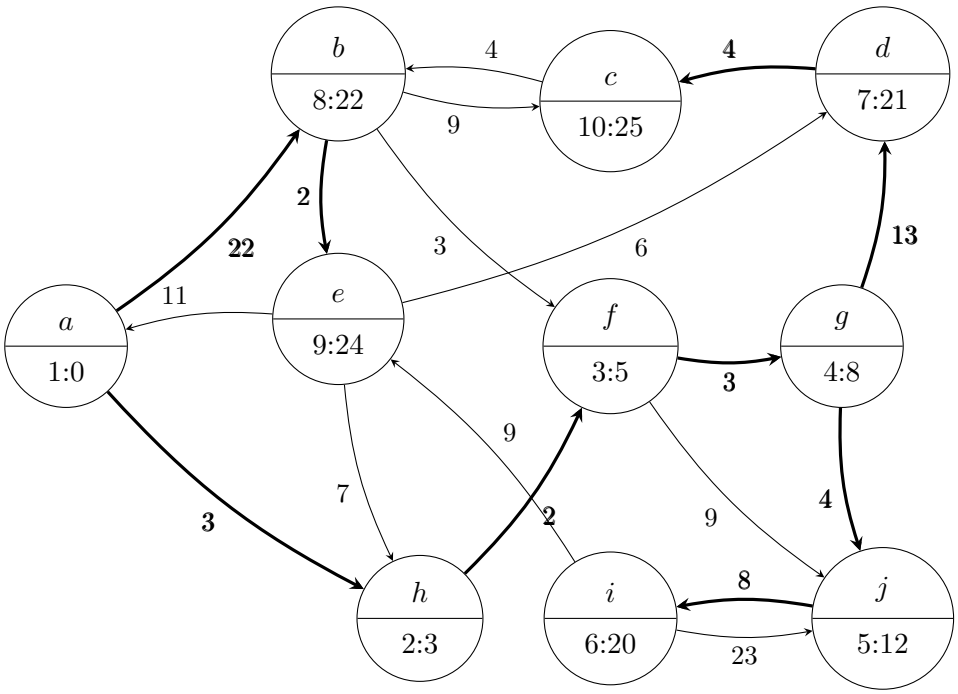
Aufgabe 3 (*Dijkstras Algorithmus,* 4 *Punkte*)

Führen Sie auf folgendem Graphen den Algorithmus von Dijkstra aus, beginnend mit Knoten a. Als Ergebnis soll in jedem Knoten folgendes stehen: links vom Doppelpunkt die Nummer des Schritts, in dem der Knoten gescannt wurde; rechts vom Doppelpunkt die Länge des kürzesten Wegs von a. Zeichnen Sie außerdem den Baum der kürzesten Wege von a aus ein.

Sie können direkt in dieses Blatt einzeichnen.



Lösung:

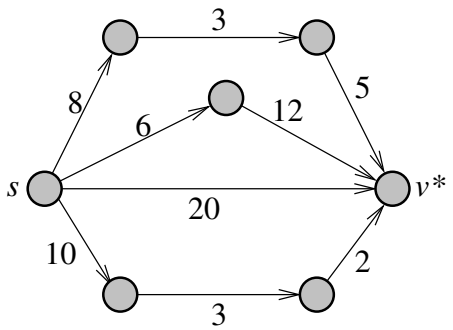


Aufgabe 4 (***Kürzeste Pfade,*** $2+2+2+2$ ***Punkte***)

1. Geben Sie einen gewichteten gerichteten Graph mit positiven Kantengewichten an, der einen Knoten v^* enthält, dessen vorläufige Distanz von Dijkstras Algorithmus mindestens dreimal mittles *decreaseKey* verringert wird. Geben Sie zusätzlich zum Graph auch den Knoten v^* und den Startknoten einer entsprechenden Dijkstra-Suche an.
2. Zeigen Sie: In gewichteten gerichteten Graphen sind Teilpfade von kürzesten Pfade wiederum kürzeste Pfade.

Lösung:

1. Der Startknoten der Dijkstra-Suche ist s, der Knoten, dessen vorläufiges Gewicht mindestens dreimal mit *decreaseKey* verringert wird, ist (wie in Aufgabenstellung bezeichnet) v^*:



2. Sei $P = \langle u, \ldots, x, \ldots, y, \ldots, v \rangle$ ein kürzester Pfad von u nach v und $Q := \langle x, \ldots, y \rangle$ ein Teilpfad. **Annahme zum Widerspruch:** Q ist kein kürzester Pfad von x nach y. Da ein Pfad Q existiert und somit der Fall, dass es keinen Pfad gibt ausgeschlossen wird, gibt es also einen kürzeren Pfad $Q' = \langle x, \ldots, y \rangle$. Ersetze in P den Teilpfad Q durch Q', das ergibt einen gültigen Pfad von u nach v der kürzer ist als P. Dies ist ein **Widerspruch,** da P kürzester Pfad. □

Aufgabe P10 (*Dijkstra: Shortest Reach, optional*)

Für die praktischen Übungen verwenden wir die Plattform `www.hackerrank.com`. Hier müssen Sie sich registrieren um an den Übungen teilzunehmen. Unter dem Link

`https://www.hackerrank.com/adsi-2023`

finden die praktischen Übungen in der Form eines Programmierwettbewerbs statt.

In der zehnten Challenge sind Sie gefragt den Algorithmus von Dijkstra zu implementieren.

Auf Moodle finden Sie wieder ein Framework für diese Aufgabe *framework10.cpp*, welches Sie statt des vorgegebenen Codes verwenden können.
Genauere Informationen, sowie ein Beispiel finden Sie auf HackerRank.