Übungsblatt 2, Algorithmen und Datenstrukturen

Aufgabe 9

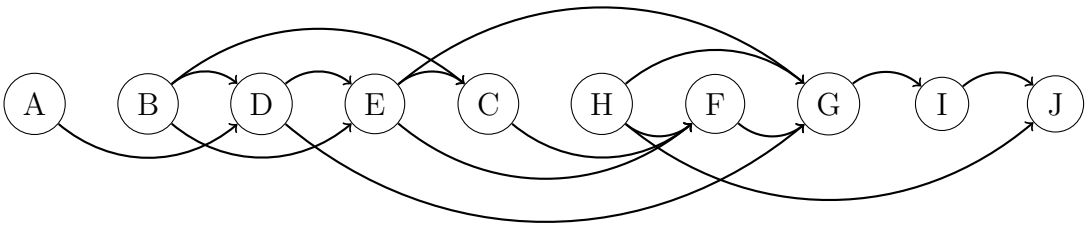
Breitensuche

$\{A, I, D, B, J, G, E, C, H, F\}$

Tiefensuche

$\{A, I, J, G, D, B, E, H, F, C\}$

Aufgabe 10



H kann 6 Positionen annehmen, gleichzeitig können A und B vertauscht werden (2 Positionen), $2 * 6 = 12$.

Aufgabe 11

Selection-Sort

1. $\{5, 17, 11, 13, 7\}$
2. $\{5, 7, 11, 13, 17\}$

3. $\{5, 7, 11, 13, 17\}$
4. $\{5, 7, 11, 13, 17\}$

10 Vergleiche mit Selection-Sort.

Insertion-Sort

1. $\{13, 17, 11, 5, 7\}$
2. $\{11, 13, 17, 5, 7\}$
3. $\{5, 11, 13, 17, 7\}$
4. $\{5, 7, 11, 13, 17\}$

10 Vergleiche mit Insertion-Sort.

Aufgabe 12

```
DFS(Discovered, V, E, v):
  i = 0
  if !Discovered[v]:
    Discovered[v] = true
    i++
    foreach (v, w) in E:
      i += DFS(Discovered, V, E, w)
  return i

MAXCLUSTER(V, E):
  Discovered[v] = false forall v in V
  iMax = 0
  iSum = 0
  foreach v in V:
    i = DFS(V, E, v)
    iSum += i
    if i > iMax:
      iMax = i
    if (|V| - iSum) < iMax:
      return iMax
```

Laufzeit

Die Laufzeitanalyse ist der des *DFSNUM*-Algorithmus aus den Folien sehr ähnlich. Die Betrachtung aller Knoten in der äußersten *for*-Schleife liegt in $O(n)$. Der i-te Aufruf

von *DFS* benötigt $O(n_i + m_i)$, wobei $\sum(n_i) = n$ und $\sum(m_i) = m$, daher ergibt sich eine Laufzeit von $O(n + n + m) = O(n + m)$.

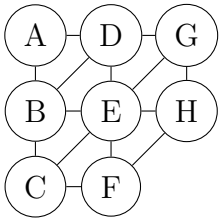
Die spezielle Abbruchbedingung in den letzten zwei Zeilen macht für die Worst-Case Laufzeit keinen Unterschied. Im Fall eines Graphen mit nur einer Zusammenhangskomponente erspart sie $n - 1$ Schleifendurchläufe, was aber auch an $\Omega(n + m)$ nichts ändert.

Korrektheit

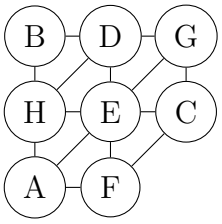
Für fast alle Knoten wird eine Tiefensuche *DFS* ausgeführt, der Zähler i gibt die Größe der zurzeit gefundenen Zusammenhangskomponente an. Falls der Knoten v bereits Teil einer gefundenen Zusammenhangskomponente ist, gilt $i = 0$. In $iMax$ wird der größte Wert für i gespeichert, daher ist $iMax$ offensichtlich die Größe der größten Zusammenhangskomponente.

Nur für *fast alle* Knoten wird *DFS* aufgerufen! Mit hoher Wahrscheinlichkeit bricht der Algorithmus schon sehr viel früher ab, und zwar wenn $|V| - iSum < iMax$. Denn wenn der unerforschte Rest des Graphen (Größe $|V| - iSum$) kleiner ist als die zurzeit größte Zusammenhangskomponente, ist es unmöglich noch eine größere Zusammenhangskomponente zu finden.

Aufgabe 13



(a) Graph 1

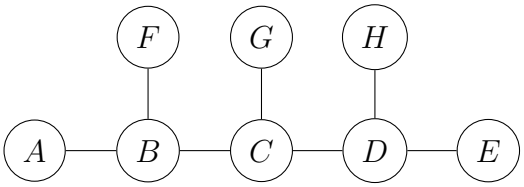


(b) Graph 2

Daraus ergibt sich folgender Isomorphismus $f : V_1 \to V_2$:

$$
\begin{aligned}
f(A) &= B \\
f(D) &= D \\
f(G) &= G \\
f(B) &= H \\
f(E) &= E \\
f(H) &= C \\
f(C) &= A \\
f(F) &= F
\end{aligned}
$$

Aufgabe 14



BFS/DFS kann überall außer bei G und C starten.

Aufgabe 15

```
DFS(V, E, V1, V2, v):
  if V2[v]:
    return false
  if !V1[v]:
    V1[v] = true
    foreach (v, w) in E:
      if !DFS(V, E, V2, V1, w):
        return false
  return true

BIPARTIT(V, E):
  V1[v] = false for all V
  V2[v] = false for all V

  for v in V:
    if not V1[v] and not V2[v]:
      if !DFS(V, E, V1, V2, v):
        return false
  return true
```

Laufzeit

Die Laufzeitanalyse ist wieder der des *DFSNUM*-Algorithmus ähnlich. Die Modifikationen für *DFS* ändern nichts an seiner Laufzeit $O(n + m)$, das ergibt eine Laufzeit von $O(n + n + m) = O(n + m)$.

Korrektheit

Bei jedem Rekursionsschritt (d.h. bei jedem Kantenübertritt) werden die zwei Sets vertauscht. Wenn ein Knoten gefunden wird der schon im gegenüberliegenden Set vorhanden ist, ist der Graph nicht bipartit und *DFS* gibt *false* zurück. Dieser Rückgabewert wird über alle Rekursionsaufrufe hinweg propagiert und ist dann auch Rückgabewert von *BIPARTIT*.

Aufgabe 16

```
MINSTUDIUM(V, E):
  L.. Linked List

  foreach v in V:
    count[v] = 0
  foreach v in V:
    foreach (v, w) in E:
      count[w]++
  foreach v in V:
    if count[v] == 0
      (Gib v zu L am Anfang hinzu)

  rv = 0
  while (L ist nicht leer):
    rv++
    L2.. Alle Elemente aus L, leere L
    foreach v in L2:
      foreach (v, w) in E:
        count[w]--
        if count[w] == 0:
           (Gib w zu L am Anfang hinzu)
  return rv
```

Laufzeit

Die Laufzeit ist dem des Kahn-Algorithmus sehr ähnlich. Die drei Schleifen zur Initialisierung liegen in $O(n+m)$, die Größe von L ist vor und während der while-Schleife in $O(n)$, die Vereinigung aller $L2$ ergibt die Menge an Elementen die jemals in L waren, das ist dasselbe wie V. Dadurch liegt die Laufzeit der while-Schleife auch in $O(n+m)$ und das ist auch die Gesamtlaufzeit.

Korrektheit

Der Algorithmus simuliert einfach, was ein "greedy" Student in jedem Semester machen würde: Einfach alles inskribieren was zurzeit nicht blockiert ist. Ähnlich wie Khan's Algorithmus haben nicht-blockierte LVAs $count = 0$. Für jeden "Block" von nicht-blockierten LVAs wird rv erhöht.