

Graphenalgorithmen

Blatt 7

Markus Vieth

Christian Stricker

21. Dezember 2016

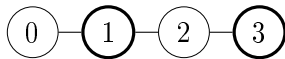
1 Aufgabe 1: FPT Vertex-Cover (10 P)

1.1 Vorüberlegung

Aus $d(v) \leq 2$ folgt, dass der Graph nur aus Einzelknoten mit $d(v) = 0$, Strecken aus 2 Knoten mit $d(v) = 1$ und beliebig vielen Zwischenknoten mit $d(v) = 2$ sowie aus Zyklen in denen für alle Knoten gilt $d(v) = 2$.

Offensichtlich gilt, wir können alle Knoten mit $d(v) = 0$ ohne Schaden löschen. Für Pfade und Zykel wird eine Fallunterscheidung gemacht:

1.1.1 Pfad mit gerader Knotenanzahl



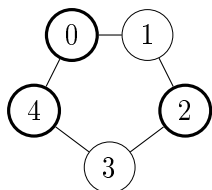
Es reicht von einer Seite anfangend jeden 2. Knoten ins Set aufzunehmen, angefangen mit dem zweiten. Durch Induktion lässt sich zeigen, dass diese Wahl immer minimal ist.

1.1.2 Pfad mit ungerader Knotenanzahl



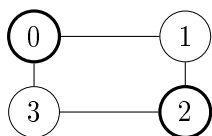
Es reicht von einer Seite anfangend jeden 2. Knoten ins Set aufzunehmen, angefangen mit dem zweiten. Durch Induktion lässt sich zeigen, dass diese Wahl immer minimal ist.

1.1.3 Zykel mit gerader Knotenanzahl



Es reicht von einem Knoten anfangend jeden 2. Knoten ins Set aufzunehmen, angefangen mit dem ersten. Durch Induktion lässt sich zeigen, dass diese Wahl immer minimal ist.

1.1.4 Zykel mit ungerader Knotenanzahl



Es reicht von einer Seite anfangend jeden 2. Knoten ins Set aufzunehmen, angefangen mit dem ersten. Durch Induktion lässt sich zeigen, dass diese Wahl immer minimal ist.

1.2 Pseudocode

```

1  Knoten[] findVertexCover( $G = (V, E)$ ) {
2      Knoten[] result;
3      zuBesuchen = V;
4      for(Knoten v in zuBesuchen) { // Laufzeit in  $O(n)$ 
5          // Lösche alle Knoten vom Grad 0
6          if(v.grad == 0) {
7              zuBesuchen.remove(v); // Bei geeigneter Datenstruktur (z.B. Hashing)  $O(1)$ 
8              continue;
9          }
10         // Bei Knoten vom Grad 1, suche Pfad ab und füge jeden 2. Knoten dem Ergebnis hinzu
11         if(v.grad == 1) {
12             boolean second = false;
13             zuBesuchen.remove(v); // Bei geeigneter Datenstruktur (z.B. Hashing)  $O(1)$ 
14             prev = NULL;
15             while(v.hasNext(prev)) { // Laufzeit in  $O(n)$ 
16                 second = !second;
17                 v = v.next(prev); //  $O(1)$ 
18                 zuBesuchen.remove(v); // Bei geeigneter Datenstruktur (z.B. Hashing)  $O(1)$ 
19                 if(second)
20                     result += [v]; //  $O(1)$ 
21             }
22             continue;
23         }
24     }
25     // Da keine Knoten mehr vom Grad  $< 2$  da sind, müssen alle anderen Knoten Zykel bilden
26     for(Knoten v in zuBesuchen) { // Laufzeit in  $O(n)$ 
27         boolean second = false;
28         zuBesuchen.remove(v); // Bei geeigneter Datenstruktur (z.B. Hashing)  $O(1)$ 
29         result += [v]; //  $O(1)$ 
30         prev = NULL;
31         first = v;
32         while(v.next(prev) != first) { // Laufzeit in  $O(n)$ 
33             second = !second;
34             v = v.next(prev); //  $O(1)$ 
35             zuBesuchen.remove(v); // Bei geeigneter Datenstruktur (z.B. Hashing)  $O(1)$ 
36             if(!second)
37                 result += [v]; //  $O(1)$ 
38         }
39     }
40 }
41 return res;
42 }

44 class Knoten {
45     // Liste benachbarter Knoten
46     Knoten[] nachbarn;

47
48     int grad() { //  $O(1)$ 
49         return nachbarn.size();
50     }

51
52     // return nächst besten Nachbarn ungleich prev
53     Knoten next(Knoten prev) { //  $O(1)$ 
54         if(prev != NULL && prev == nachbarn[0])
55             return nachbarn[1];
56         return nachbarn[0];
57     }

```

```

59     // wenn prev = null, hat dieser Knoten einen Nachbarn? sonst: Hat dieser Knoten einen Nachbarn
        ungleich prev?
60     boolean hasNext(Knoten prev) { // O(1)
61         if(prev == NULL)
62             return this.grad > 0;
63         boolean res = false;
64         for(Knoten v in this.nachbarn)
65             res |= (v == prev);
66         return res;
67     }
68 }

```

⇒ Laufzeit in $O(n) * O(n) + O(n) * O(n) = O(n^2)$ und somit in polynomialer Zeit

q.e.d.

2 Aufgabe 2: FPT Pathfind (10 P)

2.1 $k \leq d$

2.2 Pseudocode

```

1  Knoten[] findKPath(G,k) {
2      Knoten[] result;
3      if (k ≤ G.d ) {
4          Knoten v = V[0];
5          result += [v];
6          v.setMarked(true);
7          while(result.size < k) { // O(k)
8              v = v.nextUnmarked(); // O(d) ∈ O(n)
9              result += [v];
10             v.setMarked(true);
11         }
12     } // O(kn)
13     else
14     {
15         // siehe Text
16     }
17 }

19 class Knoten {
20     // Liste benachbarter Knoten
21     Knoten[] nachbarn;
22     boolean marked = false;

24     void setMarked(boolean marked) {
25         this.merked = marked;
26     }

28     // return nächst besten Nachbarn ungleich prev
29     Knoten nextUnmarked() { // O(d)
30         for(Knoten v in nachbarn)
31             if(!v.marked)
32                 return v;
33         return NULL;
34     }
35 }

```

2.3 $k > d$

Parallel zur Vorlesung nutzen wir eine Tabelle und dynamische Programmierung:

Wir speichern in der Zelle $T[i, j]$ alle erlaubten Sequenzen der Länge i startend beim Knoten v_j . Die erste Zeile beinhaltet somit $T[0, j] = \emptyset$, die zweite Zeile $T[1, j] = \{v_j\}$. Im weiteren nehmen wir alle Pfade in die Zelle auf, für die gilt, dass sie aus $[v_j] + P$ bestehen, mit $P \in T[i-1, \ell]$ mit $\ell \neq j$. Machen wir das über alle Möglichkeiten ergibt sich als Formel:

$$T[i, j] = \bigcup_{v_\ell | \{v_i, v_\ell\} \in E} \{[v_j] + P | P \in T[i-1, \ell] \wedge v_j \notin P\}$$

Weiter gilt, dass ein Knoten v_i genau d Nachbarn hat, die in der obigen Vereinigung beachtet werden und das j bis k läuft. Für jede Zellen müssen maximal d Werte betrachtet werden. Parallel zur Vorlesung gilt somit eine Tabellengröße von $d^k \cdot n$ und eine Laufzeit von $O(k^k \cdot n)$

3 Aufgabe 3: FPT Independent-Set (10 P)