Vorlesungstermin 3:

Graphenalgorithmen I

Markus Püschel
David Steurer

Algorithmen und Datenstrukturen, Herbstsemester 2018, ETH Zürich

Plan für heute

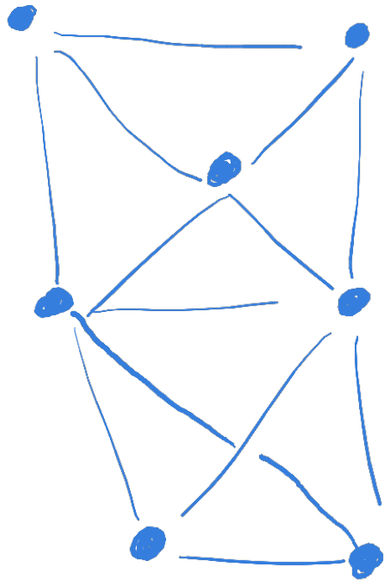
- Beispiele einfacher Graphenalgorithmen
  - Verbindung zu Induktionsbeweisen
- Effizienzbegriff von Algorithmen
  - Beispiel
  - Berechnungsmodell
  - Laufzeit (worst-case, asymptotisch)
- Repräsentation von Graphen im Computer

Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

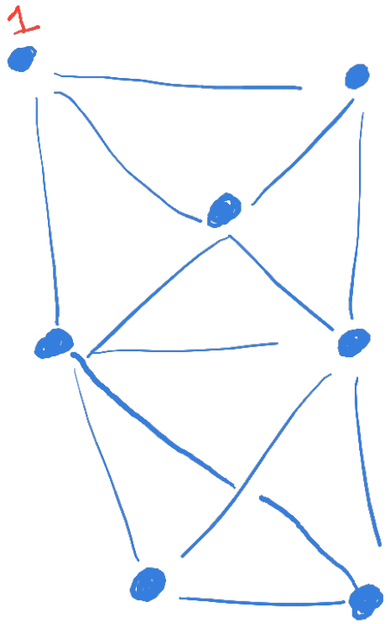


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

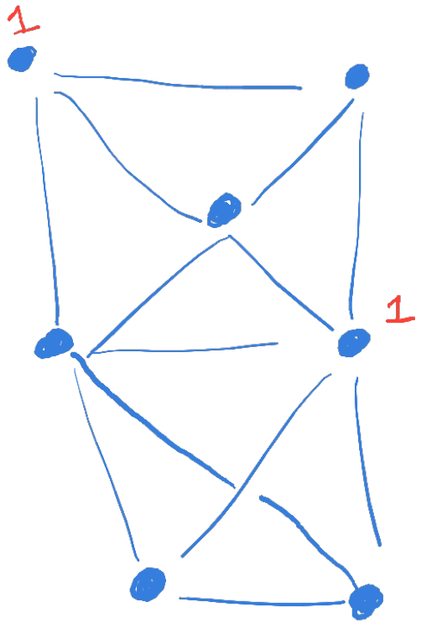


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

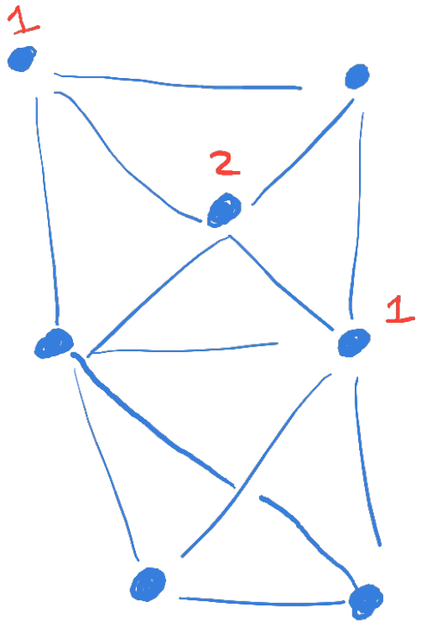


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

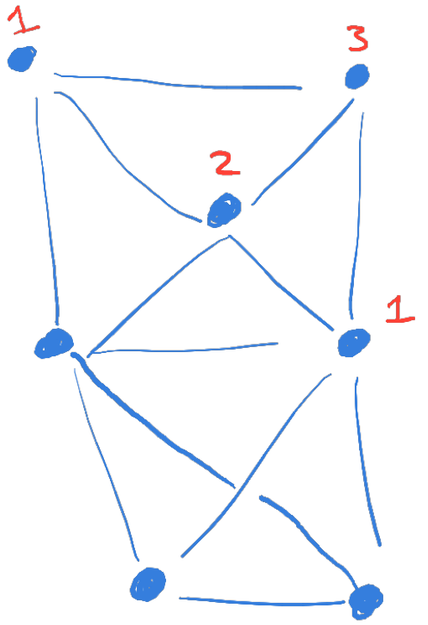


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

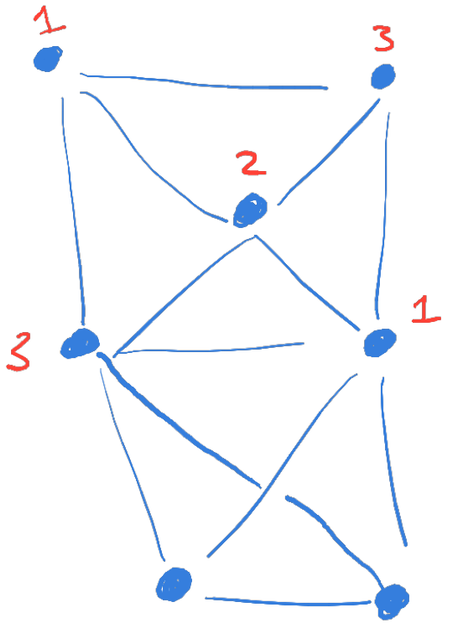


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

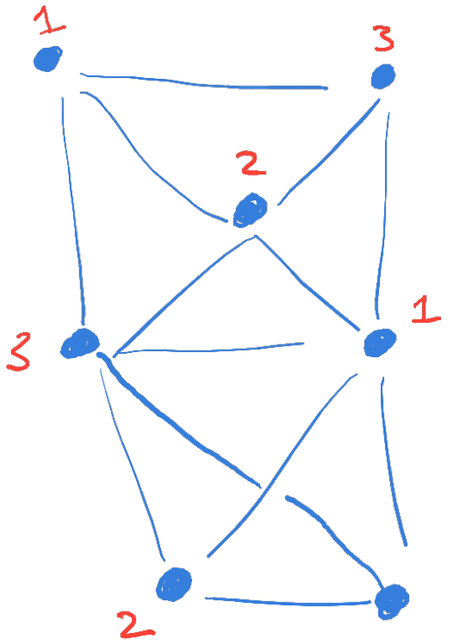


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

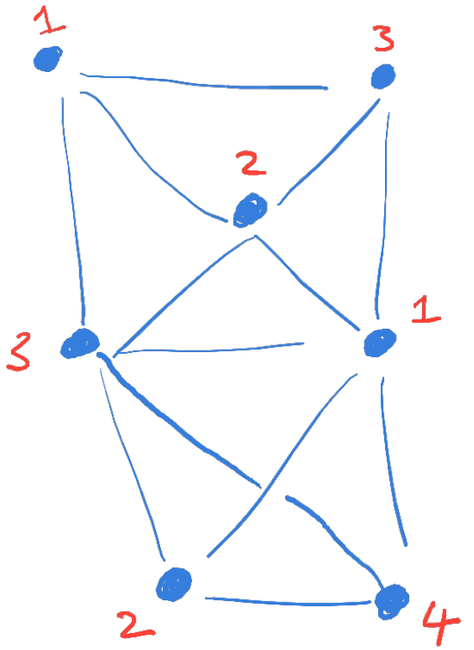


Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden



Algorithmus: Gierige-Knotenfärbung(G)

für alle Knoten v in G (in beliebiger Reihenfolge):

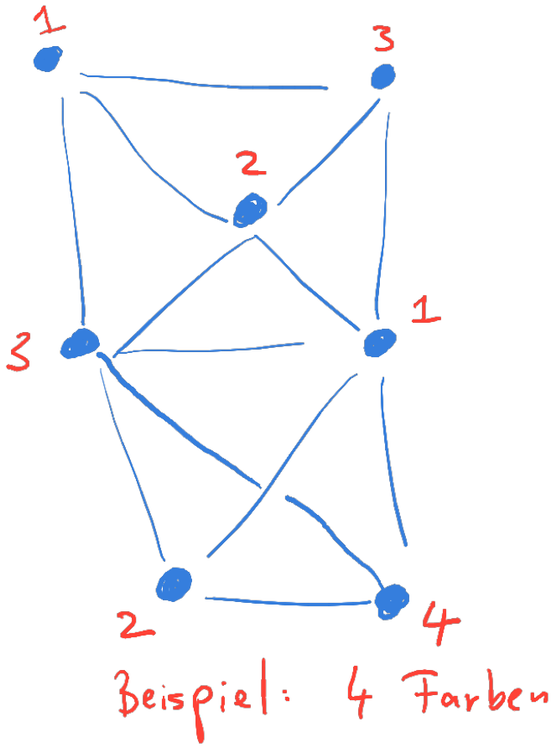
färbe v mit der ersten Farbe, die noch von keinem Nachbarn von v verwendet wird

Beachte: *natürlich-sprachliche* Beschreibung; dennoch völlig *präzise*; Algorithmus kann mit *Stift und Papier* leicht ausgeführt werden

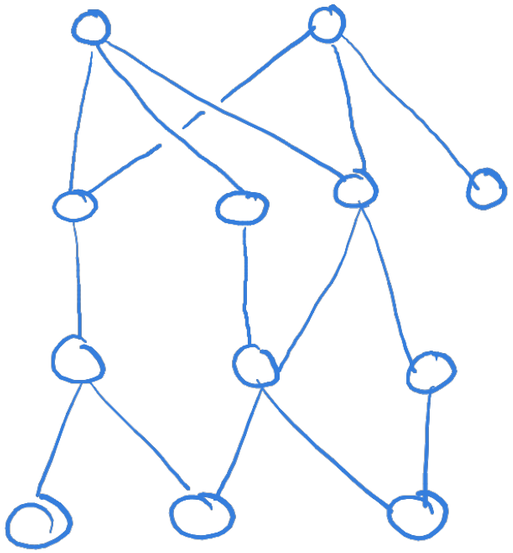
wieviele Farben verwendet dieser Algorithmus höchstens?

Satz: für Graphen mit max. Knotengrad Δ kommt der Algorithmus mit $\Delta + 1$ Farben aus

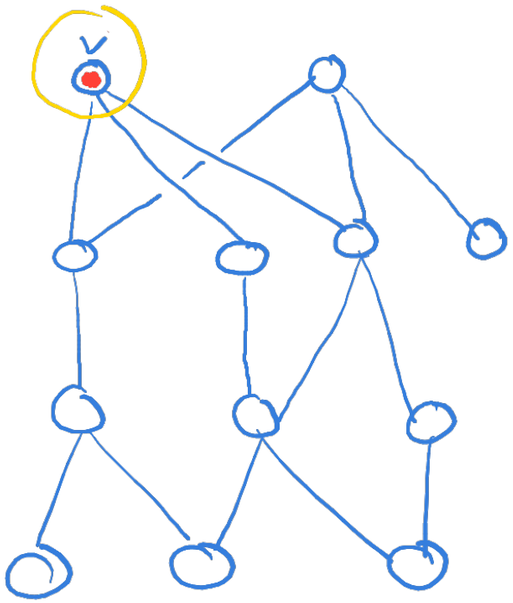
Überlege: Algorithmus entspricht *Induktionsbeweis* von letzter Woche



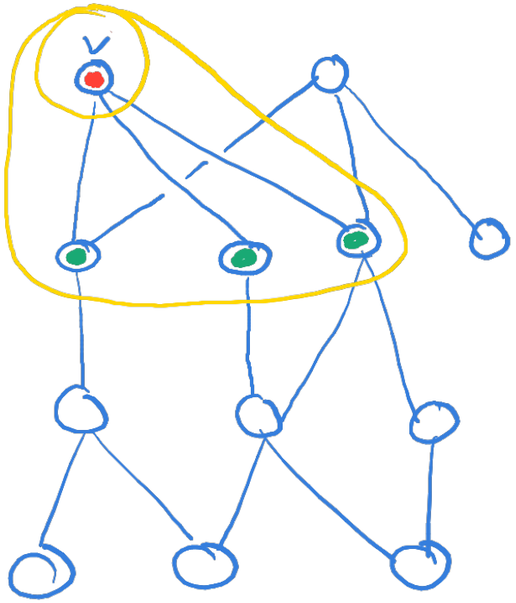
Algorithmus: Bipartition(G)



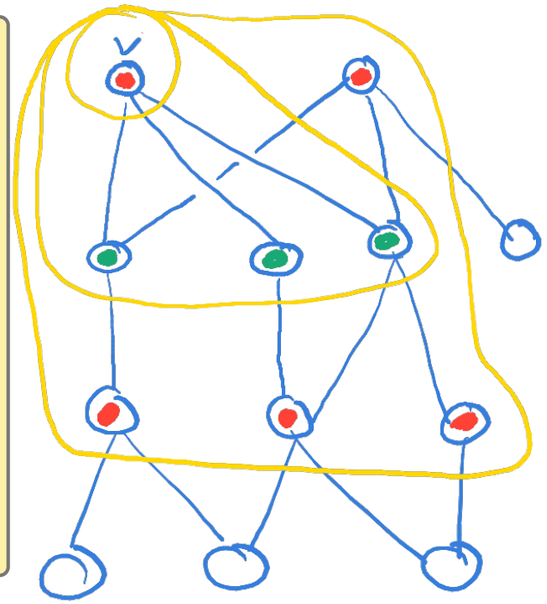
Algorithmus: Bipartition(G)



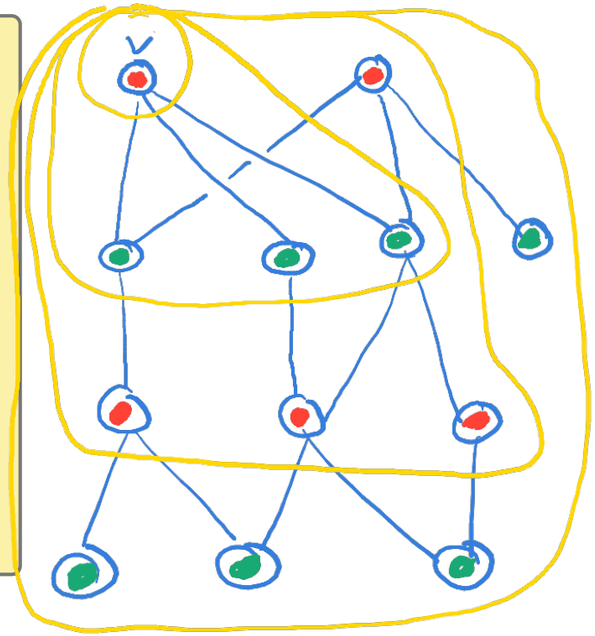
Algorithmus: Bipartition(G)



Algorithmus: Bipartition(G)

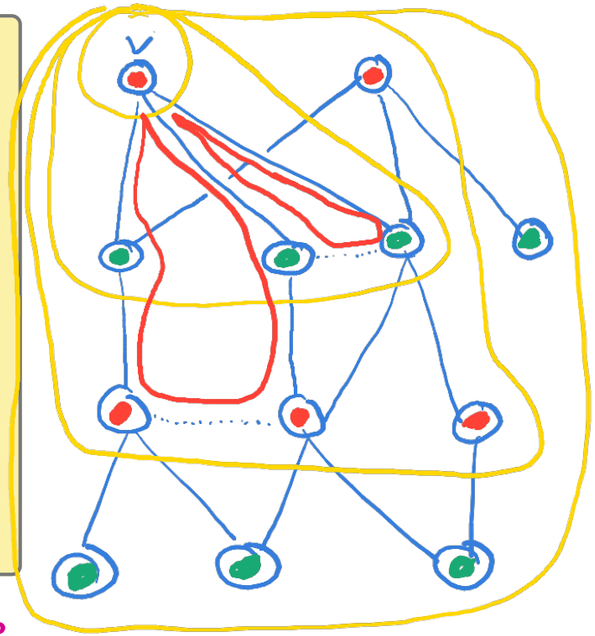


Algorithmus: Bipartition(G)



Algorithmus: Bipartition(G)

1. wähle beliebigen Knoten v und färbe ihn rot
2. setze $S \leftarrow \{v\}$ und $t \leftarrow 0$
3. wiederhole solange S Nachbarn in $V_G \setminus S$ hat
   1. färbe die Nachbarn von S in $V_G \setminus S$ grün
      falls t gerade ist oder rot falls t ungerade ist
   2. setze $S \leftarrow S \cup N_G(S)$ und $t \leftarrow t + 1$



wann färbt der Algorithmus den Graph so, dass jede Kante einen roten und einen grünen Endknoten hat?

Satz: der Algorithmus findet eine gültige Bipartition, wenn der Graph *keinen ungerade Kreis enthält* (und zusammenhängend ist)

Überlege: Algorithmus entspricht wieder *Induktionsbeweis* von letzter Woche

Effizienzbegriff von Algorithmen

es gibt *effiziente* Algorithmen (z.B. gierige Knotenfärbung) und *ineffiziente* Algorithmen (z.B. alle möglichen Färbungen eines Graphen durchprobieren)

Wie können wir die Effizienz von Algorithmen formalisieren?

die genaue *Laufzeit eines Computerprogramms* hängt von vielen Details ab (z.B. spezielle Eingabe, Computerarchitektur, und Compiler-Optimierungen)

um *effiziente und ineffiziente Algorithmen* zu unterscheiden, stellt sich heraus, dass viele dieser Details keine Rolle spielen

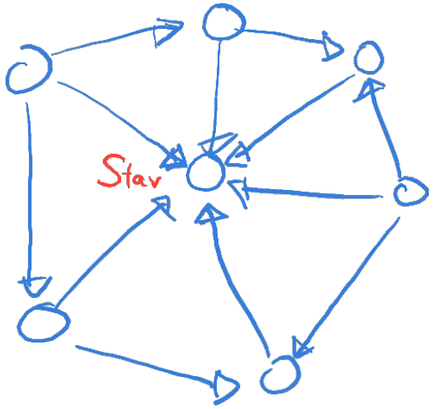
daher betrachten wir vereinfachte *Berechnungs- und Kostenmodelle*, bei denen diese Details ausgelassen werden

Vereinfachtes Beispiel: Star finden

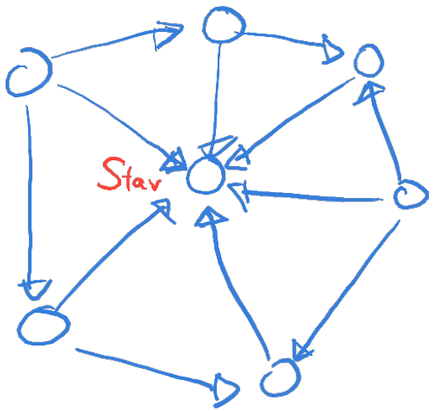
gegeben: gerichteter Graph G mit n Knoten (ohne Schleifen)

gesucht: "Star" Knoten s mit Ausgangsgrad 0 und Eingangsgrad $n - 1$

Beispiel: wenn Donald Trump jetzt zur Vorlesung kommt, kennt jeder seinen Namen; aber er würde (wohl) keinen Namen hier kennen



Vereinfachtes Beispiel: Star finden



gegeben: gerichteter Graph G mit n Knoten (ohne Schleifen)

gesucht: "Star" Knoten s mit Ausgangsgrad 0 und Eingangsgrad $n-1$

mit wieviele Kantenanfragen "$(u,v) \in E_G$?" können wir einen "Star" finden?

"prüfe" einzelnen Knoten mit $2(n-1)$ Anfragen

prüfe jeden Knoten mit $n \cdot 2(n-1)$ Anfragen

spare wiederholte Anfragen → $n \cdot (n-1)$ Anfr.

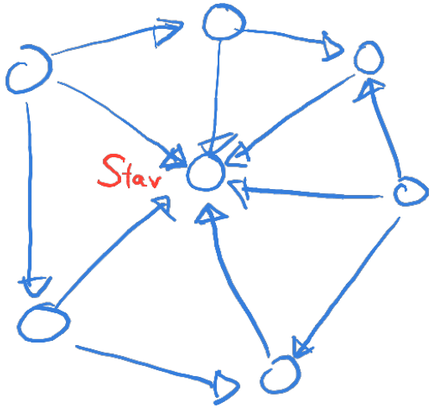
geht es besser?

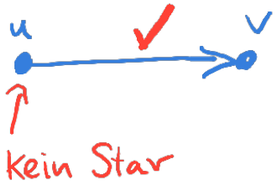
Vereinfachtes Beispiel: Star finden

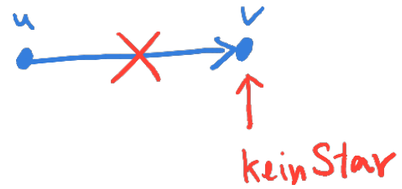
gegeben: gerichteter Graph G mit n Knoten (ohne Schleifen)

gesucht: "Star" Knoten s mit Ausgangsgrad 0 und Eingangsgrad $n-1$

Beobachtung: Jede Antwort auf "$(u,v) \in E_G$?" schliesst entweder u oder v als Star aus



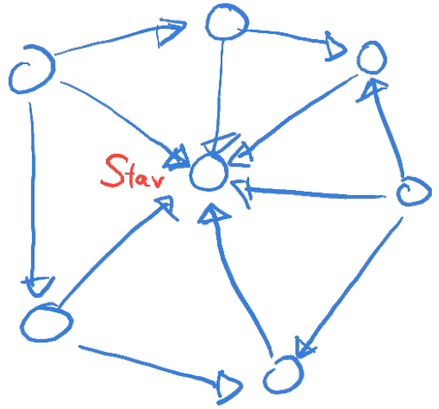




Vereinfachtes Beispiel: Star finden

gegeben: gerichteter Graph G mit n Knoten (ohne Schleifen)

gesucht: "Star" Knoten s mit Ausgangsgrad 0 und Eingangsgrad $n-1$



Algorithmus: StarFinden(G)

1. setze $S \leftarrow V_G$
2. solange S zwei Knoten $u \neq v$ enthält
   1. stelle Anfrage "$(u,v) \in E_G$?"
   2. falls $(u,v) \in E_G$, entferne u von S; falls $(u,v) \notin E_G$, entferne v von S
3. prüfe ob verbleibender Knoten in S Star ist

← genau n-1 Durchläufe

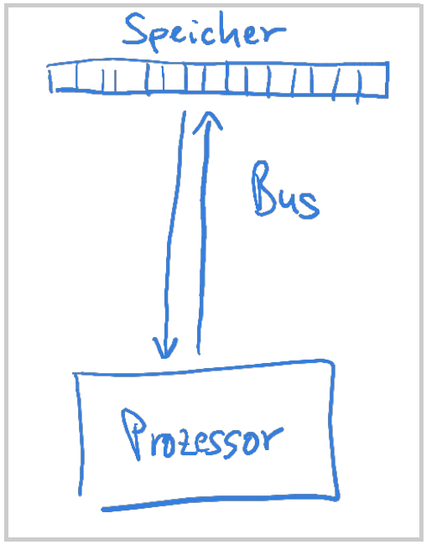
1 Anfrage pro Scheifendurchlauf

← ≤ 2·(n-1) Anfragen

Algorithmus kommt mit $3(n-1)$ Anfragen aus

Allgemeines Berechnungsmodell

Komponenten



- *Speicher:* a-priori unbeschränkt viele, adressierbare Speicherzellen ("Papier")
- *Prozessor:* führt *elementare Operationen* aus wie Rechenoperationen (Addition, Subtraktion, Multiplikation, Division), Vergleichsoperationen (=,<,>), Lese- und Schreibzugriffe auf Speicher ("Stift")
- *Bus:* verbindet Prozessor und Speicher

Anmerkung: eine Speicherzelle kann verschiedene Daten enthalten, z.B. ein Bit (0 oder 1), eine Zahl (nicht viel grösser als Zahlen in der Eingabe), konstante Anzahl von Bits oder Zahlen (unabhängig von "Grösse" der Eingabe)

Laufzeit

Begriff: *Laufzeit* eines Algorithmus für eine Eingabe ist die Anzahl der elementaren Operationen, die der Prozessor während der Berechnung ausführt

die Laufzeit könnte von vielen Details der Eingabe abhängen

daher beschränken wir die Laufzeit als Funktion der "Grösse" der Eingabe (mögliche Begriffe der "Grösse" sind problemabhängig; z.B. die Knoten- oder Kantenzahl bei Graphen)

Definition: ein Algorithmus A hat *Laufzeit* f falls $f(n) =$ *maximale Laufzeit* von A über alle Eingaben der Grösse n

bei manchen Problemen hängt die "Grösse" von mehreren Parametern ab (z.B. Knoten- und Kantenzahl bei Graphenproblemen)

dann ist die Laufzeit eine Funktion dieser Parameter

Asymptotische Notation

Überlegung: aufgrund unseres vereinfachten Berechnungsmodell macht es wenig Sinn zwischen den Laufzeiten $10 \cdot n$ und $10000 \cdot n$ zu unterscheiden (da die tatsächlichen Kosten verschiedener elementarer Operationen sich um teils grosse konstante Faktoren unterscheiden)

daher ignorieren wir *konstante Faktoren* (unabhängig von Eingabegrösse)

Definition: für Funktion $f(n)$, besteht $O(f)$ aus allen Funktionen $g(n)$, so dass gilt

$$\exists C > 0.\ \forall n \geqslant 1.\ g(n) \leqslant C \cdot f(n)$$

Falls $g(n) \in O(f)$, sagen wir "$g(n)$ hat *Grössenordnung* höchstens $f(n)$" oder "$g(n)$ ist *bis auf konstante Faktoren beschränkt* durch $f(n)$" und schreiben $g(n) \leqslant O(f(n))$

Asymptotische Notation

Definition: für Funktion $f(n)$, besteht $O(f)$ aus allen Funktionen $g(n)$, so dass gilt

$$\exists C > 0.\ \forall n \geqslant 1.\ g(n) \leqslant C \cdot f(n)$$

Falls $g(n) \in O(f)$, sagen wir "$g(n)$ hat *Grössenordnung höchstens* $f(n)$" oder "$g(n)$ ist *bis auf konstante Faktoren beschränkt* durch $f(n)$" und schreiben $g(n) \leqslant O(f(n))$

Beispiel:
naives Star-Finden macht $n \cdot (n-1) \leqslant O(n^2)$ Anfragen
cleveres Star-Finden macht $3 \cdot (n-1) \leqslant O(n)$ Anfragen

Asymptotische Notation—Übung

"$g(n) \leqslant O(f(n))$" steht für folgende Aussage über zwei Funktionen f und g:

$$\exists C > 0.\ \forall n \geqslant 1.\ g(n) \leqslant C \cdot f(n)$$

formalisiert: f kleiner gleich g bis auf konstanten Faktor

Gilt $100 \cdot n + n^2 \leqslant O(n^2)$? Ja!

Wähle $C = 101$.

Dann gilt $100 \cdot n + n^2 \leq 100n^2 + n^2 = 101n^2 = Cn^2$ für alle $n \geq 1$

Gilt $n^2 \leqslant O(1000 \cdot n)$? Nein!

Für jedes $C > 0$, gilt $n^2 > C \cdot 1000n$ wenn wir $n = C \cdot 1000 + 1$ wählen

Repräsentationen von Graphen (im Computerspeicher)

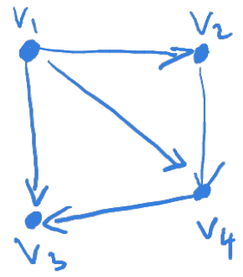
gerichteter Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$

Definition: die *Adjazenzmatrix* von G ist eine Tabelle A mit n Zeilen und n Spalten, so dass für den Eintrag A_{ij} in Zeile i und Spalte j gilt

$$A_{ij} = \begin{cases} 0 & (v_i, v_j) \notin E \\ 1 & (v_i, v_j) \in E \end{cases}$$

Laufzeit $O(1)$ um zu testen ob G bestimmte Kante (v_i, v_j) enthält

Laufzeit $O(n)$ um alle Nachfolger eines Knoten v_i aufzuzählen



	1	2	3	4
1	0	1	1	1
2	0	0	0	1
3	0	0	0	0
4	0	0	1	0

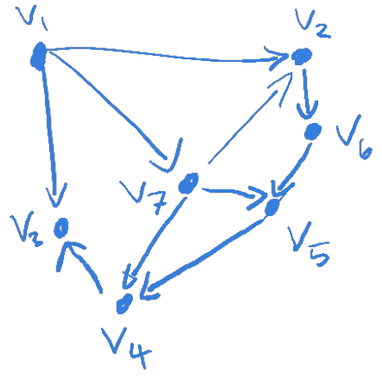
Representationen von Graphen (im Computerspeicher)

gerichteter Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$

Definition: die *Adjazenzliste* von G ist eine Tabelle A mit n Einträgen, so dass Eintrag A_i eine Liste aller Nachfolger von v_i enthält

Laufzeit $O(\deg^+(v_i) + 1)$ um zu testen ob G bestimmte Kante (v_i, v_j) enthält

Laufzeit $O(\deg^+(v_i) + 1)$ um alle Nachfolger eines Knoten v_i aufzuzählen



1	2, 3, 7
2	6
3	(leer)
4	3
5	4
6	5
7	2, 4, 5