

Effiziente Algorithmen

Zusammenfassung

Thomas Mohr

Contents

1	Grundlagen	4
1.1	Stable Matching	4
1.1.1	Propose-&-Reject	5
1.1.2	5 repräsentative Probleme	5
1.2	Zentrale Konzepte & Konventionen	7
1.2.1	\mathcal{O} -Notation	8
1.3	Graphen	9
1.3.1	Repräsentation	11
1.3.2	Bekannte Begriffe	12
1.3.3	Graphtraversierung	13
1.4	Bipartite Graphen	15
1.4.1	Starker Zusammenhang	16
1.4.2	DAG's & topologische Sortierungen	17
2	Greedyalgorithmen	18
2.1	Interval scheduling	18
2.2	Interval Partitioning	18
2.3	Verspätungsminimierung	19
2.4	Kürzeste Wege in Graphen	19
2.5	Minimale Spannbäume	20
2.6	Kodierung	20
2.6.1	Problemformulierung	21
2.6.2	Huffmann-Algorithmus	21
3	Divide-&-Conquer	21
3.1	Grundprinzip	21
3.2	Rekursionsungleichungen	21
3.2.1	Master Theorem	22
3.2.2	Zählen von Inversionen	22
3.3	Closest Pair	22

List of Algorithms

1	Propose-&-Reject	5
2	DFS	14
3	Interval Partitioning	19
4	Huffmann-Algorithmus	21

1 Grundlagen

1.1 Stable Matching

- Eingabe: Zwei gleichgroße Mengen $M = \{m_1, \dots, m_n\}$ und $W = \{w_1, \dots, w_n\}$, welche in diesem Beispiel Männer und Frauen darstellen.
- Aufgabe: Finde paarweise Zuordnung zwischen den Elementen aus M und W , so dass für jeden $m \in M$ und jede $w \in W$, die nicht m zugeordnet ist, gilt (Stabilität):
 1. m zieht ihm zugeordnete w' gegenüber w vor, oder
 2. w zieht ihr zugeordneten m' gegenüber m vor

Stabilität beschreibt hierbei, dass die Paarungen tatsächlich vorteilhaft sind für einen von beiden. D.h., wenn (m, w) ein Paar ist, aber m lieber ein Paar mit w' bilden würde, bzw. w lieber ein Paar mit m' bilden würde, so wäre ihre Verbindung instabil.

- Beispiel:

$$M = \{X, Y, Z\}$$

$$X : A < B < C$$

$$Y : B < A < C$$

$$Z : A < B < C$$

$$W = \{A, B, C\}$$

$$A : Y < X < Z$$

$$B : X < Y < Z$$

$$C : X < Y < Z$$

- Zuordnung $(X, C), (Y, B), (Z, A)$
Ist diese Zuordnung stabil? Nein! X zieht A vor und A zieht X vor.
- Zuordnung $(X, A), (Y, B), (Z, C)$
Ist die Zuordnung stabil? Ja!
 1. Niemand will mit Z oder C tauschen
 2. X hat Traumfrau
 3. Y hat Traumfrau

1.1.1 Propose-&-Reject

Algorithm 1: Propose-&-Reject

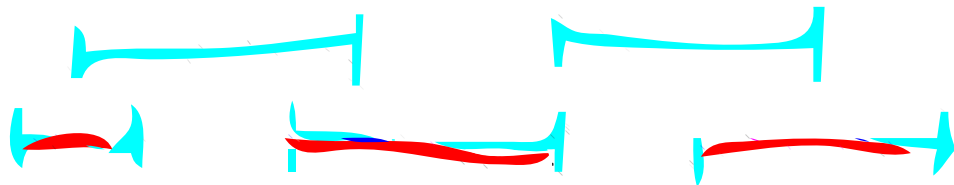
```
1 alle  $m \in M$  und alle  $w \in W$  "frei"
2 while  $\exists m \in M : m$  ist frei und  $\exists w \in W$  der  $m$  noch keinen Antrag gemacht hat
  do
3    $w \leftarrow$  erste noch "unbeantragte" Frau in  $m$ 's Präferenzfolge
4   if  $w$  ist frei then
5      $(m, w)$  wird Paar
6      $m \leftarrow$  "verlobt"
7      $w \leftarrow$  "verlobt"
8   end
9   else if  $w$  zieht  $m$  ihrem aktuellen Verlobten  $m'$  vor then
10     $(m, w)$  wird Paar
11     $m \leftarrow$  "verlobt"
12     $w \leftarrow$  "verlobt"
13     $m' \leftarrow$  "frei"
14  end
15  else
16     $w$  lehnt  $m$  ab
17  end
18 end
```

- Propose–Reject findet immer ein **perfektes Matching**, das stabil ist, und benötigt dazu $\leq n^2$ **Durchläufe** der while-Schleife.
- Jeder Mann bekommt die bestmögliche Frau zugeordnet ("männeroptimal").
- Jede Frau bekommt den schlechtestmöglichen Mann zugeordnet.

1.1.2 5 repräsentative Probleme

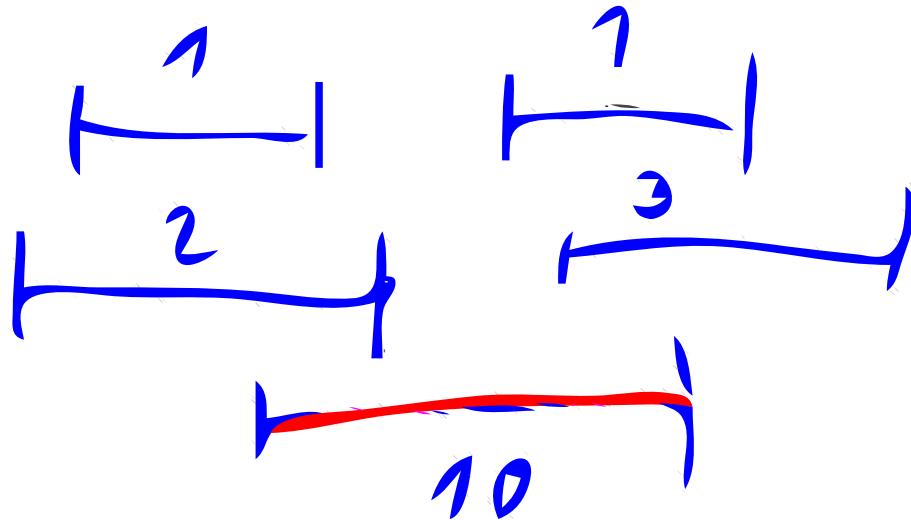
1. Interval Scheduling

- Eingabe: Intervalle mit Start- & Endzeiten
- Aufgabe: Finde größtmögliche Menge nichtüberlappender Intervalle
- Beispiel für Greedy



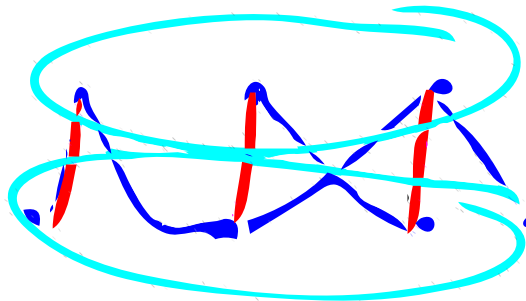
2. Gewichtetes Interval Scheduling

- Eingabe: Intervalle mit Start- & Endzeiten und positiven Gewichten
- Aufgabe: Finde Lösung mit größtmöglichem Gesamtgewicht
- Beispiel für dynamisches Programmieren:



3. Bipartites Matching

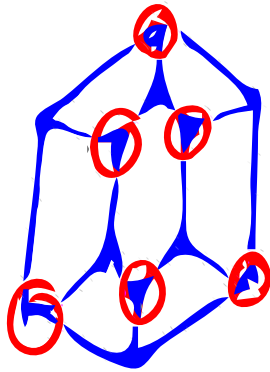
- Eingabe: Bipartiter Graph
- Aufgabe: Finde größtmögliche "unabhängige" (keine gemeinsamen Endpunkte) Kantenmenge
- Beispiel für Netzwerkflüsse:



4. Independent set

- Eingabe: Ungerichteter Graph
- Aufgabe: Finde größtmögliche "unabhängige" (paarweise nicht benachbarte) Knotenmenge

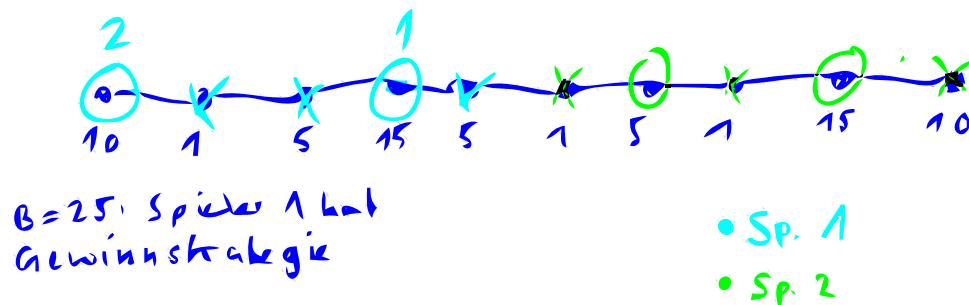
- Beispiel (NP-schwer):



- Vorige Probleme sind Spezialfälle von **Independent Set**

5. Competitive Facility Location

- Eingabe: Knotengewichteter Graph
- Regeln: Zwei Spieler wählen alternierend Knoten; gewählter Knoten wird samt Nachbarn gelöscht.
- Ziel: Spieler 1 will Knoten so wählen, dass Spieler 2 möglichst wenige Punkte macht
- Beispiel (PSPACE-vollständig):

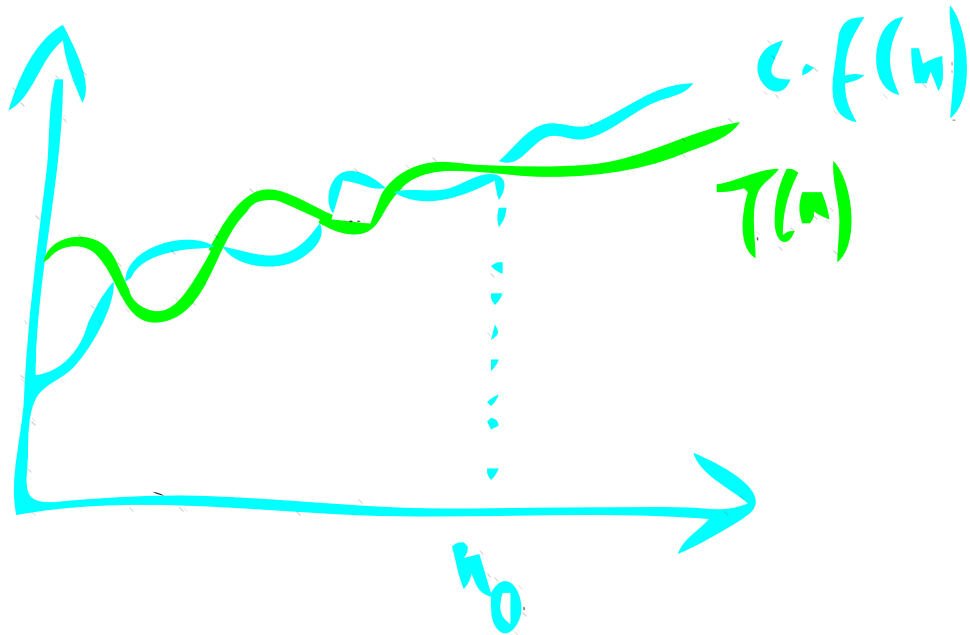


1.2 Zentrale Konzepte & Konventionen

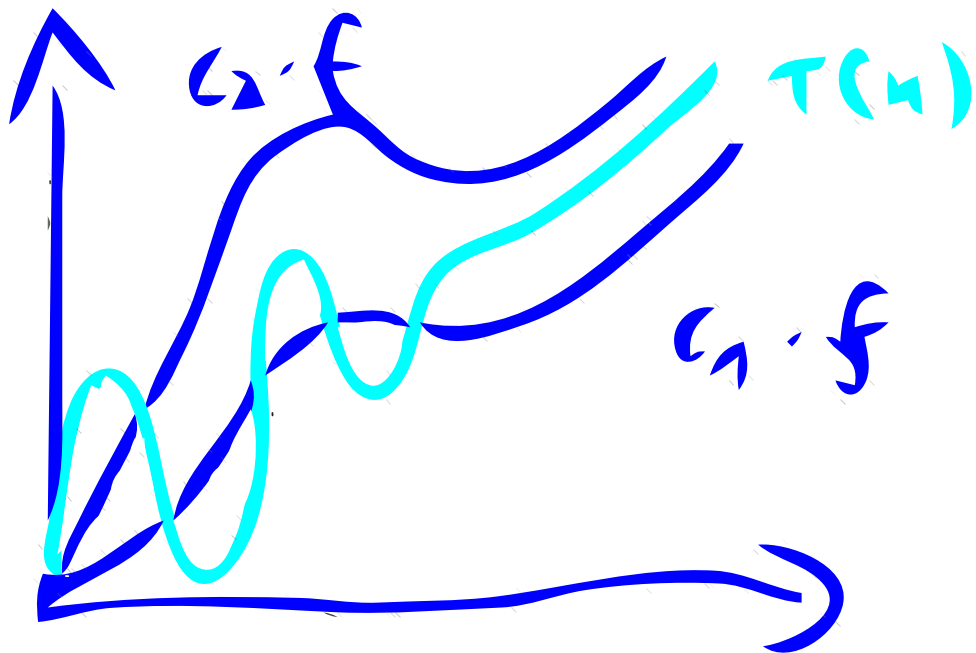
- Ziel **effizienter** Algorithmen: **polynomielle Laufzeit**, d.h. es existieren Konstanten c, d , so dass der Algorithmus bei Eingabegröße n nach $c \cdot n^d$ Schritten terminiert.
- Man beachte: **Worst-Case Analyse**

1.2.1 \mathcal{O} -Notation

- $T(n) = \mathcal{O}(f(n))$ falls $\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 : T(n) \leq c \cdot f(n)$



- $T(n) = \Omega(f(n))$ falls $\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 : T(n) \geq c \cdot f(n)$
- $T(n) = \Theta(f(n))$ falls $T(n) = \mathcal{O}(f(n))$ und $T(n) = \Omega(f(n))$



- $T(n) = o(f(n))$ falls $\forall c > 0 : \exists n_0 \geq 0 \forall n \geq n_0 : T(n) < c \cdot f(n)$
- $T(n) = \omega(f(n))$ falls $f(n) = o(T(n))$

Wenn die Eingabe n groß genug wird wächst T

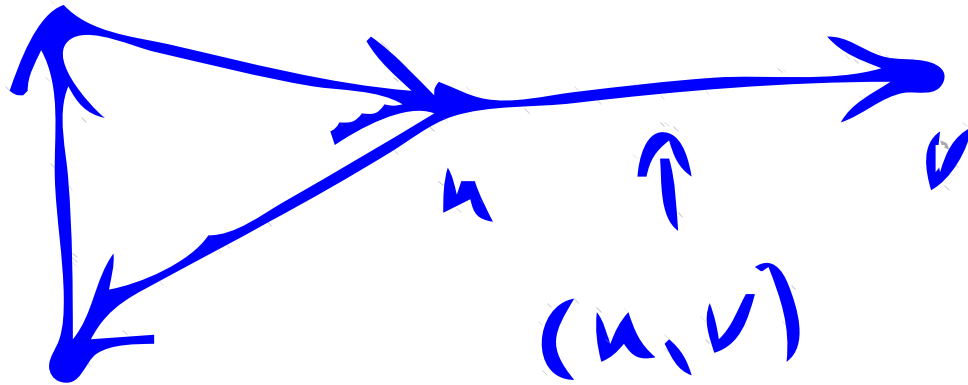
- | | |
|------------------------------|-----------------|
| • $T(n) = \mathcal{O}(f(n))$ | nicht schneller |
| • $T(n) = \Omega(f(n))$ | nicht langsamer |
| • $T(n) = \Theta(f(n))$ | genauso schnell |
| • $T(n) = o(f(n))$ | echt langsamer |
| • $T(n) = \omega(f(n))$ | echt schneller |

als f .

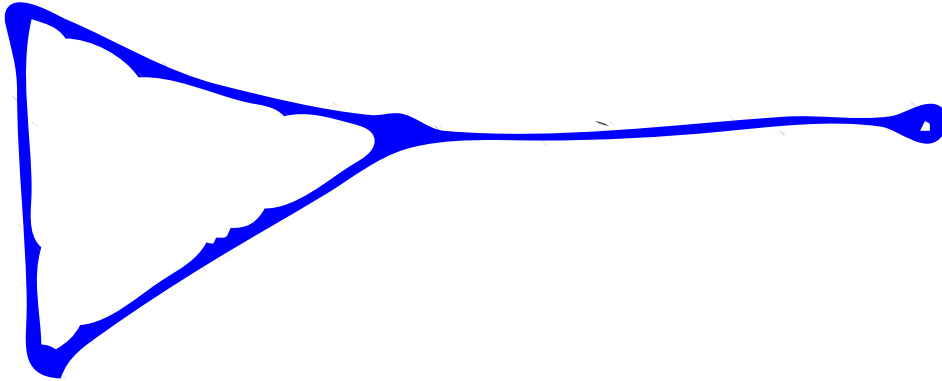
1.3 Graphen

- $G = (V, E)$
- Konvention: $n. = |V|, m := |E|$

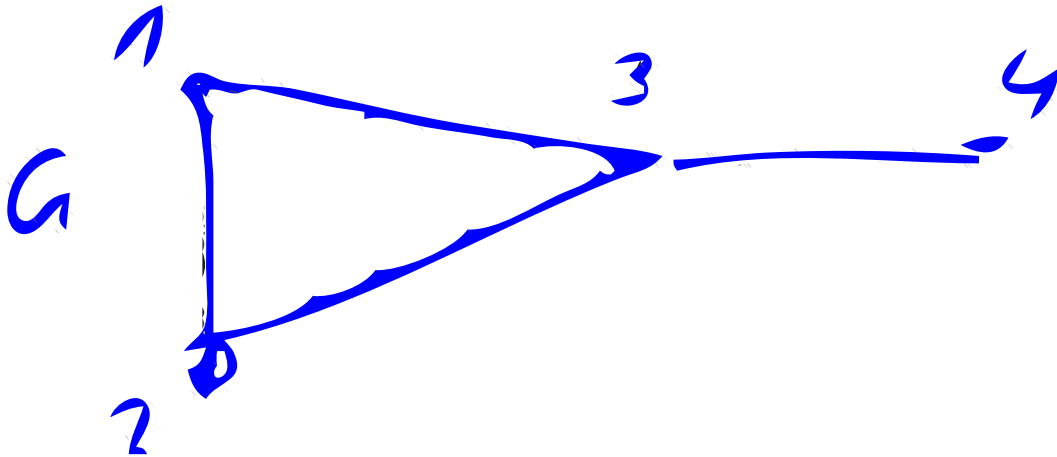
- Gerichteter Graph: $e \in E$ mit $e = (u, v), u, v \in V$ (geordnetes Paar)



- Ungerichteter Graph: $e \in E$ mit $e = \{u, v\}, u, v \in V$ (ungeordnetes Paar)



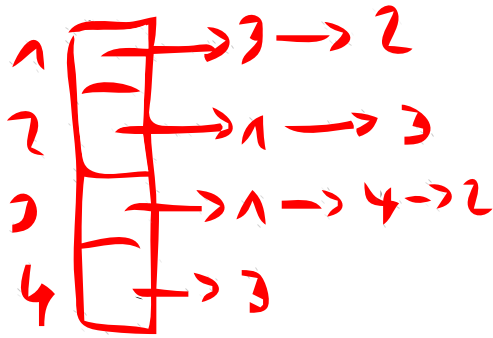
1.3.1 Repräsentation



- Adjazenzmatrix
 - $n \times n$ 0/1-Matrix
 - $A_{i,j} = 1 \iff \{v_i, v_j\} \in E$
 - Hoher Speicherbedarf für Graphen mit wenigen Kanten ("dünn", "sparse")

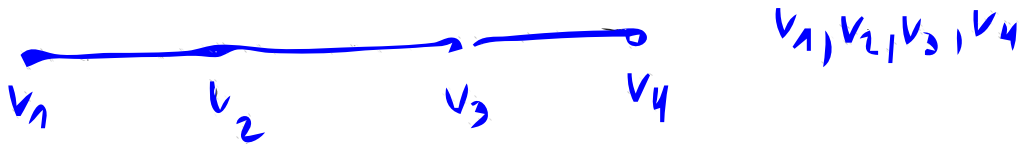
	1	2	3	4
1		1	1	
2	1		1	
3	1	1		1
4			1	

- Adjazenzliste
 - Array/Liste von Nachbarn für jeden Knoten
 - Jeder Array-Eintrag führt zur Liste von Nachbarn

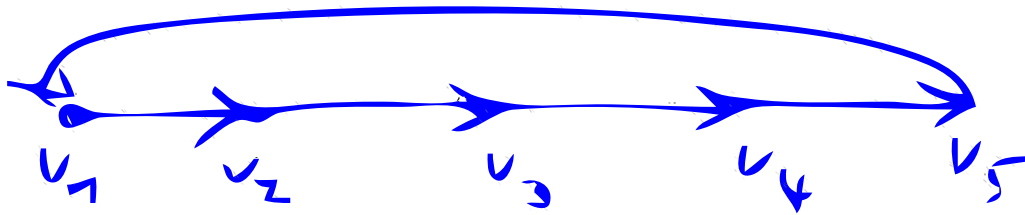


1.3.2 Bekannte Begriffe

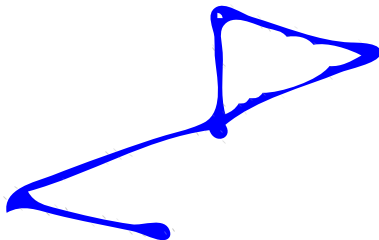
- Pfad: Folge von Knoten, aufeinanderfolgende sind benachbart



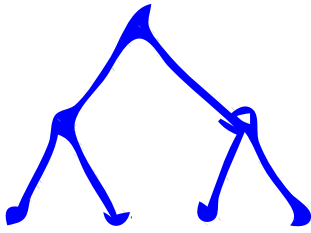
- Kreis: Pfad v_1, \dots, v_l mit $v_1 = v_l$



- Ungerichteter zusammenhängender Graph: Zwischen allen Knotenpaaren existiert ein Pfad



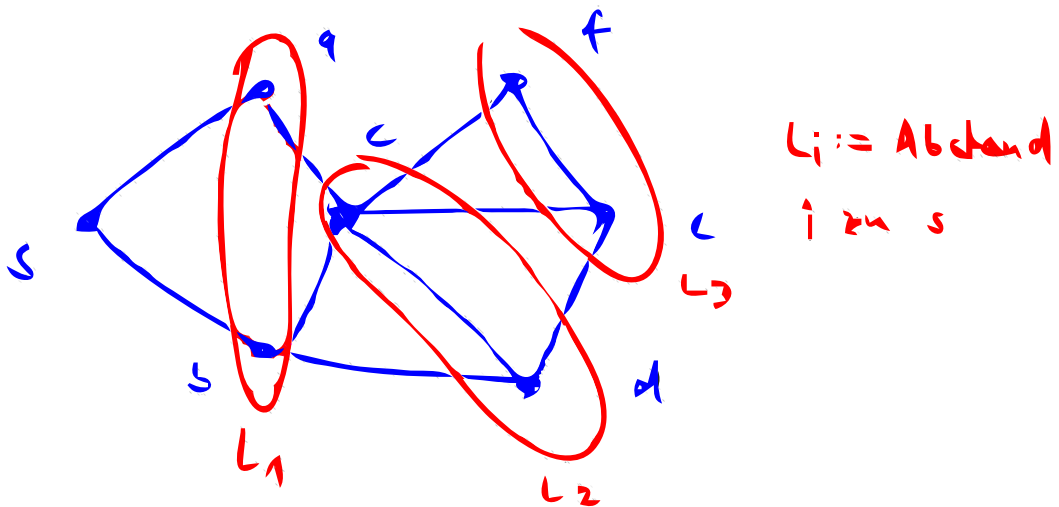
- Baum: Ungerichtet, kreisfrei, zusammenhängend



1.3.3 Graphtraversierung

Breitensuche (BFS)

- Idee
 - Beginne am Startknoten s
 - Durchforste Graph "schichtweise" (erst Abstand 1 zu s , dann Abstand 2, usw.)
- Wichtige Datenstruktur: Schlange (FIFO)
- BFS kann in $\mathcal{O}(n + m)$ Zeit durchgeführt werden
- Eventuell hoher Speicherbedarf
- Mit BFS findet man alle kürzesten Pfade ausgehend von s

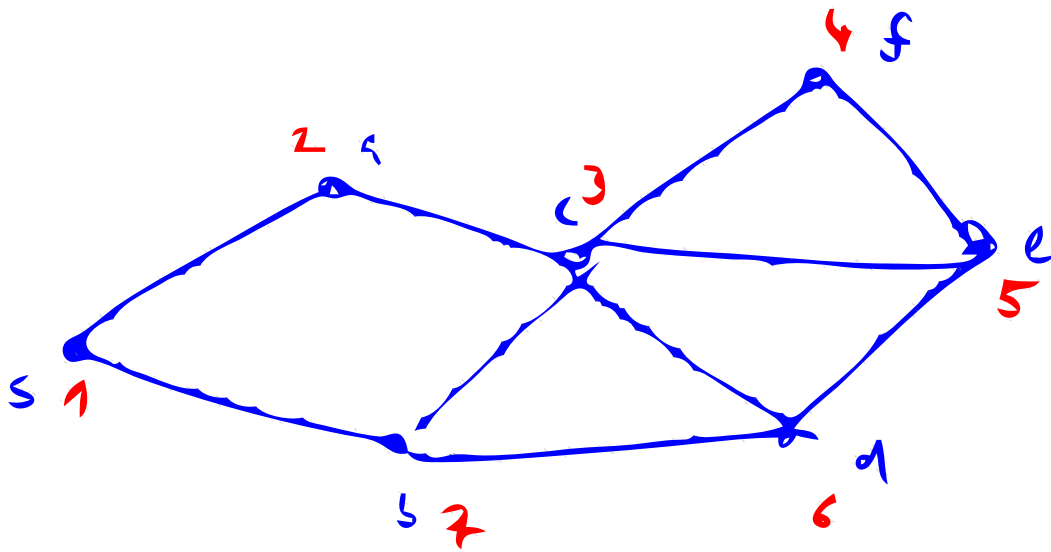


Tiefensuche (DFS)

Algorithm 2: DFS

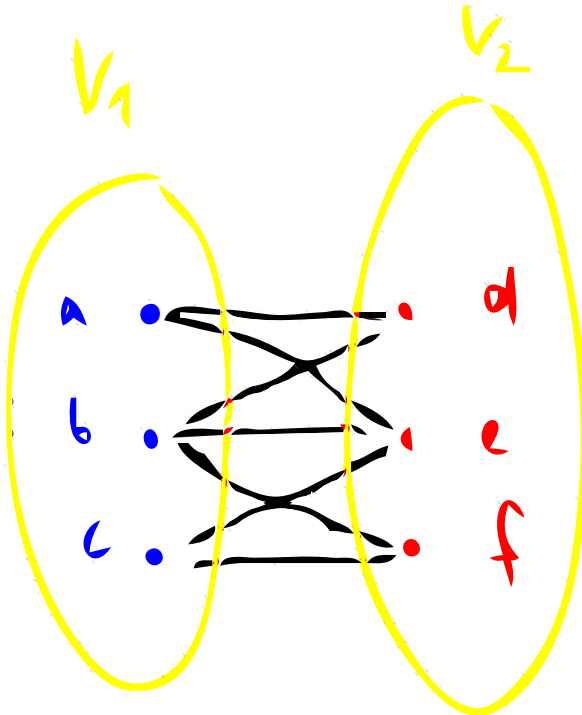
Input: Startknoten u

```
1  $R \leftarrow \emptyset$ 
2 Markiere  $u$  als besucht
3  $R \leftarrow R \cup \{u\}$ 
4 foreach  $\{u, v\} \in E$  do
5   if  $v$  nicht besucht then
6     | DFS( $v$ )
7   end
8 end
9 return  $R$ 
```

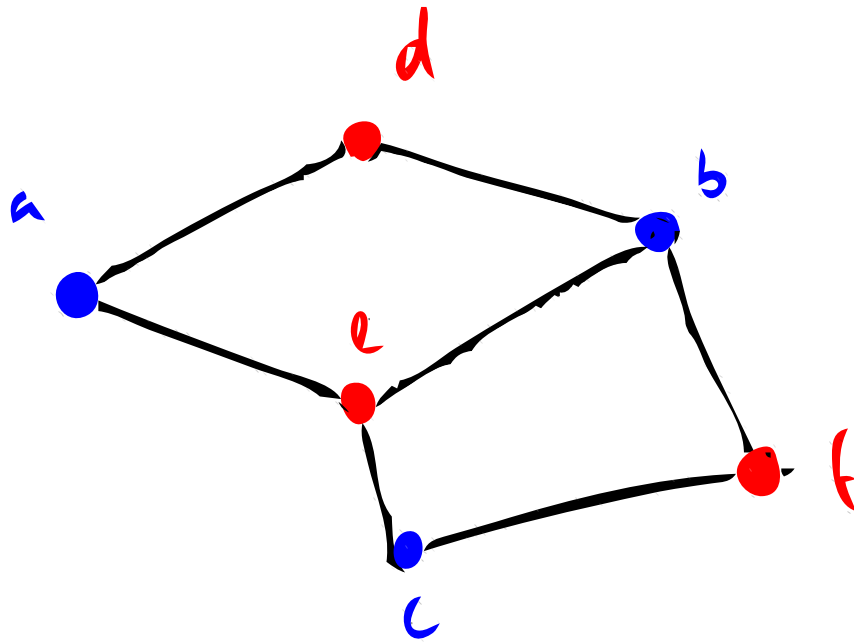


- DFS kann in $\mathcal{O}(n + m)$ Zeit durchgeführt werden.
- DFS findet in der Regel keine kürzesten Wege.
- Anwendung z.B. beim Finden von Zusammenhangskomponenten.

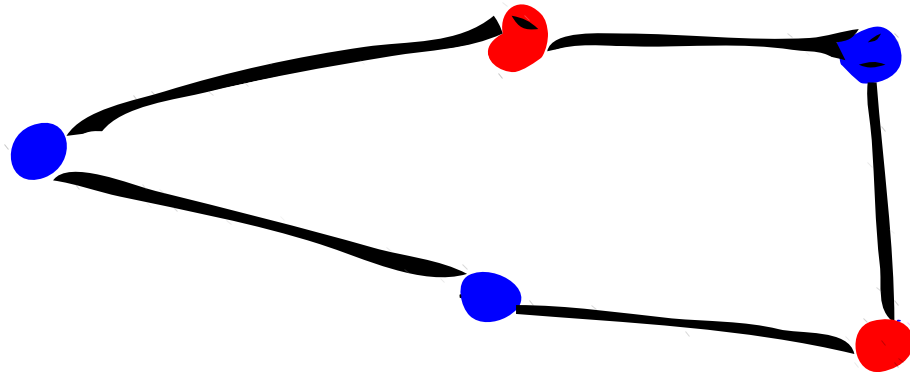
1.4 Bipartite Graphen



- Ein Graph $G = (V, E)$ ist **bipartit**, falls $V = V_1 \cup V_2$ mit $V_1 \cap V_2 = \emptyset$ und $E \subseteq V_1 \times V_2$.
- Äquivalent:
 - G ist zweifärbbar

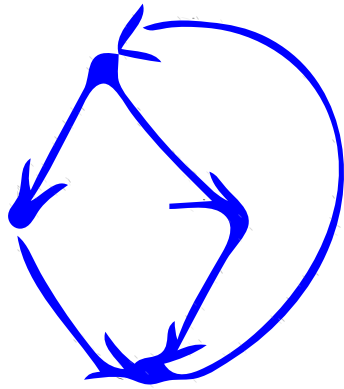


- G hat keinen Kreis ungerader Länge

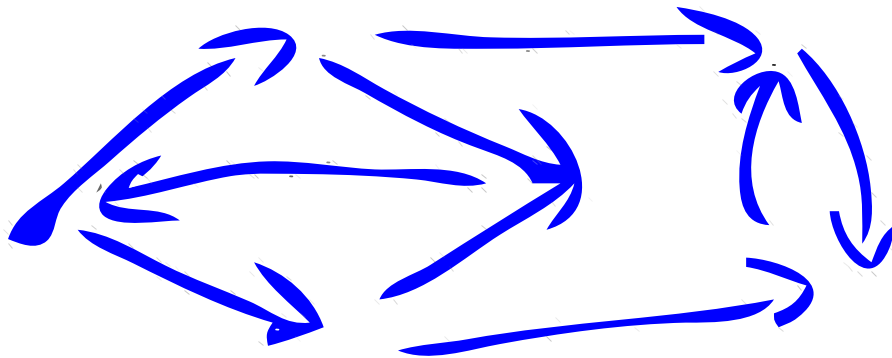


1.4.1 Starker Zusammenhang

- Ein gerichteter Graph heißt **stark zusammenhängend**, falls jedes Knotenpaar **wechselseitig** durch jeweils mind. einen gerichteten Pfad verbunden ist.
- Es kann in $\mathcal{O}(n + m)$ Zeit festgestellt werden, ob ein Graph $G = (V, E)$ stark zusammenhängend ist.
- Beispiel
 - Stark zusammenhängend



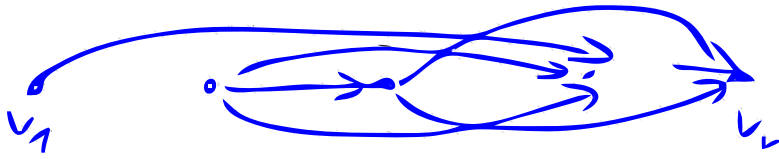
– Nicht stark zusammenhängend



1.4.2 DAG's & topologische Sortierungen

- Sei $G = (V, E)$ ein gerichteter Graph. Eine **topologische Sortierung** ist eine totale Ordnung v_1, v_2, \dots, v_n mit Knoten aus V , so dass für jede Kante $(v_i, v_j) \in E$ gilt: $i < j$.
- DAG: "directed acyclic graph": gerichteter, azyklische Graph
- Beispiel:

Task-Scheduling



$v_i \hat{=}$ Kurs in Stadium
 (v_i, v_j) heißt: Kurs v_i muss beenden sein, bevor
 v_j belegt werden kann...

- G ist gerichtet azyklisch $\iff G$ hat top. Sortierung
- Eine topologische Sortierung eines Graphen G , falls existierend, kann in $\mathcal{O}(n + m)$ Zeit gefunden werden. Beweisidee:
 1. Finde $v \in V$ ohne Eingangskante
 2. Setze v an Spitze der Sortierung
 3. Lösche v
 4. Finde Sortierung von " $G - v$ " rekursiv und setze diese hinter v

2 Greedyalgorithmen

2.1 Interval scheduling

- Eingabe: Intervalle (Jobs) mit Startzeiten s_i und Endzeiten f_i , $1 \leq i \leq n$.
- Aufgabe: Finde größtmögliche Menge nichtüberlappender Intervalle.
- Greedy-Strategie: Nimm Job mit frühestmöglicher Endzeit
- Dieser Algorithmus liefert immer eine optimale Lösung mit Laufzeit $\mathcal{O}(n \log n)$.

2.2 Interval Partitioning

- Eingabe: Intervalle (Jobs) mit Startzeiten s_i und Endzeiten f_i , $1 \leq i \leq n$.
- Aufgabe: Finde kleinstmögliche Menge von "Zeitstrahlen", so dass alle Jobs, auf diese verteilt, nicht überlappen.
- Die **Tiefe** einer Intervallmenge ist die maximale Zahl überlappender Intervalle.
- Greedy-IP liefert immer eine optimale Lösung mit Laufzeit $\mathcal{O}(n \log n)$.

Algorithm 3: Interval Partitioning

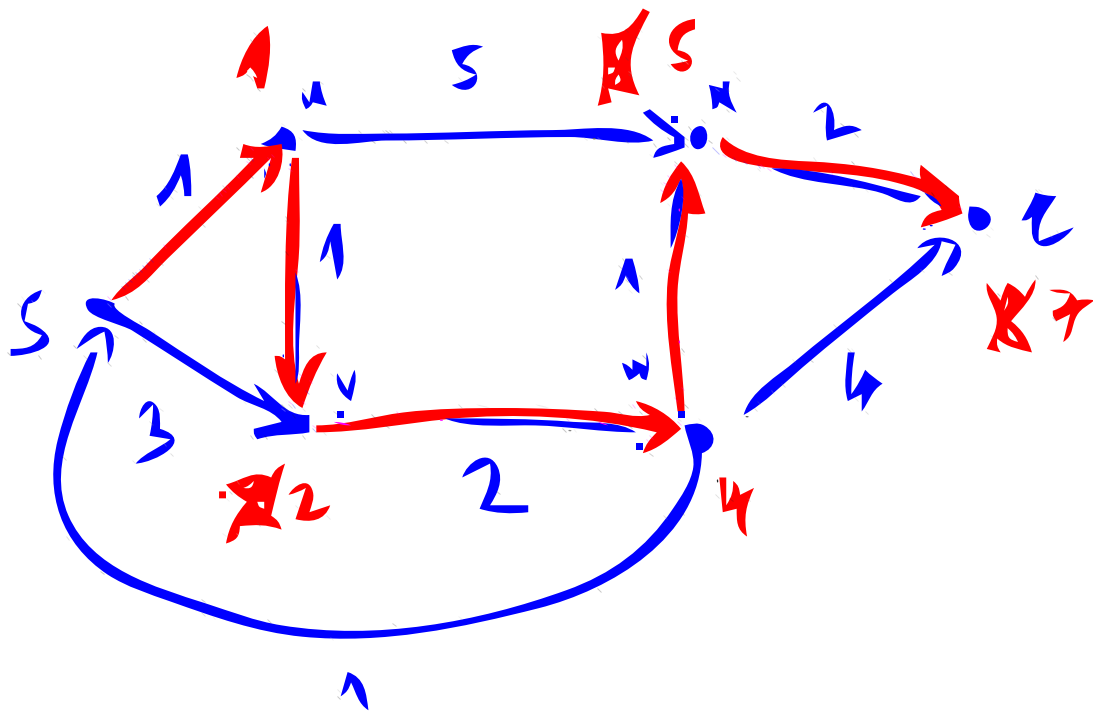
```
1 Sortiere Jobs nach aufsteigenden Startzeiten, d.h.  $s_1 \leq s_2 \leq \dots \leq s_n$ 
2  $d \leftarrow 0$ 
3 for  $r \leftarrow 1$  to  $n$  do
4   if Job  $j_r$  "passt auf Zeitstrahl"  $k \in \{1, \dots, d\}$  then
5     | Job  $j_r$  wird Zeitstrahl  $k$  zugeordnet
6   end
7   else
8     | öffne neuen Zeitstrahl  $d + 1$ 
9     | ordne Job  $j_r$  Zeitstrahl  $d + 1$  zu
10    |  $d \leftarrow d + 1$ 
11  end
12 end
```

2.3 Verspätungsminimierung

- Eingabe: Jobs $j, 1 \leq j \leq n$, mit Zeitdauer t_j und "Frist" d_j .
- Aufgabe: Finde Ausführungsreihenfolge der Jobs, so dass **maximale Verspätung** minimiert wird, d.h. minimiere $L := \max_j l_j$, wobei $l_j := \max\{0, f_j - d_j\}$, und f_j die Beendigungszeit von j in dieser Ausführungsreihenfolge ist.
- Greedy-Strategie: Führe Jobs gemäß steigender Frist d_j aus.
- Der Algorithmus liefert immer eine optimale Lösung mit Laufzeit $\mathcal{O}(n \log n)$.

2.4 Kürzeste Wege in Graphen

- Eingabe: Gerichteter Graph $G = (V, E)$ mit Längeangaben $l_e \geq 0$ für jede Kante $e \in E$, Startknoten s und Zielknoten t .
- Aufgabe: Finde kürzesten Pfad (Summe der Kantenlängen) von s nach t .
- Greedy-Ansatz
 - Starte mit $S := \{s\}$.
 - Vergrößere S schrittweise um je einen Knoten.
 - Für jeden Knoten in S ist kürzester Pfad entdeckt.
 - Kandidaten für Hinzunahme zu S sind Knoten mit mindestens einem Nachbarn in S . Erweiterung von S immer von Knoten aus, der geringste Distanz zu s hat (unter den noch nicht betrachteten Knoten).
- Mittels des Algorithmus von Dijkstra lassen sich alle kürzesten Pfade ausgehend von s in $\mathcal{O}(m \log n)$ Schritten mittels eines Priority Queue berechnen.



2.5 Minimale Spannbäume

- Eingabe: Zusammenhängender ungerichteter Graph G mit beliebigen Kantengewichten.
- Aufgabe: Finde einen Baum in G , der alle Knoten von G enthält und bei dem die Summe der Kantengewichte minimal ist.
- Berühmte Greedy-Algorithmen:
 - Kruskal: Wähle "billigste" Kante, die keinen Kreis erzeugt.
 - Prim: Erzeuge Baum ausgehend von einem Startknoten durch Erweiterung um billigste Kante.
- Beide genannten Algorithmen können das MST-Problem in $\mathcal{O}(m \log n)$ Schritten lösen.

2.6 Kodierung

- Eingabe: Zeichenkette T über endlichem Alphabet $\Sigma = \{c_1, \dots, c_n\}$, für jeden Buchstaben c_i eine "relative Häufigkeit" $f(c_i) \geq 0$, wobei $\sum_1^n f(c_i) = 1$.
- Aufgabe: Kodiere T über Binäralphabet $\{0, 1\}$, sodass der entstehende Code minimale Länge hat.
- Eine Kodierung $\gamma : \Sigma \rightarrow \{0, 1\}^+$ heißt **Präfix-Code** bzw. **präfixfrei**, falls es keine zwei Buchstaben $a, b \in \Sigma$ gibt, so dass $\gamma(a)$ ein Präfix von $\gamma(b)$ ist.

2.6.1 Problemformulierung

- **Modifizierte Aufgabenstellung:** Finde eine präfixfreie Kodierung *rightsquigarrow* Finde vollständigen Binärbaum, dessen Blätter mit de Elementen aus Σ beschriftet sind (1:1), so dass die Kosten des Baums T

$$\text{cost}(T) = \sum_{i=1}^n f(c_i) \cdot (\text{"Tiefe von } c_i \text{ im Baum"})$$

minimal sind.

2.6.2 Huffman-Algorithmus

Algorithm 4: Huffman-Algorithmus

```
1 if |  $\Sigma$  | = 2 then
2   | kodiere einen Buchstaben mit 0, den anderen mit 1
3 end
4 else
5   |  $a, b :=$  "Buchstaben mit kleinster Häufigkeit"
6   | lösche  $a$  und  $b$  aus  $\Sigma$  und füge neuen Buchstaben  $\overline{ab}$  hinzu
7   |  $f(\overline{ab}) := f(a) + f(b)$ 
8   | Konstruiere rekursiv präfixfreien Code mit Baum  $T'$ 
9   | Ersetze in  $T'$  das Blatt  $\overline{ab}$  durch den Unterbaum
10 end
```

Der Algorithmus von Huffman findet in $\mathcal{O}(n \log n)$ Zeit eine optimale präfixfreie Kodierung.

3 Divide-&-Conquer

3.1 Grundprinzip

- Zerlege Problem in mehrere (meist zwei) Teilprobleme.
- Löse jeden Teil rekursiv.
- Kombiniere die Lösungen der Teilprobleme zu Gesamtlösung.

3.2 Rekursionsungleichungen

$$T(n) = aT(\lceil \frac{n}{b} \rceil) + \mathcal{O}(n^d)$$

3.2.1 Master Theorem

- Sei $T(n) = aT(\lceil \frac{n}{b} \rceil) + \mathcal{O}(n^d)$ für Konstanten $a > 0, b > 1$ und $d \geq 0$, dann

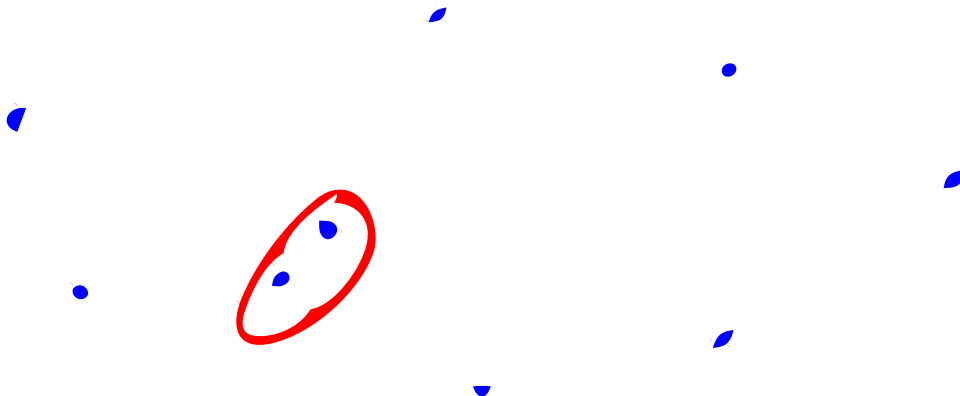
$$T(n) = \begin{cases} \mathcal{O}(n^d) & , \text{ falls } d > \log_b a \\ \mathcal{O}(n^d \log n) & , \text{ falls } d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & , \text{ falls } d < \log_b a \end{cases}$$

3.2.2 Zählen von Inversionen

- Eingabe: Eine feste Ordnung a_1, a_2, \dots, a_n der Zahlen von 1 bis n .
- Aufgabe: Bestimme die Anzahl von Inversionen im Vergleich zur Ordnung $1, 2, \dots, n$, wobei eine **Inversion** ein Paar $(i, j), 1 \leq i < j \leq n$, mit $a_i > a_j$ ist.
- Lösungsansatz:
 1. Teile Eingabe in 2 Hälften
 2. Zähle Inversionen je Hälfte
 3. Gesamtzahl der Inversionen = Addition der beiden Werte plus "Inversionen zwischen den Hälften".
- Die Zahl der Inversionen einer Folge von n verschiedenen Zahlen aus $\{1, \dots, n\}$ lässt sich in $\mathcal{O}(n \log n)$ ermitteln. (Brute-force-Ansatz bräuhete $\mathcal{O}(n^2)$ Zeit).

3.3 Closest Pair

- Eingabe: n Punkte in der Euklidischen Ebene.
- Aufgabe: Finde Punktepaar mit geringstem Abstand.
- Beispiel ($\mathcal{O}(n^2)$):



- Einfacher Spezialfall: Alle Punkte auf einer Geraden ($\mathcal{O}(n \log n)$)

