Effiziente Algorithmen
Zusammenfassung

Thomas Mohr

Contents

List of Algorithms

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:

$$
\begin{array}{cc}
M = \{X, Y, Z\} & W = \{A, B, C\} \\
X : A < B < C & A : Y < X < Z \\
Y : B < A < C & B : X < Y < Z \\
Z : A < B < C & C : X < Y < Z
\end{array}
$$

  - 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

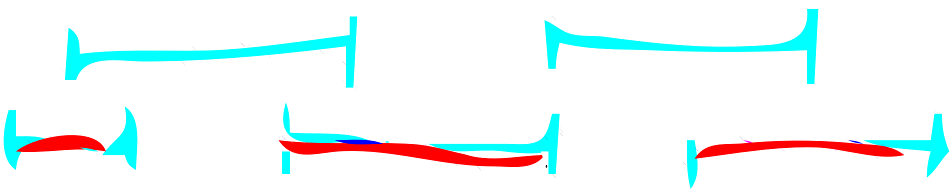
Algorithm 1: Propose-&-Reject

```
alle m ∈ M und alle w ∈ W "frei"
while ∃m ∈ M : m ist frei und ∃w ∈ W der m noch keinen Antrag gemacht hat
 do
    w ← erste noch "unbeantragte" Frau in m's Präferenzfolge
    if w ist frei then
        (m, w) wird Paar
        m ← "verlobt"
        w ← "verlobt"
    end
    else if w zieht m ihrem aktuellen Verlobten m′ vor then
        (m, w) wird Paar
        m ← "verlobt"
        w ← "verlobt"
        m′ ← "frei"
    end
    else
        w lehnt m ab
    end
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.

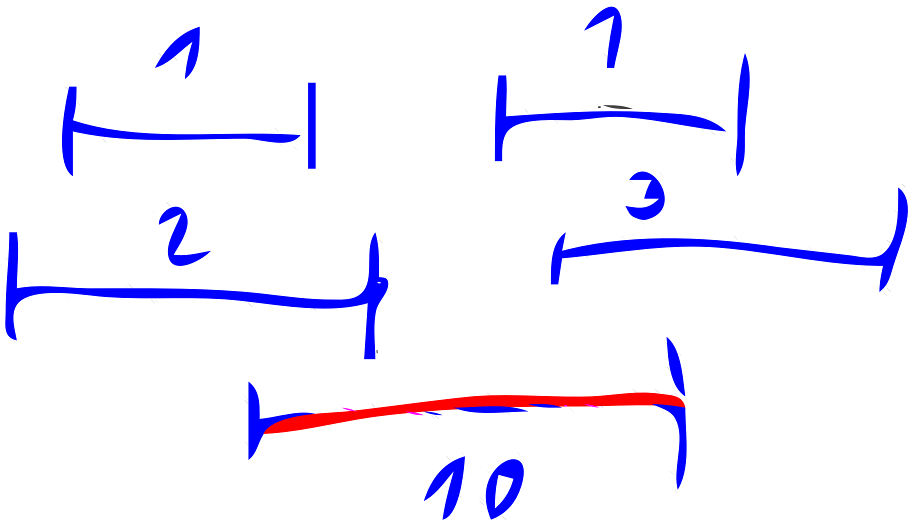
1.1.2 5 respräsentative Probleme

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

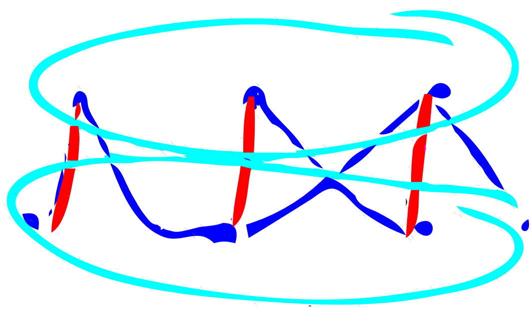


2. Gewichtetes Interval Scheduling

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

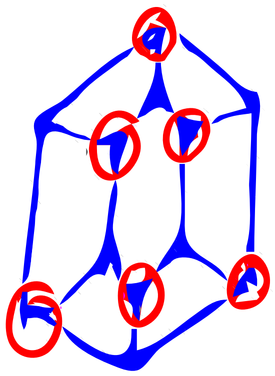


3. Bipartites Matching
   - Eingabe: Bipartiter Graph
   - Aufgabe: Finde größtmögliche "unabhängige" (keine gemeinsamen Endpunkte) Kantenmenge
   - Beispiel für Netzwerkflüsse:



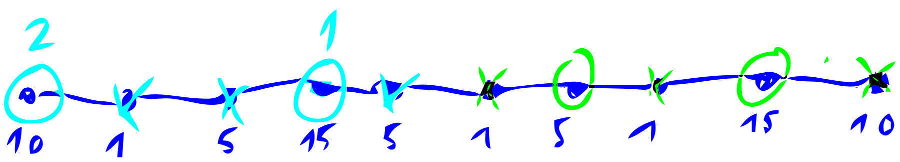
4. Independent set
   - Eingabe: Ungerichteter Graph
   - Aufgabe: Finde größtmögliche "unabhängige" (paarweise nicht benachbarte) Knotenmenge

- 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 Pnkte macht
    - Beispiel (PSPACE-vollständig):



B=25: Spieler 1 hat Gewinnstrategie

- Sp. 1
- Sp. 2

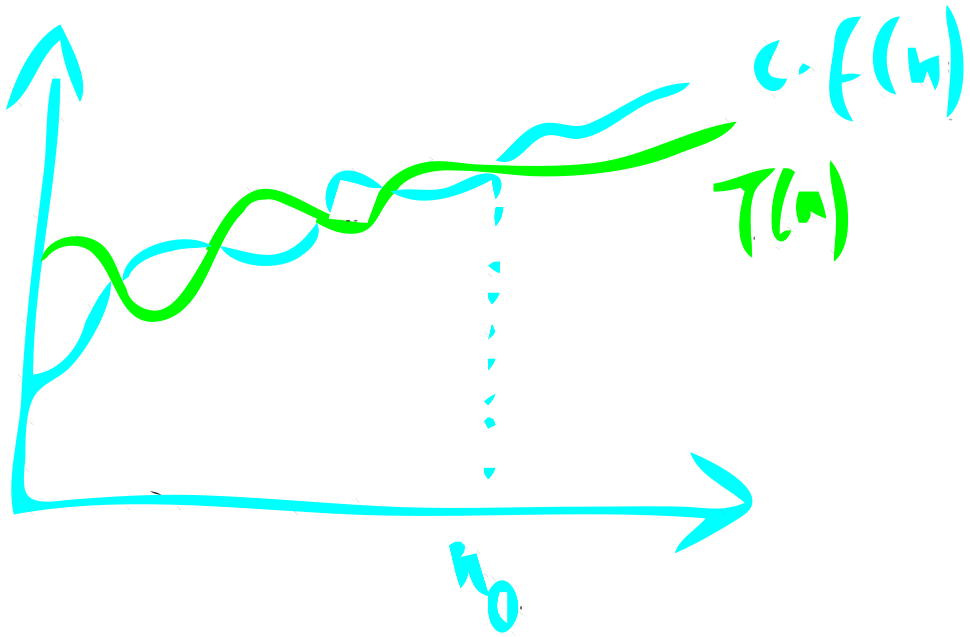
1.2 Zentrale Konzepte & Konventionen

- 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.

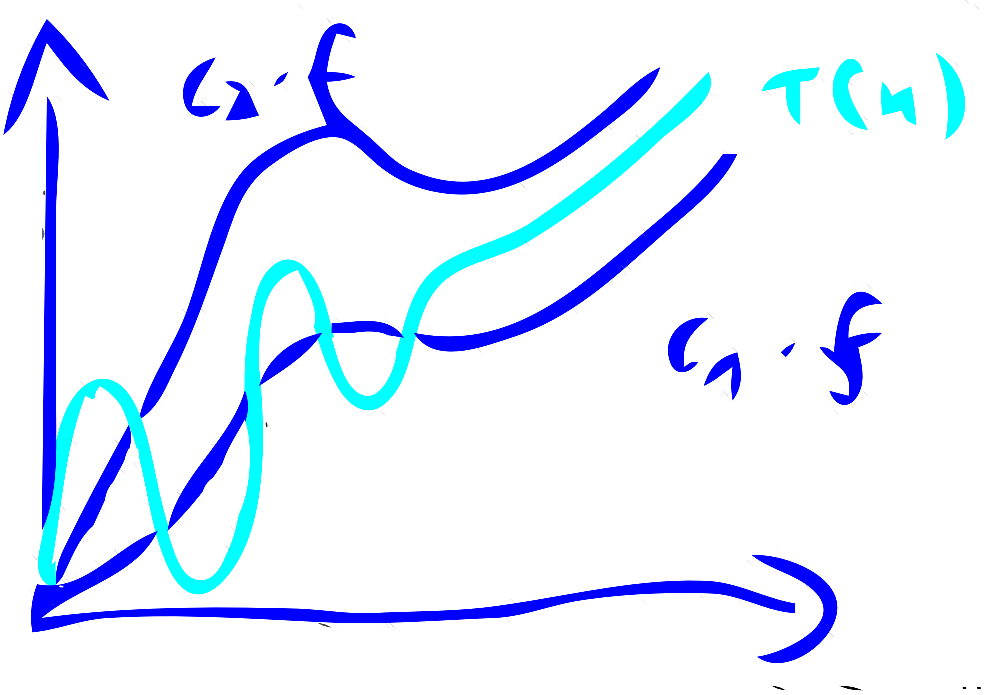
- Man beachte: **Worst-Case Analyse**

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

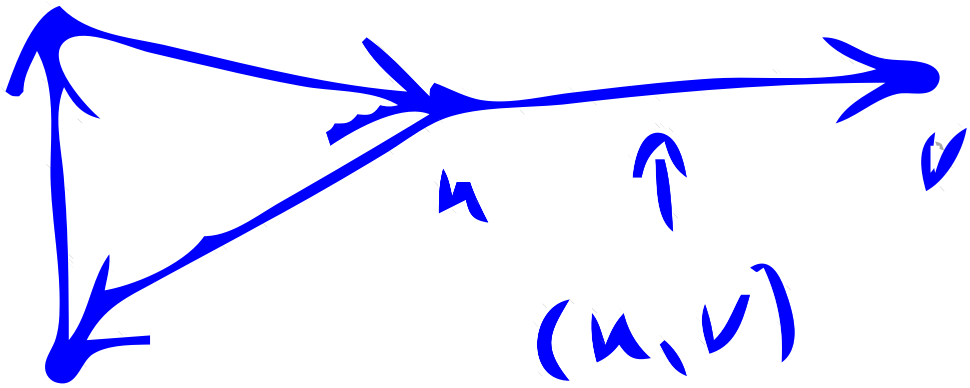
- $T(n) = \mathcal{O}(f(n))$ nicht schneller
- $T(n) = \Omega(f(n))$ nicht langsamer
- $T(n) = \Theta(f(n))$ genauso schnell
- $T(n) = o(f(n))$ echt langsamer
- $T(n) = \omega(f(n))$ echt schneller

als f.

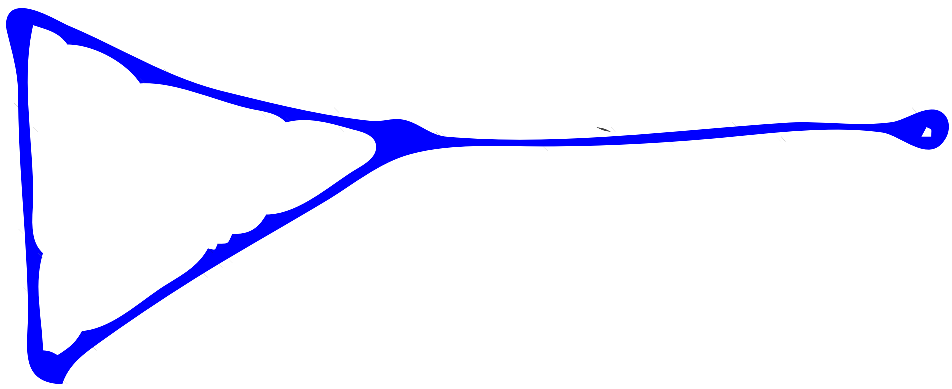
1.3 Graphen

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

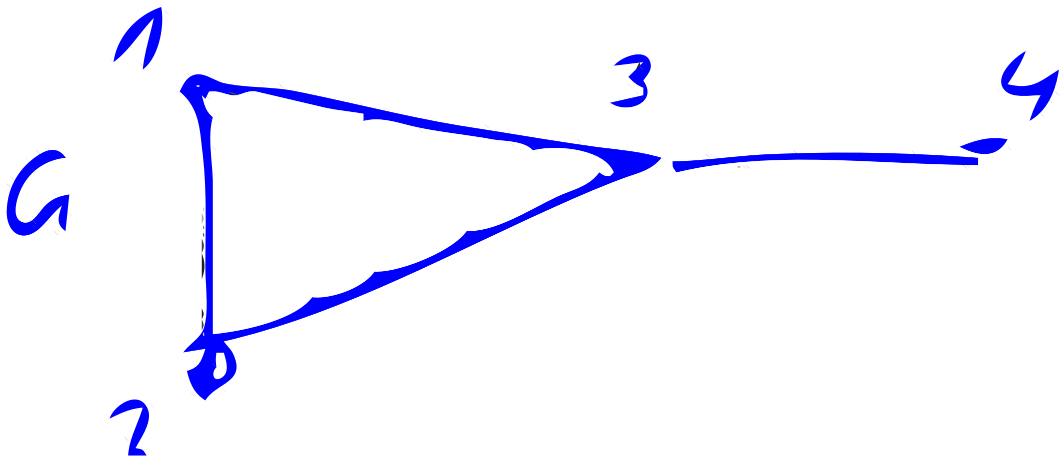
- Gerichteter Graph: $e \in E$ mit $e = (u, v), u, v \in V$ (geordnetes Paar)



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



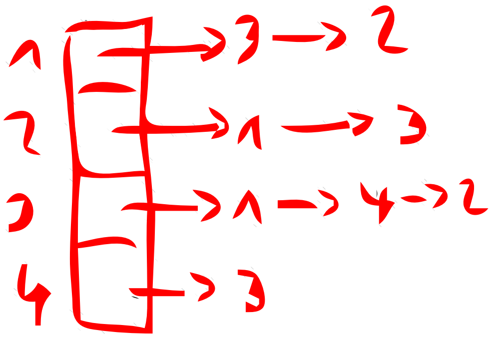
1.3.1 Repräsentation



- 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”)

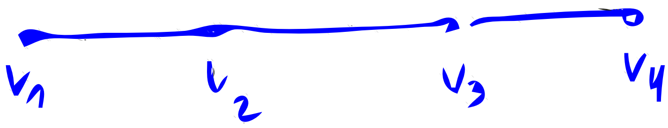
$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & & 1 & 1 & \\ 2 & 1 & & 1 & \\ 3 & 1 & 1 & & 1 \\ 4 & & & 1 & \\ \hline \end{array}$$

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



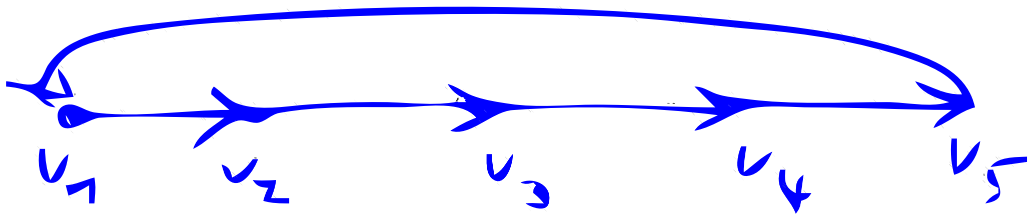
1.3.2 Bekannte Begriffe

- Pfad: Folge von Knoten, aufeinanderfolgende sind benachbart

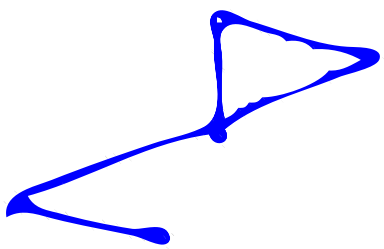


v_1, v_2, v_3, v_4

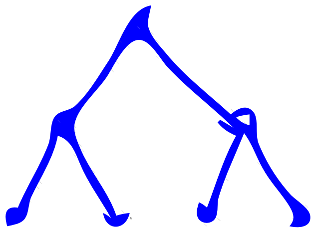
- Kreis: Pfad $v_1, \ldots, v_l$ mit $v_1 = v_l$



- Ungerichteter zusammenhängender Graph: Zwischen allen Knotenpaaren existiert ein Pfad



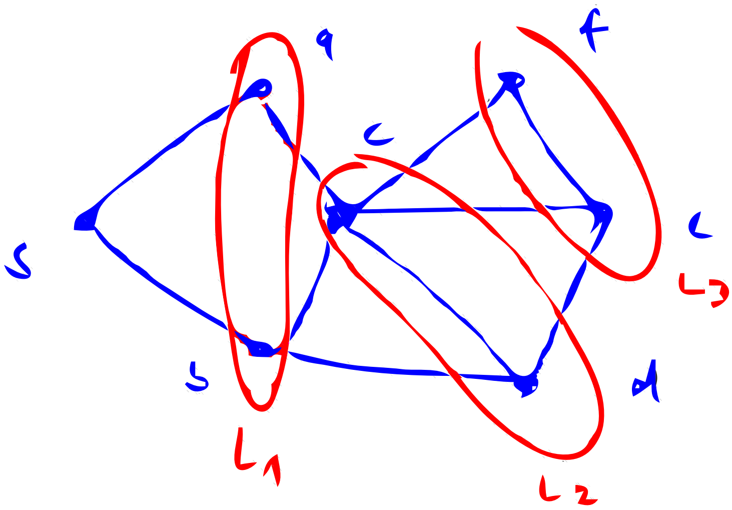
- Baum: Ungerichtet, kreisfrei, zusammenhängend



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



$L_i :=$ Abstand i zu s

Tiefensuche (DFS)

Algorithm 2: DFS

Input: Startknoten u

$R \leftarrow \emptyset$

Markiere u als besucht

$R \leftarrow R \cup \{u\}$

foreach $\{u, v\} \in E$ **do**

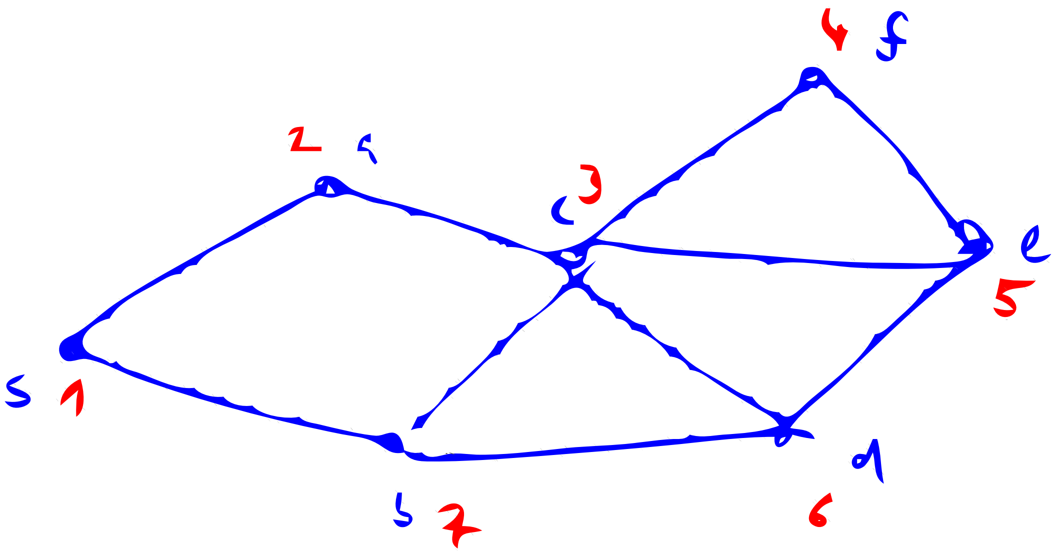
  if v *nicht besucht* **then**

    `DFS`(v)

  end

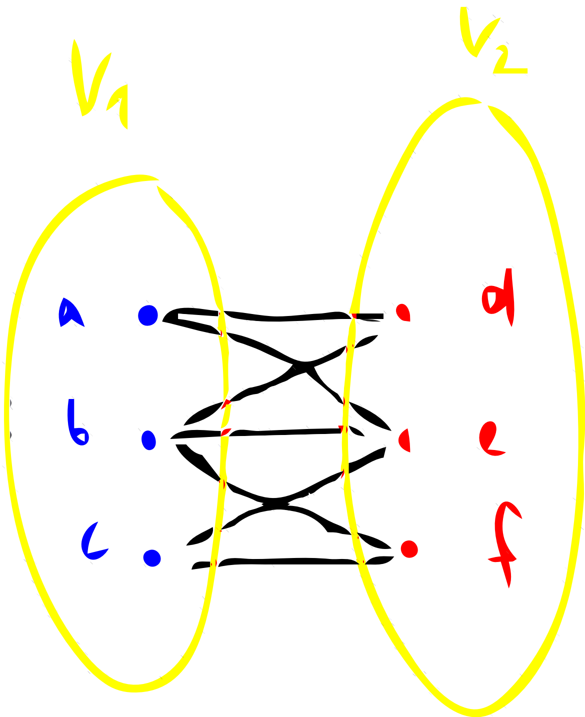
end

return R

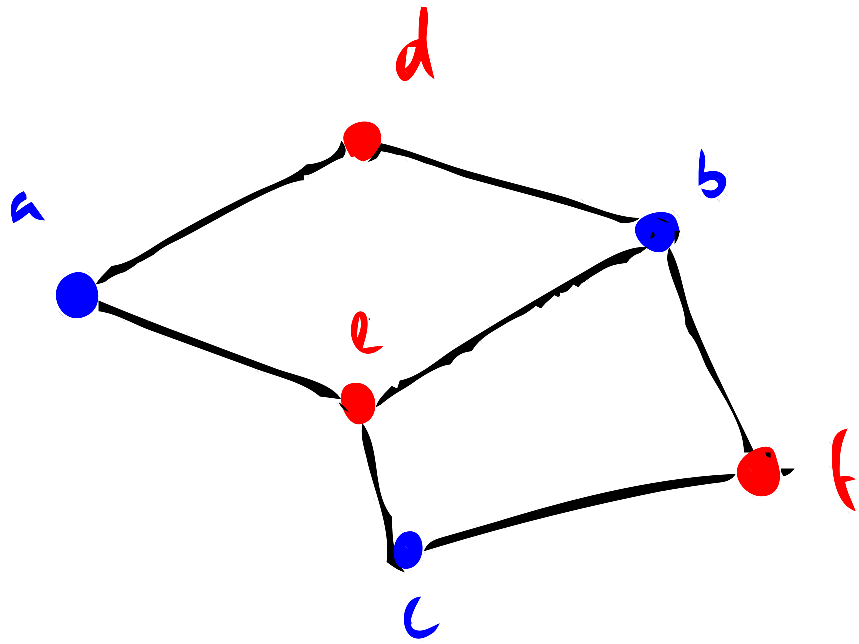


- DFS kann in $\mathcal{O}(n + m)$ Zeit durchgeführt werden.
- DFS findet in der Regel keine kürzesten Wege.
- Anwendung z.B. beim Finden von Zusammenhangskomponenten.

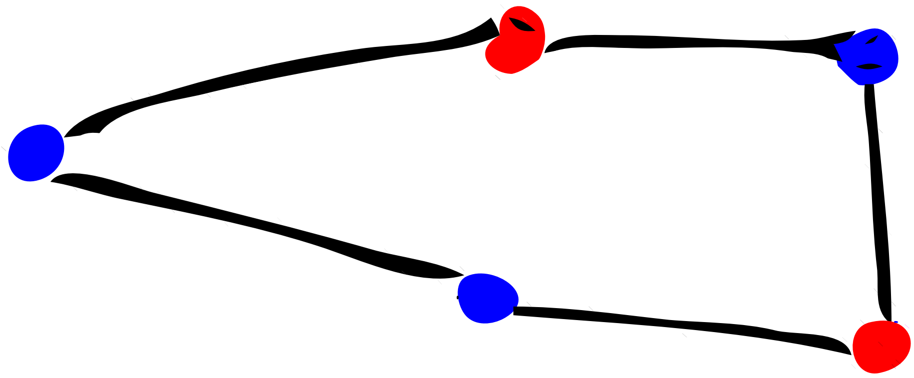
1.4 Bipartite Graphen



- 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

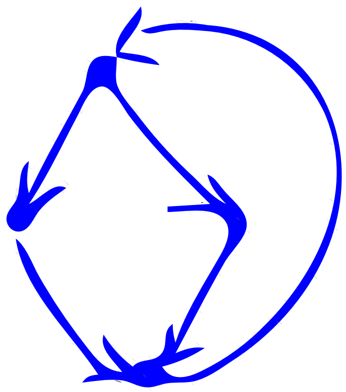


- G hat keinen Kreis ungerader Länge

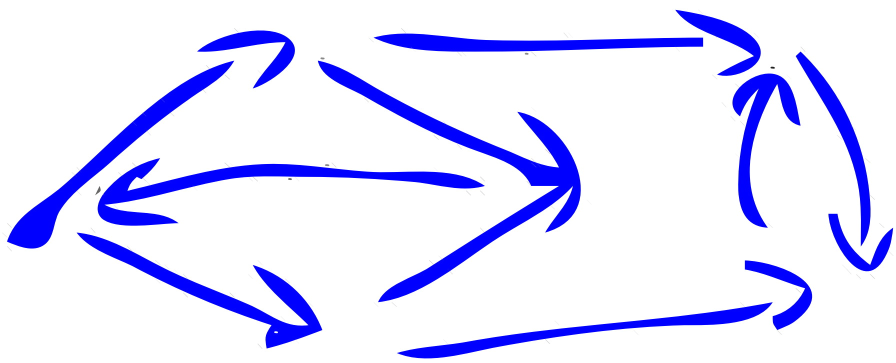


1.4.1 Starker Zusammenhang

- Ein gerichteter Graph heißt **stark zusammenhängend**, falls jedes Knotenpaar **wechselseitig** durch jeweils mind. einen gerichtetetn 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



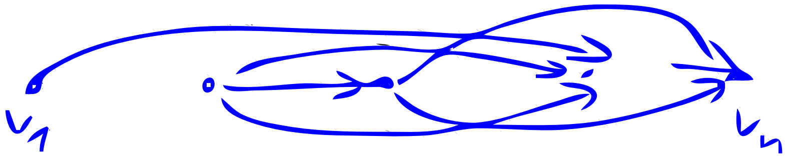
- Nicht stark zusammenhängend



1.4.2 DAG's & topologische Sortierungen

- Sei $G = (V, E)$ ein gerichteter Graph. Eine **topologische Sortierung** ist eine totale Ordnung $v_1, v_2, \ldots, v_n$ 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



$v_i \mathrel{\hat{=}}$ Kurs im Studium

(v_i, v_j) heißt: Kurs v_i muss bestanden 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

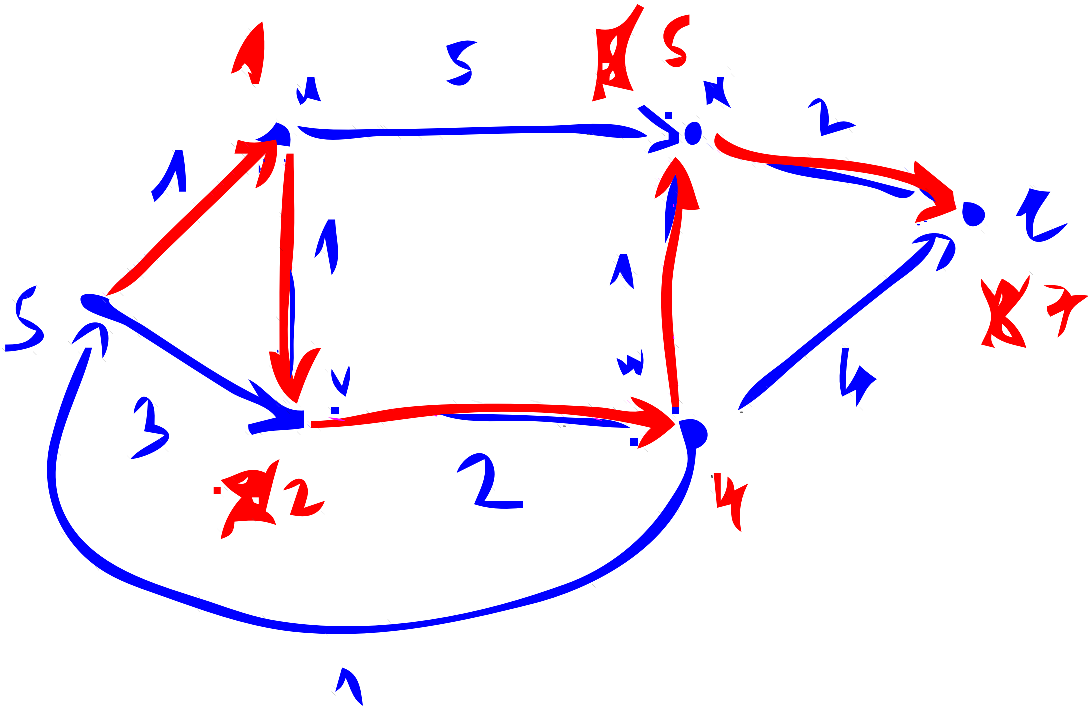
Sortiere Jobs nach aufsteigenden Startzeiten, d.h. $s_1 \leq s_2 \leq \ldots \leq s_n$
$d \leftarrow 0$
for $r \leftarrow 1$ **to** n **do**
  if *Job* j_r *"passt auf Zeitstrahl"* $k \in \{1, \ldots, d\}$ **then**
    Job j_r wird Zeitstrahl k zugeordnet
  end
  else
    öffne neuen Zeitstrahl $d + 1$
    ordne Job j_r Zeitstrahl $d + 1$ zu
    $d \leftarrow d + 1$
  end
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 Knote 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 $\Sigma = \{c_1, \ldots, c_n\}$, 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 $\gamma : \Sigma \to \{0, 1\}^+$ 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

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

  minimal sind.

2.6.2 Huffmann-Algorithmus

Algorithm 4: Huffmann-Algorithmus

```
if | Σ |= 2 then
    kodiere einen Buchstaben mit 0, den anderen mit 1
end
else
    a, b := "Buchstaben mit kleinster Häufigkeit"
    lösche a und b aus Σ und füge neuen Buchstaben ab‾ hinzu
    f(ab‾) := f(a) + f(b)
    Konstruiere rekursiv präfixfreien Code mit Baum T'
    Ersetze in T' das Blatt ab‾ durch den Unterbaum
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

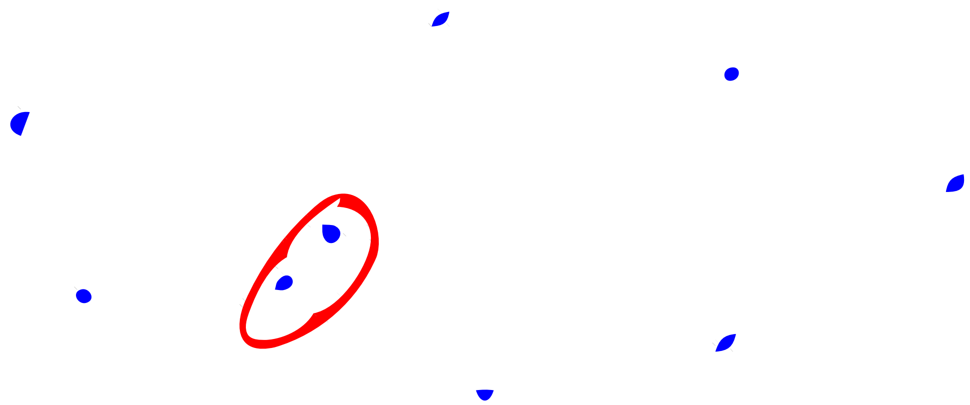
$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{, falls } d > \log_b a \\ \mathcal{O}(n^d \log n) & \text{, falls } d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & \text{, falls } d < \log_b a \end{cases}$$

3.2.2 Zählen von Inversionen

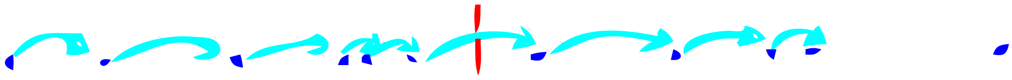
- Eingabe: Eine feste Odnung $a_1, a_2, \ldots, a_n$ der Zahlen von 1 bis n.
- Aufgabe: Bestimme die Anzahl von Inversionen im Vergleich zur Ordnung $1, 2, \ldots, n$, wobei eine **Inversion** ein Paar $(i, j), 1 \leq i < j \leq n$, mit $a_i > a_j$ ist.
- Lösungsansatz:
  1. Teile Eingabe in 2 Hälften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



3.3.1 Algorithmus

- Teile "Punktwolke" in zwei gleich große Hälften
- Paar ist entweder in einer der beiden Hälften, oder je ein Punkt in einer der beiden Hälften.
- Nur schmaler grenzstreifen ist zu untersuchen
- Jeder Punkt im Grenzstreifen ist nur mit konstant vielen anderen innerhalb des Grenzstreifens zu vergleichen.
- Closest Pair lässt sich in $\mathcal{O}(n \log n)$ Zeit lösen.

3.4 Matrixmultiplikation

- Eingabe: Zwei $n \times n$ Matrizen A und B
- Aufgabe: Berechne $C = A \cdot B$
- Schulmethode: "Zeile mal Spalte" $\rightsquigarrow \mathcal{O}(n^3)$ Elementaroperationen
- D&C-Idee: Partitionierung in vier quadratische Teilmatrizen