Entwurfsmethoden

Methoden für den Entwurf von Algorithmen

- **Greedy Algorithmen:**
  - Löse ein einfaches Optimierungsproblem durch eine Folge „vernünftiger“ Entscheidungen.
  - Eine getroffene Entscheidung wird nie zurückgenommen.
- **Dynamisches Programmieren:**
  - Zerlege ein zu lösendes Problem P in eine Hierarchie einfacherer Teilprobleme P_i.
  - Die Hierarchie wird durch den Lösungsgraphen G_P modelliert.
    - Die Knoten entsprechen den Teilproblemen.
    - Füge eine Kante (P_i, P_j) von Teilproblem P_i nach Teilproblem P_j ein, wenn die Lösung von P_i für die Lösung von P_j benötigt wird.
    - Der Graph G_P muss azyklisch ist, darf also keine Kreise besitzen.
- **Divide & Conquer** ist ein Spezialfall des dynamischen Programmierens: Der Lösungsgraph G_P ist ein Baum.
- Die **lineare Programmierung** wird für Optimierungsprobleme eingesetzt, die Lösungen in rationalen Zahlen erlauben.

Greedy-Algorithmen

- Führe eine Reihe von Entscheidungen mit Hilfe einer Heuristik aus.
- Keine Entscheidung darf jemals zurückgenommen werden. (Web)

- Dijkstras Algorithmus:
  - Die Entscheidung: Lege einen kürzesten Weg für einen neuen Knoten fest.
  - Die Heuristik: Der neue Knoten wird durch einen kürzesten S-Weg erreicht.
- DJP und Kruskals Algorithmen:
  - Die Entscheidung: Setze eine neue Kante ein.
  - Die Heuristik: Wähle eine kürzeste kreuzende Kante.

Die Fragestellung:

Welche Probleme können mit Greedy-Algorithmen gelöst werden?

Intervall-Scheduling

Intervall Scheduling auf einem Prozessor

Eingabe:
- Eine Teilmenge der Aufgaben $1, \ldots, n$ ist auf einem einzigen Prozessor auszuführen.
- Aufgabe i besitzt eine Startzeit s_i, eine Terminierungszeit t_i und wird somit durch das Zeitintervall $[s_i, t_i]$ beschrieben.

Aufgabe:
- Führe eine größtmögliche Anzahl nicht-kollidierender Aufgabe aus.
  - Zwei Aufgaben *kollidieren*, wenn sich ihre Zeitintervalle überlappen.

Intervall Scheduling: Anwendungen

Anwendungen:

- Führe eine größtmögliche Anzahl von Veranstaltungen in einem einzigen Veranstaltungsraum aus.
- Erledige möglichst viele Druckaufträge auf einem einzigen Drucker fristgerecht nach der jeweils angekündigten Anlieferung.

Heuristiken

- Führe Aufgaben nach aufsteigender Startzeit aus.

  Keine gute Idee: Wenn eine frühe Aufgabe spät terminiert, werden viele andere Aufgaben ausgeschlossen.
- Führe kurze Aufgaben zuerst aus.

  Keine gute Idee: Eine kurze Aufgabe kann zwei kollisionsfreie lange Aufgaben blockieren.
- Führe Aufgaben nach aufsteigender Terminierungszeit aus.

  Startzeiten werden ignoriert!

Frühe Terminierungszeiten zuerst

(1) Sortiere alle Aufgaben gemäß aufsteigender Terminierungszeit.
(2) Solange noch Aufgaben vorhanden sind, wiederhole

führe die Aufgabe i mit kleinster Terminierungszeit aus, entferne diese Aufgabe und alle Aufgaben j mit $s_j \leq t_i$.
// Alle kollidierenden Aufgaben werden entfernt.

- Angenommen, eine optimale Ausführungsfolge führt Aufgabe j als erste Aufgabe aus, obwohl j später als i terminiert. Also $t_i < t_j$.
- Wir erhalten eine mindestens ebenso gute Ausführungssequenz, wenn wir j durch i ersetzen!
- Es gibt also eine optimale Ausführungssequenz, in der eine Aufgabe mit kleinster Terminierungszeit als erste Aufgabe ausgeführt wird.
  - Wiederhole das Argument für die zweite, dritte .... Aufgabe.

Beispiel multiple choice

Gegeben seien folgende Aufgaben
(jeweils mit Start- und Terminierungszeit):
A_1 (2,5) A_2 (1,3) A_3 (4,8) A_4 (5,6).
Welche Aufgaben wählt der Algorithmus aus?

Auflösung: A_2 & A_4

Zusammenfassung

- Das Prinzip „Frühe Terminierungszeiten zuerst“ funktioniert:
  Eine größtmögliche Anzahl kollisionsfreier Aufgaben wird für n Aufgaben in Zeit $O(n \log_2 n)$ berechnet.
- Im gewichteten Intervall Scheduling erhalten Aufgaben einen Wert:
  - Führe eine Teilmenge von Aufgaben mit größtem Gesamtwert aus.
  - Greedy-Algorithmen sind nicht bekannt. Aber dynamisches Programmieren wird helfen.

Minimiere die Verspätung

Scheduling mit minimaler Verspätung

Eingabe:
- n Aufgaben $1, \ldots, n$ sind gegeben: Aufgabe i hat die Frist f_i und die Laufzeit l_i.
- **Alle** Aufgaben sind auf einem einzigen Prozessor auszuführen.
- Wenn Aufgabe i zum Zeitpunkt t_i fertig ist, dann ist $v_i = t_i - f_i$ ihre Verspätung.

Aufgabe:
- Bestimme eine Ausführungssequenz, so dass die maximale Verspätung

$$\max_{1 \leq i \leq n} v_i$$

kleinstmöglich ist.

Heuristiken

- Sortiere Aufgaben nach ihrem „Slack“ $f_i - l_i$ und führe Aufgaben mit kleinstem Slack zuerst aus.
  - Zwei Aufgaben mit $f_1 = 3, l_1 = 1$ und $f_2 = T + 1, l_2 = T$.
  - Zuerst wird die zweite Aufgabe ausgeführt und beansprucht das Intervall $[0, T[$.
  - Die erste Aufgabe beansprucht das Intervall $[T, T + 1[$ mit Verspätung $v_1 = T + 1 - 3 = T - 2$.
  - Besser (für $T > 2$): Zuerst Aufgabe 1 mit Intervall $[0, 1[$ und dann Aufgabe 2 mit Intervall $[1, T + 1[$.
    Verspätungen: $v_1 = 1 - 3 = -2$ und $v_2 = T + 1 - (T + 1) = 0$.
- Sortiere Aufgaben nach aufsteigender Frist und führe Aufgaben nach dem Motto „Frühe Fristen zuerst“ aus.

Sollte auch nicht funktionieren, da wir Laufzeiten ignorieren.

Frühe Fristen zuerst

(1) Sortiere alle Aufgaben gemäß aufsteigender Frist.
(2) Solange noch Aufgaben vorhanden sind, wiederhole
führe die Aufgabe i mit kleinster Frist aus und entferne diese Aufgabe.

Das Prinzip „Frühe Fristen zuerst“ funktioniert:
Eine Ausführungsfolge mit minimaler Verspätung wird für n Aufgaben in Zeit $O(n \cdot \log_2 n)$ berechnet.

Laufzeit $O(n \cdot \log_2 n)$ ist offensichtlich, denn das Sortieren dominiert.

Aber warum ist der Algorithmus korrekt?

Inversionen

opt sei eine optimale Ausführungssequenz.

- Das Paar (i, j) ist eine Inversion, wenn i in **opt** nach j ausgeführt wird, aber $f_i < f_j$ gilt.
  - Die Anzahl der Inversionen misst den „Abstand“ zwischen der optimalen Lösung und der Ausführung gemäß „Frühe Fristen zuerst“.
- Beseitige Inversionen, ohne die größte Verspätung zu erhöhen.
  - Jede Ausführungssequenz hat höchstens $\binom{n}{2}$ Inversionen.
  - Wenn das Ziel erreicht ist, dann erhalten wir „kurz über lang“ eine optimale Ausführung ohne Inversionen.
  - Aber es gibt möglicherweise viele Ausführungssequenzen ohne Inversionen!

Ausführung ohne Inversionen

Alle Ausführungen ohne Inversionen besitzen dieselbe maximale Verspätung.

- Zwei Ausführungen ohne Inversionen unterscheiden sich nur in der Reihenfolge, mit der Aufgaben mit selber Frist f ausgeführt werden.
- Die größte Verspätung wird für die jeweils zuletzt ausgeführte Aufgabe mit Frist f erreicht.
- Aber die größten Verspätungen sind identisch, denn die jeweils zuletzt ausgeführten Aufgaben werden zum selben Zeitpunkt erledigt!

Eine Inversion weniger

Es gibt Aufgaben i und j mit $f_i < f_j$, so dass Aufgabe i

direkt nach

Aufgabe j in **opt** ausgeführt wird.

Solche Aufgaben existieren, denn wenn je zwei aufeinanderfolgende Aufgaben inversionsfrei sind, dann hat die Ausführungssequenz keine Inversionen.

Was wir zeigen müssen:

Aufgabe i werde direkt nach Aufgabe j ausgeführt und es gelte $f_i < f_j$. Wenn wir die beiden Aufgaben vertauschen, dann steigt die größte Verspätung nicht an.

Verifikation des Reduktionsschritts

$f_i < f_j$, aber Aufgabe i wird direkt nach Aufgabe j erledigt.

- Wenn wir Aufgabe i vor Aufgabe j erledigen, dann nimmt die Verspätung von Aufgabe i ab.
- Aber die nach Vertauschung später ausgeführte Aufgabe j wird zu dem späteren Zeitpunkt T fertiggestellt.
  - Die neue Verspätung $v_j = T - f_j$ von Aufgabe j ist kleiner als die alte Verspätung $v_i = T - f_i$ von Aufgabe i, denn $f_i < f_j$.
  - Die maximale Verspätung kann also höchstens abnehmen.

Huffman Codes

Huffman Codes

- code : $\Sigma \to \{0,1\}^*$ ist ein Präfix-Code, wenn kein Codewort code(a) ein Präfix eines anderen Codewortes code(b) ist.
  - Die Kodierung durch einen Präfix-Code kann durch das von links-nach-rechts Lesen des Codes dekodiert werden.
- Sei $T = a_1 \dots a_n$ ein Text mit Buchstaben $a_1, \dots, a_n \in \Sigma$. Dann ist

$$\text{code}(T) = \text{code}(a_1) \dots \text{code}(a_n)$$

die Kodierung von T mit der Kodierungslänge

$$\sum_{i=1}^{n} |code(a_i)|.$$

Ein Huffman-Code für T ist ein Präfix-Code minimaler Länge.

Beispiel demogr.

Ist $a \rightarrow$ 0, $b \rightarrow$ 1, $c \rightarrow$ 00, $d \rightarrow$ 01 ein Huffman-Code?

- (1) Ja.
- (2) Nein.
- (3) Keine Ahnung.

Auflösung: (2) Nein.

Die Aufgabenstellung

Berechne einen Huffman Code für einen gegebenen Text T.

- Sei $H(a)$ die Häufigkeit mit der Buchstabe a im Text vorkommt.
  - Dann ist
    $$\sum_{a \in \Sigma} H(a) \cdot |code(a)|$$
    die Kodierungslänge des Codes.
  - Wie wird ein Huffman-Code für T aussehen?
    - ⋆ Häufige Buchstaben erhalten kurze Codewörter,
    - ⋆ seltene Buchstaben erhalten lange Codewörter.
- Zwei zentrale Fragen:
  - Wie können wir Huffman-Codes veranschaulichen?
  - Wie kodiert ein Huffman-Code die beiden seltensten Buchstaben?

Ein Beispiel

Der Text

$$T = abbacaaca$$

ist zu verschlüsseln.

- Wir erhalten die Häufigkeiten

$$H(a) = 5, H(b) = 2 \text{ und } H(c) = 2.$$

- Wie sollte der Huffman-Code für T aussehen?
  - Der häufige Buchstabe a sollte durch ein Bit kodiert werden?!
    Zum Beispiel $\text{code}(a) = 0$.
  - Aufgrund der Präfixfreiheit beginnen alle anderen Codewörter mit dem Bit 1.
    - ⋆ Dann sollten wir $\text{code}(b) = 10$ und $\text{code}(c) = 11$ setzen.

Eine Veranschaulichung von Huffman-Codes

Huffman-Codes und Binärbäume

Zu jedem Huffman-Code gibt es einen Huffman-Baum B mit den folgenden Eigenschaften:

(a) alle inneren Knoten haben genau 2 Kinder,

(b) jede Kante ist mit einem Bit $b \in \{0, 1\}$ markiert und

(c) jedem Buchstaben $a \in \Sigma$ ist genau ein Blatt b_a zugewiesen:
Die Markierung des Weges von der Wurzel zum Blatt b_a stimmt mit $\text{code}(a)$ überein.

Gibt es stets einen Huffman Baum?

Annahme:

Ein Huffman-Code produziere nur Codewörter der Länge höchstens L.

- Erzeuge den vollständigen Binärbaum B_L der Tiefe L.
- Markiere Kanten zu einem linken Kind mit 0 und Kanten zu einem rechten Kind mit 1.

Aber B_L ist doch viel zu groß!

Huffman-Bäume

- Sei V die Menge aller Knoten von B_L, deren Wege (von der Wurzel aus) mit einem Codewort markiert sind.
- Ein Huffman-Code ist ein Präfix-Code: Für **keine** zwei Knoten $u, v \in V$ ist u ein Vorfahre von v.
  - Entferne alle Knoten aus B_L, die einen Knoten aus V als Vorfahren besitzen.
  - Der entstehende Baum ist der gewünschte Huffman-Baum.
- Wirklich? Haben denn alle inneren Knoten genau zwei Kinder?
  - Wenn der Knoten v nur ein Kind besitzt, dann ist der Code unnötig lang.
  - Wir erhalten einen kürzeren Code, indem wir v entfernen und den Vater von v mit dem Kind von v direkt verbinden.

Wo sind die beiden seltensten Buchstaben?

Die Kodierung der beiden seltensten Buchstaben

Sei der Text T vorgegeben. Dann gibt es einen Huffman-Code für T mit Huffman-Baum B, so dass

- B einen inneren Knoten v maximaler Tiefe besitzt und
- v zwei Buchstaben geringster Häufigkeit als Blätter hat.

Warum?

Die beiden seltesten Buchstaben sind Geschwisterblätter

- L sei die Länge des längsten Codewortes für den Huffman-Code.
- Der zugehörige Huffman-Baum hat zwei Geschwister-Blätter α und β der Tiefe L, denn innere Knoten haben genau zwei Kinder.
- Die beiden seltensten Buchstaben x und y werden Codeworte der Länge L erhalten, denn

$$\sum_{a \in \Sigma} H(a) \cdot |\text{code}(a)|$$

ist zu minimieren.

  - Verteile die Blattmarkierungen für Blätter der Tiefe L so um, dass x und y den Blätter α und β zugewiesen werden.
  - Die Kodierungslänge für den neuen Baum ist nicht angestiegen!

Der Algorithmus

(1) Füge alle Buchstaben $a \in \Sigma$ in einen anfänglich leeren Heap ein. Der Heap sei gemäß kleinstem Häufigkeitswert geordnet.
// Wir konstruieren einen Huffman-Baum B. Zu Anfang ist B ein
// Wald von Knoten v_a, wobei v_a dem Buchstaben a zugeordnet ist.

(2) Solange der Heap mindestens zwei Buchstaben enthält, wiederhole:

(2a) Entferne die beiden Buchstaben x und y geringster Häufigkeit.

(2b) Füge einen neuen Buchstaben z in den Heap ein und setze $H(z) = H(x) + H(y)$.

- Schaffe einen neuen Knoten v_z für z
- Mache v_x und v_y zu Kindern von v_z.
- Markiere die Kante $\{v_x, v_z\}$ mit 0 und die Kante $\{v_y, v_z\}$ mit 1.

Zusammenfassung

Eine schnelle Berechnung von Huffman-Codes

Der Algorithmus findet einen Huffman-Code für einen Text über einem Alphabet mit n Buchstaben. Die Laufzeit ist durch $O(n \log_2 n)$ beschränkt.

Warum? Es werden höchstens $4n$ Heap-Operationen durchgeführt:

- Am Anfang werden alle Buchstaben einmal eingefügt.
- Nach Entfernen von zwei Buchstaben und...
- Einfügen eines neuen Buchstabens...

...hat der Heap insgesamt einen Buchstaben verloren.

- Es gibt bessere Kodierverfahren als Präfix-Codes, zum Beispiel Lempel-Ziv Codes und arithmetische Codes.
- Aber Präfix-Codes gehören zu den mächtigsten Codes, für die ein bester Code effizient gefunden werden kann.

Huffmanbäume demogr.

Sei n die Länge des Textes T. Ein Huffmanbaum für T kann in Zeit $O(n \log n)$ berechnet werden. Ginge es auch in Zeit $\Theta(n)$?

- (1) Ja.
- (2) Nein.
- (3) Keine Ahnung.

Auflösung: (1) Ja, mit einfacher Integer-Prioritätswarteschlange.

Stabiles Matching

Dating

- $2n$ Personen sind auf der Suche nach dem perfekten Partner.
- Menge $A = \{1, \ldots, n\}$ von n Frauen
- Menge $B = \{n+1, \ldots, 2n\}$ von n Männern.
- Jede/r Frau/Mann hat eine Präferenzordnung über alle potenziellen Partner (des jeweils anderen Geschlechts)

- Die Präferenzordnung von Frau $i \in A$ ist eine Sortierung $\succ_i = (b_1^i, \ldots, b_n^i)$ aller Männer in absteigender Reihenfolge. Wenn i Mann $j \in B$ lieber mag als Mann $k \in B$, schreiben wir $j \succ_i k$.

- Präferenzordnung für Männer analog.

Match-Maker

- Wir suchen ein Matching $M \subseteq A \times B$.
- Jede/r Teilnehmer/in ist lieber gematcht als nicht gematcht.
- Es gibt viele mögliche Matchings! Welches sollen wir wählen?
- Wenn Leute in ihren Beziehungen unglücklich sind, dann brennen sie u.U. mit anderen Partnern durch...

Blockierendes Paar

Ein Matching M hat ein blockierendes Paar $(i, j) \in A \times B$ wenn

- $(i, j) \notin M$ und
- $(i, k) \in M$ und $j \succ_i k$, oder i ungematcht in M; und
- $(j, \ell) \in M$ und $i \succ_j \ell$, oder j ungematcht in M.

Ein stabiles Matching ist ein Matching ohne ein blockierendes Paar.

Stabiles Matching

Eingabe:
- $2n$ Agenten in Mengen A und B mit $|A| = |B| = n$
- Jeder Agent $i \in A$ hat eine vollständige Präferenzordnung $\succ_i$ über alle Agenten in B
- Jeder Agent $j \in B$ hat eine vollständige Präferenzordnung $\succ_j$ über alle Agenten in A

Aufgabe:
- Bestimme eine stabiles Matching $M \subseteq A \times B$ ohne blockierendes Paar.

Beispiel multiple choice

Betrachte die folgende Instanz mit $A = \{1, 2, 3\}$ und $B = \{4, 5, 6\}$ sowie

$\succ_1$: $(4, 5, 6)$ $\quad$ $\succ_4$: $(1, 2, 3)$
$\succ_2$: $(4, 6, 5)$ $\quad$ $\succ_5$: $(3, 1, 2)$
$\succ_3$: $(4, 5, 6)$ $\quad$ $\succ_6$: $(3, 2, 1)$

und das Matching $M = \{(1, 4), (2, 5), (3, 6)\}$.

Was gilt?

- (1) $(1, 5)$ ist kein blockierendes Paar.
- (2) $(2, 6)$ ist ein blockierendes Paar.
- (3) $(3, 5)$ ist ein blockierendes Paar.
- (4) M ist ein stabiles Matching.

Auflösung: (1) und (3)

Stabile Matchings

Gibt es überhaupt immer ein stabiles Matching?
Und wenn ja, wie kann man es (schnell) berechnen?

- Wir nutzen einen Greedy-Algorithmus.
- Eine Seite A ist aktiv und macht Anträge.
- Die andere Seite B ist passiv und nimmt Anträge (vorläufig) an oder lehnt sie (endgültig) ab.
- Der Algorithmus berechnet immer ein stabiles Matching!
  (und zeigt damit, dass so ein Matching immer existiert).

Deferred-Acceptance Algorithmus

(1) Setze $s_i = 1$ für alle $i \in A$.

(2) Solange ein Agent $i \in A$ existiert, der nicht vorläufig gematcht ist:

(2a) i macht einen Antrag an den Agenten an Position s_i in seiner Sortierung (sei dies Agent $j \in B$).

(2b) Wenn j keinen vorläufigen Partner hat, wird i der vorläufige Partner von j.

(2c) Sonst sei $k \in A$ der vorläufige Partner von j. Wenn $i \succ_j k$, dann

- j nimmt den Antrag von i an und lehnt den Antrag von k ab.
- i wird der neue vorläufige Partner von j.
- Erhöhe s_k um 1.

(2d) Andernfalls:

- j behält weiter den Antrag von k und lehnt den Antrag von i ab.
- k bleibt der vorläufige Partner von j.
- Erhöhe s_i um 1.

(3) Setze $M = \{$ alle vorläufig gematchten Paare $\}$

Laufzeit

Der Algorithmus benötigt Laufzeit $O(n^2)$:

- Es werden höchstens n^2 viele Anträge gemacht.
- Beim Antrag müssen wir die Güte von $i, k \in A$ für $j \in B$ vergleichen.
- Wir berechnen am Anfang einmal für jede Präferenzordnung eines Agenten in B die Umkehrfunktion.
- Dann gelingt jeder Güte-Vergleich in Zeit $O(1)$, also in Zeit $O(n^2)$ insgesamt.
- Die anfängliche Berechnung einer Umkehrfunktion benötigt Laufzeit $O(n)$, es gibt n Präferenzordnungen für Agenten in B.

Korrektheit

Für die Korrektheit beobachten wir, dass im Laufe des Algorithmus...

(1) ... der vorläufige Partner von $j \in B$ nur besser wird.

(2) ... der vorläufige Partner von $i \in A$ nur schlechter wird.

Das ist beides klar: $j \in B$ tauscht den vorläufigen Partner nur aus wenn ein besserer Antrag kommt. Dagegen geht $i \in A$ in der Präferenz der Anträge immer nur runter.

Daraus folgt: M ist ein stabiles Matching.

Warum?

Korrektheit

M ist ein stabiles Matching.

- Wir nehmen an M ist nicht stabil.
- Sei (i, j) ein blockierendes Paar.
- Also ist i ungematcht oder hat einen schlechteren Partner in M als j.
- i hat irgendwann einen Antrag an j gemacht und wurde abgelehnt.
- Zu der Zeit hatte j einen besseren vorläufigen Partner als i (am Ende in M sogar noch besser wegen (1)).
- Aber dann will j doch nicht mit i durchbrennen!
- (i, j) ist kein blockierendes Paar, Widerspruch.

Damit gilt auch: Kein Agent bleibt ungematcht.
(und in Zeile (2a) des Algorithmus kommt es nie zu $s_i \geq n + 1$).

Divide & Conquer

Divide & Conquer

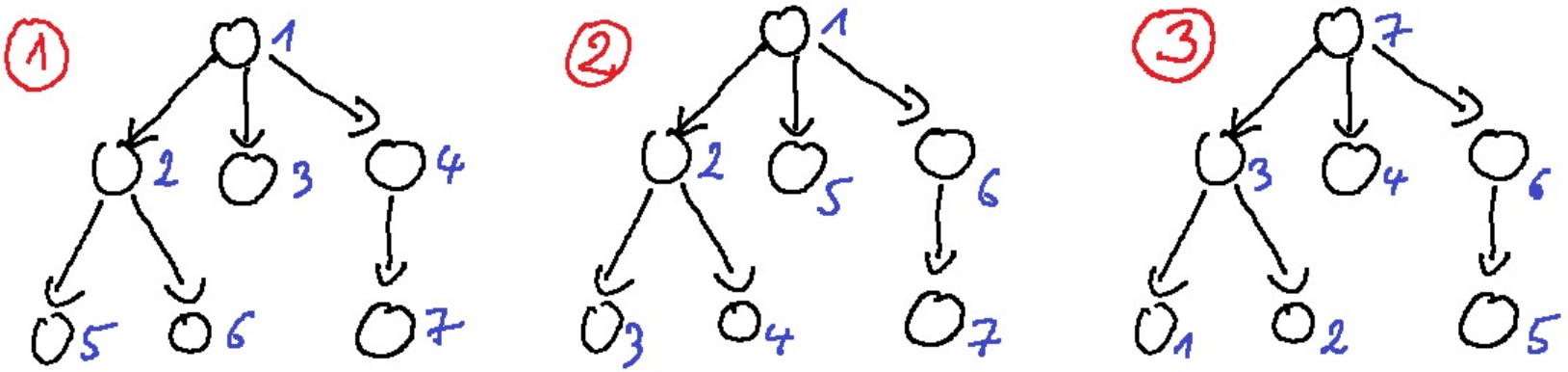
Wir kennen bereits erfolgreiche Divide & Conquer Algorithmen:

- Binärsuche,
- Präorder und Postorder Traversierung von Bäumen,
- Tiefensuche.

Jedesmal wird das Ausgangsproblem rekursiv gelöst, nachdem ähnliche, aber kleinere Probleme gelöst wurden.

Postorder demogr.

Welches ist eine Postorder-Traversierung?



Auflösung: (3).

Die schnelle Multiplikation natürlicher Zahlen

Die Multiplikation natürlicher Zahlen

Eingabe: Zwei n-Bit Zahlen

$$x = \sum_{i=0}^{n-1} a_i 2^i \quad \text{und} \quad y = \sum_{i=0}^{n-1} b_i 2^i$$

sind gegeben.

Aufgabe: Bestimme die Binärdarstellung des Produkts $x \cdot y$.

Die Schulmethode

Addiere die Zahlen $x \cdot 2^i$ für jedes i mit $b_i = 1$.

Der Aufwand in **Bitoperationen:**

- Bis zu n Zahlen mit bis zu $2n - 1$ Bits müssen addiert werden.
- Für eine Addition werden bis zu $2n$ Bitoperationen benötigt.

Insgesamt werden $\Theta(n^2)$ Bit-Operationen benötigt.

Ein erster rekursiver Algorithmus

Wir nehmen an, dass n eine Zweierpotenz ist.
(Wenn nicht, dann fülle mit führenden Nullen auf.) Dann ist

$$x = x_2 2^{n/2} + x_1 \text{ und } y = y_2 2^{n/2} + y_1,$$

wobei x_1, x_2, y_1 und y_2 Zahlen mit jeweils $n/2$ Bits sind.

Berechne die vier Produkte

$$x_1 \cdot y_1, x_1 \cdot y_2, x_2 \cdot y_1 \text{ und } x_2 \cdot y_2$$

rekursiv und addiere

$$x \cdot y = x_1 \cdot y_1 + 2^{n/2} \cdot (x_1 \cdot y_2 + x_2 \cdot y_1) + 2^n x_2 \cdot y_2.$$

Und welche Laufzeit erzielen wir?

Laufzeit des ersten rekursiven Algorithmus

- Sei $\mathrm{Bit}(n)$ die Anzahl der benötigten Bit-Operationen.
  Wir erhalten die Rekursionsgleichung

$$\mathrm{Bit}(n) = 4 \cdot \mathrm{Bit}(n/2) + c \cdot n.$$

  Der additive Term $c \cdot n$ zählt die Anzahl der nicht-rekursiven Bit-Operationen, also die drei Shifts und Additionen.
- Was ist die Lösung dieser Rekursionsgleichung ?
  Stimmen wir ab . . .

Lösung der Rekursionsgleichung demogr.

$$\text{Bit}(n) = 4 \cdot \text{Bit}(n/2) + c \cdot n.$$

hat die Lösung

- (1) $\Theta(n \cdot \log^4 n)$
- (2) $\Theta(n \cdot n^{4/3})$
- (3) $\Theta(n^2)$
- (4) $\Theta(n^2 \log n)$
- (5) $\Theta(n^4)$

Auflösung: (3) $\Theta(n^2)$, nicht schneller als die Schulmethode!

Wenn wir aber mit drei statt vier rekursiven Multiplikationen auskommen würden, dann wären wir im Geschäft.

Drei statt vier Multiplikationen

- Berechne $z = (x_1 + x_2)(y_1 + y_2)$ sowie
- $z_1 = x_1 \cdot y_1$ und $z_2 = x_2 \cdot y_2$. Dann ist

$$\begin{aligned} x \cdot y &= x_1 \cdot y_1 + 2^{n/2} \cdot (x_1 \cdot y_2 + x_2 \cdot y_1) + 2^n \cdot x_2 \cdot y_2 \\ &= z_1 + 2^{n/2} \cdot (x_1 \cdot y_2 + x_2 \cdot y_1) + 2^n \cdot z_2 \\ &= z_1 + 2^{n/2} \cdot (z - z_1 - z_2) + 2^n \cdot z_2. \end{aligned}$$

Es genügen 3 (vorher 4) Multiplikationen, aber wir haben jetzt 6 (vorher 3) Shifts und Additionen. Wir erhalten die neue Rekursion

$$\text{Bit}(n) = 3 \cdot \text{Bit}(n/2) + d \cdot n.$$

- Unser Algorithmus benötigt nur $\text{Bit}(n) = \Theta(n^{\log_2 3}) = O(n^{1.59})$ Bitoperationen.
- „Weltrekord“ ist $O(n \cdot \log_2 n)$ Bitoperationen (Web).

Matrizenmultiplikation, schneller als in der Schule

Die Multiplikation von Matrizen

Eingabe: Zwei $n \times n$-Matrizen A und B

Aufgabe: Bestimme die Produktmatrix $C = A \cdot B$.

Die Schulmethode

- Multipliziere die $i$te Zeile von A mit der $j$ten Spalte von B, um den Eintrag $C[i, j]$ zu berechnen: $O(n)$ arithmetische Operationen.
- Insgesamt sind $\Theta(n^3)$ arithmetische Operationen erforderlich, da C n^2 Einträge hat.

Wir nehmen wieder an, dass n eine Zweierpotenz ist. Wir zerlegen A und B in $n/2 \times n/2$ Teilmatrizen:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Um C zu berechnen, benötigt die Schulmethode 8 Multiplikationen.

7 Matrizenmultiplikationen reichen

$$
\begin{aligned}
M_1 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \\
M_2 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
M_3 &= (A_{1,1} - A_{2,1}) \cdot (B_{1,1} + B_{1,2}) \\
M_4 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
M_5 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
M_6 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
M_7 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1},
\end{aligned}
$$

denn für die vier Teilmatrizen $C_{i,j}$ gilt

$$
\begin{aligned}
C_{1,1} &= M_1 + M_2 - M_4 + M_6 \\
C_{1,2} &= M_4 + M_5 \\
C_{1,3} &= M_6 + M_7 \\
C_{1,4} &= M_2 - M_3 + M_5 - M_7.
\end{aligned}
$$

Zusammenfassung

Unser Algorithmus benötige $A(n)$ arithmetischen Operationen, um zwei $n \times n$ Matrizen zu multiplizieren. Dann ist

$$A(n) = 7A(n/2) + c \cdot n^2,$$

wenn $c \cdot n^2$ die Additionen zählt.

Schnelle Multiplikation: Zahlen und Matrizen

(a) Zwei n-Bit Zahlen können mit $O(n^{\log_2 3}) = O(n^{1.59})$ Bitoperationen multipliziert werden.

(b) Zwei $n \times n$ Matrizen können mit $O(n^{\log_2 7}) = O(n^{2.81})$ arithmetischen Operationen multipliziert werden.

Da viele Additionen ausgeführt werden, erhalten wir Ersparnisse nur für große Eingabelängen n. (Große Konstanten im $O()$.)

Dynamische Programmierung

Dynamische Programmierung: Die Methode

Zerlege das Ausgangsproblem wie in Divide & Conquer in eine Hierarchie ähnlicher, aber kleinerer Teilprobleme (Web).

- **Der Lösungsgraph der dynamischen Programmierung:**
  - Die Teilprobleme P_i sind Knoten des Graphen.
  - Füge die Kante $P_i \to P_j$ ein, wenn die Lösung von P_i in der Lösung von P_j benötigt wird.
- **Der wesentliche Unterschied zu Divide & Conquer:**
  - Der Lösungsgraph ist ein kreisfreier gerichteter Graph, für Divide & Conquer erhalten wir stets einen Baum.
  - Die Lösung eines Teilproblems wird mehrmals benötigt und muss deshalb abgespeichert werden.
- **Die wesentlichen Schritte im Entwurf:**
  - Definiere die **Teilprobleme** und
  - bestimme die **Rekursionsgleichungen:** Löse schwierigere Teilprobleme mit Hilfe bereits gelöster, einfacherer Teilprobleme.

Der Orakel-Ansatz: Ein Gedankenexperiment

Ein Orakel beantwortet Fragen über eine optimale Lösung $\mathcal{O}$.

- Wir stellen unsere Frage über $\mathcal{O}$ und erhalten die Orakel-Antwort.
  - Nicht jede Frage ist sinnvoll, denn letztlich müssen wir unsere Fragen selbst beantworten!
  - Die Bestimmung einer optimalen Lösung, *mit Hilfe der Antwort*, sollte auf ein etwas leichteres, ähnliches Problem führen.
- Um dieses etwas leichtere Problem zu lösen, stellen wir eine zweite Frage und denken diesen Prozess fortgesetzt, bis eine Lösung offensichtlich ist.
  - Die in diesem Prozess gestellten Fragen definieren unsere **Teilprobleme.**
  - Die **Rekursionsgleichung** beschreibt, wie das jeweilige Teilproblem zu lösen ist.

TSP: Das Travelling Salesman Problem

Das Travelling Salesman Problem

Eingabe: n Städte sind gegeben, wobei $\texttt{laenge}(u, v)$ die Distanz zwischen Stadt u und Stadt v ist.

Aufgabe: Bestimme eine Permutation π der Städte, so dass die Länge der Rundreise, also

$$\sum_{i=1}^{n-1} \texttt{laenge}(\pi(i), \pi(i+1)) + \texttt{laenge}(\pi(n), \pi(1)),$$

kleinstmöglich ist.

Der Brute Force Ansatz

- TSP ist sehr schwierig und effiziente Algorithmen sind nicht zu erwarten.
- Eine triviale Lösung: Zähle alle Permutationen von $\{1, \ldots, n\}$ auf und bestimme die Beste.
  - fürchterliche Laufzeit $\Theta(n!)$: 3,6 Millionen für $n = 10$, 6,2 Milliarden für $n = 13$, 1,3 Billionen für $n = 15$ und 6,3 Billiarden Permutationen für $n = 18$.
- Ziel: Eine substantielle Verbesserung der Laufzeit $\Theta(n!)$.

Der Orakel-Ansatz für TSP

Unsere erste Frage: Welche Stadt wird nach Stadt 1 in einer optimalen Rundreise $\mathcal{O}$ besucht?

- Ziemlich naive Frage?!
- Aber die Antwort hilft in der Bestimmung einer optimalen Rundreise.
  - Wenn die Stadt j als nächste besucht wird, dann benutzt $\mathcal{O}$ die Kante $1 \to j$.
  - Die zweite Frage: Welche Stadt wird nach j besucht?
  - Weitere Fragen nach der jeweils nächsten Stadt decken die optimale Rundreise Schritt für Schritt auf.

Aber leider müssen wir unsere Fragen selbst beantworten.

Die Teilprobleme

Zu jedem Zeitpunkt arbeiten wir mit einem Wegstück $1 \to j \to \cdots \to k$ und fragen

> nach einem kürzesten Weg, der in k startet, alle noch nicht besuchten Städte besucht und dann in 1 endet.

- Sei $L(k, S)$ die Länge eines kürzesten Weges, der in Stadt k beginnt, alle Städte in der Menge $S \subseteq V$ besucht und in 1 endet.
  - Ein Teilproblem (k, S) für jedes $k \neq 1$ und für jede Teilmenge $S \subseteq \{2, \ldots n\} \setminus \{k\}$.
- Die Teilprobleme $(k, \emptyset)$ sind sofort lösbar, denn

$$L(k, \emptyset) = \text{länge}(k, 1).$$

Die Rekursionsgleichungen

Um $L(k, S)$ zu bestimmen, müssen wir alle Möglichkeiten für die erste Kante (k, v) eines kürzesten Weges durchspielen.

Wenn wir Knoten v probieren, dann

- muss der kürzeste Weg von k nach v springen,
- alle Knoten in $S \setminus \{v\}$ besuchen und
- schließlich im Knoten 1 enden.

Wir erhalten **die Rekursionsgleichung:**

$$L(k, S) = \min_{v \in S} \{ \text{länge}(k, v) + L(v, S \setminus \{v\}) \}.$$

Der Aufwand

- Die Anzahl der Teilprobleme ist höchstens

$$(n-1) \cdot 2^{n-2},$$

denn es ist stets $k \neq 1$ und $S \subseteq \{2, \ldots n\} \setminus \{k\}$.

- Da wir für jedes Teilproblem (k, S) alle Elemente $v \in S$ probieren, kann jedes Teilproblem in $O(n)$ Schritten gelöst werden.

Der Aufwand für TSP

Das allgemeine Traveling Salesman Problem für n Städte kann in Zeit $O(n^2 \cdot 2^n)$ gelöst werden.

Wir haben eine riesige Zahl von Teilproblemen produziert, aber

- für $n = 15$: Die triviale Lösung benötigt über 1,3 Billion Schritte, wir benötigen knapp 7,2 Millionen Operationen.
- für $n = 18$: Die triviale Lösung benötigt über 6,3 Billiarden Schritte, wir benötigen knapp 83 Millionen Schritte.

Das gewichtete Intervall Scheduling

Das gewichtete Intervall Scheduling

Eingabe: n Aufgaben sind gegeben, wobei

Aufgabe i durch das Zeitintervall $[s_i, t_i]$ beschrieben wird und den Wert w_i erhält.

Aufgabe: Wähle eine kollisionsfreie Menge $A \subseteq \{1, \ldots, n\}$ von Aufgaben aus und führe sie auf einem einzigen Prozessor aus.

Der Gesamtwert der ausgeführten Aufgaben ist zu maximieren.

Das gewichtete Intervall Scheduling: Der Greedy Algorithmus

Der Greedy-Algorithmus „Frühe Terminierungszeiten zuerst“ funktioniert nicht mehr: Die Aufgabenwerte werden nicht beachtet.

Wir brauchen neue Ideen.

Der Orakel Ansatz

Wir greifen zuerst die Grundidee unserer Greedy-Lösung auf: Die Aufgaben seien nach aufsteigender Terminierungszeit angeordnet.

- Aufgabe 1 endet als erste, ...
- Aufgabe n endet als letzte.

- Welche Aufgaben können vor Aufgabe i ausgeführt werden?
  - $a(i)$ sei die Aufgabe mit spätester Terminierungszeit, die vor Aufgabe i endet. Offensichtlich ist $a(i) < i$.
  - Nur Aufgaben $j \leq a(i)$ können vor Aufgabe i ausgeführt werden.
- **Unsere Frage an das Orakel:** Wird Aufgabe n in einer optimalen Lösung ausgeführt?
  - Wenn n ausgeführt wird, dann ist das kleinere Intervall Scheduling Problem mit Aufgaben in $\{1, \ldots, a(n)\}$ optimal zu lösen.
  - Sonst ist eine optimale Lösung für die Aufgaben in $\{1, \ldots, n-1\}$ zu bestimmen.

Die Teilprobleme und ihre Rekursionsgleichungen

Bestimme nicht nur eine optimale Lösung für die Aufgaben in $\{1, \ldots, n\}$, sondern für alle Teilmengen $\{1, \ldots, i\}$:

$$\begin{aligned} \text{opt}(i) \quad &= \quad \text{Der Gesamtwert einer optimalen Lösung für} \\ &\qquad \text{die Aufgaben in } \{1, \ldots, i\} \end{aligned}$$

Wir erhalten die Rekursionsgleichung

$$\text{opt}(i) = \max\{\text{opt}(a(i)) + w_i, \text{opt}(i-1)\}$$

- Entweder eine optimale Lösung führt Aufgabe i aus und wir erzielen den Gesamtwert $\text{opt}(a(i)) + w_i$.
- Oder Aufgabe i wird nicht ausgeführt und wir erzielen den Gesamtwert $\text{opt}(i-1)$.

Der Algorithmus

(1) Sortiere alle n Aufgaben nach aufsteigender Termininierungszeit. Wir nehmen $t_1 \leq t_2 \leq \cdots \leq t_n$ an.

(2) Für jedes i
- bestimme $a(i)$, die Aufgabe mit spätester Terminierungszeit, die vor Aufgabe i endet.
- Wenn es keine vor Aufgabe i endende Aufgabe gibt, dann setze $a(i) = 0$.

(3) Setze $\text{opt}(0) = 0$.
// Wenn keine Aufgabe auszuführen ist, dann entsteht kein Wert.

(4) for ($i = 1$, $i \leq n$; i++)
$\text{opt}(i) = \max\{\text{opt}(a(i)) + w_i, \text{opt}(i-1)\}$,

Finden von $a(i)$ demogr.

Wie schnell kann in Schritt (2) des Algorithmus eines der $a(i)$ berechnet werden?

- (1) $\Theta(\log n)$
- (2) $\Theta(\sqrt{n})$
- (3) $\Theta(1)$
- (4) $\Theta(n \log n)$
- (5) $\Theta(n)$

Auflösung: (1) $\Theta(\log n)$, binäre Suche!

Wie bestimmt man denn ein optimales Scheduling?

Wir kennen den Wert einer optimalen Auswahl, aber nicht die optimale Auswahl selbst!

- Wir definieren ein Array V, so das $V[i] = j \Leftrightarrow$
  j ist die letzte, „vor“ Aufgabe i ausgeführte Aufgabe.
  - Setze $V[0] = 0$.
  - Wenn $\text{opt}(i) = \text{opt}(a(i)) + w_i$, dann gehört i zur optimalen Auswahl unter den ersten i Aufgaben. Wir setzen $V[i] = i$.
  - Ansonsten ist $\text{opt}(i) = \text{opt}(i-1)$. Wir setzen $V[i] = V[i-1]$.
- Berechne die optimale Auswahl $A \subseteq \{1, \ldots, n\}$ durch:

(1) $A = \emptyset$; $i = n$; // Initialisierung.
(2) while ($i \neq 0$)
  $A = A \cup \{V[i]\}$; // $V[i]$ ist die letzte ausgeführte Aufgabe
  $i = a(V[i])$; // und $V[a(V[i])]$ die vorletzte.
(3) Entferne die imaginäre Aufgabe 0 aus A, falls vorhanden.

Das All-Pairs-Shortest-Path Problem

Das All-Pairs-Shortest-Path Problem

Eingabe: Gegeben ist ein gerichteter Graph $G = (\{1, \ldots, n\}, E)$ sowie die Kantenlängen

$$\text{länge} : E \to \mathbb{R}.$$

Aufgabe: Für je zwei Knoten u und v ist

$$\text{distanz}[u][v] \quad = \quad \text{die Länge eines kürzesten Weges von } u \text{ nach } v$$

zu berechnen.

Der Orakel-Ansatz

Unsere Frage: Was ist die erste Kante (u, w) eines kürzesten Weges $W(u, v)$ von u nach v?

- $W(u, v)$, nach Herausnahme der Kante (u, w), ist ein kürzester Weg von w nach v, und dieser Weg hat eine Kante weniger.
- Unsere zweite Frage gilt der zweiten Kante von $W(u, v)$. Der verbleibende Weg hat jetzt zwei Kanten weniger.
- Um unsere eigenen Fragen beantworten zu können, müssen wir für alle Knoten u, v und alle i, die Länge eines kürzesten Weges von u nach v mit höchstens i Kanten bestimmen.

Die Teilprobleme und ihre Rekursionsgleichungen

Wir führen die Teilprobleme

$$\text{distanz}_i[u][v] \quad = \quad \text{die Länge eines kürzesten Weges von } u \text{ nach } v \text{ mit höchstens } i \text{ Kanten}$$

ein und erhalten sofort die **Rekursionsgleichungen**

$$\text{distanz}_i[u][v] = \min_{w,(u,w)\in E} \{ \text{länge}(u,w) + \text{distanz}_{i-1}[w][v] \}$$

Die Länge eines kürzesten Weges von u nach v mit höchstens i Kanten ist das Minimum über die Längen aller kürzesten Wege, die

- von u zu einem Nachbarn w
- und dann von w über höchstens $i-1$ Kanten nach v laufen.

Der Bellman-Ford Algorithmus

(1) Setze $\text{distanz}[u][v] = \begin{cases} \text{länge}(u,v) & (u,v) \in E, \\ \infty & \text{sonst.} \end{cases}$

// Nächster[u][v] soll stets der auf u folgende Knoten auf einem
// kürzesten Weg von u nach v sein.

$\text{Nächster}[u][v] = \begin{cases} v & (u,v) \in E, \\ \text{nil} & \text{sonst} \end{cases}$

(2) for ($i = 1$; $i \leq n-1$, i++)
for ($u = 1$; $u \leq n$, u++)
für alle Knoten v: $\text{distanz}[u][v] = \min_{w,(u,w)\in E}\{ \text{länge}(u,w) + \text{distanz}[w][v] \}$.

Wenn das Minimum im Knoten w angenommen wird,
dann setze $\text{Nächster}[u][v] = w$.

Warum können wir $\text{distanz}_i[u][v]$ durch $\text{distanz}[u][v]$ ersetzen?

The Good and the Bad

- **Die guten Nachrichten:** Wir können einen kürzesten Weg von u nach v rekonstruieren, wenn wir von u nach $u_1 = \text{Nächster}[u][v]$ und von u_1 nach $u_2 = \text{Nächster}[u_1][v]$ .... wandern.
- **Die guten Nachrichten:** Auch bei negativen Kantengewichten funktioniert Bellman-Ford, solange es keine Kreise negativer Länge gibt.
- **Die schlechten Nachrichten:** Der Aufwand ist sehr hoch.
  - Wir haben $n-1$ „Runden“, wobei n die Anzahl der Knoten ist.
  - Pro Runde werden die Summen $\text{länge}(u,w) + \text{distanz}[w][v]$ für alle Knoten v und alle Kanten (u,w) bestimmt. (Keine Neu- berechnung erforderlich, wenn sich distanz$[w][v]$ nicht ändert.)
  - Pro Runde werden $O(|V| \cdot |E|)$ und insgesamt $O(|V|^2 \cdot |E|)$ Operationen ausgeführt.

Dijkstra's Algorithmus, für jeden Knoten ausgeführt, benötigt höchstens $O(|V| \cdot |E| \cdot \log_2 |V|)$ Operationen.

Bellman-Ford mit Parameter demogr.

Wenn alle kürzesten Pfade in G nur aus höchstens k Kanten bestehen, wie schnell kann dann Bellman-Ford ausgeführt werden?

- (1) $\Theta(|V| \cdot |E|)$
- (2) $\Theta(|V| \cdot |E| \cdot \log k)$
- (3) $\Theta(|V| \cdot |E| \cdot \log k \cdot \log |V|)$
- (4) $\Theta(|V| \cdot |E| \cdot k)$
- (5) $\Theta(|V|^2 / \log k \cdot |E|)$

Auflösung: (4) $\Theta(|V| \cdot |E| \cdot k)$.

Der Algorithmus von Floyd

Der Algorithmus von Floyd: Der Orakel-Ansatz

Ziel: Eine Beschleunigung von Bellman-Ford.

Unsere Frage an das Orakel: Durchläuft ein kürzester Weg $W(u, v)$ von u nach v den Knoten n als inneren Knoten oder nicht?

- **Wenn ja,** dann erhalten wir $W(u, v)$, indem wir zuerst einen kürzesten Weg $W(u, n)$ von u nach n und dann einen kürzesten Weg $W(n, v)$ von n nach v durchlaufen!

  Und was ist die große Erkenntnis?
  Weder $W(u, n)$ noch $W(n, v)$ besitzen n als inneren Knoten.
- **Wenn nein,** dann wissen wir auch mehr, denn $W(u, v)$ benutzt n **nicht** als inneren Knoten.

Die Teilprobleme und ihre Rekursionsgleichungen

Wir führen deshalb die **Teilprobleme**

$$\text{distanz}_i[u][v] \quad = \quad \text{die Länge eines kürzesten Weges von } u \text{ nach } v \text{ mit } \textit{inneren} \text{ Knoten aus der Menge } \{1, \ldots i\}.$$

ein. Die Teilprobleme für $i = 0$ sind trivial (keine inneren Knoten):

$$\text{distanz}_0[u][v] = \begin{cases} \text{länge}(u, v) & (u, v) \in E, \\ \infty & \text{sonst} \end{cases}$$

Die **Rekursionsgleichungen**: $\text{distanz}_i[u][v] =$

$$\min\{\ \text{distanz}_{i-1}[u][v],\ \text{distanz}_{i-1}[u][i] + \ \text{distanz}_{i-1}[i][v]\}.$$

Entweder lasse i aus oder laufe von u nach i und dann nach v.

Implementierung (1/2)

Die Matrix A:

$$A[u][v] = \begin{cases} \text{länge } (u,v) & \text{falls } (u,v) \in E, \\ \infty & \text{sonst.} \end{cases}$$

Der Algorithmus:

```
for (u=1; u ≤ n; u++)
    for (v=1; v ≤ n; v++)
        distanz_0[u][v] = A[u][v];
for (i=1; i ≤ n; i++)
    for (u=1; u ≤ n; u++)
        for (v=1; v ≤ n; v++) {
            temp = distanz_{i-1}[u][i] + distanz_{i-1}[i][v];
            distanz_i[u][v] = (distanz_{i-1}[u][v] > temp) ?
                                    temp : distanz_{i-1}[u][v];
        }
```

Implementierung (2/2)

Die Berechnung von $\text{distanz}_i[u][v]$ ist jetzt wesentlich schneller. Laufzeit insgesamt ist $O(n^3)$.

- Aber $O(n^3)$ Speicherplatz inakzeptabel.
- Es ist stets $\text{distanz}_{i-1}[u][i] = \text{distanz}_i[u][i]$ sowie $\text{distanz}_{i-1}[i][v] = \text{distanz}_i[i][v]$.
- Wir können die Abhängigkeit von i unterschlagen.

```
for (i=1; i ≤ n; i++)
    for (u=1; u ≤ n; u++)
        for (v=1; v ≤ n; v++) {
            temp = A[u][i] + A[i][v];
            A[u][v] = (A[u][v] > temp) ?  temp : A[u][v];
        }
```

Zusammenfassung

Für Graphen mit n Knoten und m Kanten:

Bellman-Ford

- **Vorteile:** Arbeitet sogar in (asynchronen) verteilten System korrekt.
  (Wichtig für Internet Anwendungen)
  Relativ schnell, wenn distanz-Werte sich selten ändern.
- **Nachteile:** Lausige worst case Laufzeit $O(n^2 \cdot m)$.
  Reagiert selbst bei kleinen Änderungen der Kantenlängen sehr träge.

Floyd

- **Vorteile:** Schneller als Bellman-Ford mit Laufzeit $O(n^3)$.
- **Nachteile:** Langsamer als Dijkstra für schlanke Graphen.
  Kann nicht in (asynchronen) verteilten Systemen eingesetzt werden.

Das paarweise Alignment

Das paarweise (globale) Alignment

Für DNA-Sequenzen, RNA-Sequenzen und Proteine: die Ähnlichkeit von Sequenzen impliziert „häufig“ ihre funktionale Ähnlichkeit.

DNA-Sequenzen, RNA-Sequenzen und Proteine entsprechen in ihrer Primärstruktur Strings über einem vier- bzw. 20(21)-elementigem Alphabet (Nukleotide bzw. Aminosäuren):

Führe eine Ähnlichkeitsanalyse von zwei Strings durch.

Strings

- Sei Σ eine endliche Menge, das Alphabet.
- Für $n \in \mathbb{N}$ ist Σ^n die Menge der Strings über Σ der Länge n.
- $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ ist die Menge aller Strings über Σ.
- Für einen String $w \in \Sigma^*$ ist w_i der $i$te Buchstabe von w.
- $-$ bezeichnet das Blanksymbol.

Ähnlichkeitsanalyse

Strings $u^*, v^* \in (\Sigma \cup \{-\})^*$ bilden ein Alignment der Strings $u, v \in \Sigma^*$ genau dann, wenn

- u (bzw. v) durch Entfernen der Blanksymbole aus u^* (bzw. v^*) entsteht und
- u^*, v^* dieselbe Länge besitzen.

Das paarweise globale Alignment Problem für Strings u, v und ein Ähnlichkeitsmaß

$$d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$$

Das Ziel: Bestimme ein Alignment u^*, v^* mit maximaler „Übereinstimmung“

$$q(u^*, v^*) = \sum_i d(u_i^*, v_i^*).$$

Eine andere Perspektive

Ein Beispiel: Die Strings

$$\begin{aligned} u &= GCTGATATAGCT \\ v &= GGGTGATTAGCT \end{aligned}$$

besitzen das Alignment

$$\begin{aligned} u^* &= -\,G\,C\,T\,G\,A\,T\,A\,T\,A\,G\,C\,T \\ v^* &= G\,G\,G\,T\,G\,A\,T\,-\,T\,A\,G\,C\,T \end{aligned}$$

Wie könnte v aus u durch „Punktmutationen“ entstanden sein?

- Füge G an Position 1 in u hinzu,
- ersetze C in Position 3 durch G und
- lösche A in Position 8.

Der Orakel-Ansatz

Alignments sollen also die „Mutationsdistanz“ zwischen zwei Strings messen.

Unsere Frage an das Orakel: Was sind die letzten Buchstaben u_N^* und v_N^* eines optimalen Alignments u^* und v^*?

- **Fall 1:** $u_N^* \neq -$ und $v_N^* \neq -$. Problem zurückgeführt auf ein optimales Alignment von u und v nach Streichen des jeweilig letzten Buchstabens.
- **Fall 2:** $u_N^* = -$ und $v_N^* \neq -$. Problem zurückgeführt auf ein optimales Alignment von u und v nach Streichen des letzten Buchstabens von v.
- **Fall 3:** $u_N^* \neq -$ und $v_N^* = -$. Problem zurückgeführt auf ein optimales Alignment von u und v, nach Streichen des letzten Buchstabens von u.

Die Teilprobleme

Wir werden stets von einem optimalen Alignment für u und v auf ein optimales Alignment für Präfixe von u und v geführt.

Deshalb führen wir die Teilprobleme „Bestimme $D(i,j)$“ ein mit

$$D(i,j) \quad = \quad \text{der Wert eines optimalen Alignments für die Präfixe } u_1 \cdots u_i \text{ und } v_1 \cdots v_j.$$

Eine Lösung der trivialen Teilprobleme:

$$D(i,0) = \sum_{k=1}^{i} d(u_k, -) \text{ und } D(0,j) = \sum_{k=1}^{j} d(-, v_k),$$

denn wir müssen ausschließlich Buchstaben löschen bzw. hinzufügen.

Die Rekursionsgleichungen

- **Fall 1:** $u_i^* \neq -$ und $v_j^* \neq -$

$$D(i,j) = D(i-1,j-1) + d(u_i, v_j),$$

- **Fall 2:** $u_i^* = -$ und $v_j^* \neq -$

$$D(i,j) = D(i,j-1) + d(-, v_j),$$

- **Fall 3:** $u_i^* \neq -$ und $v_j^* = -$

$$D(i,j) = D(i-1,j) + d(u_i, -)$$

und wir erhalten die Rekursionsgleichungen

$$\begin{aligned} D(i,j) \quad = \quad & \max\{\ D(i-1,j-1) + d(u_i, v_j), \\ & \qquad D(i,j-1) + d(-, v_j), \\ & \qquad D(i-1,j) + d(u_i, -)\ \}. \end{aligned}$$

Das Programm

Der String u habe n und der String v habe m Buchstaben.

(1) // **Initialisierung**.
$D(0,0) = 0$;
for $(i = 1; i <= n; i++)$
$D(i,0) = \sum_{k=1}^{i} d(u_k, -)$;
for $(j = 1; j <= m; j++)$
$D(0,j) = \sum_{k=1}^{j} d(-, v_k)$;

(2) // **Iteration**.
for $(i = 1; i <= n; i++)$
for $(j = 1; j <= m; j++)$

$$\begin{aligned} D(i,j) \quad = \quad \max\{ & D(i-1,j-1) + d(u_i, v_j), \\ & D(i,j-1) + d(-, v_j), \\ & D(i-1,j) + d(u_i, -) \}. \end{aligned}$$

Laufzeit paarweises Alignment demogr.

Wie schnell wird das Alignment für Strings der Größen n und m berechnet?

- (1) $\Theta(n+m)$
- (2) $\Theta((\min\{n,m\})^2)$
- (3) $\Theta(n \cdot m)$
- (4) $\Theta((\max\{n,m\})^2)$
- (5) $\Theta(n^m)$

Auflösung: (3) $\Theta(n \cdot m)$

Zusammenfassung

Das globale Alignment für zwei Strings der Längen n und m wird in Zeit höchstens $O(n \cdot m)$ bestimmt.

Unser Ansatz kann auch für andere Alignment Varianten wie

- das semi-globale Alignment und
- das lokale Alignment

benutzt werden.

Unser Algorithmus ist zu langsam für die Suche in großen Datenbanken langer Strings:
Heuristiken wie BLAST oder FASTA werden verwandt.

Die RNA-Sekundärstruktur

Die RNA-Sekundärstruktur

- Während die Doppelhelix der DNA aus zwei komplementären Strängen mit den komplementären Basenpaaren (A, T) und (G, C) besteht, besitzt die RNA nur einen einzigen Strang.
- Wichtig für die Funktionalität eines RNA Moleküls ist seine *Sekundärstruktur*:

  Der Strang des RNA Moleküls faltet sich, um Bindungen zwischen den komplementären Basen $\{A, U\}$ und $\{C, G\}$ einzugehen.

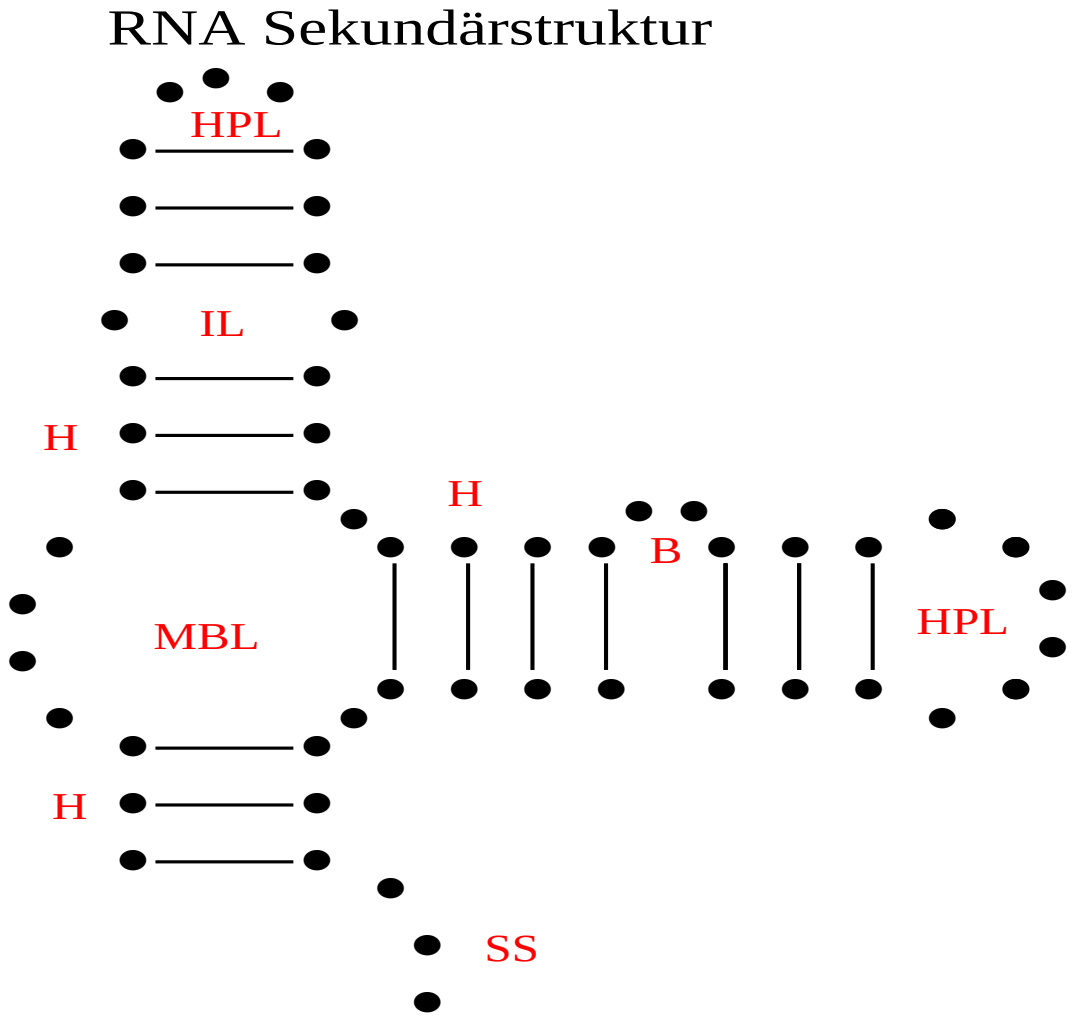
Das Ziel: Bestimme die Sekundärstruktur der Primärstruktur

$$R \in \{A, C, G, U\}^n.$$

Welche Eigenschaften hat die Sekundärstruktur?

Die RNA-Sekundärstruktur: Ein Beispiel

(www.zib.de/MDGroup/temp/lecture/l5/p_secondary/rna_second.pdf)



B bulge, H helix, HPL hairpin loop, IL internal loop, MBL multibranched loop or junction, SS single strand

„Formalisierung“ der Sekundärstruktur

$R \in \{A, C, G, U\}^n$ sei die Primärstruktur eines RNA Moleküls.

Eine Menge $P \subseteq \{ \{i, j\} \mid 1 \leq i \neq j \leq n\} \}$ von Paarmengen ist eine Sekundärstruktur, falls:

- Für jedes Paar $\{i, j\} \in P$ gilt $|i - j| \geq 5$.
  - Bindungen nur ab Distanz 5: Sonst knickt die Faltung zu „scharf ab“.
- Die Paare in P bilden ein Matching. Jede Position $i \in \{1, \ldots, n\}$ gehört zu höchstens einem Paar in P.
- Paare in P entsprechen komplementären Basen. Für jedes Paar $(i, j) \in P$ ist entweder $\{R_i, R_j\} = \{A, U\}$ oder $\{R_i, R_j\} = \{C, G\}$.
- Die Faltung besitzt keine „Überkreuzungen“:
  - Wenn $\{i, j\} \in P$ (für $i < j$) und $(k, l) \in P$ (für $k < l$),
  - dann ist $i < k < j < l$ ausgeschlossen.

Das Optimierungsproblem

Die obigen Eigenschaften werden von den „meisten“ RNA-Molekülen angenommen.

- **Annahme:** Der String R faltet sich, um die Anzahl der Bindungen zu maximieren.
- **Das Ziel:** Bestimme eine Sekundärstruktur $P \subseteq \{ \{i, j\} \mid 1 \leq i \neq j \leq n \}$ mit maximaler Anzahl Bindungen.

Der Orakel-Ansatz

Unsere Frage an das Orakel:

- Besitzt eine optimale Faltung eine Bindung für die letzte Position?
- Wenn ja, mit welcher Position i wird die Bindung eingegangen?

Angenommen eine optimale Sekundärstruktur P_{opt} enthält das Paar $\{i, n\}$.
Dann zerfällt P_{opt} in zwei optimale Sekundärstrukturen:

- die Sekundärstruktur für den Präfix von R zu den Positionen $1, \ldots, i-1$ und
- die Sekundärstruktur für den Suffix von R zu den Positionen $i+1, \ldots, n-1$.

Die Teilprobleme und ihre Rekursionsgleichungen

Die Idee:
Berechne optimale Sekundärstrukturen für alle Teilstrings von R.

Wir definieren deshalb die Teilprobleme

$$D(i,j) \quad = \quad \text{Die Größe einer optimalen Sekundärstruktur für den Teilstring } R_i \cdots R_j.$$

- **Fall 1:** j gehört zu keinem Paar in P_{opt}. Es ist $D(i,j) = D(i,j-1)$.
- **Fall 2:** Ein Paar (k,j) gehört zu P_{opt}.
  Dann ist $D(i,j) = D(i,k-1) + 1 + D(k+1,j-1)$.
  - Aber wir kennen k nicht!
  - Kein Problem, maximiere über alle $i < k \leq j-5$.

Die Rekursionsgleichungen

Das Paar (k, j) heißt wählbar, falls

- $i \leq k \leq j - 5$: Keine zu scharfen Knicke und
- $\{R_k, R_j\} = \{A, U\}$ oder $\{R_k, R_j\} = \{C, G\}$:
  Wir fordern komplementäre Basen.

Die Rekursionsgleichungen:

$$D(i,j) = \max_{k,(k,j) \text{ ist wählbar}} \{D(i,k-1) + D(k+1,j-1) + 1, D(i,j-1)\}$$

Die Intervallgrenzen $(i, k-1)$ und $(k+1, j-1)$ sichern die Matching-Eigenschaft:

Weder k noch j treten in einem weiteren Paar auf.

Der Algorithmus

(1) for ($l = 1$; $l <= 4$; l++)
    for ($i = 1$; $i <= n - l$; i++)
        Setze $D(i, i + l) = 0$; // $D(i, i + l) = 0$ für $l \leq 4$.

(2) for ($l = 5$; $l <= n - 1$; l++)
    // l beschreibt die Länge des Teilstrings.
    for ($i = 1$; $i <= n - l$; i++)
    // i ist die erste Position des Teilstrings und $j = i + l$ die letzte.

$$
\begin{aligned}
j &= i + l \\
D(i,j) &= \max_{k,(k,j) \text{ ist waehlbar}} \{D(i, k-1) + D(k+1, j-1) + 1, D(i, j-1)\}
\end{aligned}
$$

Laufzeit RNA Sekundärstruktur demogr.

Wie schnell kann die RNA Sekundärstruktur für Stringgröße n berechnet werden?

- (1) $\Theta(n)$
- (2) $\Theta(n^2)$
- (3) $\Theta(n^3)$
- (4) $\Theta(n^4)$
- (5) $\Theta(n^5)$

Auflösung: (3) $\Theta(n^3)$

Die Laufzeit ist $O(n^3)$, denn
jedes der n^2 Teilprobleme kann in Zeit $O(n)$ gelöst werden.

Dynamische Programmierung: Der Orakel-Ansatz

- Der zentrale Schritt ist der Entwurf einer Hierarchie von Teilproblemen.
  - Der Orakel-Ansatz: Wir befragen ein Orakel über eine optimale Lösung und tasten uns sukzessive an die optimale Lösung heran.

    Aber Achtung: Wir müssen unsere Fragen selbst beantworten.
- Was war unsere jeweils erste Frage?
  - Für das gewichtete Intervall Scheduling: Wird die Aufgabe mit spätester Terminierungszeit ausgewählt oder nicht?
  - Für kürzeste Wege-Probleme: Wird Knoten n durchlaufen ? (Floyd) Welche Kante wird als erste gewählt (TSP und Bellman-Ford)?
  - Für das paarweise Alignment: Wird der letzte Buchstabe von u gegen den letzten Buchstaben von v oder gegen das Blank-Symbol aligniert?

Dynamische Programmierung: Gute Fragen

- Wenn wir die Antwort auf die erste Frage kennen, dann sollte das Auffinden einer optimalen Lösung, im Vergleich zum Ausgangsproblem, ein ähnliches, aber einfacheres Problem sein.
- Wir stellen weitere Fragen, da das „Restproblem“ immer noch zu hart sein wird.
  - Beschreibe die Menge aller insgesamt gestellten Fragen,
  - oder äquivalent dazu: Formuliere die Teilprobleme.
- Stelle die Rekursionsgleichungen auf, um alle Fragen zu beantworten, bzw. alle Teilprobleme zu lösen.
  - Stelle sicher, dass die Anzahl der Teilprobleme nicht zu groß ist
  - und dass die Teilprobleme ausreichend schnell lösbar sind!

Aber wie stellt man gute Fragen? Gute Frage!

Lineare Programmierung

Lineare Programmierung

Die Lineare Programmierung ist eine sehr mächtige Technik für die Lösung von Optimierungsproblemen.

- Bedingungen werden durch ein System von linearen Ungleichungen definiert.
- Eine optimale Lösung muss alle Ungleichungen erfüllen und eine lineare Zielfunktion optimieren.

Formal:

$$\max \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i \text{ für } i = 1, \ldots, m$$

$$x_1, \ldots, x_n \geq 0. (\mathit{Web})$$

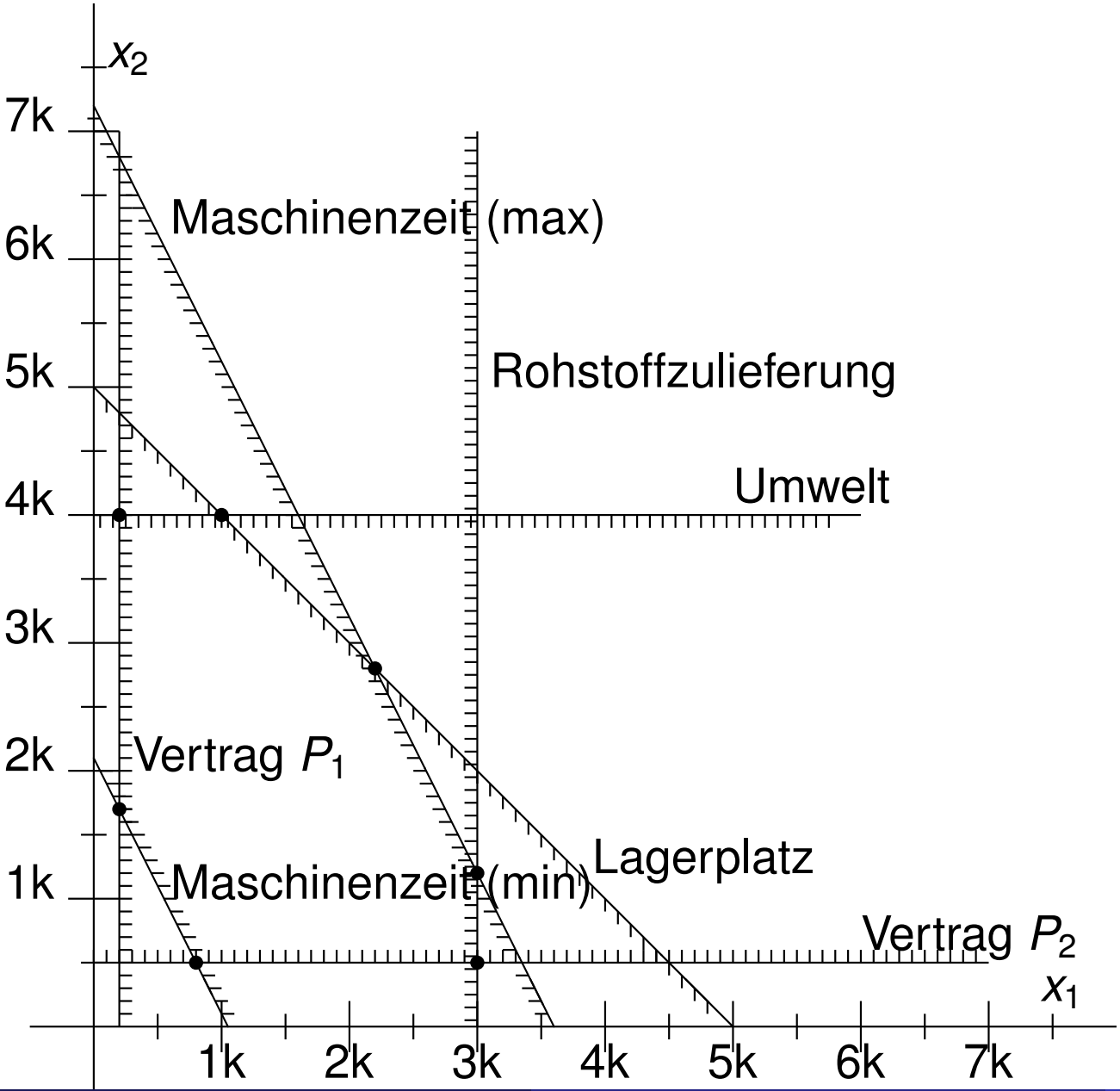
Ein Beispiel

- Ein Unternehmen verfügt über eine Maschine, auf der zwei verschiedene Produkte P_1 und P_2 produziert werden können.
- Eine Einheit von P_1 bringt 7 Euro Gewinn, eine Einheit P_2 bringt 4 Euro. Die Zielfunktion ist $7 \cdot x_1 + 4 \cdot x_2$.
- Die Produktion einer Einheit von P_1 nimmt die Maschine 4 Minuten in Anspruch, eine Einheit P_2 ist in 2 Minuten fertig.
  Die Maschine steht im Monat für 240 Stunden, also für 14400 Minuten, zur Verfügung. Wir erhalten die Ungleichung $4 \cdot x_1 + 2 \cdot x_2 \leq 14400$.
- Weitere Bedingungen wie Rohstoffverbrauch, Absatzmöglichkeiten etc. führen auf weitere Ungleichungen.

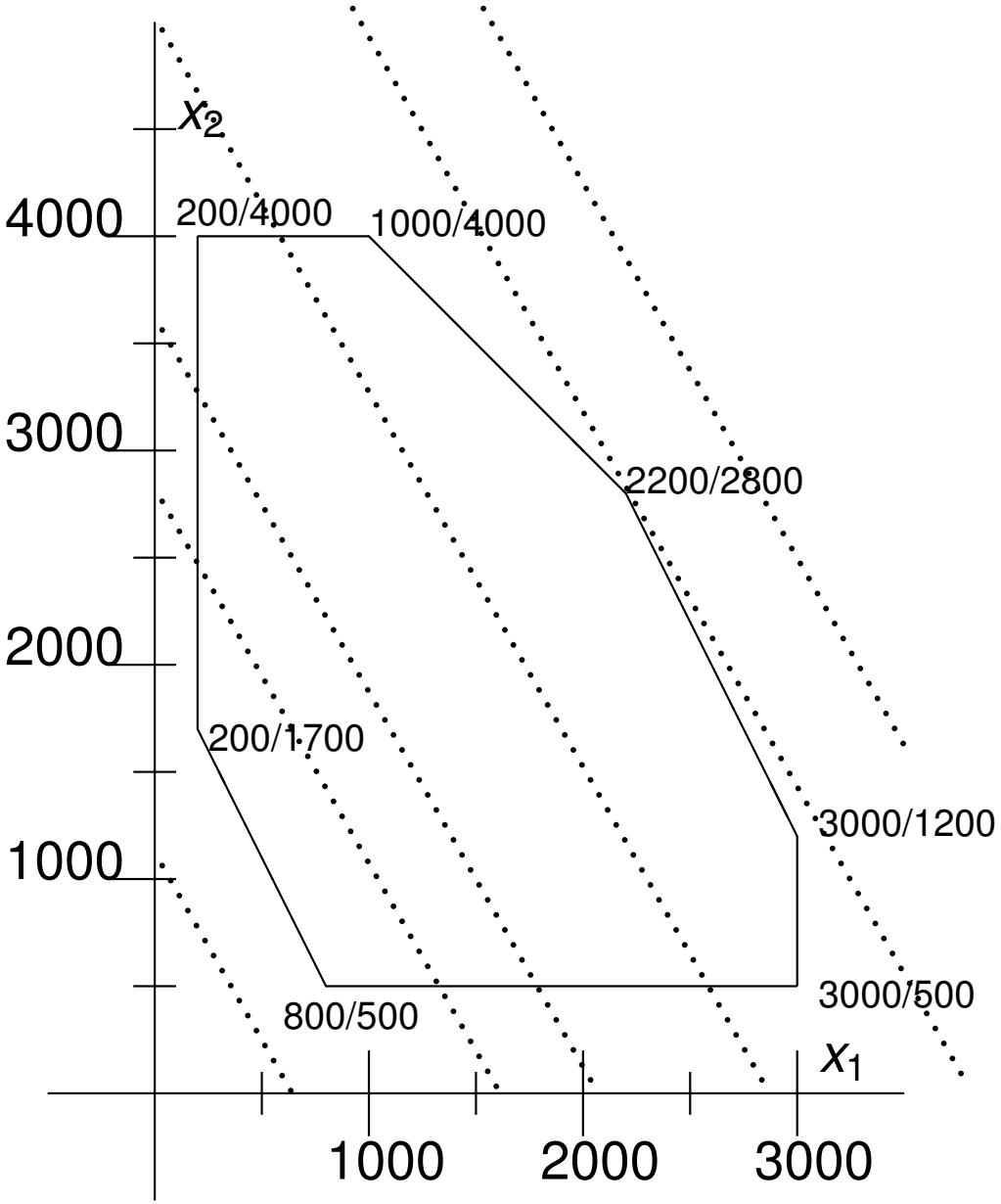
Wie kann eine optimale Lösung bestimmt werden?

Zuerst eine geometrische Interpretation.

Die Ungleichungen



Der Lösungsraum und die Zielfunktion



Effiziente Algorithmen für die Lineare Programmierung

Man bezeichnet den Lösungsraum auch als Polytop.

- Um eine optimale Lösung zu bestimmen, verschiebe die Gerade $7 \cdot x_1 + 4 \cdot x_2 = 0$ so weit wie möglich „nach oben“.
- Die Gerade wird dann auf eine Ecke treffen. (Ecken erfüllen zwei oder mehrere Ungleichungen exakt.)
- Die Situation im *n*-dimensionalen ist ungleich komplizierter, aber es gibt schnelle Algorithmen.
  - Der Simplex-Algorithmus sucht nach einer optimalen Ecke.
  - Interior-Point Verfahren durchstossen das Innere des Polytops und sind häufig schneller.

Das Zuordnungsproblem (bipartites Matching)

- n Angestellte und m Aufgaben sind gegeben.
- $k_{i,j}$ ist die Kompetenz des Angestellten i für die Aufgabe j.
- Weise jeder Aufgabe j höchstens einen Angestellten $i(j)$ und jedem Angestellten höchstens eine Aufgabe zu. Maximiere die Kompetenz

$$\sum_{j=1}^{m} k_{i(j),j}.$$

Die Formulierung als ein lineares Program:

$$\max \sum_{i,j} k_{i,j} \cdot x_{i,j} \quad \text{so dass} \quad \sum_{i} x_{i,j} \leq 1 \text{ für alle Aufgaben } j,$$

$$\sum_{j} x_{i,j} \leq 1 \text{ für alle Angestellten } i,$$

$$x_{i,j} \geq 0 \text{ für alle } i, j$$

Integrale und fraktionale Lösungen

- Wenn $x_{i,j}$ stets entweder 0 oder 1 ist, dann beschreibt x eine legale Zuordnung.
- Im Zuordnungsproblem ist jede Ecke integral, das heißt jede Komponente ist ganzzahlig.
  - Jede Ecke hat nur 0- oder 1-Komponenten.
  - Die lineare Programmierung löst also das Zuordnungsproblem, wenn wir eine optimale Ecke zum Beispiel mit dem Simplex Algorithmus berechnen.
- Für viele andere Probleme wird es aber fraktionale Ecken geben.
  - **Zentrale Frage:** Wie erhält man aus einer optimalen fraktionalen Lösung eine gute integrale Lösung?

Lineare Nebenbedingungen demogr.

$$\max \sum_{i=1}^{n} c_i \cdot x_i \quad \text{so dass} \quad \sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i \text{ für } i = 1, \ldots, m$$
$$x_1, \ldots, x_n \geq 0.$$

Kann man die Bedingung $x_2 = 2x_5$ in einem linearen Programm ausdrücken?

- Ja.
- Nein.
- Keine Ahnung

Auflösung: Ja, benutze $x_2 - 2x_5 \leq 0$ und $-x_2 + 2x_5 \leq 0$.