Es ist bekannt, dass sowohl das Erfüllbarkeitsproblem der Aussagenlogik SAT als auch 3-SAT, das Erfüllbarkeitsproblem mit Beschränkung auf Klauseln mit nur 3 Literalen, $\mathcal{NP}$-vollständig sind. Das Problem 3-CLIQUE ist wie folgt definiert:

Problem: 3-CLIQUE
Gegeben: Ein ungerichteter Graph $G = (V, E)$
Gesucht: Gibt es eine Clique (vollständig verbundener Teilgraph) der Größe 3 in G?

Zeigen Sie, dass 3-CLIQUE **nicht** $\mathcal{NP}$-vollständig ist!

Wir geben einen Algorithmus an, der 3-CLIQUE in Polynomialzeit entscheidet. Da wir in dieser Aufgabe von der Annahme $\mathcal{P} \neq \mathcal{NP}$ ausgehen, kann 3-CLIQUE nicht $\mathcal{NP}$-vollständig sein, denn sonst wäre $\mathcal{P} = \mathcal{NP}$. Das CLIQUE-Problem gehört zu den "fixed parameter tractable"- Problemen. Das heißt, ist die Größe einer Clique fest vorgegeben, in diesem Fall 3, so ist das zugehörige Entscheidungsproblem in Polynomialzeit lösbar. Der Algorithmus verfolgt dabei schlicht den Brute-Force-Ansatz: Man betrachtet alle Tripel von Knoten und testet, ob diese eine Dreierclique bilden. Ist nach durchlaufen aller Tripel eine Dreierclique gefunden, so gibt man als Ausgabe "JA" aus, ansonsten "NEIN". Es gibt insgesamt $\binom{|V|}{3}$ Tripel von Knoten, daher ist der Aufwand des Algorithmus im wesentlichen $O(\binom{|V|}{3}) = O(|V|^3)$, was polynomial in der Eingabe ist.

Nun beweisen wir die Korrektheit der Reduktion. 3DM

1. Ein dreidimensionales Matching aus P_F muss für jedes $i = 1, 2, \ldots, n$ die Mengen $B(x_i)$ und $G(x_i)$ überdecken. Diese Elementen sind, wie oben dargestellt, entweder alle in einem der „Kreise" $\bar{x}_i^{(j)}$ für $j = 1, 2 \ldots, r$ oder alle in einem der „Dreiecke" $x_i^{(j)}$ für $j = 1, 2 \ldots, r$. Die durch die Überdeckung der Paare (β_j, γ_j) $(j = 1, 2, \ldots, r)$ aus dem Klauselteil definierte Auswahlfunktion α ist folglich widerspruchsfrei.
2. Ist F erfüllbar, und ist α eine widerspruchsfreie Auswahlfunktion, so füllen wir die zu α gehörige Überdeckung M' der Paare $\{(\beta_j, \gamma_j) \mid j = 1, 2, \ldots, r\}$ mit $|M'| = r$ wie folgt zu einem dreidimensionalen Matching M auf:
   - 2.1 Ist $\alpha(K_j) = x_i$, so wählt man für die Mengen $B(x_i)$ und $G(x_i)$ die Überdeckung mit Kreisen.
   - 2.2 Ist $\alpha(K_j) = \bar{x}_i$, so wählt man für die Mengen $B(x_i)$ und $G(x_i)$ die Überdeckung mit Dreiecken
   - 2.3 Gibt es für ein x_i keine Klausel K mit $\alpha(K) \in \{x_i, \bar{x}_i\}$, so ist die Überdeckung der Mengen $B(x_i)$ und $G(x_i)$ nicht festgelegt.
   - 2.4 Alle bisher nicht überdeckten Elemente aus H_F werden durch den Auffüllteil überdeckt.

CLIQUE-> HALFCLIQUE

1.Zeigen das HalfClique in NP
2.von Clique auf Halfclique reduzieren

Die Reduktion kann durch folgende zwei Fälle beweisen werden
Bem.: m = |V|, k = Eingabe Clique

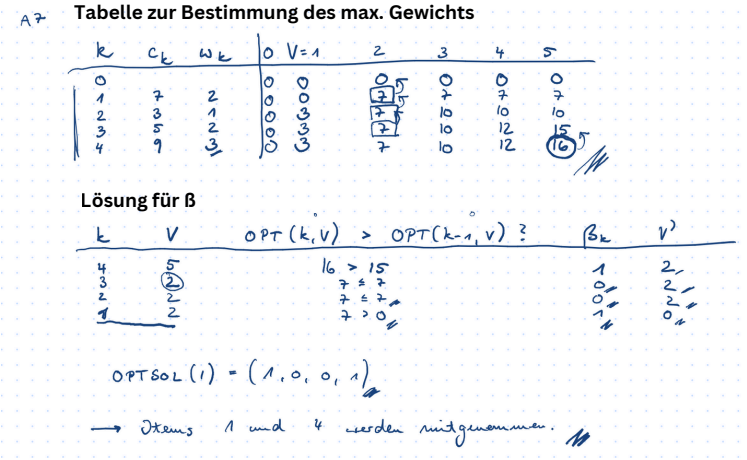
Fall 1 k >= m/2:
neuer Graph G' bekommt genau t= 2k-m neue Knoten
diese werden mit keinen anderen Knoten verbunden -> sind CLIQUE der Größe 1 -> ändern nichts am Ergebnis
-> dadurch wird das Verhältnis von Knoten berichtigt

- wir haben jetzt genau 2k Konten
- nun hat der alte Graph G genau dann eine Clique der Größe k, wenn G' eine Clique der Größe k hat

Fall 2 k < m/2:
neuer Graph G' bekommt genau t= m-2k neue Knoten
diese werden mit allen anderen Knoten verbunden -> G' enthält jetzt genau eine Clique der Größe k+t (k+t= m+t/2) die mindestens k alte Konten enthalten muss, welche in Graph G eine Clique gebildet haben
-> durch diese Konstruktion enspricht Cliquengröße in G' genau n/2 (n Knotenanzahl neuer knoten)

KNAPSACK- dynamische Programmierung

Tabelle zur Bestimmung des max. Gewichts



Lösung für ß

Datenstrukturen- Kruskal

Union Find: effiziente Möglichkeit (alle Operationen in O(log_2(n)))um Zusammenhangskomponenten in einem Graph zu verwalten -> mit dieser können schnell Zusammenhangskomponenten ermittelt, zusammengefügt und aktualisiert werden -> diese eignet sich um Kruskal effizient zu implementieren, da dieser schnell überprüfen muss ob durch die neue Kante eventuell Zyklen entstehen könnten für die Färbung

Listen: Für die Sortierung der Kanten eignen sich Listen

Partition-> ThirdPartition

Def. ThirdPartition:

- Eingabe: Multimenge S
- Ausgabe: Akzeptiere gdw es existiert eine Teilmenge A von S sodass
  - 2 * sum_{a in A} a = sum_{a notin A} a

1. Zeige dass ThirdPartition in NP (Raten, dann Verifizieren in P)
2. Reduziere Partition auf ThirdPartition:

Transformation der Eingabe I aus Partition in eine Eingabe I' aus ThirdPartition:
Sei SUM := sum_{i in I} i, sei e1 := 3/2 * SUM und e2 := 7/2 * SUM (alles in PTIME berechenbar). Sei die transformierte Eingabe I' := S u {e1} u {e2}. Damit ist sum_{i in I'} i = 6 * SUM.
Wird die Eingabe I von Partition akzeptiert, existiert die partitionierende Teilmenge A von I mit sum_{a in A} = SUM/2. Damit I' von ThirdPartition akzeptiert wird, muss eine Teilmenge B von I' existieren mit sum_{b in B} b = 6 * SUM / 3 = 2 * SUM. Wir nehmen also B := A u {e1}.

Problem : Input (G,B,H,P)-> P ist Relation -> Frage: gibt es in P eine Teilmenge s.d. in jedem Tupel genau jedes Element von g,b vorkommt und genau zweimal jedes Element in H

3DM-> Multi3DM

1.Zeigen das Multi3DM in NP
2.von 3DM auf Multi3DM reduzieren

Eingabe 3DM umwandeln in eine Eingabe von Multi3DM

- G' = G u {alle Elemente in G mit neuem Index}
- B' = B u {alle Elemente in B mit neuem Index}
- H' = H
- P' = P u {alle Tupel nehmen und b ersetzen mit }

Bounded Partition in P

Lösung durch dynamische Programmierung [Bearbeiten | Quelltext bearbeiten]

Mithilfe der dynamischen Programmierung kann man das Partitionsproblem in pseudopolynomieller Zeit entscheiden.[4] Hierbei werden systematisch kleinere Teilprobleme betrachtet und ihre Lösungen tabelliert und rekursiv zusammengefügt.

Sei S die Summe der N gegebenen Zahlen. Falls sie ungerade ist, existiert offensichtlich keine perfekte Aufteilung. Andernfalls wird für alle $0 \leq i \leq N$ und alle $0 \leq j \leq S/2$ geprüft, ob es eine Auswahl von Zahlen in der Familie $\{a_1, a_2, \ldots, a_i\}$ der ersten i Zahlen gibt, deren Summe genau j ergibt. Für $i = 0$ und $j = 0$ ist dies offenbar der Fall, genauso für $i = 1$ und $j = a_1$. Für $i > 1$ und alle anderen j dagegen nicht. Dies ist der Anfang der Rekursion, der in der ersten Zeile einer Tabelle notiert wird. Für die weiteren Zeilen ergeben sich die Einträge nach folgender Rekursion: Eine Auswahl für (i, j) existiert genau dann, wenn entweder bereits eine für $(i-1, j)$ existiert, oder wenn $j > a_i$ ist und eine Auswahl für $(i-1, j-a_i)$ existiert. Die Antwort auf das Entscheidungsproblem gibt der letzte Eintrag in der Tabelle (für $i = N$ und $j = S/2$) an.

Die Komplexität dieses Algorithmus ist $\mathcal{O}(N \cdot S)$.

Mergesort

Prinzip: Divide and Conquer

- Eingabe Array wird solange rekursiv halbiert bis nur noch Teilarrays der Größe >= 2 übrig bleiben
- die Teilarrays werden anschließend sortiert falls die Größe gleich 2 ist
- danach werden die Teilarrays mit ihren Teilungspartnern aus der Teile-Phase gemergt
- dafür werden die Teilarrays von links nach rechts betrachtet
- stets das kleinere Element wird in das resultierende(zussammengemischte Array) eingefügt
- dies passiert solange bis alle Teilarrays wieder zu einem großen zusammengefügt wurden

Laufzeit: O(n* log n) -> log n Teilarrays entstehen
-> jede merge Operation benötigt O(n) Schritte

Hashing

offenes Hashing: jedes Bucket als Liste implementiert
geschlossenes Hashing: niemals zwei Schlüssel im gleichen Bucket

- **lineares Sondieren:** (I.) Hashwert berechnen (II.) ist Bucket voll gehen einen Bucket weiter (III.) Mache das solange bis freies Bucket gefunden
- **quadratisches Sondieren:** (I.)(II.) Ist Bucket belegt-> suche freies Bucket mit Muster: +1,-1,+4, -4, +9, -9, +16, -16, ...

Inorder - Preorder

Behauptung: Der Binärbaum zu geg. **inorder** und **preorder** ist eindeutig bestimmt

Begründung: Zu Beginn ist die Wurzel des Binärbaums eindeutig durch die preorder Anordnung bestimmt-> an erster Stelle
Suchen wir die Wurzel nun in der Inorder- Anordnung so wissen wir, dass in der Anordnung links von der Wurzel alle Werte im linken Teilbaum der Wurzel stehen, und alle Werte rechts der Wurzel stehen im rechten Teilbaum der Wurzel.
Betrachten wir nun die Teilbäume, können wir diese rekursiv wie gehabt behandeln.
In der preorder können wir die Wurzel eindeutig bestimmen(erste Position der Zahlensequenz des aktuell betrachteten Teilbaumes) und über die inorder- Anordnung können wir die restlichen dem linken oder rechten Teilbaum zuordnen.
Da es bei der Positionierung der Werte keine Entscheidungsfreiheit gibt, ist diese bzw. der Binärbaum eindeutig bestimmt.

Inorder - Postorder

Behauptung: Der Binärbaum zu geg. **inorder** und **postorder** ist eindeutig bestimmt

Begründung: Zu Beginn ist die Wurzel des Binärbaums eindeutig durch die postorder Anordnung bestimmt-> an letzter Stelle
Suchen wir die Wurzel nun in der Inorder- Anordnung so wissen wir, dass in der Anordnung links von der Wurzel alle Werte im linken Teilbaum der Wurzel stehen, und alle Werte rechts der Wurzel stehen im rechten Teilbaum der Wurzel.
Betrachten wir nun die Teilbäume, können wir diese rekursiv wie gehabt behandeln.
In der postorder können wir die Wurzel eindeutig bestimmen(letzte Position der Zahlensequenz des aktuell betrachteten Teilbaumes) und über die inorder- Anordnung können wir die restlichen dem linken oder rechten Teilbaum zuordnen.
Da es bei der Positionierung der Werte keine Entscheidungsfreiheit gibt, ist diese bzw. der Binärbaum eindeutig bestimmt.

Alg. In-/Preorder -> Binärbaum

KonstBinärbaum(IN, PRE)

```
n<- length(IN)
empty Tree T
if n==1 : return T
root <- PRE[0]
rootPos <- -1
for i=0 to n-1 do:
    if IN[i]== root: rootPos <- i; break;
leftIN <- Teilarray(IN, 0, rootPos -1);
rightIN <- Teilarray(IN, rootPos+1, n-1);
leftPRE <- Teilarray(PRE, 1, length(leftIN));
rigthPRE <- Teilarray(PRE, length(leftIN)+1, n-1);
T.left <- KONSTBinärbaum(leftIN, leftPRE)
T.right <- KONSTBinärbaum(rightIN, rightPRE);
return T
```

L**aufzeit:** O(n)

Suchbaum preorder

Behauptung: zu einer geg. preorder passende binäre Suchbaum ist eindeutig

Begründung: die erste Zahl ist eindeutig die Wurzel r

- alle Zahlen < r gehören eindeutig zum linken Teilbaum(Suchbaum Eigenschaft)
- alle zahlen >r zum rechten Teilbaum
- betrachten wir die Teilbäume zu können wir obiger Argument rekursiv anwenden
- da die Op < / > eindeutig sind ist auch der Baum eindeutig, da jeder Knoten max. 2 Kinder (v.l / v.r hat)

Alg. Suchbaum/preorder

KonstSuchbaum(PRE)

```
n<- length(PRE)
empty Tree T
if n<=1 : return T
root <- PRE[0]
Pos <- n
for i=1 to n-1 do:
    if PRE[i]>root: Pos <- i; break;
leftPRE <- Teilarray(PRE,1, Pos-1)
rigthPRE <- Teilarray(PRE, Pos, length)
T.left <- KONSTBinärbaum(leftPRE)
T.right <- KONSTBinärbaum(rightPRE);
return T
```

L**aufzeit:** O(n^2)-> wenn der Baum vollständig unbalanciert ist

Heapsort

- aufgebaut auf der Datenstruktur des binärheaps
- im ersten Schritt wird die Eingabe in einen Binärheap umtransformiert
- dann wird die Wurzel getauscht mit dem Element an der Stelle heapsize-1 (also ans Ende des Arrays)
- heapsize wird verringert da nun an letzter Stelle unseres Arrays das größte Element steht und das sozusagen den sortieren Teil angibt
- nun muss um das nächst kleinere Element zu finden wird das Array wieder geheapt werden und dann werden die obrigen Schritte solange wiederholt bis heapsize 1 ist -> dann sind alle Elemente sortiert

Laufzeit: O(n *log n)

- Erstellung eines Heap Baumes O(n *log n) -> da Baum eine Höhe von O(log n) und jeder Konten max O(log n) mal überprüft/ getauscht wird -> da es n Konten gibt O(n*log n)

PRIM

Beim Algorithmus von Prim wird ein MSt zu einem Graphen bestimmt, indem von einem Startknoten ausgehend zunächst alle adjazenten Knoten/ Kanten betrachtet werden. Es wird stets die Kante mit dem kleinsten Gewicht gewählt, di die aktuelle Menge gewählter Knoten mit einem neuen unbesuchten Knoten verbindet. Dies geschieht nach der Prim-Grünfärbbarkeitsregel. Es werden Prioritätswarteschlangen eingesetzt, um alle Kanten entsprechend zu verwalten.

Laufzeit:

- *Einfügen und Entfernen der Knoten in der Prio- Warteschlange:* O(|V|* log|V|)
- *Siftup- Operationen mit Heap Implementation der Prio-Warteschlange: O(|E|*log|V|))*
- *==> O(|V|*log|V|+ |E|*log|V|)*

Kruskal

Beim Algorithmus von Kruskal wird ein MST zu einem Graphen bestimmt, indem alle Kanten eines Graphen aufsteigend nach des Gewicht sortiert werden.Gehe Liste von vorn nach hinten durch. Färbe Kante grün falls sie zwei Zusammenhangskomponenten von Wt := (V, {e | e ist nach t Durchläufen grüngefärbt}) verbindet, ansonsten rot. Es wird Union Find eingesetzt um alle Zusammenhangskomponenten entsprechend zu verwalten.

Laufzeit:

- *Sortieren der Kanten:* O(|E|* log|E|)= O(|E|*log|V|)(da E<=V^2 ist)
- *Überprüfen ob die Kante zwei Zusammenhangskomponenten verbindet-> |V|-makeSet Operationen, |V|-1 Union und 2|E| find Operationen* ==> O(|E|* log_2^*|V|)
- ==> O(|E|* log_2|V|)

Tiefensuche

Tiefensuche bestimmt ausspannenden Baum zu einem Graphen. Der Algorithmus funktioniert wie folgt: Makiere den Starknoten als besucht und füge ihn zum Stack hinzu. Wähle einen nicht besuchten Nachbarn des akt. Knoten, markiere ihn als besucht und füge ihn zum Stack hinzu und kennzeichne Kante als treeedge. Wenn kein noch nicht besuchter Nachbar mehr gefunden wird entferne den Knoten vom Stack und gehe zum vorherigen Knoten im Pfad zurück. Wenn alle Knoten besucht wurden.

Laufzeit:

- es müssen alle möglichen Pfade zu allen möglichen Knoten betrachtet werden => O(|V|+ |E|)

Datenstrukturen

Stack

Tiefensuche bestimmt einen aufspannenden Baum zu einem gegebenen Graphen. Man startet bei einem Startknoten v; diesen markiert man. Die Markierung wird genutzt um zu wissen, welche Knoten schon besucht wurden. Von v geht man alle Kanten e, die von v ausgehen, durch. Ist der Zielknoten e_u von e noch nicht markiert, färbt man e als TreeEdge und führt rekursiv Tiefensuche mit e_u als Startknoten aus. Andernfalls färbt man einfach e als OtherEdge.

Toposort- Vorlesungen

Konstruiere Graph G:

- Erstelle für jede Vorlesung v_i einen Knoten i in G
- erstelle für jedes Voraussetzungspaar(v_i, v_j) eine gerichtete Kante (i,j) in G
- Wende Algorithmus toposort auf Graph G an

=> da die Vorlesungsvoraussetzungen einer Vorgängerbeziehung entspricht wird mit der topologischen Sortierung eine gültige Reihenfolge gefunden.
Die Laufzeit der Graphenkonstruktion ist in O(n+m) wobei n die Anzahl der Vorlesungen ist und m die Anzahl der Vorraussetzungspaare.
Dazu kommt die Laufzeit von toposort, die ebenfalls in O(n+m) ist

minimaler Pfad von allen Knoten zu einem Knoten a

Dijskstra-Algorithmus für Graph mit Kantengewichten

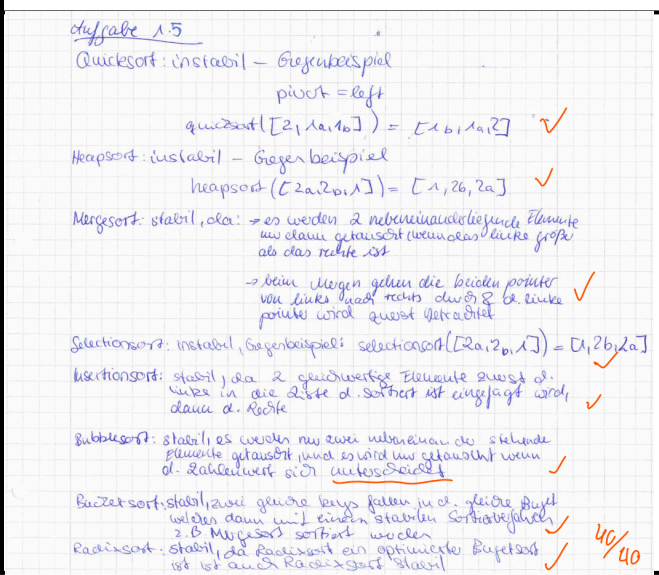
Breitensuche für Graph ohne Kantengewichte

Breitensuche

Breitensuche bestimmt minimal aufspannenden Baum in Graph nach folgendem Verfahren: Markiere zuerst Starknoten als besucht. Füge Startknoten in die Warteschlange ein. Solange die Warteschlange nicht leer ist, wähle den ersten Knoten u aus der Warteschlange und gehe alle Kanten durch die bisher noch als unclassified_edge makiert sind. Wenn die Kante u mit einem noch nicht besuchten Nachbarn verbindet markiere die Kante zwischen den besuchten Nachbarn und u als treeEdge und füge Nachbarn der Warteschlange hinzu und makiere ihn als besucht. Anderenfalls färbe Kante als other_edge.

Laufzeit: O(|V|+ |E|) jede Kante und jeder Knoten wird genau einmal besucht

Breitensuche bestimmt einen aufspannenden Baum zu einem gegeben Graphen. Man initialisiert zunächst eine leere Queue q. Man startet bei einem Startknoten v; diesen markiert man. Die Markierung wird genutzt um zu wissen, welche Knoten schon besucht wurden. Von v geht man alle Kanten e, die von v ausgehen, durch. Ist der Zielknoten e_u von e noch nicht markiert, färbt man e als TreeEdge, packt e_u an das Ende von q, und markiert e_u. Andernfalls färbt man e als OtherEdge. Schließlich nimmt man den ersten Knoten v aus q heraus und führt den ganzen Prozess mit v als Startknoten wieder durch. Kann man dies nicht machen, weil q leer ist, so sind wir fertig.



Quicksort

Erklärung: Wähle ein Pivot(z.B. das mittlere Element) und vertausche die Elemente s.d. am Ende eines Aufrufs alle kleineren Elemente links, alle größeren rechts stehen. Dies passiert folgendermaßen: es wird immer das am weitestens links stehende Element welches >= dem Pivot ist mit dem am weitest rechts stehenden Element was <= pivot ist getauscht. Am Ende dieses Schrittes steht das Pivot Element auf jeden Falll an seinem endgültigen Platz im Array. Danach wird Quicksort rekursiv auf das linke und rechte Teilfeld angewendet.

Laufzeit: O(n^2)(worst case) wenn feld sortiert ist
durchschnittliche Laufzeit O(n*log n)