> ***Frage**: Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?*

Alle oben genannten Probleme sind in NP und damit schnell nichtdeterministisch lösbar. Die besten bekannten deterministischen Algorithmen arbeiten allerdings im Grunde so wie oben im allgemeinen Fall illustriert mit einer Brute-Force-Methode und benötigen daher expoentielle Laufzeit.

Die Aussage stimmt nicht ganz, da mittlerweile für etliche NP-Probleme auch etwas bessere Algorithmen bekannt sind. Diese Algorithmen sind allerdings weiterhin exponentiell, so dass die generelle Aussage gültig bleibt. Aus Praxissicht sind diese Algorithmen dann aber immerhin etwas besser als die Brute-Force-Methode und tatsächlich ist es ein aktives Foschungsfeld hier Algorithmen zu entwickeln, die mit immer größeren Eingabeinstanzen fertig werden.

Die Frage ist nun, geht es wirklich nicht schneller? Wenn wir dies beantworten könnten, so könnten wir entweder versuchen schnellere Algorithmen zu entwerfen oder können dies ohne schlechtes Gewissen unterlassen. Um die Suche nach unteren Schranken soll es im nächsten Abschnitt zur NP-Vollständigkeit gehen.

5.3 NP-Vollständigkeit

Für die erwähnten Probleme aus NP des vorherigen Abschnittes wie z.B. das Mengenpartitionsproblem sind keine deterministischen Algorithmen in Polynomialzeit bekannt. Wir würden dies nun gerne zeigen können, d.h. wir würden gerne zeigen können, dass sich das Problem für kein k in $O(n^k)$ lösen lässt. Leider fehlen uns hierfür trotz intensiver Forschung die nötigen Techniken. Wir verfolgen daher einen anderen Ansatz. Wir zeigen für ein Problem X (von dem wir vermuten, dass es nicht in P ist), dass es bestimmte Eigenschaften hat, so dass, sollte X doch in P liegen, sehr unwahrscheinliche Dinge folgen würden.

5.3.1 Reduktionen

Um unser oben beschriebenes Ziel zu erreichen, wollen wir von einem Problem sagen können, dass es zu den schwierigsten Problemen seiner Klasse gehört. Hierzu brauchen wir den Begriff der Reduktion.

Definition 5.3.1 (Reduktion)**.** *Seien $L_1, L_2 \subseteq \{0,1\}^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2* in polynomialer Zeit reduziert wird*, wenn eine in Polynomialzeit berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$ existiert mit*

$$x \in L_1 \text{ genau dann wenn } f(x) \in L_2$$

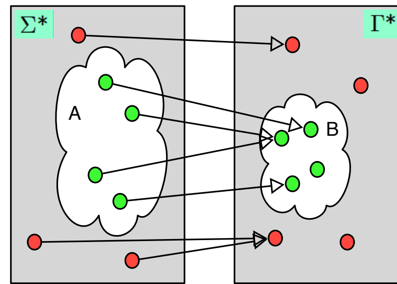

Abbildung 5.1: Visualisierung einer Reduktion

für alle $x \in \{0,1\}^*$.

Hierfür schreiben wir dann $L_1 \leq_p L_2$. f *wird als* Reduktionsfunktion, *ein Algorithmus, der* f *berechnet, als* Reduktionsalgorithmus *bezeichnet.*

Andere Symbole für die Reduktion sind $L_1 \leq_{pol} L_2$ oder auch $L_1 \leq_m^p L_2$. Das m steht dabei für "many-one", da man zwei (oder mehr) $x, y \in L_1$ auf das gleiche $f(x) = f(y) = u \in L_2$ abbilden darf. (Ebenso darf man zwei (oder mehr) $x', y' \notin L_1$ auf das gleiche $f(x') = f(y') = u' \notin L_2$ abbilden.)

Allgemeiner kann man eine Reduktion zu zwei Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ als eine Funktion $f : \Sigma^* \to \Gamma^*$ mit $x \in A$ gdw. $f(x) \in B$ für alle $x \in \Sigma^*$ definieren. Den Sprachen können also verschiedene Alphabete zugrunde liegen und die Reduktion muss (zunächst) nicht in Polynomialzeit möglich sein. Man kann dann unterschiedliche Zeitreduktionen einführen und so z.B. auch P-vollständige Probleme definieren (was dann die schwierigsten Probleme in P sind).

Hat man eine Reduktion $f : \{0,1\}^* \to \{0,1\}^*$ von L_1 auf L_2, so ist das wichtige, dass jedes $x \in L_1$ (man spricht hier auch von *Ja-Instanzen*, da dies ein x ist, bei dem eine Turingmaschine für L_1 (sofern eine existiert) mit *ja* antworten würde) auf ein $y \in L_2$ abgebildet wird, also auf eine Ja-Instanz von L_2. Ja-Instanzen werden also auf Ja-Instanzen und Nein-Instanzen auf Nein-Instanzen abgebildet. Die Fragen "Ist $x \in L_1$?" und "Ist $f(x) \in L_2$?" haben also stets die gleiche Antwort. Dies werden wir gleich ausnutzen. Abbildung 5.1 verdeutlicht noch einmal eine Reduktion. Wir haben hier den allgemeinen Fall zweier Sprachen $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ skizziert. Man beachte, dass Ja-Instanzen auf Ja-Instanzen und Nein-Instanzen auf Nein-Instanzen abgebildet werden und dass mehrere Ja-Instanzen auf eine Ja-Instanz abgebildet werden können (und genauso mehrere Nein-Instanzen auf eine Nein-Instanz).

Da Ja-Instanzen auf Ja-Instanzen abgebildet werden und Nein-Instanzen auf Nein-Instanzen und da die Reduktion per Definition berechenbar ist lassen sich Reduktionen ausnutzen, um Probleme durch andere zu lösen. Die Kernidee dabei ist, erst die Reduktion zu benutzen und dann das Problem, auf das reduziert

wurde, zu lösen (und damit dann auch das ursprüngliche). Der folgende Satz konkretisiert dies.

Satz 5.3.2. *Seien $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, dann folgt aus $L_2 \in P$ auch $L_1 \in P$.*

Beweis. Wir wissen aus den Voraussetzungen zunächst, dass wegen $L_1 \leq_p L_2$ eine in Polynomialzeit berechenbare Reduktionsfunktion f existiert mit $x \in L_1$ genau dann, wenn $f(x) \in L_2$. Ferner gibt es wegen $L_2 \in P$ einen Algorithmus A_2, der L_2 in Polynomialzeit entscheidet.

Wir müssen damit zeigen, dass es einen Algorithmus A_1 gibt, der L_1 in Polynomialzeit entscheidet. Die Idee ist zunächst die Reduktion zu berechnen und dann A_2 zu benutzen. Im einzelnen arbeitet A_1 auf einer Eingabe $x \in \{0,1\}^*$ wie folgt:

1. Berechne $f(x)$.

2. Nutze A_2, um $f(x) \in L_2$ zu entscheiden.

3. Antworte genau so wie A_2.

Da f berechenbar ist, klappt der erste Schritt. Da A_2 ebenfalls ein Algorithmus ist, der stets terminiert, klappt auch der zweite Schritt. Und zuletzt produziert der dritte Schritt wegen $f(x) \in L_2$ gdw. $x \in L_1$ gerade die richtige Antwort.

Wir sind damit fast feritg. Wir müssen noch zeigen, dass A_1 auch in Polynomialzeit arbeitet (bisher haben wir nur gezeigt, dass er die korrekten Ergebnisse liefert). Die richtige Laufzeit sieht man so ein: Zunächst kann f in Polynomialzeit berechnet werden. Daher ist die Länge des Wortes, das bei der Reduktion berechnet wird (also die Länge von $f(x)$), polynomiell in x beschränkt, d.h. es ist $|f(x)| \in O(|x|^c)$ (mit einer Konstanten c). Die Eingabe von A_2 ist $f(x)$, daher ist zur Berechnung der Laufzeit von A_2 dann $|f(x)|$ relevant. Die Laufzeit von A_2 ist durch $O(|f(x)|^d) = O(|x|^{c \cdot d})$ beschränkt, wobei d wieder eine Konstante ist. Damit ist eine obere Schranke für die Laufzeit von A_1 dann $O(|x|^c + |x|^{c \cdot d}) = O(|x|^{c \cdot d})$ und somit läuft A_1 in Polynomialzeit und es ist $L_1 \in P$. □

Der obige Satz erlaubt es ein Problem (das L_1 in obigem Satz) durch ein anderes (das L_2) zu lösen. Darum auch die Benennung *Reduktion*. Statt einen Algorithmus für L_1 zu finden und so L_1 zu lösen, nutzt man einen für L_2 und löst so nicht nur L_2, sondern (dank des Reduktionsalgorithmus) auch L_1. Das Problem, L_1 zu lösen, ist also darauf "reduziert" worden, das Problem L_2 zu lösen. Man beachte aber, dass ein Reduktionsalgorithmus auch nichts anderes ist als ein ganz normaler Algorithmus. Die Aufgabe eines Reduktionsalgorithmus ist es aber, eine Probleminstanz in eine andere umzuwandeln. Also z.B. eine Instanz eines Graphenproblems in eine Instanz eines zahlentheoretischen Problems. (Man kann daher eine Reduktion auch als *Transformation* bezeichnen.)

Wenn wir nun für zwei Probleme A und B zeigen, dass $A \leq_p B$ gilt, so wissen wir, dass A höchstens so schwierig wie B ist (höchstens bezieht sich hier auf den polynomiellen Mehraufwand, der hier (im Falle von Problemen in P und NP) als akzeptabel angsehen wird). A kann nämlich nicht schwieriger sein, denn sonst löst man A einfach indem man erst die Reduktion benutzt und dann B löst. Daher auch die Notation mit dem Kleiner-Gleich-Zeichen. Reduziert man nun jede Sprache aus NP auf eine (neue) Sprache L, so ist L mindestens so schwierig wie *ganz* NP, denn löst man nun L, so kann man (mittels des Umwegs über die Reduktion) nun jedes Problem aus NP lösen. Das macht dann $L \in NP$ sehr unwahrscheinlich, denn dann würde $P = NP$ gelten, was als sehr unwahrscheinlich angesehen wird, da sich Generationen von Wissenschaftlern mit den verschiedenen Problemen in NP beschäftigt haben und nie für irgendeines dieser Probleme auf einen Algorithmus in P gekommen sind. – Es gibt noch weitere gute Gründe, warum $P = NP$ unwahrscheinlich ist, diese führen hier aber etwas zu weit. Wir merken uns an dieser Stelle, dass es gute Gründe gibt $P \neq NP$ anzunehmen. Bewiesen ist dies aber nicht!

Wir wollen nun obiges mit dem Begriff der NP-Vollständigkeit formalisieren und dann an Beispielen illustrieren, wie uns dies helfen kann für Probleme zumindest zu zeigen, dass es unter der Annahme $P \neq NP$ keinen Algorithmus in Polynomialzeit geben kann. Da die Annahme ein recht starkes Fundament hat, ist dies dann schon eine recht starke Aussage.

5.3.2 Definition der NP-Vollständigkeit

Wir führen den wichtigen Begriff der NP-Vollständigkeit ein, der in der theoretischen Informatik und der Algorithmik eine fundamentale Bedeutung hat.

Definition 5.3.3. *Eine Sprache $L \subseteq \{0,1\}^*$ wird als NP-vollständig bezeichnet, wenn*

1. *$L \in NP$ und*

2. *$L' \leq_p L$ für jedes $L' \in NP$ gilt.*

Kann man für L zunächst nur die zweite Eigenschaft beweisen, so ist L NP-schwierig (-schwer/-hart).

Alle NP-vollständigen Probleme bilden die Komplexitätsklasse NPC.

Da wir dies nun formal definiert haben, können wir den folgenden wichtigen Satz beweisen:

Satz 5.3.4. *Sei $L \in NPC$. Ist nun $L \in P$, so ist $NP = P$.*

Beweis. Wir erinnern an den Satz aus dem letzten Abschnitt: Sind $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, dann folgt aus $L_2 \in P$ auch $L_1 \in P$.

Sei hier nun $L \in NPC$ und $L \in P$ also $L \in NPC \cap P$. Wir wollen $P = NP$ zeigen. Dabei ist $P \subseteq NP$ klar. Zu zeigen ist also noch $NP \subseteq P$. Sei dazu nun $L' \in NP$. Wegen $L \in NPC$ gilt $L' \leq_p L$ und aus $L \in P$ folgt mit dem oben wiederholten Satz aus dem letzten Abschnitt $L' \in P$ und wir sind bereits fertig. □

Eine äquivalente Formulierung des Satzes von eben ist: Gibt es ein $L \in NP \setminus P$, so ist $NPC \cap P = \emptyset$. Es ist eine gute Übung sich zu überlegen, warum dies gilt.

Der letzte Satz rechtfertigt nun gerade die Aussage, dass ein Problem in NPC (also ein NP-vollständiges Problem) höchstwahrscheinlich nicht effizient lösbar ist (also in P ist). Dann würde nämlich $P = NP$ gelten und damit wären alle Probleme in NP (darunter auch all die komplizierten aus NPC) effizient lösbar (in P), was aller Erfahrung widersprechen würde.

Wir betonen noch einmal, dass es sehr gute Gründe gibt, $P \neq NP$ anzunehmen und dass es daher eine sehr starke Aussage ist, wenn man sagt, dass ein bestimmter Sachverhalt nur gilt, wenn $P = NP$ ist. Der Sachverhalt kann dann praktisch ausgeschlossen werden. Trotzdem ist es nicht sicher, dass $P = NP$ gilt. Man ist sich in dieser Hinsicht allerdings so sicher, dass teilweise Sicherheitsprotokolle auf dieser Annahme basieren.

Unser Ziel ist es nun Probleme als NP-Vollständig nachweisen zu können. Wir könnten dann immer versuchen einen schnellen Algorithmus für ein neues Problem zu finden und, sollte dies nicht klappen, versuchen zu zeigen, dass das Problem NP-vollständig ist. Gelingt letzteres, können wir im Allgemeinen aufhören zu versuchen einen schnellen Algorithmus zu finden (und können beginnen andere Ansätze zu verfolgen, z.B. Algorithmen entwickeln, die nur eine fast optimale Lösung liefern und ähnliches). Um dies zu erreichen, zeigen wir zunächst noch zwei weitere wichtige Sätze (mit den obigen sind es dann drei wichtige). Der erste Satz sagt uns etwas über die Transitivität von $\leq_p$, was man auch interpretieren kann als eine Hintereinanderausführung von Reduktionsalgorithmen. Der zweite Satz sagt uns darauf aufbauend etwas darüber, wie Probleme als NP-vollständig nachgewiesen werden können.

Satz 5.3.5. *Ist $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, so ist $L_1 \leq_p L_3$.*

Beweis. Das Argument ist ähnlich wie bei dem Beweis, dass $L_1 \in P$ aus $L_1 \leq_p L_2$ und $L_2 \in P$ folgt. Seien f und g die Reduktionsfunktionen aus $L_1 \leq_p L_2$ bzw. $L_2 \leq_p L_3$. Bei Eingabe x mit $|x| = n$ berechnen wir zunächst $f(x)$ in Polynomialzeit $p(n)$. Dann berechnen wir $g(f(x))$ in Zeit $q(|f(x)|) \leq q(p(n))$. Insgesamt ist der Aufwand dann bei Eingaben der Länge n durch $p(n) + q(p(n))$ nach oben beschränkt, was ein Polynom ist. (Die Eigenschaft $x \in L_1$ gdw. $(g \circ f)(x) \in L_3$ folgt direkt aus den gegebenen Reduktionen. Die hier gesuchte Reduktionsfunktion ist also $g \circ f$.) □

Satz 5.3.6. *Sei L eine Sprache und $L' \in NPC$. Gilt $L' \leq_p L$, so ist L NP-schwierig. Ist zusätzlich $L \in NP$, so ist L NP-vollständig.*

Beweis. Wegen $L' \in NPC$ gilt $L'' \leq_p L'$ für jedes $L'' \in NP$. Aus $L' \leq_p L$ und dem vorherigen Satz folgt dann $L'' \leq_p L$, L ist also NP-schwierig. Ist zusätzlich $L \in NP$, so ist L nach Definition NP-vollständig. □

Aus dem letzten Satz lässt sich ein Verfahren ableiten, wie vorgegangen werden kann, wenn man von einem neuen Problem L zeigen will, dass es NP-vollständig ist.

1. Zeige $L \in NP$.
2. Wähle ein $L' \in NPC$ aus.
3. Gib einen Algorithmus an, der ein f berechnet, das jede Instanz $x \in \{0,1\}^*$ von L' auf eine Instanz $f(x)$ von L abbildet (also eine Reduktion).
4. Beweise, dass f die Eigenschaft $x \in L'$ gdw. $f(x) \in L$ für jedes $x \in \{0,1\}^*$ besitzt.
5. Beweise, dass f in Polynomialzeit berechnet werden kann.

Hierbei zeigen die letzten drei Punkte $L' \leq_p L$. Mit dem letzten Satz folgt daraus und aus den ersten beiden Punkten dann $L \in NPC$.

Das Problem ist nun, dass, um im zweiten Schritt des Verfahrens ein $L' \in NPC$ auswählen zu können, man erstmal welche haben muss! Je mehr man hier kennt, desto besser ist es später, aber ein erstes brauchen wir und dort werden wir tatsächlich *alle* Probleme aus NP auf dieses reduzieren müssen! (Denn für das erste Problem können wir obiges Verfahren noch nicht anwenden, da der zweite Schritt nicht möglich ist.)

Als erstes NP-vollständige Problem nehmen wir das auch historisch erste NP-vollständige Problem nämlich das *Erfüllbarkeitsproblem der Aussagenlogik* `SAT` (für *Satisfiability*).

Definition 5.3.7 (`SAT`)**.** *Das Erfüllbarkeitsproblem der Aussagenlogik ist definiert als das Entscheidungsproblem `SAT` mit*

$$\texttt{SAT} = \{\langle\phi\rangle \mid \phi \textit{ ist eine erfüllbare aussagenlogische Formel}\}$$

Satz 5.3.8. *`SAT` ist NP-vollständig.*

Beweis. Um `SAT` $\in NP$ zu zeigen, raten wir gegeben eine Formel ϕ eine Belegung (es gibt 2^n viele bei n verschiedenen Variablen in ϕ) und verifizieren in Polynomialzeit, ob sie die Formel erfüllt.

Um zu zeigen, dass `SAT` vollständig ist für NP müssen wir alle Probleme aus NP auf `SAT` reduzieren. Die Beweisidee ist zu einer Sprache $L \in NP$ die

Turingmaschine M mit $L(M) = L$ zu betrachten und alle Kopfbewegungen und damit mögliche Konfigurationsübergänge in einer Formel zu kodieren. Ist diese Formel erfüllbar, gibt es eine akzeptierende Rechnung, sonst nicht. Die genaue Konstruktion ist interessant, führt hier aber zu weit. Man findet den Beweis z.B. im Buch von Hopcroft, Ullman und Motwani. □

Hat man nun erstmal ein NP-vollständiges Problem, so kann man (dem Plan oben entsprechend) nun dieses benutzen, um weitere Probleme als NP-vollständig nachzuweisen. Der umständliche Weg *alle* NP-Probleme auf ein neues zu reduzieren entfällt so (bzw. man kriegt dies insb. wegen der Transitivität von $\leq_p$ geschenkt). Je größer dann der Vorrat an NP-vollständigen Problemen ist, desto größer ist die Auswahl an Problemen, von denen man eine Reduktion auf ein neues Problem, dessen Komplexität noch unbekannt ist, versuchen kann. (Es gibt natürlich immer die Möglichkeit alle Probleme aus NP auf ein neues zu reduzieren, aber i.A. wird dies nicht gemacht, sondern wie im Verfahren oben beschrieben vorgegangen.)

Wir wollen neben `SAT` noch zwei weitere Probleme als NP-vollständig voraussetzen. Die Beweise finden sich wieder im Buch von Hopcroft, Ullman und Motwani.

Satz 5.3.9. *Die Probleme*

$$\texttt{CNF} = \{\langle\phi\rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Formel in KNF}\}$$

und

$$\texttt{3CNF} = \{\langle\phi\rangle \mid \phi \in CNF, \text{jede Klausel hat genau drei verschiedene Literale}\}$$

sind NP-vollständig.

Beweis. $\texttt{CNF}, \texttt{3CNF} \in NP$ ist klar. Man kann dann $\texttt{SAT} \leq_p \texttt{CNF}$ und $\texttt{CNF} \leq_p \texttt{3CNF}$ zeigen (siehe wieder das Buch von Hopcroft, Ullman und Motwani). □

Hierauf aufbauend wollen wir nun unser erstes Problem tatsächlich als NP-vollständig nachweisen. Dies ist das Problem eine Clique in einem Graphen zu finden.

Definition 5.3.10 (Clique)**.** *Das Cliquenproblem ist definiert als*

$$\texttt{Clique} = \{\langle G, k\rangle \mid G \text{ enthält einen } K^k \text{ als Teilgraphen}\}$$

Dabei ist ein K^k eine k-Clique, d.h. ein (Teil-)Graph bestehend aus k Knoten, wobei alle Knoten paarweise miteinander verbunden sind.

Ein Dreieck ist z.B. eine 3-Clique. Für eine 5-Clique nimmt man 5 Knoten und verbindet je zwei Knoten miteinander. Bei obigen Clique-Probleme erhält man einen Graphen G als Eingabe und eine Zahl k und die Frage ist, ob in dem Graph k Knoten zu finden sind, die alle paarweise miteinander verbunden sind. Der nachfolgende Satz sagt uns, dass es nicht einfach ist, Cliquen zu bestimmen.

Satz 5.3.11. *`Clique` ist NP-vollständig.*

Beweis. Als Zertifikat nehmen wir eine Menge $V' \subseteq V(G)$ von Knoten, die eine Clique bilden. Dieses Zertifikat ist polynomial in der Eingabelänge und zudem lässt sich leicht in polynomialer Zeit prüfen, ob alle Knoten verbunden sind, indem man für je zwei Knoten u, v aus V' einfach testet, ob $\{u, v\}$ eine Kante in $E(G)$ ist. Dies zeigt `Clique` $\in NP$.

Nun zeigen wir noch `3CNF` $\leq_p$ `Clique`. Sei dazu $\phi = C_1 \wedge \ldots \wedge C_k$ eine Instanz von `3CNF` mit k Klauseln. Seien ferner l_1^r, l_2^r, l_3^r für $r = 1, 2, \ldots, k$ die drei verschiedenen Literale in der Klausel C_r. Wir konstruieren eine Instanz (G, k) von `Clique` wie folgt: Zu jeder Klausel $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ nehmen wir ein Tripel v_1^r, v_2^r, v_3^r in V auf. Zwei Knoten v_i^s und v_j^t sind nun genau dann miteinander verbunden, wenn $s \neq t$ gilt und die zugehörigen Literale nicht zueinander komplementär sind (d.h. das eine ein positives das andere ein negatives Literal der selben Variable ist). Der Wert k der Instanz von `Clique` entspricht der Anzahl der Klauseln von ϕ.

Sei zur Illustration die Formel ϕ durch

$$\phi = (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z)$$

gegeben.

Als Graph ergibt sich nach der Konstruktion folgender Graph. Die drei Clique ist bereits eingezeichnet. Eine erfüllende Belegung macht y falsch, z wahr und wertet x beliebig aus.

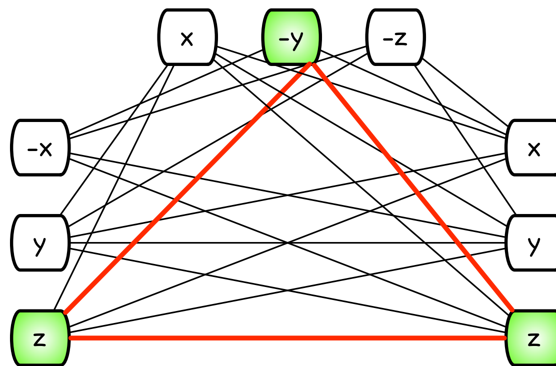

Abbildung 5.2: Visualisierung der Reduktion von SAT auf Clique

Diese Konstruktion ist in Polynomialzeit möglich, da man durch einmal lesen der Formel die auftretenden Variablen und Klauseln kennt und so die Knoten erzeugen kann. Die Kanten erzeugt man dann schlimmstenfalls in dem man jeden Knoten mit allen Knoten der anderen Tripel vergleicht und prüft ob die zugehörigen Literale komplementär sind. Sind sie es nicht, fügt man eine Kante hinzu. Dies geht dann in $O(V^2) = O(\phi^2)$.

Wir müssen noch zeigen, dass dies wirklich eine Reduktion ist, der gegebene Graph also genau dann eine Clique enthält, wenn die Formel erfüllbar ist. Sei die Formel erfüllbar, dann gibt es eine Belegung die in jeder Klausel mindestens ein Literal wahr macht. Nimmt man nun aus jeder Klausel eines dieser wahren Literale und dann die jeweils zugehörigen Knoten aus den Tripeln so hat man eine k-Clique, denn es sind k Knoten (da es k Klauseln sind) und zu zwei Knoten v_i^r, v_j^s gilt $r \neq s$ (da die Literale aus verschiedenen Klauseln, die Knoten also aus verschiedenen Tripeln gewählt wurden) und ferner sind die zu den Knoten gehörigen Literale nicht komplementär, da die Belegung dann nicht beide wahr machen könnte. Die Knoten sind also durch eine Kante verbunden.

Gibt es andersherum eine k-Clique V', so muss jeder Knoten aus einem anderen Tripel sein, da die Knoten in einem Tripel nicht miteinander verbunden sind. Wir können nun dem zu einem Knoten $v_i^r \in V'$ zugehörigem Literal l_i^r den Wert 1 zuweisen ohne dadurch dem Literal und seinem Komplement den Wert 1 zuzuweisen, da dann zwei Knoten in V' sein müssten, die nicht miteinander verbunden wären (was nicht sein kann, da V' eine Clique ist). Damit ist dann jede Klausel erfüllt, da aus jedem Tripel ein Knoten und damit aus jeder Klausel ein Literal beteiligt ist und wir haben somit eine erfüllende Belegung. Damit ist alles gezeigt. □

Wir haben nun ein weiteres NP-vollständiges Problem, das wir nutzen können, um weitere Probleme als NP-vollständig nachzuweisen. Wir wollen zur Illustration noch ein weiteres recht ähnlich aussehendes Problem betrachten, bei dem der Beweis recht knifflig wird.

Definition 5.3.12 (Big-Clique)**.** *Im Problem `Big-Clique` ist ein ungerichteter Graph G gegeben. Die Frage ist, ob G eine Clique enthält, die aus mindestens $m/2$ Knoten besteht, wobei m die Anzahl der Knoten von G ist.*

Satz 5.3.13. *`Big-Clique` ist NP-vollständig.*

Beweis. Dass `Big-Clique` in NP ist, sieht man leicht ein: Das Zertifikat sind $m/2$ Knoten. Der Verifikationsalgorithmus überprüft dann, ob diese $m/2$ Knoten tatsächlich eine Clique bilden.

Wir geben nun eine Reduktion von `Clique` auf `Big-Clique` an. Sei $\langle G, k\rangle$ eine Instanz von Clique, wobei G m Knoten habe. Wir wollen eine Instanz H von `Big-Clique` erzeugen.

Nach einigem probieren, merkt man, dass der Fall $k = m/2$ einfach ist. In diesem Fall kann man als H einfach direkt G nehmen. G hat dann nämlich eine k-Clique genau dann, wenn H $(= G)$ eine Clique hat, die aus mindestens der Hälfte der Knoten besteht. Es sind m Knoten, also ist k gerade die Hälfte.

Weitere Überlegungen führen dazu, dass die Fälle $k > m/2$ und $k < m/2$ auch getrennt behandelt werden müssen. Ist nämlich $k < m/2$, so müssen, sollte G eine k-Clique enthalten, zu G weitere Knoten hinzugefügt werden, die mit den k Knoten zusammen eine größere Clique bilden. Ist hingegen $k > m/2$,

so müssen, sollte G eine k-Clique enthalten, zu G isolierte Knoten hinzugefügt werden, damit die Clique (in H) nicht zu groß ist.

Wir behandeln die Fälle nun genauer.

Fall $k < m/2$. Hat man hier in G eine k-Clique (Ja-Instanz), so kann es sein, dass die größte Clique tatsächlich den Wert k hat und daher für H zu klein ist. Als H also einfach G zu nehmen wie im Falle $k = m/2$ klappt nicht. Wir müssen die Clique irgendwie vergrößern (und damit auch den Graphen G anpassen). Die gradlinige Idee ist zu G eine $(m/2 - k)$-Clique hinzuzufügen und diese mit allen anderen Knoten zu verbinden. Dann bilden nämlich die ursprüngliche k-Clique und die neue $(m/2 - k)$-Clique eine Clique der Größe $m/2-k+k = m/2$. Das sieht gut aus, *aber* das Problem ist, dass wir ja gerade einige Knoten hinzugefügt haben. In dem so entstandenen Graphen H ist daher $m/2$ gar nicht mehr die Hälfte der Knoten! Wir müssen also noch mehr Knoten hinzufügen (also eine größere Clique hinzufügen). Um darauf zu kommen, wie viele Knoten hinzugefügt werden müssen, machen wir folgendes: Bisher hatten wir einen Graphen G mit m Knoten und einer k-Clique. Die Anzahl der Knoten, die wir wie oben beschrieben hinzufügen wollen, nennen wir j. Wir wollen dann, dass $k + j$ (die Größe der neuen Clique) gerade $(m + j)/2$ ist (das ist nämlich gerade die Hälfte der nun vorhandenen $m + j$ Knoten im Graphen!). Setzen wir $k + j = (m + j)/2$ und lösen zu j auf, erhalten wir $j = m - 2k$. Unsere bisherigen Überlegungen führen zu dieser Zahl und zu der Behauptung, dass, wenn wir so viele Knoten wie oben beschrieben hinzufügen gerade gilt, dass G genau dann eine k-Clique enthält, wenn H eine Clique aus mindestens der Hälfte seiner Knoten enthält. Um ganz sicher zu sein, dass das oben ermittelte j passt, muss man dies noch beweisen.

Sei also angenommen G hat eine k-Clique. Dann bilden aber die $k+j$ Knoten mit den j neuen Knoten eine $k+j$-Clique in H und da $k+j = k+m-2k = m-k$ für die Größe der Clique und $m + j = m + m - 2k = 2m - 2k = 2(m - k)$ für die Anzahl der Knoten in H gilt, besteht die $k + j$-Clique gerade aus der Hälfte der Knoten in H und damit ist H in `Big-Clique`. Sei andersherum H in `Big-Clique`, dann gibt es wie bei obigen Rechnungen in H eine Clique aus mindestens $k + j$ Knoten (die Anzahl der Knoten in H ist $2(m - k)$, die Hälfte davon ist $m-k$ und wenn man oben die Gleichung von rechts nach links liest, ist $m - k = k + j$). Da nun nur j Knoten hinzugefügt wurden, müssen mindestens k Knoten dieser Clique schon in G vorhanden gewesen sein und dort eine k-Clique bilden, also ist $\langle G, k\rangle$ in `Clique`.

Noch ein Hinweis: Oben wurde $k + j$ gleich $(m + j)/2$ gesetzt. Streng genommen, will man ja, dass die Clique in H aus *mindestens* $|V(H)|/2$ Knoten besteht. Mit einer Gleichung zu arbeiten ist oft aber einfacher als mit einer Ungleichung und wir wollen ja einen festen Wert für j bestimmen. Außerdem besitzt H ja tatsächlich eine Clique, die mindestens die Größe $|V(H)|/2$ hat, wenn sie eine hat, die genau diese Größe hat.

Nun der Fall $k > m/2$. Hier geht es so ähnlich wie oben. Nun ist es aber so, dass man $\langle G, k\rangle$ nicht auf G abbilden kann, da eine Nein-Instanz von `Clique` evtl. eine Ja-Instanz von `Big-Clique` ist (Hat G z.B. 10 Knoten und hat G eine 6-Clique, so ist $\langle G, 8\rangle$ eine Nein-Instanz von Clique, aber G eine Ja-Instanz von `Big-Clique`.) Wir müssen es hier schaffen, zu G Knoten hinzuzufügen, damit die (mögliche) große Clique in G nur noch die Hälfte der Knoten (in H) ausmacht. Hatten wir vorhin $\langle G, 8\rangle$, so müssen wir dafür sorgen, dass H dann 16 Knoten hat. Dann wäre eine 8-Clique nun nur noch die Hälfte von H und wenn in G so eine nicht existiert, dann ist H nun auch eine Nein-Instanz von `Big-Clique`. Wir erreichen unser Ziel hier, indem wir isolierte Knoten hinzufügen. Wie viele hinzugefügt werden müssen, kann man sich ähnlich zu obigem überlegen. Sei j wieder die Anzahl der Knoten, die wir hinzufügen wollen (diesmal isolierte Knoten). Dann wollen wir nun, dass k (die Größe der Clique) gleich $(m + j)/2$ (die Hälfte der Größe von H) ist, also $k = (m + j)/2$. Aufgelöst zu j ergibt sich $j = 2k - m$. Wir müssen nun zeigen, dass mit dieser Wahl von j die Konstruktion tatsächlich funktioniert, dass also G genau dann eine k-Clique hat, wenn H wie oben konstruiert eine Clique hat, die aus mindestens der Hälfte der Knoten aus H besteht. Habe also G eine k-Clique. Dann ist das auch eine k-Clique in H, aber dort haben wir $m + j = m + (2k - m) = 2k$ Knoten und also macht die k-Clique gerade die Hälfte der Knoten in H aus. Ist andersherum H in `Big-Clique`, so gibt es eine Clique aus mindestens k Knoten (da H ja $2k$ Knoten hat wie oben). Diese Clique kann keinen der neuen Knoten enthalten, da die ja isoliert sind. Also sind diese k Knoten auch in G und bilden dort eine k-Clique.

In Gänze arbeitet die Reduktion also wie folgt: Zunächst wird die Anzahl m der Knoten von G ermittelt. Dann wird k mit $m/2$ verglichen und entsprechend eine der obigen drei Konstruktionen ausgeführt. Diese Reduktion geht in Polynomialzeit und damit haben wir `Clique` auf `Big-Clique` in Polynomialzeit reduziert, was mit obigen Argument, dass `Big-Clique` in NP ist, zeigt, dass `Big-Clique` NP-vollständig ist.

Ein deutlich schnellerer Beweis, wie man ihn von der Art her oft auch in der Literatur findet, ist nachfolgend aufgeführt. Man kann diesem Beweis Schritt für Schritt folgen, und ist dann von der Richtigkeit der Aussage überzeugt, man erhält aber aus dem Beweis wenig Anhaltspunkte dafür, wie man auf das j kommt. Je nach Zielgruppe kann der Beweis so formuliert aber auch genügen.

Der Beweis nun noch einmal deutlich schneller formuliert. Zunächst ist `Big-Clique` in NP klar. Wir unterscheiden nun drei Fälle:

1. $k = m/2$. Dann setzen wir einfach $H = G$ und es ist $\langle G, k\rangle \in$ `Clique` gdw. $\langle H\rangle \in$ `Big-Clique`.

2. $k < m/2$. Dann erhalten wir H, indem wir zu G $j := m - 2k$ Knoten hinzufügen, die alle paarweise miteinander verbunden sind und ferner mit jedem Knoten aus dem ursprünglichen G verbunden sind. H hat

damit $m + j = 2m - 2k$ Knoten. Hat H nun eine Clique aus mindestens $(m + j)/2 = m - k$ Knoten, so müssen mindestens $m - k - j = m - k - m + 2k = k$ dieser Knoten von den alten Knoten aus G sein und dort eine k-Clique bilden. Hat andersherum G eine k-Clique, so bilden diese Knoten zusammen mit den j neuen Knoten in H eine Clique, die aus der Hälfte der Knoten in H bestehen. Also gilt auch hier wie gewünscht $\langle G, k\rangle \in$ `Clique` gdw. $\langle H\rangle \in$ `Big-Clique`.

3. $k > m/2$. Dann erhalten wir H, indem wir zu G $j = 2k - m$ Knoten hinzufügen, die mit keinem anderen Knoten verbunden sind. H hat dann $m + j = 2k$ Knoten. Hat G nun eine k-Clique, so ist dies eine Clique in H, die aus der Hälfte der Knoten in H besteht. Hat andersherum H eine Clique aus mindestens k Knoten, so kann hier keiner der j neuen Knoten dabei sein. Alle Knoten der Clique sind also schon in G und bilden dort eine Clique, die sogar aus mehr als k-Knoten besteht (k Knoten dieser Clique bilden dann eine k-Clique). Auch hier gilt somit $\langle G, k\rangle \in$ `Clique` gdw. $\langle H\rangle \in$ `Big-Clique`

In der Reduktion bestimmen wir also einmal $m/2$, vergleichen den Wert dann mit k und konstruieren dann H wie oben angegeben. Die Konstruktion gelingt in Polynomialzeit (da die eben genannten Rechnungen schnell gelingen und auch die Konstruktion von H wie oben beschrieben leicht in Polynomialzeit möglich ist) und nach obigen Ausführungen gilt auch $\langle G, k\rangle \in$ `Clique` gdw. $\langle H\rangle \in$ `Big-Clique`, womit wir fertig sind. □

Neben den eben betrachten `Clique`- und `Big-Clique`-Problem sind auch die anderen im letzten Abschnitt eingeführten NP-Probleme, das Mengenpartitonsproblem, das Teilsummenproblem und das Färbungsproblem alle NP-vollständig. Daneben gibt es noch tausende weitere und für keines dieser Probleme ist ein effizienter Algorithmus (d.h. ein Algorithmus, der in Polynomialzeit läuft) bekannt. Dies bestärkt uns in dem Glauben, dass ein Nachweis von $L \in NPC$ bedeutet, dass L nicht in P liegt. Denn wäre es in P, würden all die tausenden von Probleme, für die kein effizienter Algorithmus gefunden werden konnte, plötzlich in P sein.

Daher kann man nun, gegeben ein Problem für das man einen Algorithmus entwickeln will, dies zunächst versuchen. Fallen einem aber nach einiger Zeit und etlichem Nachdenken stets nur Algorithmen ein, die *im Prinzip den ganzen Suchraum durchgehen*, so ist das Problem vermutlich NP-vollständig (oder schlimmer). Dies kann man dann versuchen nachzuweisen. Gelingt dies, kann man die Suche nach einem effizienten Algorithmus zugunsten von anderen Ansätzen verwerfen. Möglichkeiten diese Probleme dennoch zu attackieren sind z.B. Einschränkungen der Eingabe zu betrachten (vielleicht braucht man ja nur mit Bäumen zu arbeiten statt mit allgemeinen Graphen), Approximationsalgorithmen zu entwickeln (mit denen man i.A. nicht das Optimum ermittelt,

aber vielleicht einen Wert, der gut genug ist), randomisierte Algorithmen zu entwickeln (die manchmal kein Ergebnis liefern) oder Heuristiken zu entwickeln (bei denen sowohl die Laufzeit als auch die Güte des Ergebnisses meist unklar sind). Diese Möglichkeiten sind Inhalt von weiterführenden Veranstaltungen zur Algorithmik.

Fragen

1. Welche Aussage gilt unter der Annahme $P \neq NP$?
   a) $P \subsetneq NP \subsetneq NPC$
   b) $P \subsetneq NPC \subsetneq NP$
   c) $P \subsetneq NP$ und $NPC \subsetneq NP$ und $P \cap NPC = \emptyset$
   d) $P \subsetneq NPC$ und $NP \subsetneq NPC$
2. Sei NPH die Klasse der NP-schwierigen Probleme. Was gilt?
   a) $NPH \subseteq NPC$
   b) $NPC \subseteq NPH$
   c) $NPC \cap NPH = \emptyset$
   d) $NPC \cap NPH \neq \emptyset$ aber auch $NPC \setminus NPH \neq \emptyset$ und $NPH \setminus NPC \neq \emptyset$
3. Sei $L_{NPC} \in NPC$ und die Komplexität von $L_?$ unbekannt. Welche Reduktion müssen Sie zeigen, um $L_?$ als NP-vollständig nachzuweisen?
   a) $L_?$ auf L_{NPC} reduzieren
   b) L_{NPC} auf $L_?$ reduzieren
   c) Beide oben genannten Reduktionen
   d) Welche Richtung ist wegen der Eigenschaft der Reduktion ($x \in L_1$ gdw. $f(x) \in L_2$) egal.
4. Sei nochmal $L_{NPC} \in NPC$ und $L_?$ ein Problem mit unbekannter Komplexität. Was wissen Sie, wenn Sie doch $L_? \leq_p L_{NPC}$ zeigen?
   a) Nichts! (Zumindest nichts hilfreiches!)
   b) $L_? \in NPC$
   c) $L_? \in NP$
   d) L_{NPC} "erbt" die Komplexität von $L_?$, wenn wir diese ermittelt haben.