

BWINF 35, Runde 2 - Aufgabe 1

Rosinen picken

Robin Schmöcker

Inhaltsverzeichnis

1	Lösungsidee.....	3
2	Verworfenne Lösungsideen.....	4
2.1	Greedy	4
2.2	Brute Force	4
3	Programmteile	5
3.1	Zyklen entfernen.....	5
3.2	Positive Blätter kaufen (Regel 1a).....	8
3.3	Negative Wurzeln nicht kaufen (Regel 1b)	9
3.4	Positive Knoten mit nur einer ausgehenden Kante mit Zielknoten vereinen (Regel 2a).....	10
3.5	Negative Knoten mit einer nur eingehenden Kante mit Quellknoten vereinen (Regel 2b)	11
3.6	Die Null (0)	12
3.7	Vereinfachungsregeln Anwendung	12
3.8	Redundante Kanten entfernen	12
3.9	Graphen in zusammenhängende Graphen unterteilen	13
3.10	Maximale geschlossene Teilmenge bestimmen für restliche Knoten.....	14
3.10.1	Überführung zu Max-Flow	14
3.10.2	Max-Flow Problem lösen	15
3.10.3	Bestimmung der Teilmenge durch minimalen Schnitt	17
3.10.4	Beispiel.....	18
4	Programmablauf	19
4.1	Gesamter Programmablauf.....	19
4.2	Struktogramm	20
4.3	Anleitung GUI.....	21
4.4	Erstellung der Lösungsdatei zu vorgegebener Aufgabe	23
5	Lösungsbeispiele	24
5.1	Diverse Beispiele.....	24
5.2	Lösungen aller Beispielaufgaben der BwInf-Seite	29
6	Laufzeiten und Speicherverbrauch	30
6.1	Laufzeiten.....	30
6.1.1	Graphen unterteilen	30

6.1.2	Zyklen entfernen.....	30
6.1.3	Vereinfachungsregeln.....	30
6.1.4	Redundante Kanten entfernen	30
6.1.5	Maximales Closure bestimmen	31
6.1.6	Gesamtlaufzeit.....	31
6.2	Speicherverbrauch	31
6.2.1	Laden des Graphen	31
6.2.2	Graphen unterteilen	31
6.2.3	Zyklen entfernen.....	32
6.2.4	Vereinfachungsregeln.....	32
6.2.5	Redundante Kanten entfernen	32
6.2.6	Maximales Closure bestimmen	32
6.2.7	Gesamtspeicherverbrauch.....	33
6.3	Messdaten.....	33
6.3.1	Laufzeiten	33
6.3.2	Speicherverbrauch	36
6.4	Grenzen des Programms.....	37
7	Programm-Dokumentation	39
8	Quelltextauszüge	44

1 Lösungsidee

Das Firmenkonglomerat kann als gewichteter, gerichteter Graph interpretiert werden. Die Unternehmen stellen die Knoten dar, die Beziehungen zwischen den Unternehmen werden durch gerichtete Kanten dargestellt.

Eine optimale Lösung des Problems ist eine Menge von Knoten, die die Voraussetzung der Aufgabe erfüllt, also eine Kombinationen an Knoten, dessen Wert von keiner anderen erlaubten Kombination an Knoten überboten werden kann. Da mit jedem Knoten auch alle nachfolgenden Knoten gekauft werden müssen, sind zu jedem Knoten auch alle nachfolgenden Knoten in der Menge vorhanden. Daher führt jede ausgehende Kante der optimalen Lösung zu einem weiteren Knoten, der auch in der optimalen Lösung liegt. Somit kann aus so einer optimalen Menge von Knoten keine Kante aus der Menge herausgehen. Eine solche Menge nennt sich Closure bzw. geschlossene Teilmenge und repräsentiert die wertvollste gesuchte Teilmenge.

„In graph theory and combinatorial optimization, a closure of a directed graph is a set of vertices with no outgoing edges”

https://en.wikipedia.org/wiki/Closure_problem

Allgemeiner ist eigentlich jede erlaubte Lösung der Aufgabenstellung aus den eben angeführten Gründen ein Closure. Umgekehrt ist jedes Closure auch eine erlaubte Lösung, da es die Bedingungen der Firmenabhängigkeiten beinhaltet. Somit gilt es, dass die Suche nach der Lösung für die Aufgabe, der Suche nach einem maximalen Closure entspricht.

Das Closure-Problem ist für einen knotengewichteten, gerichteten Graphen, ein maximales Closure zu finden und entspricht somit unserer Aufgabe.

https://en.wikipedia.org/wiki/Closure_problem

Somit ist mein Lösungsansatz das Closure-Problem für den Ausgangsgraphen zu lösen. Um die Laufzeit des Programms zu verbessern, werden vorher diverse schnellere Verfahren angewendet, um das Problem bzw. den Graphen zu vereinfachen. Also Methoden, um die Knoten- und/oder Kantenanzahl zu reduzieren.

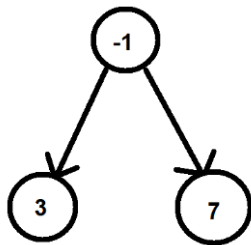
2 Verwerfene Lösungsideen

Hinweis: Ab diesem Kapitel werden der Einfachheit halber in den Beispielen die Knoten nicht mit ihrer ID + ihrem Wert gekennzeichnet, sondern es wird ausschließlich auf den Wert Bezug genommen. Die Beispiele wurden so ausgewählt und dargestellt, dass eigentlich keine Verwechslungen auftreten sollten, sind aber hierdurch deutlich übersichtlicher.

2.1 Greedy

Eine zunächst sinnvoll erscheinende Idee ist es immer den Knoten zu kaufen, der inklusive all seiner mitzukaufenden Knoten, den maximalen Gewinn verspricht. Ein solches Verfahren ist zwar sehr schnell und einfach zu implementieren, wird jedoch nicht immer eine korrekte Lösung liefern. Da schon bereits aus der Aufgabenstellung hervorgeht, dass es bei der Aufgabe gerade darauf ankommt eine maximale Lösung zu finden, wird dieser Ansatz verworfen.

Beispiel für die Fehleranfälligkeit des Greedy Ansatzes:



Um die möglichst größte Teilmenge hier zu erhalten, würde man Knoten 3 und 7 kaufen. Der Greedy Ansatz würde aber die -1 mitkaufen, da dieser zu Anfang den direkten größten Gewinn versprechen würde, da $9 (-1+3+7) > 3$ oder 7 .

2.2 Brute Force

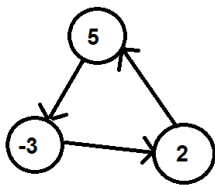
Eine weitere Lösung des Problems wäre die Anwendung eines Brute-force Algorithmus. Dieser könnte dann, wie bereits dem Namen zu entnehmen ist, alle möglichen Käufe ausprobieren und die Teilmenge kaufen, die den größten Wert erzielt. Wenn alle Möglichkeiten durchgegangen werden, ist davon auszugehen, dass der Algorithmus immer die korrekte Lösung liefern wird. Das Problem wird jedoch daran liegen, dass eine solche Lösung wegen der exponentiellen Laufzeit schnell an seine Grenzen stoßen wird (exponentiell, weil die Anzahl der möglichen Teilmengen $= 2^n$).

3 Programmteile

Im Folgenden sind zuerst Verfahren beschrieben, die die Knoten bzw. Kantenanzahl des Graphen verringern. Danach wird das abschließende Verfahren beschrieben zum Bestimmen der optimalen Teilmenge. In welcher Reihenfolge diese Verfahren letztlich angewendet werden, wird in Kapitel 5 erklärt.

3.1 Zyklen entfernen

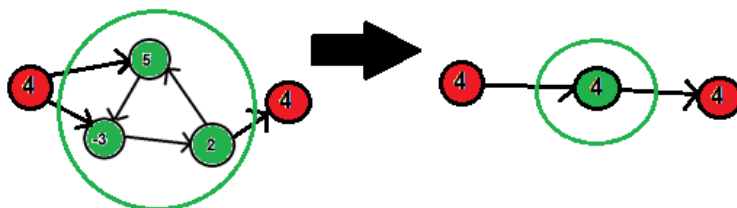
Ein Zyklus in einem Graphen beschreibt einen Weg, dessen Startpunkt dem Endpunkt entspricht. Bezogen auf das Firmenkonglomerat wäre dies eine Firma, deren Kauf, bedingt durch die Handelsbeziehungen, zum Kauf von sich selber führen würde. Folgendes Beispiel zeigt ein zyklisches Firmenkonglomerat.



Das Vereinigen aller Knoten eines Zyklus sorgt dafür, dass sich die Anzahl der Kanten und Knoten im Graphen verringert. Deshalb sollten alle Zyklen eines Graphen entfernt bzw. vereinfacht werden.

Würde man einen beliebigen Knoten eines Zyklus kaufen, müsste man automatisch alle anderen Knoten des Zyklus mitkaufen, sowie deren Abhängigkeiten. Würde man einen Knoten des Zyklus nicht kaufen wollen, so dürfte man auch keinen anderen Knoten kaufen, der Teil des Zyklus ist. Man kann einen Zyklus also so vereinfachen, indem man einen neuen Knoten erstellt, dessen Wert der Summe aller Knotenwerte des Zyklus entspricht. Alle aus- und eingehenden Kanten der Knoten des Zyklus werden nun mit dem neuen Knoten verbunden. Alle Knoten des Zyklus werden aus dem Graphen entfernt. Folgendes Beispiel veranschaulicht, wie ein Zyklus vereinfacht wird.

Vereinfachung des Graphen: - (Zyklengröße-1 Knoten), - mindestens so viele Kanten wie der Zyklus Knoten hat (eventuell mehr – siehe Beispiel ausgehende Kanten der 4 in den Zyklus). Wenn ein Graph viele Zyklen enthält, wird dieses Verfahren effektiv die Graphengröße verringern.



Zyklen werden mittels einer Tiefensuche entdeckt. Findet man während der Tiefensuche einen Knoten, der bereits in der gleichen Tiefensuche entdeckt wurde, so existiert ein Zyklus. Der Zyklus kann bestimmt werden indem für jeden Knoten der Tiefensuche gespeichert wird über welchen dieser aufgerufen wurde.

Allen Knoten sind während der Tiefensuche drei Stadien zuzuweisen:

1. Ein Knoten kann bereits entdeckt worden sein, jedoch wurde er noch nicht als nicht zu einem Zyklus gehörend identifiziert. Hat ein Knoten, der gerade von der Tiefensuche

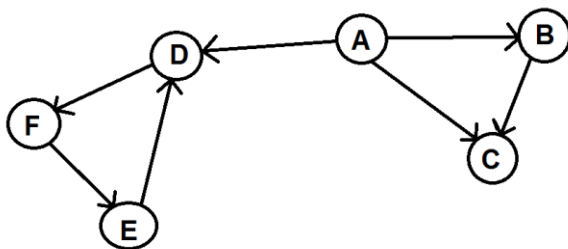
aufgerufen wurde, einen angrenzenden, erreichbaren Knoten in diesem Stadium, wurde ein Zyklus identifiziert.

2. Auch kann ein Knoten noch gar nicht von der Tiefensuche entdeckt worden sein.
3. Zuletzt kann ein Knoten zu keinem Zyklus gehörend klassifiziert werden. In diese Phase gerät der Knoten, wenn durch die Tiefensuche keiner seiner ausgehenden Knoten einen Weg zurück zum Knoten gefunden hat. Würde ein Knoten einen eindeutig nicht zu einem Zyklus gehörenden Knoten aufrufen, kann dieser einfach ignoriert werden, da von ihm kein Weg zum Knoten, der ihn aufrufen würde, gefunden werden kann.

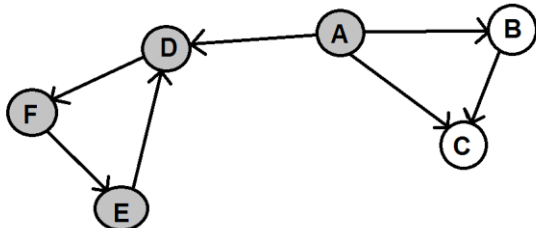
Wurde ein Zyklus durch die Tiefensuche identifiziert, so kann dieser wie oben beschrieben vereinfacht werden. Nachdem eine Tiefensuche ausgehend von einem Knoten ausgeführt wurde, können immer noch Zyklen im Graphen existieren. Es werden also solange Tiefensuchen gestartet, bis alle Knoten des Graphen (auch die neu Entstandenen durch das Zusammenfassen) als nicht zu einem Zyklus gehörend kategorisiert wurden. Weitere Tiefensuchen sollten aber nur von Knoten starten, die noch zu einem Zyklus gehören könnten also sich nicht in Stadium 3 befinden.

Damit Informationen aus vorherigen Tiefensuchen nicht verschwendet werden, werden alle Knoten gespeichert, die sich in Stadium 3 befinden.

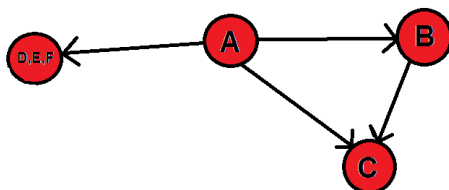
Im Folgenden ist ein Beispiel, das visualisiert, wie dieses Verfahren funktioniert.



In diesem Graphen befindet sich ein Zyklus bestehend aus den Knoten D, F und E.



Die Tiefensuche, beginnend bei A, ruft Knoten D auf, welcher F aufruft, der E aufruft. Im nächsten Schritt würde Knoten E D aufrufen. Hier grau markiert sind die Knoten, die sich gerade in Stadium 1 befinden. Da Knoten D sich bereits in Stadium 1 befindet, muss ein Zyklus existieren. Nun wird die Aufrufliste zurückverfolgt. Knoten D wurde von E aufgerufen, welcher von F aufgerufen wurde und F wurde von D aufgerufen. Diese Knoten können nun zu einem Zyklus zusammengefasst werden. Der daraus entstehende Graph sieht wie folgt aus.



In diesem Fall ist ein nicht zyklischer Graph entstanden – aber das weiß der Algorithmus noch nicht. Es könnte immer noch der Fall sein, dass weitere Zyklen im Graphen existieren. In diesem

speziellen Fall können keine Informationen aus der vorherigen Tiefensuche übernommen werden, da kein Knoten in Stadium 3 verschoben wurde. Die Tiefensuche beginnt erneut bei einem Knoten. In diesem Falle ist es wieder Knoten A. A beginnt die Suche indem er den neuen Knoten „D, E, F“ aufruft. Da an Knoten „D, E, F“ kein weiterer ausgehender Knoten angrenzt, kann sich kein Folgeknoten im Stadium 1 oder 2 befinden, welches die Voraussetzung für das Weitersuchen an einem Knoten ist. Somit kann „D, E, F“ als nicht zu einem Zyklus gehörend identifiziert werden. Auf der Abbildung werden Knoten in Stadium 3 rot markiert. Knoten A sucht die nächste Kante ab und ruft Knoten B auf, welcher C aufruft. C kann keinen Knoten mehr aufrufen und wird daher in Stadium 3 verschoben. Darauf folgt dann, dass nun Knoten B keinen Knoten mehr aufrufen kann, und wird nun ebenfalls in Stadium 3 verschoben. Die letzte Möglichkeit der Tiefensuche ist nun, dass Knoten A seine letzte Kante untersucht. C ist aber bereits in Stadium 3 und daher ist die Tiefensuche beendet. Da alle Knoten sich nun in Stadium 3 befinden, wurden alle Knoten des Graphen erfolgreich entfernt und der Algorithmus terminiert.

3.2 Positive Blätter kaufen (Regel 1a)

Jeder positive Knoten mit keiner ausgehenden Kante wird gekauft und damit aus dem Graphen entfernt.

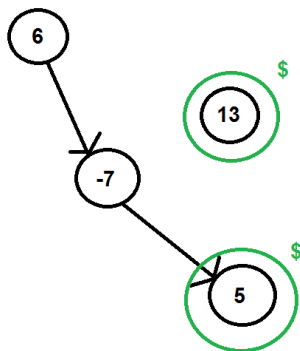
Wenn ein positiver Knoten keine ausgehende Kante besitzt, würde dessen Kauf zum Kauf keines weiteren Knotens führen und damit auch keine Verluste einbringen. Würde der Kauf eines Knotens der eingehenden Kanten des aktuellen Knotens erst dann Gewinn bringen, wenn der aktuelle Knoten automatisch mitgekauft wird, macht es ohnehin keinen Sinn einen von diesen zu kaufen, da der aktuelle Knoten auch einzeln gekauft werden kann. Das heißt, dass man diesen Knoten aus dem Graphen entfernen kann und den Wert des Knotens zum Wert der optimalen Teilmenge des Restgraphen addiert, um die optimale Teilmenge und Wert des Gesamtgraphen zu erhalten.

Vereinfachung des Graphen: -1 Knoten, $-$ Anzahl der eingehenden Kanten des Knoten.

Beispiel:

Der Knoten mit dem Wert 13 wird gekauft, da dieser keine ausgehenden Kanten besitzt. Sein Kauf verursacht keine weiteren Käufe.

Auch der Knoten mit dem Wert 5 kann ohne Bedenken gekauft werden, da sein Kauf ebenfalls in keine weiteren Käufe resultiert. Der Kauf des Knotens mit dem Wert 6 würde nur dann einen Gewinn erzielen, wenn der Knoten 5 automatisch mitgekauft wird. Dies wäre jedoch nicht die größte Teilmenge, da Knoten 5 auch einzeln kaufbar ist. Das ist sogar noch besser ersichtlich, wenn man das gekaufte Blatt 5 einfach aus dem Graph entfernt. Dann bleibt nur noch die Kette 6, -7 übrig und man sieht, dass man keine dieser Knoten kaufen sollte.



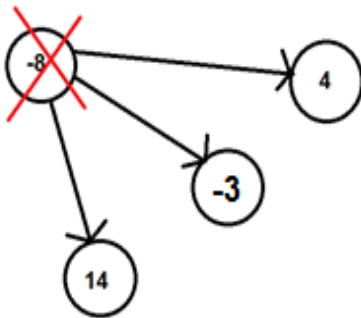
3.3 Negative Wurzeln nicht kaufen (Regel 1b)

Diese Regel ist eigentlich analog zur Regel 1a mit umgekehrten Vorzeichen. Jeder negative Knoten mit keiner eingehenden Kante kann aus dem Graphen entfernt werden. Kein Kauf irgendeines Knotens zwingt einen dazu diesen negativen Knoten zu kaufen. Da man auch die Knoten, welche automatisch durch den Kauf des negativen Knotens gekauft werden einzeln kaufen kann, gibt es keinen Grund, diesen Knoten zu kaufen. Das heißt, dass man so eine Wurzel problemlos aus dem Graph entfernen kann, ohne die maximale Teilmenge zu beeinflussen.

Vereinfachung des Graphen: -1 Knoten, - Anzahl der ausgehenden Kanten des Knoten.

Beispiel:

Hier kann der Knoten -8 entfernt werden, da er keine eingehenden Kanten hat. Die positiven Knoten einzeln zu kaufen, statt über einen automatischen Kauf durch die -8, erzielt die maximale Teilmenge.



3.4 Positive Knoten mit nur einer ausgehenden Kante mit Zielknoten vereinen (Regel 2a)

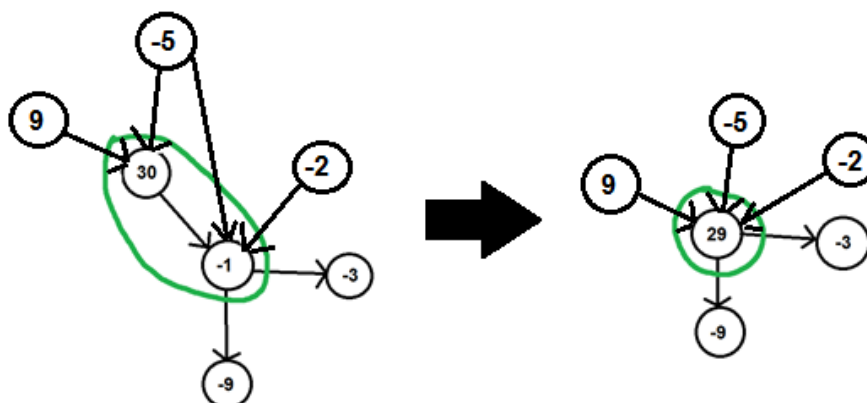
Jeder positive Knoten V_P , der nur eine ausgehende Kante besitzt kann mit dem mit der Kante verbundenen Knoten V_I zusammengefasst werden. Das heißt, es entsteht ein neuer Knoten V_{neu} mit der Summe der Werte der Knoten V_P und V_I . Außerdem werden alle ein- und ausgehenden Kanten von V_P und V_I nun mit dem neuen Knoten V_{neu} verbunden.

Würde man sich den Knoten V_P kaufen, resultiert sein Kauf in jedem Fall zum Kauf des weiterführenden Knotens V_I und dessen Abhängigkeiten. Würde man sich hingegen V_I kaufen, werden durch die Unternehmensbeziehungen auch alle Abhängigkeiten gekauft. Dieser Vorgang kann jedoch nur einmal geschehen, da Firmen nicht mehrere Male gekauft werden können. Jeder positive, eingehende Knoten V_P von V_I , dessen Kauf ausschließlich im Kauf von V_I resultiert (kann nur dann passieren, wenn V_P nur eine ausgehende Kante hat), kann nur noch Gewinn bringen und sollte deswegen gekauft werden. Dies liegt daran, dass V_P positiv ist und auch keine „Firmenbeziehungskettenreaktion“ mehr auslösen kann, weil diese von V_I ausgelöst wurde.

Vereinfachung des Graphen: -1 Knoten, - mindestens verbindende Kante (eventuell mehr – siehe Beispiel Kanten von $-5 \rightarrow -1$ und $-5 \rightarrow 30$).

Beispiel:

Der Kauf der 30 alleine würde in allen Fällen automatisch zum Kauf der -1 und damit auch zum Kauf ihrer Abhängigkeiten führen. Kauft man die -1 alleine, wäre die Kettenreaktion ausgelöst und der Kauf der 30 würde keine weiteren Reaktionen auslösen, die ein negatives Geschäft sein könnten. Die 30 hat nur eine ausgehende Kante und ist positiv und kann daher mit der -1 zusammengefasst werden. Es ergibt sich ein neuer Knoten mit dem Wert 29 ($30+(-1)$) und allen eingehenden und ausgehenden Kanten von einem der beiden Knoten zu einem anderen Knoten sind nun mit dem neuen Knoten verbunden.



3.5 Negative Knoten mit einer nur eingehenden Kante mit Quellknoten vereinen (Regel 2b)

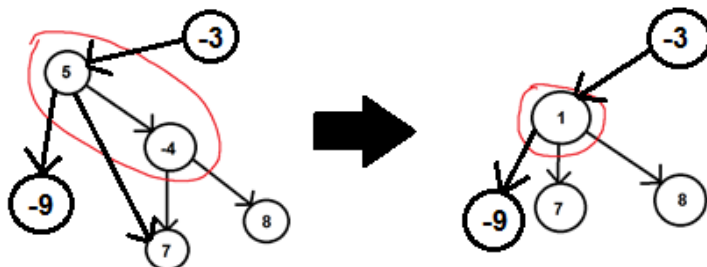
Diese Regel ist analog zu Regel 2a mit umgekehrten Vorzeichen. Jeder negative Knoten V_N mit nur einer eingehenden Kante kann mit dem mit der Kante verbundenen Knoten V_I wie in der vorherigen Vereinfachungsregel zusammengefasst werden.

Zunächst würde der Kauf von V_I in jedem Falle auch zum Kauf von V_N führen. Sich V_N zu kaufen, ergibt aber nur dann Sinn, wenn er das Resultat eines Kaufes des Vorgängerknotens ist. Da V_N nur eine eingehende Kante hat, kann der Kauf dieses Knotens nur das Resultat seiner eingehenden Kante werden, weshalb V_N und V_I zusammengefasst werden können.

Vereinfachung des Graphen: -1 Knoten, - mindestens verbindende Kante (eventuell mehr – siehe Beispiel Kanten von $5 \rightarrow 7$ und $4 \rightarrow 7$).

Beispiel:

Auf der linken Seite befindet sich der Graph vor Anwendung der Vereinfachungsregel, auf der rechten Seite der Graph nach der Anwendung. Hier ist gut zu erkennen, dass es nie Sinn macht die -4 einzeln zu kaufen. Wollte man an die Teilmengen, die unter der -4 liegen, so könnte man diese auch einzeln kaufen. Die -4 sollte nur über das Resultat eines anderen Kaufes gekauft werden. Da es in diesem Falle nur eine Möglichkeit gibt die -4 zu kaufen, nämlich über die 5, kann die -4 mit dieser zusammengefasst werden. Im neuen Graphen auf der rechten Seite entsteht ein neuer Knoten mit den addierten Werten ($5 + (-4) = 1$), sowie alle Kanten, die mit der -4 oder 5 verbunden waren.

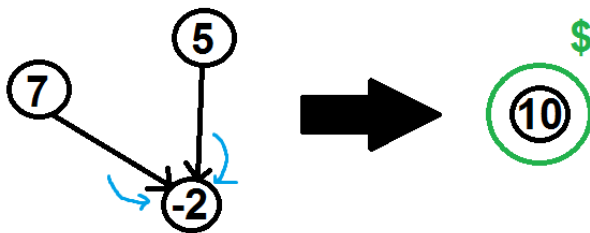


3.6 Die Null (0)

Knoten mit dem Gewicht 0 sind weder positiv noch negativ. Die Regeln funktionieren jedoch auch bei Knoten des Wertes 0. Da das Ziel der Regeln ist, den Graphen um eine möglichst große Zahl von Knoten und Kanten zu verkleinern, kann bei jeder Regel der Wert 0 als positiv oder negativ interpretiert werden, wenn damit eine der Regeln zur Anwendung kommen kann.

3.7 Vereinfachungsregeln Anwendung

Auf den Graphen sollen die Vereinfachungsregeln 1a-2b solange angewendet werden, bis keine dieser Regeln mehr auf den Graphen anwendbar sind. Durch die Anwendung einer Regel auf einen Knoten, kann eine Regel auf einen Knoten anwendbar sein, auf welchen sie es vorher nicht war. Ein Beispiel hierfür wäre ein negativer Knoten, der zwei eingehende positive Knoten mit nur einer ausgehenden Kante hat. Zunächst kann keine Vereinfachungsregel auf den negativen Knoten angewendet werden. Die beiden positiven Knoten können aber jeweils nach Regel 2a mit dem negativen Knoten zusammengefügt werden. Ist dies geschehen kann der ursprünglich negative Knoten aber vereinfacht werden und zwar mit Regel 1b oder 1a, abhängig davon ob sein Wert nach den Vereinfachungen immer noch negativ ist. Dies sähe wie folgt aus.



Somit ergibt sich, dass die Knoten des Graphen solange iteriert werden, unter Anwendung der Vereinfachungsregeln, bis bei einer Iteration keine Veränderung im Graph stattgefunden hat, also keine Regel mehr angewandt wurde. Um die Laufzeit zu verringern, werden nach der ersten Iteration, die alle Knoten des Graphen beinhaltet, nur noch über die Knoten iteriert, die mit einem Knoten verbunden waren, auf den eine Vereinfachungsregel angewendet wurde. Dies ist damit zu begründen, dass diese die einzigen Knoten sind auf welche nach der einer Iteration, durch die Vereinfachungsregeln, die Vereinfachungsregeln möglicherweise erneut anwendbar sind.

Gerade bei Graphen mit einem geringen Kanten zu Knoten Verhältnis werden die meisten Knoten wenige Kanten haben, so dass die Effektivität dieser Regeln dann besonders hoch ist.

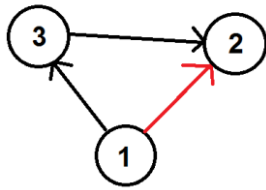
3.8 Redundante Kanten entfernen

Eine weitere Möglichkeit den Graphen zu vereinfachen ist es redundante Kanten zu entfernen. Eine ausgehende Kante eines Knotens kann dann entfernt werden, wenn ein Weg zwischen einer anderen ausgehenden Kante zu dem Knoten auf den die redundante Kante gerichtet ist, existiert.

Würde man sich einen Knoten, kaufen führt dessen Kauf sofort zu dem Kauf aller seiner anliegenden Knoten auf welche eine seiner Kanten gerichtet ist. Würde der Kauf einer seiner ausgehenden Knoten über mehrere Verbindungen hinweg zum Kauf eines direkt angrenzenden Knotens führen, dann ist die Kante zu diesem Knoten nicht nötig, da der Knoten so oder so gekauft werden würde.

Vereinfachung des Graphen: -1 Kante.

Im Folgenden ist ein Beispiel einer redundanten Kante zu sehen. Die Kante von $1 \rightarrow 2$ ist redundant, da der Kauf von 1 eh über 3 zur 2 führt.



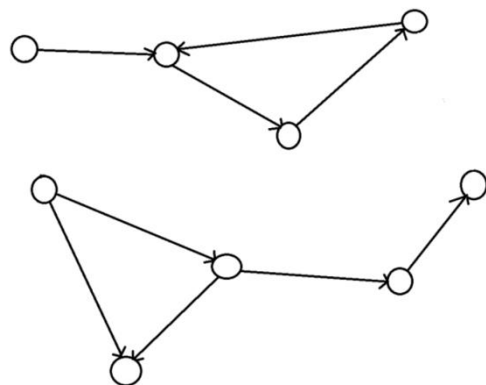
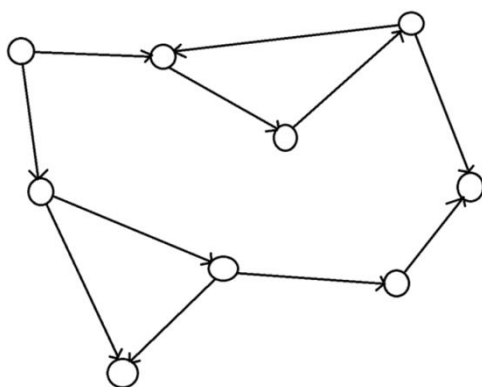
Das Entfernen der redundanten Kanten wird algorithmisch mittels einer Breitensuche ausgehend von den Ursprungsknoten jeder Kante umgesetzt. Vom Ursprungsknoten einer Kante wird eine Breitensuche gestartet ausgehend von allen Zielknoten aller ausgehenden Kanten des Ursprungsknoten, abgesehen von dem Zielknoten der zu untersuchenden Kante. Stößt die Breitensuche auf den Zielknoten der zu untersuchenden Kante, so ist diese redundant und kann entfernt werden.

Gerade bei Graphen mit einem großen Kanten zu Knoten Verhältnis ist es wahrscheinlicher, dass redundante Kanten auftreten, so dass dann die Effektivität dieses Verfahrens besonders hoch ist.

3.9 Graphen in zusammenhängende Graphen unterteilen

Um den Originalgraphen zu vereinfachen, kann dieser in zusammenhängende Teilgraphen unterteilt werden. Ein Graph ist dann zusammenhängend wenn sich je zwei Knoten mit einer Kantenfolge verbinden lassen. Die Kanten des Graphen sind zwar gerichtet, können aber während der Unterteilung als ungerichtet betrachtet werden. Da sich der Kauf eines Knotens in einem Teilgraphen nur auf Knoten auswirken kann, die mit ihm zusammenhängen, kann man alle zusammenhängenden Teilgraphen einzeln betrachten und das Gesamtergebnis ist dann die Summe der Ergebnisse der Teilgraphen. Dies ist insofern sinnvoll, als dass man bestimmte Verfahren, deren Effizienz abhängig von der Größe des Gesamtgraphen ist, dann auf alle einzelnen Teilgraphen anwenden kann. Da die Teilgraphen kleiner sind als der Gesamtgraph und meine Lösung polynomielle Laufzeit hat (Wird später gezeigt), ist die Gesamtlaufzeit = der Summe der Teilgraphenlaufzeiten und der Aufteilung.

Hier ein Beispiel für einen zusammenhängenden (links) und einen nicht zusammenhängenden (rechts) Graphen.



Um den Graphen in zusammenhängende Graphen zu unterteilen, kann eine Breitensuche verwendet werden. Die Suche startet bei einem beliebigen Knoten und findet alle mit diesem Knoten verbundenen Knoten. Die gefundenen Knoten werden zu einem Teilgraphen zusammengefasst. Solange noch Knoten existieren, die noch nicht von der Breitensuche gefunden wurden, existiert mindestens noch ein weiterer zusammenhängender Graph. Daher wird die Breitensuche solange mit den verbleibenden Knoten wiederholt bis alle Knoten besucht wurden.

3.10 Maximale geschlossene Teilmenge bestimmen für restliche Knoten

Das Verfahren, das Lösen des Max-Closure Problems, und somit dieser Teil der Dokumentation beruht unter anderem auf folgenden Quellen.

<http://riot.ieor.berkeley.edu/~dorit/pub/scribe/lec11/Lec11posted.pdf>

https://en.wikipedia.org/wiki/Closure_problem

Das Finden der gesuchten Knotenmenge gliedert sich in drei Schritte:

1. Überführung des Closure-Problems zu einem Max-Flow Problem
2. Lösen des Max-Flow Problems
3. Bestimmung der gesuchten Menge durch minimalen Schnitt

In der Aufgabenstellung wurde gefragt, ob das angewendete Verfahren ein bestmögliches Ergebnis findet, oder diesem nur nahe kommt. Eine optimale geschlossene Teilmenge ist so ein bestmögliches Ergebnis. Meine Vorgehensweise, inklusive der vorangegangenen Vereinfachungsverfahren, garantiert somit, dass immer eine optimale Lösung gefunden wird.

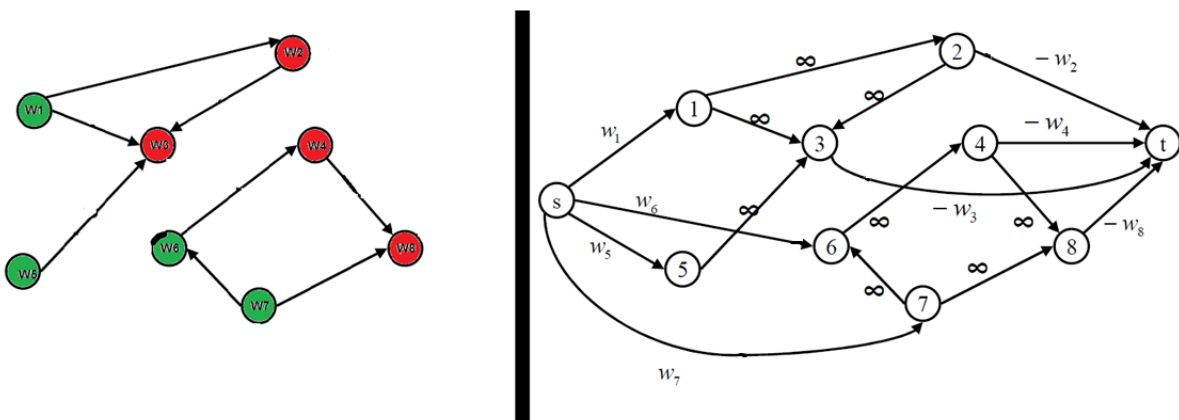
3.10.1 Überführung zu Max-Flow

Es wurde von *Jean-Claude Picard* bewiesen, dass man einen Graphen so anpassen kann, dass die wertvollste Teilmenge über das Bestimmen eines minimalen Schnitts bestimmt werden kann. Die wertvollste Teilmenge besteht dann aus den Knoten, die der minimale Schnitt eingrenzt.

Zunächst werden dem Graphen zwei zusätzliche Knoten, S und T, hinzugefügt. Diese Knoten fungieren später als Start- und Zielknoten für die Bestimmung des maximalen Flusses. Die Kapazitäten aller Kanten, die im Graph existieren, werden auf Unendlich gesetzt. Für jeden Knoten mit einem positiven Wert, wird eine Kante mit der Kapazität gleich dem Wert des Knoten vom hinzugefügten Quellknoten S zu diesen positiven Knoten hinzugefügt. Für jeden negativen Knoten wird eine Kante mit einer Kapazität gleich dem Absolutwert des Knotenwertes vom Knoten zum Zielknoten T erstellt. Zu welcher Kategorie Knoten mit dem Wert 0 hinzugefügt werden, ist irrelevant, da sie aber einer Kategorie hinzugefügt werden müssen, habe ich mich dazu entschieden, dass Knoten mit dem Wert 0 in die Kategorie positive Knoten fallen. Folgendes Beispiel verdeutlicht wie ein Graph durch diesen Algorithmus umgewandelt wird.

Beispiel:

Der Graph auf der linken Seite wird wie oben beschrieben umgewandelt. Die grünen Knoten haben einen positiven Wert, die Roten einen negativen Wert (Das rechte Bild ist auch auf der Wikipedia zu finden unter https://en.wikipedia.org/wiki/Closure_problem#/media/File:Closure.png).



3.10.2 Max-Flow Problem lösen

Um Knoten zu bestimmen, die ein minimaler Schnitt eingrenzt, ist es nötig vorher einen Residualgraphen durch das Anwenden eines Max-Flow Algorithmus zu bilden, da dies wie im nächsten Kapitel beschrieben, ermöglicht einen minimalen Schnitt zu bilden. Der Residualgraph besteht aus den gleichen Knoten wie der Originalgraph. Nur sind in diesem alle Kanten vollständig ausgelastet, sowie hinzugefügte Rückkanten. Genauer hier:

https://de.wikipedia.org/wiki/Fl%C3%BCsse_und_Schnitte_in_Netzwerken#Residualnetzwerk

Um das Max-Flow Problem zu lösen, existieren diverse Verfahren.

https://en.wikipedia.org/wiki/Maximum_flow_problem#Solutions

Die Verfahren, welche ich verstanden habe und somit auch debuggen kann, beschränken sich auf den Edmond-Karp Algorithmus, sowie den Ford-Fulkerson Algorithmus. Da der Edmond-Karp Algorithmus sicher terminiert, wurde sich für diesen entschieden. Außerdem ist seine Laufzeit nicht von dem maximalen Fluss abhängig.

https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm

Folgende Erklärungen basieren zum Teil auf diesem Wikipedia Artikel:

https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm

Ein Max-Flow Problem existiert nur in einem Netzwerk mit einem Quell- und Zielknoten. Der durch die Überführung zum Max-Flow Problem entstandene Graph stellt solch ein Netzwerk dar, da dieser aus gewichteten, gerichteten Kanten sowie einem benötigten Quell- und Zielknoten besteht.

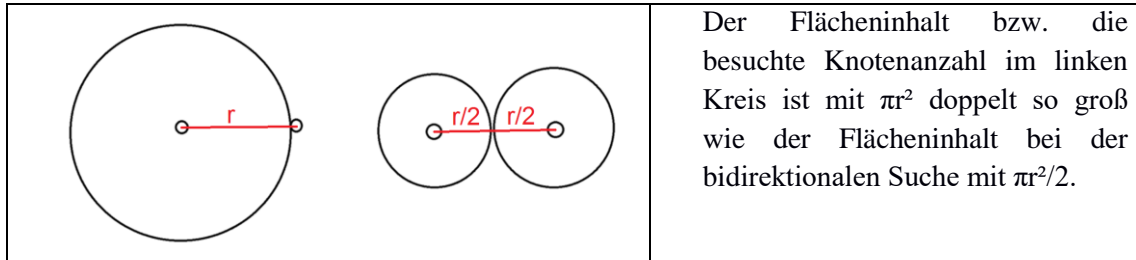
Der Start- und Ziel Knoten sind in diesem Falle, die durch die Reduktion von Closure zu Max-Flow hinzugefügten Knoten S und T. Zu Beginn des Algorithmus wird zu jeder Kante des Graphen eine Rückkante mit der Kapazität 0 gebildet. Solange ein Pfad vom Start zum Zielknoten mit der Kapazität > 0 existiert, wird der maximale Fluss des Pfades ermittelt und der jeweiligen Rückkante hinzuaddiert und von den Kanten des Pfades subtrahiert. Falls mehrere Pfade vom Start zum Zielknoten existieren, wird der Weg genutzt, der aus den wenigsten Knoten besteht. Der Edmond-Karp Algorithmus wird nicht an Beispielen verdeutlicht, da diese bereits genau dokumentiert im Internet existieren. Ich verweise auf den englischen Wikipedia Artikel zu dem Algorithmus (https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm).

Teil des Algorithmus ist es den kürzesten Weg zwischen zwei Knoten S und T zu finden und ist wie folgt umgesetzt.

Um den kürzesten Pfad von Knoten S zu Knoten T zu finden, wird eine Breitensuche verwendet. Es werden also alle erreichbaren Knoten in einem Schritt bestimmt, danach alle erreichbaren Knoten in zwei Schritten usw. Es dürfen nur Knoten über eine Kante mit der Kapazität > 0 erreicht werden. Wird während der Suche dann der Zielknoten gefunden, ist davon auszugehen, dass es sich um einen der schnellsten Wege zu dem Knoten handelt, da der Knoten ja nicht mit weniger Schritten erreicht wurde. Um die Laufzeit der Breitensuche zu verbessern, wird eine bidirektionale Breitensuche verwendet, das heißt eine Breitensuche ausgehenden vom Zielknoten und eine ausgehend vom Startknoten wird ausgeführt. Treffen sich die beiden Suchen, wurde ein schnellster Weg gefunden. Findet eine der Breitensuchen keine neuen Knoten ist der Algorithmus beendet, da dies beweist, dass kein Weg zwischen Knoten S und T besteht. Falls die Breitensuche auf einen Knoten stößt, den sie bereits gefunden hat, wird dieser nicht weiter behandelt, da bereits ein Weg zu diesem gefunden wurde mit weniger Schritten. Es werden zwei Listen während den Breitensuchen geführt, die speichern, welcher Knoten von welchem Knoten aufgerufen wurde um später den Weg via „Backtracking“ zu finden.

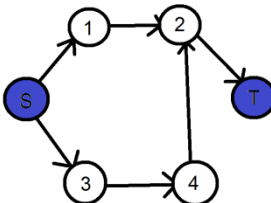
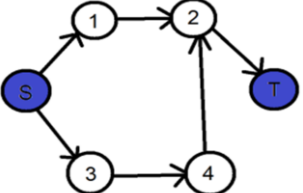
Folgendes Bild verdeutlicht die Effizienz der bidirektionalen Breitensuche. Man stelle sich einen Graphen in \mathbb{R}^2 vor, dessen Knoten gleichmäßig verteilt liegen. Ausgehend von Zielknoten wird nun radialsymmetrisch nach dem Zielknoten gesucht. Der abgesuchte Bereich kann durch einen Kreis

visualisiert werden. Die Größe des abgesuchten Bereichs, also die abgesuchte Knotenanzahl, ist dementsprechend der Flächeninhalt des Kreises. Auf der linken Seite ist der kürzeste Weg zwischen den zwei Knoten via Breitensuche bestimmt worden. Auf den rechten Seiten hingegen wurde der Weg mittels bidirektionaler Breitensuche bestimmt. Hier ist zu erkennen, dass die abgesuchte Fläche, welche die Laufzeit bestimmt, bei der bidirektionalen Suche geringer ist als die der normalen Breitensuche.

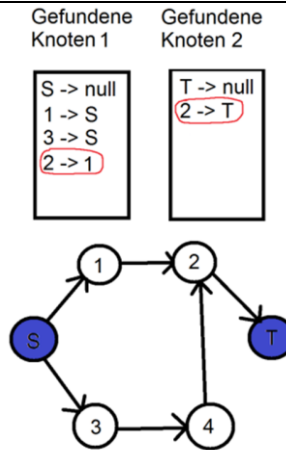


Die folgenden Beispiele illustrieren, wie ein schnellster Weg zwischen zwei Knoten gefunden wird.

Beispiel 1 – Es gibt eine Lösung

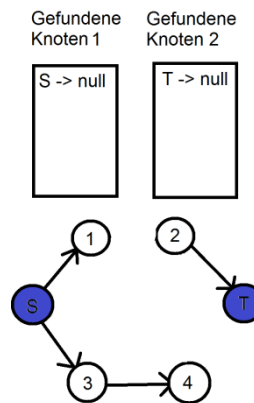
<p>Zu Beginn werden Knoten S und T in die Listen eingetragen. Die eine Breitensuche startet von Knoten S, die andere von Knoten T.</p>	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Gefundene Knoten 1</p> <div style="border: 1px solid black; padding: 5px; width: 60px;">S -> null</div> </div> <div style="text-align: center;"> <p>Gefundene Knoten 2</p> <div style="border: 1px solid black; padding: 5px; width: 60px;">T -> null</div> </div> </div> 
<p>Beide Breitensuchen führen eine Iteration durch. Alle gefundenen Knoten werden in die Listen eingetragen. Nun sind in den Listen alle Knoten vorhanden, die entweder von S mit einem Schritt oder von T mit einem Schritt erreichbar sind. Die Breitensuche ausgehend von T besucht nur eingehende Knoten, die Breitensuche von S nur ausgehende Knoten.</p>	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p>Gefundene Knoten 1</p> <div style="border: 1px solid black; padding: 5px; width: 60px;">S -> null 1 -> S 3 -> S</div> </div> <div style="text-align: center;"> <p>Gefundene Knoten 2</p> <div style="border: 1px solid black; padding: 5px; width: 60px;">T -> null 2 -> T</div> </div> </div> 

Die nächste Iteration startet mit den gerade neu eingetragenen Knoten. Da in der Iteration der Breitensuche ausgehend von Knoten S ein Knoten gefunden wird, der bereits in der anderen Liste vorhanden ist, nämlich Knoten 2, wurde ein schnellster Weg gefunden. Nun wird der Weg wie folgt ermittelt. Der gefundene Knoten ist der Schnittpunkt der beiden Breitensuchen. Man stelle es sich so vor, Liste 1 beschreibt von wo der Knoten aufgerufen wurde, die andere Liste beschreibt, welche Knoten der Knoten aufgerufen hat. Also Knoten 2 wurde von 1 aufgerufen, 1 von S, S ist der Start. Also ist der Weg zum Knoten S-1-2. 2 hat T aufgerufen, T ist der Endknoten. Also ist der gesamte Weg S-1-2-T.



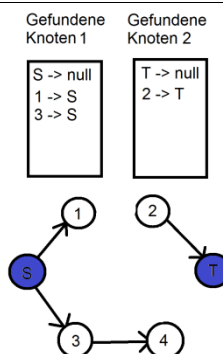
Beispiel 2: Keine Lösung

Zu Beginn der bidirektionalen Breitensuche werden Start- und Zielknoten in die Listen eingetragen. In der folgenden Iteration werden alle Knoten gefunden, die innerhalb eines Schrittes erreichbar sind.



In der Iteration wurde in der Breitensuche ausgehend von S Knoten 1 und 3 gefunden. Die Breitensuche ausgehend von T findet den Knoten 2.

In der folgenden Iteration wird von der ersten Breitensuche der Knoten 4 entdeckt. Die Breitensuche ausgehend von T findet jedoch keine neuen Knoten und hat auch noch keine Verbindung zur ersten Breitensuche gefunden. Trotz der Tatsache, dass die erste Breitensuche noch weitere Knoten besuchen könnte, kann der Algorithmus an dieser Stelle abgebrochen werden, da die zweite Breitensuche festgestellt hat, dass kein Weg zwischen den Knoten S und T existiert.



3.10.3 Bestimmung der Teilmenge durch minimalen Schnitt

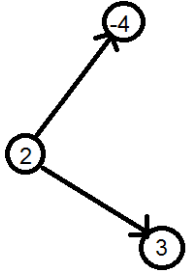
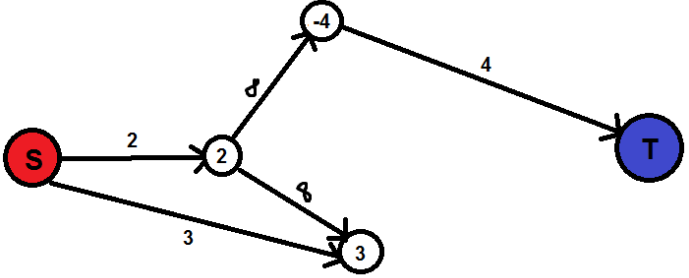
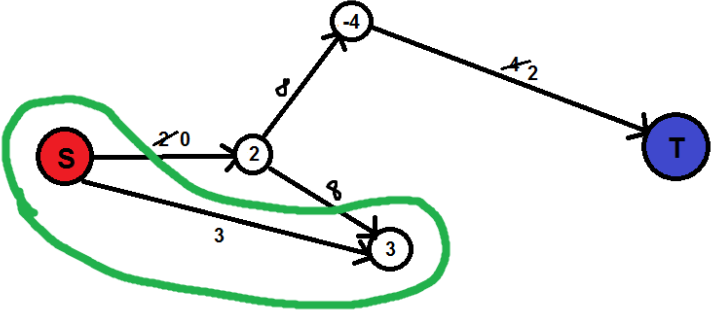
Laut Theorie von Picard besteht nach Bildung des Residualgraphs die gesuchte Knotenmenge aus den Knoten, die ein minimaler Schnitt auf der Seite des Quellknotens einschließt.

Ein minimaler Schnitt kann bestimmt werden indem alle Knoten gefunden werden, zu denen ein Weg über Kanten mit Kapazität > 0 vom Startknoten S existiert. Kurz gesagt verläuft ein minimaler Schnitt nur durch Kanten dessen ausgelastete Kapazität 0 im Residualgraphen beträgt. Dies lässt sich mit dem Max-Flow-Min-Cut Theorem belegen. Genauer dazu ist hier zu finden.

https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

Diese Knoten werden mittels einer Breitensuche ausgehend von S bestimmt, welche nur Knoten über Kanten mit einer Kapazität von > 0 besucht.

3.10.4 Beispiel

<p>Es wird die Teilmenge mit dem oben beschriebenen Verfahren in diesem Graphen bestimmt. Man sieht direkt, dass hier nur der Knoten mit Wert 3 gekauft wird.</p>	
<p>Hinzufügen der Knoten S und T sowie das Erstellen der Kanten und setzen der Kantenkapazitäten.</p>	
<p>Anwendung des Edmond-Karp Algorithmus führt zu folgendem Residualgraphen. Da von S aus nur noch der Knoten mit Wert 3 erreichbar ist (Weg zu 2 nicht möglich, da Kapazität 0 beträgt), bildet dieser das maximale Closure (grün umrandet). Man sieht auch hier, dass nur der Knoten mit dem Wert 3 darin liegt und somit gekauft werden muss.</p>	

4 Programmablauf

4.1 Gesamter Programmablauf

Um die Laufzeit der Anwendung aller Algorithmen auf den Graphen zu verringern, muss eine sinnvolle Reihenfolge der Algorithmen existieren. In den folgenden Teilen der Dokumentation wird die Laufzeit der verschiedenen Schritte abgeschätzt und diese Abschätzung dazu verwendet die Reihenfolge der Schritte zu optimieren. Es sollte jedoch erwähnt werden, dass es auch möglich ist die Lösung durch den letzten Schritt (Reduktion zum Max-Flow Problem, welches dann gelöst wird), alleine zu bestimmen. Die vorangehenden Schritte sind lediglich dazu da die Größe des Graphen und damit auch die Laufzeit des letzten Schrittes zu verbessern.

Zum Anfang werden die Vereinfachungsregeln angewendet, da ihre Effizienz am größten ist. Die Praxis hat gezeigt, dass die Anwendung der Regeln vor dem Aufteilen in Teilgraphen effizienter ist als umgekehrt.

Danach wird der Graph in Teilgraphen unterteilt, da dies auch nur eine lineare Laufzeit benötigt. Durch das vorherige Anwenden der Vereinfachungsregeln ist die Anzahl dieser Graphen im Optimalfall nicht allzu hoch. Dadurch, dass durch das Anwenden der anderen Algorithmen keine neuen Teilgraphen entstehen, wird dieser Algorithmus nur einmal angewendet. Die Reihenfolge der nachfolgenden Schritte bezieht sich jeweils auf einen Teilgraph.

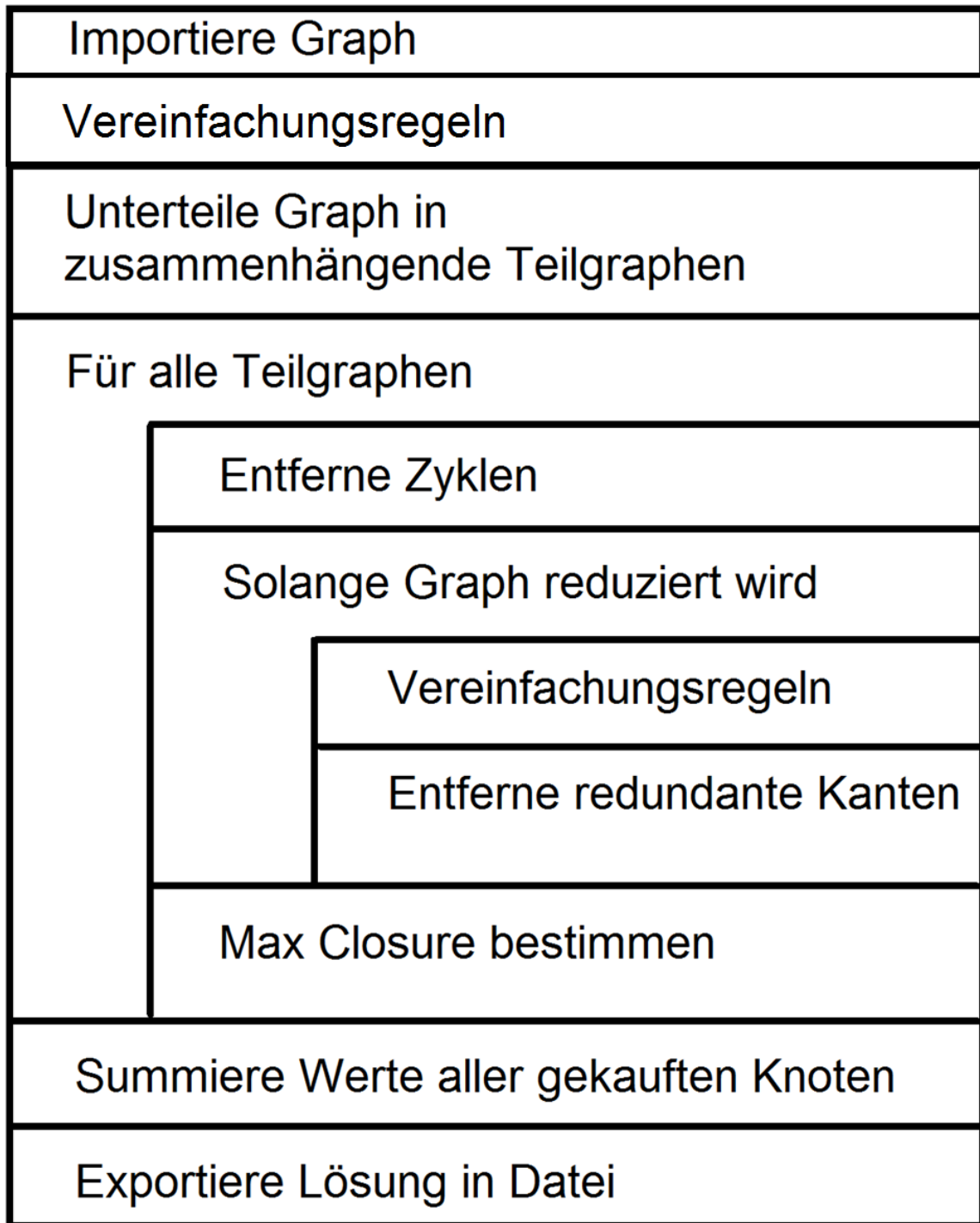
Das Entfernen der Zyklen hat wie später gezeigt wird eine bessere oder ähnliche Laufzeit wie alle anderen folgenden Verfahren. Das Anwenden aller später hier besprochenen Verfahren kann jedoch nicht dazu führen, dass neue Zyklen entstehen. Andererseits ist es möglich, dass durch Entfernen von Zyklen zum Beispiel Vereinfachungsregeln wieder anwendbar sind. Daher werden zuerst die Zyklen aus dem aktuellen Teilgraph entfernt, da dieser Schritt danach nie wieder ausgeführt werden muss.

Nach dem Entfernen der Zyklen ist es trotz der anfänglichen Anwendung der Vereinfachungsregeln möglich, dass die Vereinfachungsregeln wieder anwendbar sind. Abwechselnd werden die Vereinfachungsregeln und das Entfernen redundanter Kanten auf die einzelnen Teilgraphen angewendet. Durch das Entfernen der redundanten Kanten können neue Vereinfachungsregeln entstehen sowie umgekehrt. Da die Laufzeit der Vereinfachungsregeln besser ist, als die des Entfernens der redundanten Kanten, wird mit den Vereinfachungsregeln begonnen.

Es ist möglich, dass die vorherigen Schritte noch keine Entscheidung über alle Knoten getroffen haben (kaufen oder nicht kaufen). Ab diesem Punkt ist es auch nicht mehr möglich den Graphen weiter zu vereinfachen und das Closure-Problem muss gelöst werden. Da dieser Teil die schlechteste Laufzeit besitzt, wird dieser erst zum Schluss angewendet.

4.2 Struktogramm

Im Folgenden befindet sich ein Struktogramm des gesamten Programmablaufs bezüglich des Findens der wertvollsten Teilmenge.



4.3 Anleitung GUI

Um die Anwendung des Programmes angenehmer zu gestalten und um die Ausführung der einzelnen Algorithmen an Graphen zu visualisieren, wird eine Benutzeroberfläche benötigt.

Anforderungen an die Benutzeroberfläche:

- Darstellung eines gewichteten, gerichteten Graphen mit variabler Größe
- Möglichkeit Effekte der benutzen Algorithmen anzuzeigen
- Graphen importieren

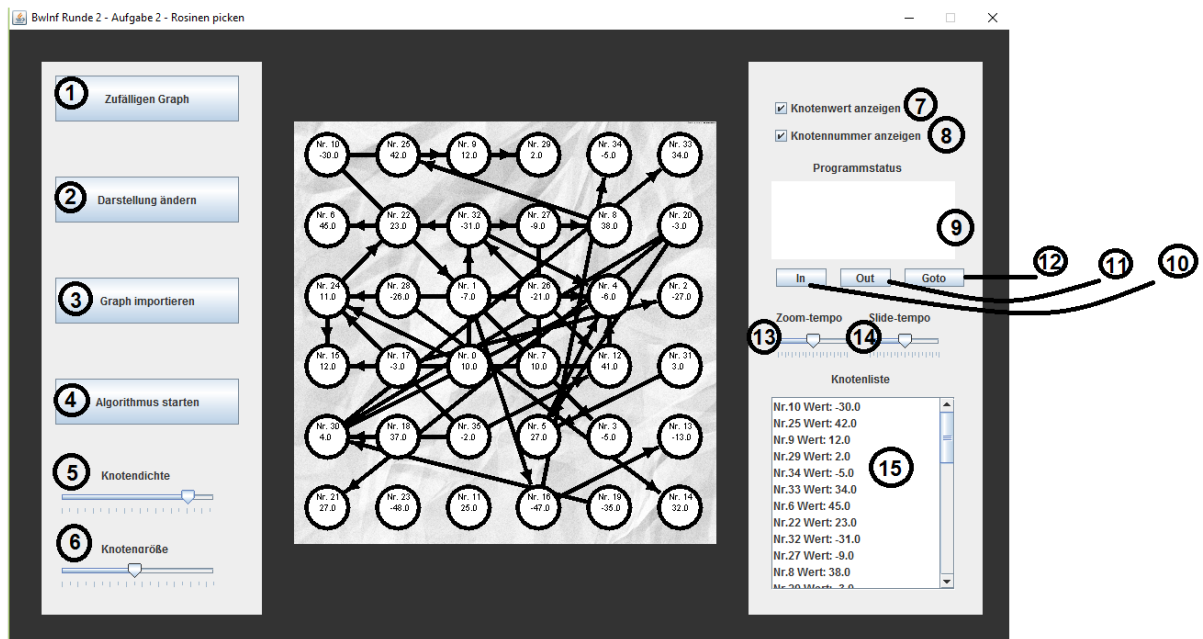
Die Benutzeroberfläche sollte zwar möglichst selbsterklärend sein, jedoch erhält die Anwendung hier eine kurze Anleitung. Der Ablauf wie aus einer Textdatei die Lösung erhalten wird, ist im nächsten Kapitel beschrieben.

Um sowohl kleine als auch große Graphen darzustellen, erlaubt das Programm Heraus- und Hereinzoomen sowie Bewegen innerhalb des Graphen. Zoomen ist mithilfe des Mausekkrads möglich. Man kann sich innerhalb des Graphen bewegen in dem man an einer beliebigen Stelle des Graphen mit der Maus zieht.

Das Programm besitzt zwei Modi. Der erste Modus erlaubt es, einen Graphen zu erzeugen und dessen Darstellung zu ändern. Der zweite Modus ermöglicht die Anwendung aller Algorithmen auf den Graphen - schrittweise und sofort.

Hier ist Modus 1 zu sehen, die einzelnen Funktionen werden im Folgenden aufgelistet.

1. Ein zufälliger Graph mit benutzerdefinierter Kanten- und Knotenmenge wird erzeugt
2. Wechsel der Anordnung der Knoten. Die Knoten werden entweder in einem Kreis oder innerhalb eines Quadrates angeordnet
3. Ermöglicht es dem Benutzer einen Graphen zu importieren in dem dieser eine Datei auswählt (Der Dateiselektor startet im Unterordner „*graphen*“, der sich im gleichen Verzeichnis befinden muss, wie die .jar Datei)
4. Wechselt in den zweiten Modus, welcher erlaubt mithilfe der Anwendung der verschiedenen Algorithmen den Graphen zu bearbeiten
5. Variiert den Abstand zwischen den Knoten
6. Variiert die Größe der Knoten
7. Ermöglicht es die Werte der Knoten entweder anzuzeigen oder nicht anzuzeigen. Dies kann hilfreich sein, um Lag bei extrem großen Graphen zu verringern
8. Ermöglicht es die Nummer der Knoten entweder anzuzeigen oder nicht anzuzeigen. Dies kann hilfreich sein, um Lag bei extrem großen Graphen zu verringern
9. Gibt den aktuellen Status des Programmes aus, während ein Graph vollständig gelöst wird
10. Zoomt so nah an Knoten heran wie möglich
11. Zoomt soweit heraus wie möglich
12. Zentriert die Kamera auf einen ausgewählten Knoten
13. Reguliert das Tempo mit dem sich innerhalb des Graphen bewegt werden kann
14. Reguliert das Zoomtempo
15. Eine Liste aller im Graphen vorhandenen Knoten



Im nächsten Bild ist das Programm im Modus 2 zu sehen. Dieser Modus erlaubt es einem die Vereinfachungsregeln oder den Hauptalgorithmus schrittweise oder sofort anzuwenden. Werden Algorithmen schrittweise ausgeführt, wird farblich markiert, wie sich der Graph verändert hat. Alle rot markierten Knoten wurden aus dem Graphen entfernt, alle Grünen gekauft und alle Blauen zu einem anderen Knoten zusammengefügt. Lediglich die linke Funktionsseite unterscheidet sich zu Modus 1. Der Graph kann immer noch durch die Funktionen auf der rechten Seite betrachtet werden.

Im Folgenden eine Liste aller neuen Funktionen:

1. Wechsel in Modus 1
2. Einstellung, dass bei Auswahl eines Algorithmus dieser sofort vollständig angewendet wird
3. Einstellung, dass bei Auswahl eines Algorithmus dieser schrittweise ausgeführt wird, um genau Veränderungen im Graphen einzusehen
4. Auswahl des Zyklen entfernen Algorithmus
5. Auswahl der Vereinfachungsregeln
6. Auswahl des Entfernen redundanter Kanten
7. Auswahl des Hauptalgorithmus
8. Führt einen einzelnen Schritt des ausgewählten Algorithmus aus. Wurde kein Algorithmus ausgewählt geschieht nichts
9. Führt alle Algorithmen in einer festgelegten Reihenfolge aus. Der Graph wird so bearbeitet wie in der gesamten Doku beschrieben (Erste Teilung des Graphen dann... etc.). Wenn der Graph vollständig gelöst wurde, wird eine Ausgabedatei ausgegeben. Diese Funktion kann nur angewendet werden, wenn vorher noch keine andere Funktion auf den geladenen Graphen angewendet wurde
10. Eine Liste, die alle Knoten beinhaltet, die zum aktuellen Stand des Graphen gekauft wurden
11. Der Graph wird wieder in seine Ursprungsverfassung zurückversetzt

Bwlnf Runde 2 - Aufgabe 2 - Rosinen picken

1 Abbruch

2 Sofort 3 Schrittweise

4 1) Zyklen 5 2) Regeln

6 3) Redund... 7 4) Closures

8 Weiter

9 Alles sofort

Kaufenliste

10 Nr.29
Nr.33
Nr.15
Nr.18

11 Reset

☒ Knotenwert anzeigen

☒ Knotennummer anzeigen

Programmstatus

In Out Goto

Zoom-tempo Slide-tempo

Knotenliste

- Nr.28 Wert: -26.0
- Nr.30 Wert: 4.0
- Nr.22 Wert: 23.0
- Nr.21 Wert: 27.0
- Nr.15 Wert: 12.0
- Nr.32 Wert: -31.0
- Nr.6 Wert: 45.0
- Nr.0 Wert: 10.0
- Nr.20 Wert: -3.0
- Nr.1 Wert: -7.0
- Nr.26 Wert: -21.0
- Nr.40 Wert: -25.0

4.4 Erstellung der Lösungsdatei zu vorgegebener Aufgabe

Um die optimale Teilmenge aus einem beliebigen Graphen zu finden, muss zunächst der Graph importiert („Graph importieren“) werden. Danach wird in Modus 2 über den Knopf „Starte Algorithmus“ gewechselt und der „Alles sofort“ Knopf muss gedrückt werden, um das in der Dokumentation beschriebene Verfahren anzuwenden. Die Lösung wird dann in das Verzeichnis geschrieben aus welchem der Graph importiert wurde. Der Dateiname der Lösung setzt sich aus dem Namen der importierten Datei + „_solution.txt“.

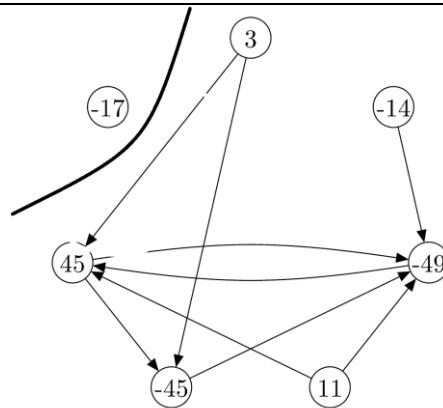
5 Lösungsbeispiele

5.1 Diverse Beispiele

Der gesamte Programmablauf wird im Folgenden an mehreren Beispielen demonstriert. Um nicht unnötig große Beispiele zu verwenden, wird der erste Schritt, das einmalige Anwenden der Vereinfachungsregeln, ausgelassen, da dieser Schritt leider sehr effizient darin ist einfache Beispiele zu verkleinern. Da die Vereinfachungsregeln aber auch später im Verfahren benutzt werden, werden diese trotzdem in den Beispielen behandelt.

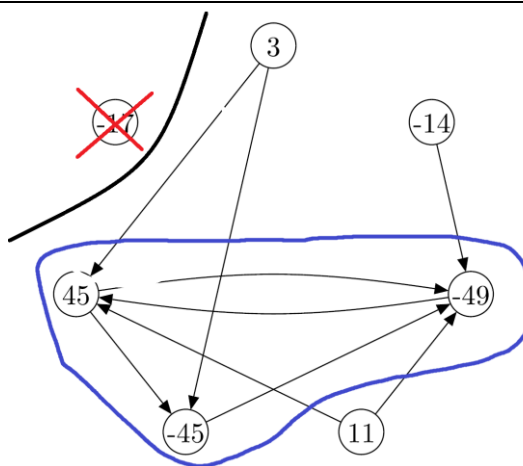
Als erstes Beispiel wurde das Firmenkonglomerat **Zufall-7.txt** der Beispielaufgaben gewählt.

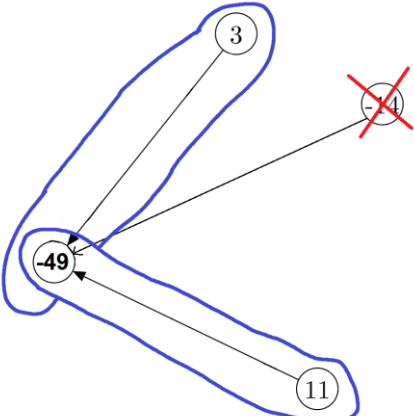

Im ersten Schritt wird der Graph in zusammenhängende Knoten unterteilt. Dieses Beispiel besteht aus zwei Teilgraphen, welche aus dem Knoten mit dem Wert -17 und den restlichen Knoten bestehen.



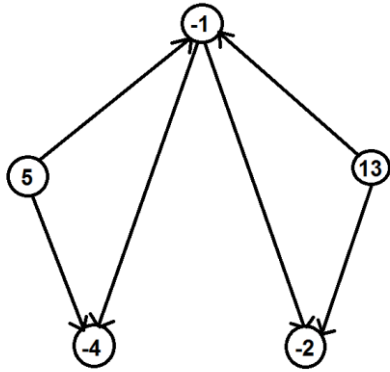
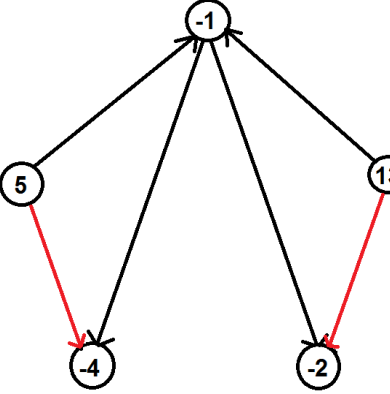
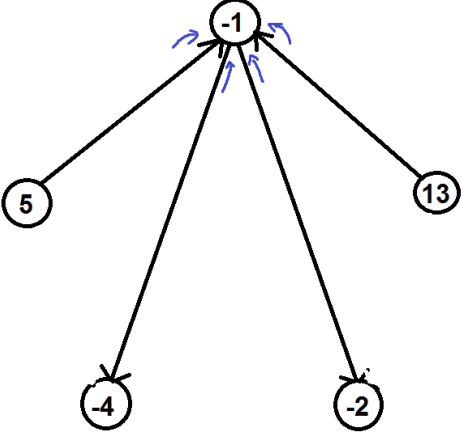

Als nächstes werden die Teilgraphen konsequent abgearbeitet. Der erste Teilgraph besteht nur aus einem Knoten. Es können keine Zyklen entfernt werden, jedoch ist Regel 1b anwendbar, wodurch identifiziert wird, dass dieser Knoten nicht gekauft wird.

Der erste Schritt im anderen Teilgraphen besteht darin alle Zyklen zu entfernen. Hier ist ein Zyklus vorhanden, der aus den Knoten mit den Werten -45, -49 und 45 besteht. Diese lassen sich zu einem neuen Knoten mit dem Wert -49 zusammenfassen. Folgende Besonderheiten sind hier zu beachten. Einerseits fällt die Kante $45 \rightarrow -49$, weg da ihr Zielknoten und Startknoten nach dem Zusammenfassen der gleiche wäre. Solch eine Kante existiert nicht. Ebenfalls ist zu beachten, dass Knoten 11 zwei Kanten zu Knoten innerhalb des Zyklus besitzt. Nach dem Zusammenfassen würde dann



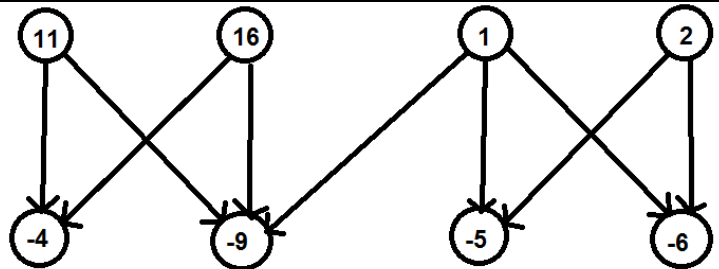
<p>zweimal die gleiche Kante existieren. Dies ist ebenfalls nicht möglich und es existiert nur eine Kante von 11 zum zusammengefassten Knoten.</p>	
<p>Als nächstes werden die Vereinfachungsregeln angewendet. Es ergibt sich, dass die -14 mit Regel 1b entfernt werden kann. 3 und 11 lassen sich jeweils mit Regel 2a zu der -49 zusammenfassen. Danach besteht der Graph nur noch aus einem Knoten mit dem Wert -35.</p>	
<p>Da während der ersten Iteration der Vereinfachungsregeln mindestens ein Knoten vereinfacht wurde, werden die Regeln erneut auf die verbleibenden Knoten angewendet. Knoten -35 wird mit 1b vereinfacht. Das Programm wird beendet, da der Graph aus keinen Knoten mehr besteht. Es ergibt sich, dass kein Knoten gekauft wurde.</p>	

Das nächste Beispiel wurde so gewählt, dass das Entfernen von redundanten Kanten ebenfalls zur Anwendung kommt und visualisiert werden kann.

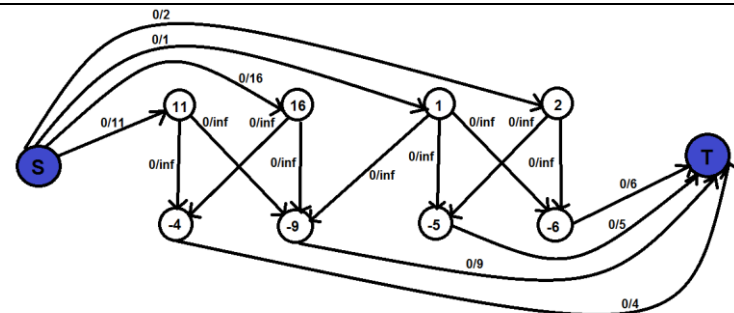
<p>Es wird die größte Teilmenge aus folgendem Graph bestimmt. Dieser Graph lässt sich nicht in Teilgraphen zerlegen. Außerdem sind keine Zyklen enthalten und Vereinfachungsregeln lassen sich zunächst auch nicht anwenden.</p>	
<p>Können die Vereinfachungsregeln den Graphen mit mehr vereinfachen, werden redundante Kanten entfernt. In diesem Graphen existieren zwei solcher Kanten, welche entfernt werden können. Die Kante von $5 \rightarrow -4$ ist redundant, da der Kauf von 5 zu -1 führt, welcher nun zum Kauf von -4 resultiert. Durch diesen Grund ist die Kante von $13 \rightarrow -2$ auch redundant.</p>	
<p>Da mindestens eine redundante Kante entfernt wurde, werden die Vereinfachungsregeln erneut angewendet. Dies führt dazu, dass Knoten 5 und 13 durch Regel 2a mit -1 zusammengefasst werden. Knoten -4 und -2 werden durch Regel 2b ebenfalls zur -1 zusammengefasst. Es entsteht so ein Knoten mit dem Wert 11.</p>	
<p>Dadurch, dass während einer Iteration der Vereinfachungsregeln mindestens eine Regel erfolgreich angewendet wurde, wird erneut probiert potentiell betroffene Knoten zu vereinfachen. In diesem Fall lässt sich der Knoten 11 durch Regel 1a vereinfachen. Die optimale Teilmenge in diesem Graphen besteht somit aus dem Kauf vom Knoten 11. Da dieser durch das Zusammenfassen von allen Knoten des Graphen entstanden ist, werden alle originalen Knoten des Graphen gekauft (5,-4,-1,-2,13).</p>	

Das nächste Beispiel verdeutlicht wie die optimale Teilmenge gefunden wird, wenn keine anderen Vereinfachungen mehr möglich sind, so dass die maximale Teilmenge mit dem in 3.10 beschriebenen Verfahren bestimmt werden muss.

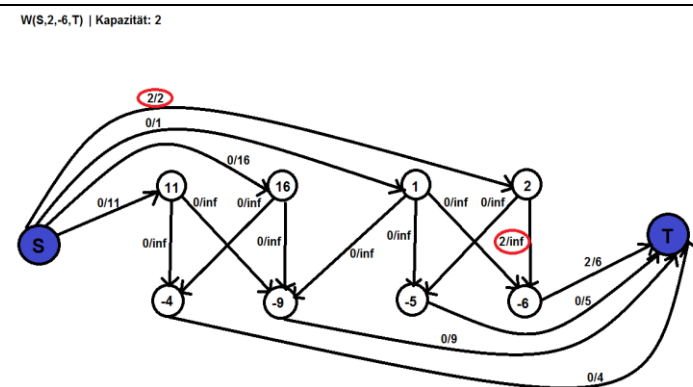
Dieser Graph kann weder in Teilgraphen unterteilt werden, noch enthält dieser Zyklen. Auch Vereinfachungsregeln, sowie das Entfernen von redundanten Kanten ist nicht möglich.



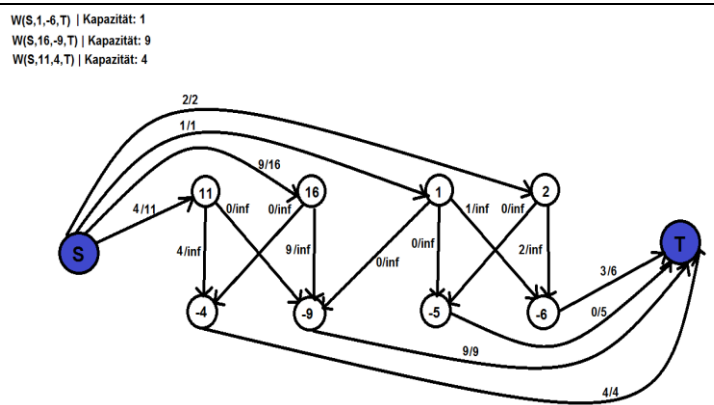
Es wird nun das maximale Closure bestimmt. Zunächst wird der Graph angepasst. Es werden zwei Knoten S und T hinzugefügt. Zusätzlich wird die Kapazität aller alten Kanten auf ∞ gesetzt. Neue Kanten zu den Knoten S und T werden ebenfalls entsprechend erstellt. Es liegt nun ein Netzwerk vor.



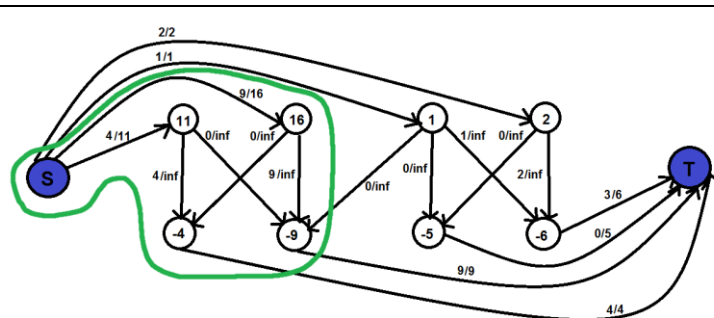
Der Edmond-Karp Algorithmus wird nun angewendet. Es werden, solange es möglich ist, kürzeste, mögliche Wege von S zu T gesucht. Wege sind dann möglich, wenn jede gelaufene Kante noch nicht voll ausgelastet ist. Die geringste Kapazität aller Kanten, die sich auf dem Weg befinden, wird allen Kanten des Pfades subtrahiert. Die Abbildung zeigt die Kantenauslastung nach dem ein Weg $(S \rightarrow 2 \rightarrow -6 \rightarrow T)$ behandelt wurde. Die Kante $S \rightarrow 2$ ist nun vollständig ausgelastet und kann im folgenden Schritt nicht mehr benutzt werden.



Es werden solange kürzeste Wege bestimmt und die Kanten entsprechend angepasst, bis dies nicht mehr möglich ist. Um in diese Phase zu gelangen wurden die Wege
 $S \rightarrow 1 \rightarrow -6 \rightarrow T$ (Kapazität: 1),
 $S \rightarrow 16 \rightarrow -9 \rightarrow T$ (Kapazität: 9),
 $S \rightarrow 1 \rightarrow 4 \rightarrow T$ (Kapazität: 4)
gewählt. Der so entstandene Residualgraph ist rechts abgebildet. Hier ist es nicht mehr möglich einen Weg von S zu T zu finden über nicht ausgelastete Kanten.



Zuletzt müssen nur noch die Knoten bestimmt werden, die von S aus über nicht ausgelastete Kanten erreicht werden können. Knoten S erreicht alle grün eingekreisten Knoten (11, 16, -4, -9). Die Kanten $S \rightarrow 2$, $S \rightarrow 1$, $-9 \rightarrow T$, $-4 \rightarrow T$ sind ausgelastet. Die Rückkante $-9 \rightarrow 1$ ist dies ebenfalls, so dass keine nicht ausgelastete Kante aus der Menge herausführt. Die Knoten 11, 16, -4, -9 bilden also die optimale Teilmenge und werden gekauft.



5.2 Lösungen aller Beispielaufgaben der BwInf-Seite

Das Programm wurde auf alle Beispielaufgaben der BwInf-Homepage angewendet. Alle Aufgaben wurden innerhalb weniger Millisekunden, also ohne wahrnehmbare Verzögerung, von dem Programm gelöst. Diese Lösungsdateien befinden sich im beigelegten Ordner „Aufgabe1-Implementierung\graphen“ der Einsendung.

Dateiname	Kaufwert	Anzahl Knoten	Gekaufte Knoten
Kreis-8.txt	4	8	0, 1, 2, 3, 4, 5, 6, 7
Quadrat-6.txt	333	25	1, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 18, 21, 22, 24, 25, 27, 29, 30, 31, 33, 34
Quadrat-8.txt	446	28	2, 4, 8, 9, 14, 15, 17, 21, 22, 23, 26, 27, 31, 32, 33, 35, 37, 38, 41, 44, 50, 52, 53, 54, 57, 60, 61, 62
Quadrat-13.txt	961	78	7, 8, 10, 12, 13, 16, 18, 23, 25, 27, 28, 29, 35, 36, 39, 41, 43, 48, 57, 59, 60, 62, 68, 72, 74, 75, 83, 84, 86, 88, 89, 90, 91, 92, 93, 94, 96, 99, 100, 103, 104, 105, 106, 109, 110, 111, 112, 114, 115, 116, 118, 119, 120, 121, 122, 123, 124, 129, 130, 131, 134, 135, 137, 138, 141, 146, 148, 152, 153, 154, 155, 158, 161, 162, 163, 165, 166, 168,
Zufall-7.txt	0	0	
Zufall-40.txt	504	31	0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 20, 22, 23, 24, 25, 26, 27, 28, 31, 32, 34, 35, 36, 37, 38
Zufall-100.txt	613	21	0, 2, 4, 10, 15, 20, 25, 29, 34, 41, 47, 53, 57, 62, 63, 66, 67, 81, 96, 97, 98
<p>Es wurden außerdem 8 zufällig generierte Eingabedateien mit jeweils 1000 Knoten erstellt. 4 davon enthalten Zyklen, die anderen 4 nicht. Außerdem enthalten die Beispieldateien unterschiedliche Knoten zu Kanten Verhältnisse, da das Verhältnis Auswirkungen auf die Effektivität der unterschiedlichen Verfahren nimmt. Das kleinste Verhältnis ist 1000:1000, das größte Verhältnis ist 1000:32000.</p> <p>Die Beispieldateien, sowie die dazugehörigen Lösungsdateien befinden sich im Ordner „Aufgabe1-Implementierung\graphen“ der Einsendung.</p>			

6 Laufzeiten und Speicherverbrauch

Es folgen theoretische Überlegungen zur Laufzeit und zum Speicherverbrauch sowie in der Praxis gesammelte Messdaten, die die Abschätzungen abgleichen. Die Messdaten auf welche hier Bezug genommen wird, sind im Messdaten Kapitel abgebildet.

6.1 Laufzeiten

Alle Laufzeiten werden für einen Graphen G mit $|V|$ Knoten und $|E|$ Kanten analysiert.

6.1.1 Graphen unterteilen

Das Unterteilen in zusammenhängende Graphen erfordert wie in 2.8 beschrieben, dass jeder Knoten und jede Kante einmal behandelt wird. Es ergibt sich also eine Laufzeit von $O(|V|+|E|) = O(N)$. In den zugehörigen Messreihen bestätigt sich diese Abschätzung.

6.1.2 Zyklen entfernen

Das Finden eines Zyklus wird mit $O(|V|+|E|)$ abgeschätzt, weil schlimmstenfalls alle Knoten und Kanten einmal besucht werden. Beim Zusammenfassen des Zyklus verringert sich die Anzahl der Knoten um $(\text{Zykluslänge}-1)$ und die Anzahl der Kanten mindestens um die Zykluslänge. Schlimmstenfalls – falls der Zyklus nur aus zwei Knoten besteht – verringert sich der Graph auch nur um einen Knoten und zwei Kanten. Der schlimmste Fall beinhaltet auch, dass bei der Tiefensuche kein Knoten in Stadium 3 verschoben wird. Somit ergibt sich eine Worst-Case Laufzeit von $O((|E|+|V|)^2) \leq O(N^2)$, wenn man $|V|-1$ mal Zyklen zusammenfasst. In den Messreihen ist deutlich zu erkennen, dass die Laufzeit dieses Verfahrens in der Praxis eher im linearen Bereich liegt, da vermutlich der Worst-Case sehr unwahrscheinlich ist.

6.1.3 Vereinfachungsregeln

Eine Regel auf einen Knoten anzuwenden hat eine konstante Laufzeit $O(1)$. Alle Knoten des Graphen zu iterieren und auf alle die Vereinfachungsregeln anwenden kann mit $O(|V|+|E|) \leq O(N)$ beschrieben werden. Wie bereits erwähnt, wird beim erfolgreichen Anwenden jeder Vereinfachungsregel der Graph um genau einen Knoten reduziert und um die mit ihm verbundenen Kanten. Zwar werden in den folgenden Iterationen nur Knoten, verbunden mit einem Knoten, der vereinfacht wurde, behandelt, so dass eigentlich eine lineare Laufzeit in zufälligen Graphen vermutet wird. Aber im Worst-Case wird pro Iteration nur 1 Knoten entfernt, so dass man $|V|-1$ mal die Vereinfachungsregeln anwenden muss, was zu einer Laufzeit von $O((|V|+|E|)*|V|) \leq O(N^2)$ führt. Ähnlich wie bei den Zyklen ist das Eintreten des Worst-Case bei den generierten Beispielen sehr unwahrscheinlich, da die gemessene Laufzeit eher linear ist.

6.1.4 Redundante Kanten entfernen

Da für jede Kante eine Breitensuche gestartet werden muss, die im schlechtesten Fall bis zu $O(|V|+|E|)$ betragen kann, ergibt sich eine gesamte Laufzeit von $O(|E|) * O(|V|+|E|) = O(|V|*|E|+|E|^2) \leq O(N^2)$. In den zugehörigen Messreihen ist zu erkennen, dass sich diese Laufzeitabschätzung bestätigen lässt.

6.1.5 Maximales Closure bestimmen

Um ein Closure-Problem auf ein Max-Flow Problem zu reduzieren ist es von Nöten zunächst jede einzelne Kante zu iterieren um ihre Kapazität auf ∞ zu setzen. Außerdem muss jeder Knoten iteriert werden, um eine Kante zu den extra Start- und Zielknoten zu erstellen. Es ergibt sich hierfür also eine Laufzeit von $O(|E|+|V|) = O(N)$.

Um den Residualgraphen zu bilden wird der Edmond-Karp Algorithmus angewendet. In der Theorie beträgt die Laufzeit dieses Algorithmus $O(|V| * |E|^2) \leq O(N^3)$. Dies kann hier nachgelesen werden.

https://en.wikipedia.org/wiki/Maximum_flow_problem#Solutions

Nach dem Bestimmen des maximalen Closures wird der minimale Schnitt gebildet, bzw. alle Knoten, die erreichbar sind ohne eine Kante mit einer Kapazität von 0 zu überqueren, vom Startknoten aus, werden gefunden. Da dies mit einer Breitensuche gelöst wurde, beträgt für diesen Schritt die Laufzeit ebenfalls $O(|V|+|E|) = O(N)$.

Die gesamte Laufzeit setzt sich aus der Laufzeit der einzelnen Teilschritte zusammen. Es ergibt sich $O(N) + O(N^3) + O(N) = O(N^3)$.

Betrachtet man die Messreihen ist zu erkennen, dass die Laufzeit leicht schlechter als quadratisch ist, aber deutlich besser als kubisch. Dies ist vielleicht damit zu begründen, dass der Worst-Case mit dem der Edmond-Karp Algorithmus abgeschätzt werden musste, bei den betrachteten Graphen nicht auftritt.

6.1.6 Gesamtlaufzeit

Die Gesamtlaufzeit ergibt sich aus der Summe der Einzelschritte.

$$O_{\text{Gesamt}} = O(N) + O(N^2) + O(N^2) + O(N^2) + O(N^3) = O(N^3).$$

In der zugehörigen Messreihe ist zu erkennen, dass die echte Laufzeit deutlich unter der des Worst-Case liegt. Diese liegt nämlich im Bereich zwischen linear und quadratisch. In den Messreihen ist zu erkennen, dass bei zufällig generierten Graphen die Laufzeit der Einzelschritte eher linear beziehungsweise bei den Schritten redundante Kanten entfernen und maximales Closure bestimmen eher quadratisch sind, so dass die Gesamtlaufzeit bei zufällig generierten Graphen je nach dem welches Verfahren die meisten Knoten bzw. Kanten vereinfacht, abhängt.

6.2 Speicherverbrauch

Alle Speicherverbräuche werden für einen Graphen G mit $|V|$ Knoten und $|E|$ Kanten analysiert.

6.2.1 Laden des Graphen

Es wird ein Speicher für alle angegebenen Knoten und Kanten benötigt, weshalb sich ein Speicherverbrauch von $O(|V|+|E|) = O(N)$ ergibt.

6.2.2 Graphen unterteilen

De facto werden wird der Graph durch das Unterteilen kopiert. Also die Teilgraphen zusammen besitzen die gleiche Kanten und Knotenanzahl. Es kann nicht mehr Teilgraphen als Knoten geben. Daraus ergibt ein Speicherverbrauch von $O(2*(|V|+|E|)+V) = O(N)$. Diese Annahme lässt sich in den zugehörigen Messdaten so bestätigen.

6.2.3 Zyklen entfernen

Das Finden eines Zyklus benötigt eine Tiefensuche und verbraucht daher $O(N)$ Speicher. Gleichzeitig befindet sich jeder Knoten in einem von 3 Stadien. Also auch hier $O(N)$. Nach der Zyklusentfernung wird der eben genannte Speicher nicht mehr benötigt und kann bei der nächsten Zyklusentfernung wieder verwendet werden. Das Zusammenfassen eines Zyklus entfernt einige Knoten, dafür wird die relevante Information dieser Knoten in den zusammengefassten Knoten überführt, so dass hier kein zusätzlicher Speicher benötigt wird $O(1)$.

Daraus folgt, ein Speicherverbrauch während des Ausführens von $O(N+N+1) = O(N)$. Auch diese Annahme lässt sich eindeutig in den Messdaten bestätigen.

6.2.4 Vereinfachungsregeln

Da lediglich Knoten angeschaut werden und, dass das Reduzieren eines Knotens diesen entweder entfernt oder mit einem anderen zusammenfügt, wird hier wie beim Zusammenfassen im Zyklusentfernen kein zusätzlicher Speicher benötigt, also $O(1)$. Diese Theorie kann ebenfalls mit den Messdaten bestätigt werden. Diese bilden zwar einen linearen Speicherverbrauch ab, jedoch entsteht dieser automatisch durch das Speichern des Graphen. Da keine größere Steigung als linear in den Messdaten aufzufinden ist, kann man hier von einem konstanten Speicherverbrauch ausgehen.

6.2.5 Redundante Kanten entfernen

Es werden lediglich alle Knoten einmal iteriert, was an sich keinen zusätzlichen Speicher benötigt. Jedoch wird pro Untersuchung von einem Knoten eine Breitensuche durchgeführt, welche $O(|V|+|E|)$ Speicher benötigt, der nach der Breitensuche wieder frei zur Verfügung steht. Während des Verfahrens ergibt sich also ein Speicher von $O(N)$. Diese Annahme wird auch mit den Messdaten verifiziert, welche einen linearen Speicherverbrauch abbilden.

6.2.6 Maximales Closure bestimmen

Der erste Schritt, die Reduzierung zu einem Max-Flow Problem, benötigt keinen Speicher, da lediglich jede Kante und jeder Knoten einfach iteriert wird. Das Ändern der Kapazität benötigt keinen Speicher, wohingegen das Hinzufügen einer Kante pro Knoten in einen gesamten Speicherverbrauch von $O(|V|) \leq O(N)$ resultiert. Das Hinzufügen eines Start- und Zielknotens ist irrelevant, da der Speicher hier $O(1)$ ist.

Der zweite Schritt, die Anwendung des Edmond-Karp Algorithmus, besteht aus dem Finden des kürzesten Weges mittels bidirektionaler Breitensuche $O(N)$ und der Aktualisierung der Kantenkapazitäten $O(1)$. Dies entspricht einem Speicherverbrauch von $O(N)$.

Der letzte Schritt, das Abrufen der optimalen geschlossenen Teilmenge, besteht aus einer Breitensuche startend vom hinzugefügten Startknoten. Hierbei wird wieder ein Speicher von $O(N)$ benötigt.

Somit ergibt für diesen Teil des Programms auch wieder ein Speicherverbrauch von $O(N) + O(1) + O(N) + O(N) = O(N)$.

Die Messdaten bilden auch hier die Vermutung ab. Es sollte jedoch erwähnt werden, dass der Speicherverbrauch in den Messdaten leicht schlechter als linear ist. Die Vermutung wird trotz der geringen Abweichung unterstützt.

6.2.7 Gesamtspeicherverbrauch

Der Gesamtspeicherverbrauch ergibt sich aus der Summe der Einzelschritte.

$$O_{\text{Gesamt}} = O(N) + O(N) + O(N) + O(1) + O(N) + O(N) = O(N).$$

Dass der gesamte Speicherverbrauch linear ist, lässt sich anhand der zugehörigen Messreihe bestätigen.

6.3 Messdaten

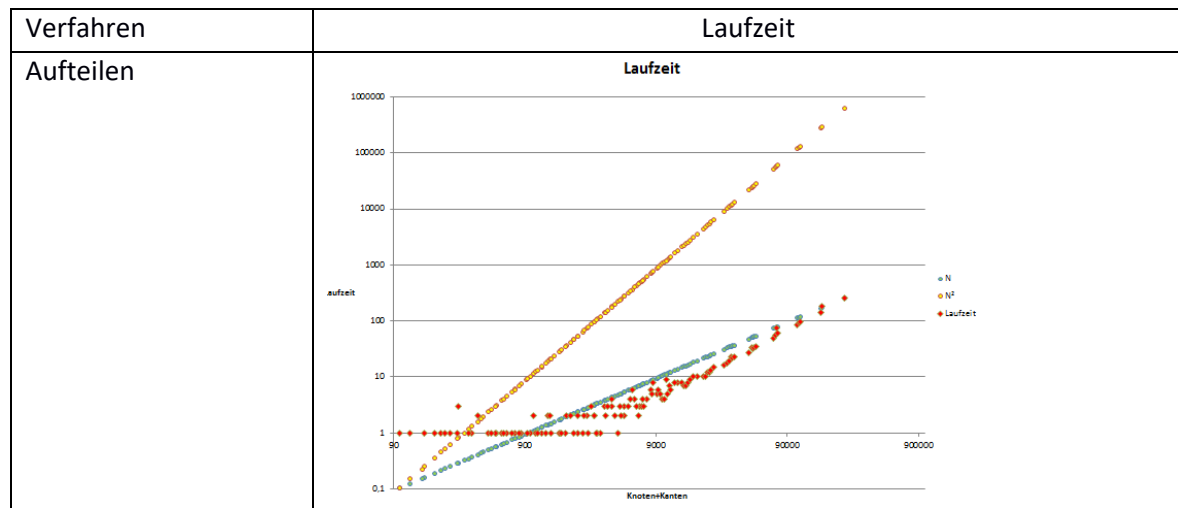
Im Folgenden sind Messungen der Laufzeit und des Speicherverbrauchs zu den einzelnen Programmteilen, sowie die Anwendung aller Programmteile zusammen, abgebildet.

Die Grafiken wurden in Excel erstellt und die Achsen sind logarithmisch skaliert, so dass man visuell etwa erkennen kann, ob es sich um ein lineares, quadratisches oder kubisches Verhalten handelt. Orientierungsgraden für quadratisches und lineares Verhalten sind zum Vergleich ebenso eingezeichnet. Der Vergleich der Vorhersagen in den vorherigen Kapiteln mit den gemessenen Ergebnissen findet sich jeweils in den zugehörigen Kapiteln wieder.

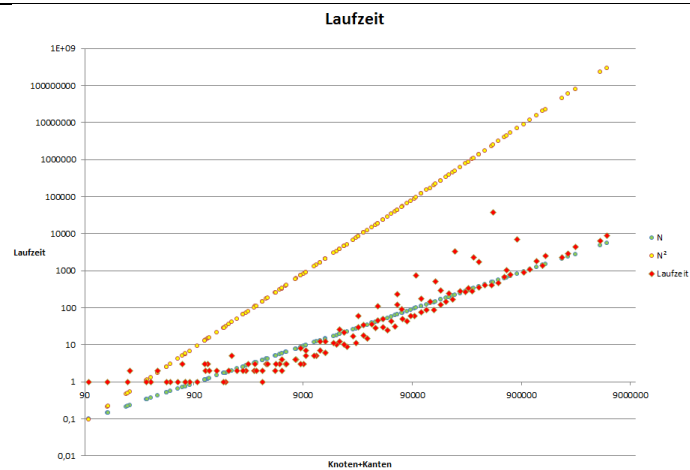
Die x-Achse gibt die Graphengröße an (Kanten + Knoten), wohingegen die y-Achse die jeweils benötigte Laufzeit bzw. Speicher angibt. Da die Laufzeit der Verfahren abhängig von dem Verhältnis von Knoten zu Kanten sein könnte, wurde dieses Verhältnis für jeden Graphen mit der gleichen Knotenanzahl variiert. Die Graphen wurden so generiert, dass jeder Knoten einen zufälligen Wert im Bereich $[-50; 50]$ besitzt. Kanten wurden immer zwischen zwei zufällig selektierten Knoten erstellt.

6.3.1 Laufzeiten

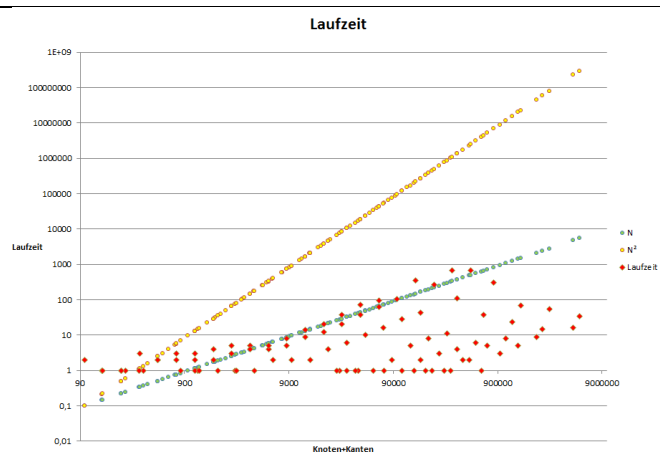
Die folgende Tabelle zeigt die Laufzeit jedes Verfahrens in einer Grafik.



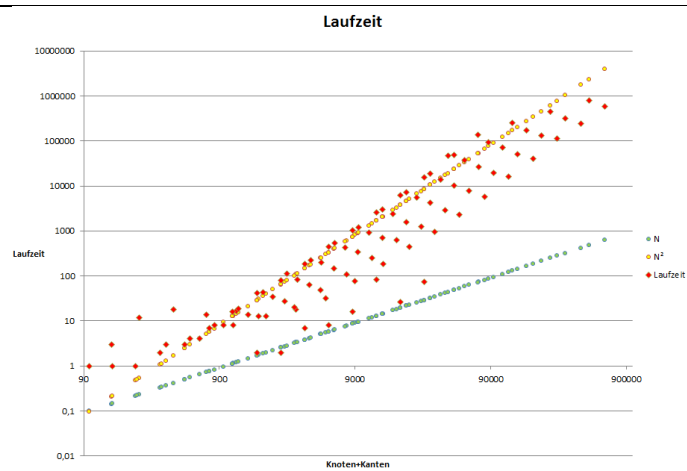
Zyklen entfernen



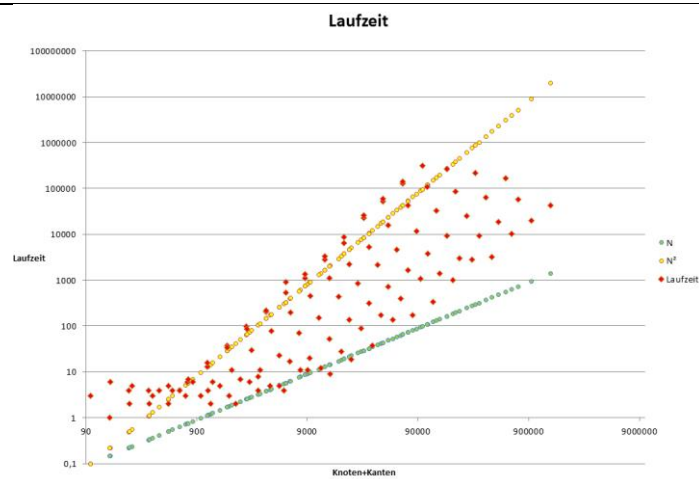
Vereinfachungsregeln



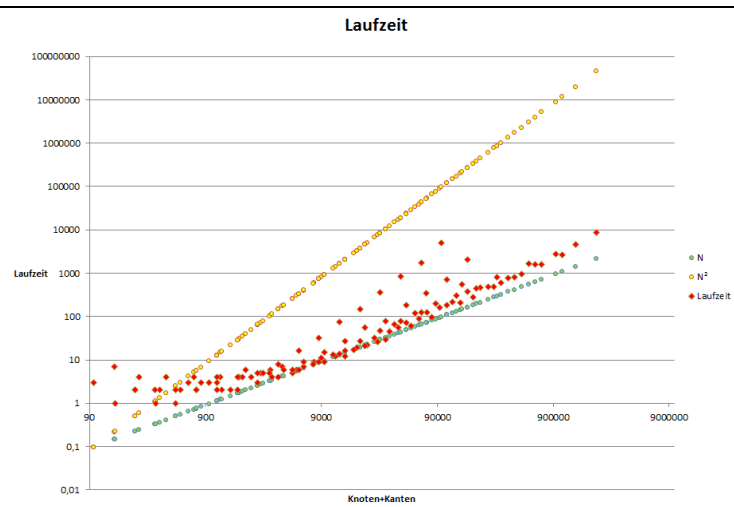
Entfernen
redundanter Kanten



Maximales Closure
bestimmen



Alle Verfahren in
korrekter Reihenfolge

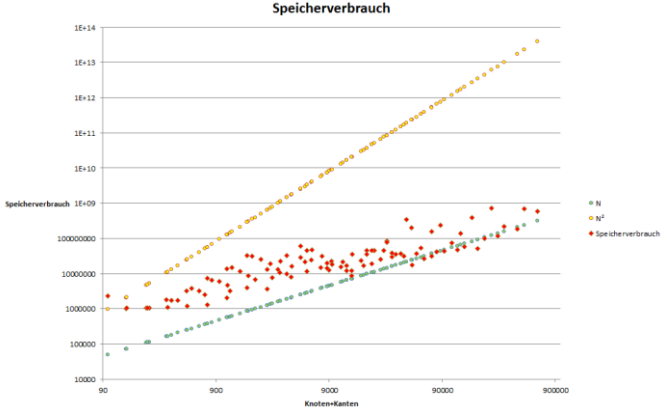
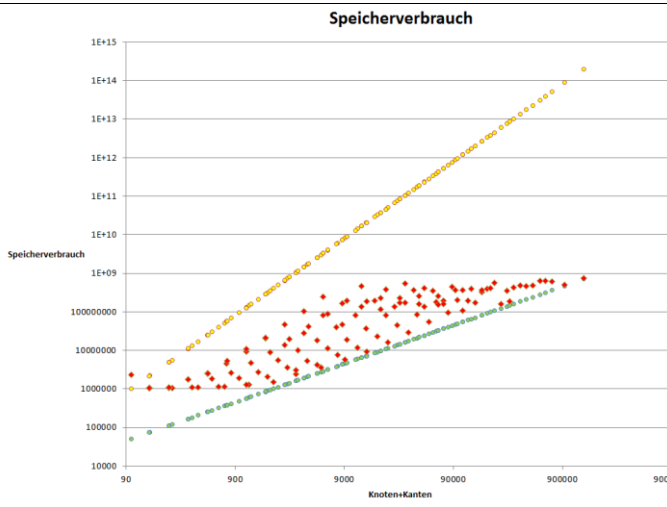
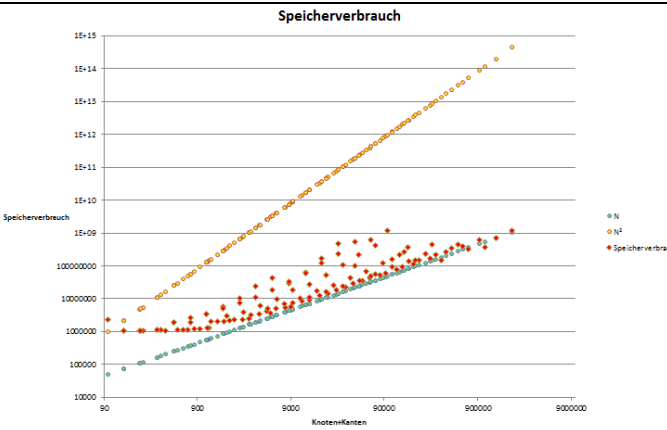


6.3.2 Speicherverbrauch

Die folgende Tabelle zeigt den Speicherverbrauch. Es wurde der Speicherverbrauch gemessen, den das Programm während des jeweiligen Verfahrens benötigt hatte. Dazu gehört einerseits der Speicher den das Programm schon vor Beginn des Verfahrens benötigt, um die Knoten- und Kantendaten zu halten sowie der Speicher den das GUI benötigt, als auch den zusätzlichen Speicher der durch das Verfahren benötigt wird.

Der in den meisten Grafiken zu erkennende konstante Speicherverbrauch für kleine Graphen ist damit zu erklären, dass die GUI und die Java-Umgebung einen konstanten Speicherverbrauch benötigen, jedoch das Verhältnis zum Speicherverbrauch des Verfahrens sowie der Kanten und Knoten des Graphen hier noch sehr groß ist.

Verfahren	Speicherverbrauch
Aufteilen	
Zyklen entfernen	
Vereinfachungsregeln	

Entfernen redundanter Kanten	
Maximales Closure bestimmen	
Alle Verfahren in korrekter Reihenfolge	

6.4 Grenzen des Programms

Neben dem Sammeln der Messdaten wurden ebenfalls die Grenzen des Programms ausgelotet. Es wurden mehrere große Graphen mit verschiedenen Knoten zu Kanten Verhältnissen getestet. Da die Dateien bzw. Lösungen zu groß sind, um sie der Einsendung beizulegen, wird hier beschrieben wie man diese erzeugen kann und wie die zu erwartende Laufzeit ist.

- Der erste Graph besteht aus 1000000 Knoten und 1000000 Kanten und wurde über zufällig erstellt mit dem Seed 1 und enthält keine Zyklen. Die Laufzeit des Programms betrug hierfür etwa 1 Minute.

- Der zweite Graph, der getestet wurde, besteht aus 500000 Knoten und 1000000 Kanten und wurde ebenfalls zyklensfrei mit dem Seed 1 generiert. Die Laufzeit betrug für diesen Graph etwa 3 Minuten.
- Auch ein Graph mit hohem Kanten zu Knotenanteil wurde getestet. Dieser dritte Graph besteht aus 5000 Knoten und 500000 Kanten. Die Laufzeit hier betrug etwa 15 Minuten.

7 Programm-Dokumentation

Die Lösungsidee wurde in Java umgesetzt. Das Programm fundiert auf den Klassen **Graph**, **Vertex**, **Edge** welchen einen gewichteten, gerichteten Graph darstellen. Die Hauptprozedur liest über **Import_Graph** ein Firmenkonglomerat ein. Der Graph wird durch **Split_Graph** in Teilgraphen unterteilt, in welchen chronologisch durch **Remove_Cycles**, **Shorten_Rules**, **Remove_Redundant_Edges**, **MinCut_MaxFlow** die optimale geschlossene Teilmenge bestimmt wird. **Solve_Graph** verbindet alle Teilverfahren, indem es diese in korrekter Reihenfolge auf einen Graphen anwendet. Anschließend wird die Lösung in das angegebene Dateiformat mittels **Export_Solution** formatiert und in eine Datei gespeichert.

Im Folgenden befindet sich eine Übersicht der erwähnenswerten, für die Lösung der Aufgabe relevanten Klassen und deren Methoden und Attribute. Die wichtigsten Quelltext-Schnipsel, verantwortlich für die Umsetzung der beschriebenen Algorithmen, befinden sich kommentiert im Anschluss. Diese Methoden sind mit ^(*) gekennzeichnet.

- **Graph - Klasse**

Die Implementierung der Algorithmen erfordert eine Klasse, welche einen gewichteten, gerichteten Graphen repräsentiert. Diese Klasse heißt **Graph** und ihre wichtigsten Methoden und Attribute sind:

Methoden:

- ***addVertex^(*)*** – Fügt dem Graphen einen Knoten mit einem Gewicht und einer festgelegten Nummer hinzu
- ***removeVertex^(*)*** – Entfernt aus dem Graphen einen Knoten inklusive all seiner Kanten. Weiterhin wird von allen Knoten, die mit ihm verbunden waren, die Kante zu ihm entfernt
- ***buyVertex^(*)*** – Entfernt einen Knoten mit ***removeVertex*** aus dem Graphen. Dieser wird zusätzlich in einer Liste gespeichert, welche sich alle Knoten merkt, die über diese Methode entfernt wurden
- ***getVertices*** – Liefert eine Liste aller im Graphen befindlichen Knoten
- ***getVertex*** – Liefert ein Knotenobjekt über dessen Wiedererkennungswert (Nummer)
- ***mergeVertices^(*)*** – Verschmilzt bestimmte Knoten zu einem einzigen Knoten und gibt dem Knoten die Information welche Knoten zu ihm zusammengefasst wurden
- ***addEdge^(*)*** – Fügt dem Graphen eine Kante zwischen zwei Knoten hinzu
- ***removeEdge^(*)*** – Entfernt eine Kante aus dem Graphen, in dem die ausgehende Kante des Quellknoten, sowie die eingehende Kante des Zielknoten entfernt werden
- ***getEdges*** – Liefert eine Liste aller im Graphen befindlichen Kanten
- ***getBoughtVertices^(*)*** – Liefert alle Knoten, die durch ***buyVertex*** aus dem Graphen entfernt wurden

Attribute:

- ***vertices*** – Dieses Attribut speichert alle Knoten des Graphen in Form eines HashSets. Dies bietet dem Vorteil gegenüber einer normalen Liste, dass mit konstanter Laufzeit überprüft werden kann, ob sich ein Knoten im Graph befindet. Außerdem können bestimmte Elemente schnell aus der Liste entfernt werden

- **vertices_by_number** – Hierbei handelt es sich um eine HashMap, die als Keywert Knotennummern und als Value das dazugehörige Knotenobjekt speichert. Dies ermöglicht es mit nahezu konstanter Laufzeit auf Knoten über ihre ID zugreifen zu können
- **edges** – Dieses Attribut speichert alle Kanten in Form eines HashSets, um wie bei **vertices** schnell überprüfen zu können, ob ein bestimmtes Element im Graph enthalten ist und um bestimmte Kanten schnell aus der Liste zu entfernen
- **bought_vertices** – Eines der wichtigsten Attribute speichert alle Knoten, die gekauft wurden. Hierzu wurde ein Stapel (Stack) verwendet. Da anfangs nicht klar ist wie viele Knoten gekauft werden und da diese zum Schluss nur noch iteriert werden müssen, bietet sich ein Stapel an durch konstante Laufzeit beim Hinzufügen eines Elements

- **Vertex - Klasse**

Die **Vertex** Klasse, die als Knotenobjekte von der **Graph** Klasse verwendet werden, besitzt folgende wichtige Methoden und Attribute.

Methoden:

- **getValue** – Der aktuelle Wert des Knoten wird abgerufen
- **changeValue** – Der Wert des Knoten wird geändert
- **getNumber** – Der einzigartige Wiedererkennungswert des Knoten wird abgerufen
- **getIncomingEdges** – Methode, die eine Liste aller eingehenden Kanten zurückliefert
- **getOutgoingEdges** – Methode, die eine Liste aller ausgehenden Kanten zurückliefert
- **getMergedVertices** – Gibt eine Liste aller Knoten zurück, die zu diesem Knoten zusammengefasst wurden

Attribute:

- **outgoing_edges** – Speichert alle ausgehenden Kanten des Knoten in Form eines HashSets. Dies bietet den Vorteil, dass nahezu konstant überprüft werden kann, ob eine bestimmte Kante mit dem Knoten verbunden ist. Außerdem ermöglicht ein HashSet ein schnelles Entfernen eines bestimmten Elements
- **incoming_edges** - Analog
- **outgoing_vertices** – Speichert alle Zielknoten aller ausgehenden Kanten innerhalb einer HashMap. Das Schlüsselement ist ein Zielknoten einer ausgehenden Kante, der Wert ist das dazugehörige Kantenobjekt. Dies bietet den Vorteil, dass schnell überprüft werden kann, ob der Knoten mit einem anderen Knoten über ausgehende Kanten verbunden ist. Dieses Attribut wird auch dazu verwendet das Kantenobjekt, das zwei Knoten verbindet, schnell zu ermitteln, indem dieses Attribut des Quellknotens nach dem Zielknoten durchsucht wird
- **incoming_vertices** - Analog

- **Edge - Klasse**

Die beiden vorherigen Klassen arbeiten beide mit Kantenobjekten. Die **Edge** Klasse ermöglicht es mit gewichteten Kanten, zu arbeiten. Die wichtigsten Methoden folgen.

Methoden:

- *getCapacity* – Die Kapazität einer Kante wird abgerufen
- *setCapacity* – Die Kapazität einer Kante wird gesetzt
- *isCapacityInfinite* – Überprüfung, ob Kante eine unendliche Kapazität besitzt
- *setCapacityInfinite* – Setzt die Kapazität einer Kante auf ∞
- *getSourceVertex* – Gibt den Quellknoten der Kante zurück, also den Knoten aus dem die Kante entspringt
- *getTargetVertex* – Gibt den Knoten zurück, auf welchen einer Kante gerichtet ist

• MinCut_MaxFlow - Klasse

Die Aufgabe der Klasse **MinCut_MaxFlow** ist es alle Knoten zu bestimmen, die zum maximalen Closure einer **Graph** Instanz gehören und diese zu kaufen. Die Klasse beinhaltet einerseits die Überführung vom Closure-Problem zum Max-Flow-Problem und die Anwendung des Edmond-Karp Algorithmus. Im Folgenden sind die wichtigsten Methoden dieser Klasse aufgelistet.

Methoden:

- *closure_problem_to_max_flow* – Passt den Graphen nach dem in 2.9.1 beschriebenen Prinzip an. Die dadurch hinzugefügten Knoten S und T werden zurückgeliefert.
- *shortest_path* – Findet den kürzesten Weg zwischen zwei Knoten. Diese Methode wird für den Edmond-Karp Algorithmus benötigt. Es wird eine bidirektionale Breitensuche verwendet. Der Weg wird anhand der Anzahl der besuchten Knoten gemessen
- *get_residual_graph* – Bildet den Residualgraphen eines Graphen unter Anwendung des Edmond-Karp Algorithmus. Durch *shortest_path* wird, solange es möglich ist, der kürzeste mögliche Weg von S zu T gefunden. Die geringste Kapazität aller auf dem Weg liegenden Kanten wird von allen Kanten subtrahiert und den jeweiligen Gegenkanten mit *setCapacity* addiert.
- *getVerticesByMinCut* – Findet alle Knoten die vom Source Knoten aus erreichbar sind, welche die Knoten sind, die auf der Seite des Source Knoten liegen würden, würde man mit einem minimalen Schnitt in den Graph schneiden.
- *buyClosure* – Das gefundene maximale Closure wird vollständig gekauft
- *derive_max_closure* – Diese Methode kombiniert die vorher genannten Methoden, der Klassen um das maximale Closure zu ermitteln und zu kaufen.

• Remove_Cycles - Klasse

Die Aufgabe der Klasse **Remove_Cycles** ist es in einer **Graph**-Instanz alle Zyklen zu vereinfachen. Die wichtigsten Funktionen sind im Folgenden zu finden.

Methoden:

- *removeAllCycles* – Entfernt alle Zyklen des Graphen auf einmal
- *removeCycleIteration* – Sucht mittels Tiefensuche im Graphen nach einem Zyklus. Wurde einer gefunden, wird dieser zusammengefasst. Zurückgeliefert wird, ob ein Zyklus vereinfacht wurde und welche Knoten als eindeutig nicht zu einem Zyklus gehörend identifiziert worden sind.

Attribute:

Jeder Knoten befindet sich immer in einem Stadium. Es werden hierzu 3 HashSets verwendet, die alle Knoten im jeweiligen Stadium speichern. Dies ermöglicht es, Knoten schnell von einem in ein anderes Stadium zu verschieben. Zusätzlich kann schnell überprüft werden in welchem Stadium sich ein Knoten befindet. Diese drei HashSets heißen:

- **graySet** (Stadium 1)
- **whiteSet** (Stadium 2)
- **blackSet** (Stadium 3)

• Shorten_Rules - Klasse

Diese Klasse hat die Aufgabe alle Vereinfachungsregeln (1a, 1b, 2a, 2b) auf den Graphen anzuwenden. Um dies zu ermöglichen ergeben sich folgende wichtige Funktionen.

Methoden:

- **apply_rules** – Iteriert den Graphen, bzw. angrenzende Knoten an vereinfachte Knoten, solange bis eine Iteration keine Vereinfachungsregel erfolgreich angewendet hat.
- **apply_rules_iteration** – Iteriert einen Graphen einmal und probiert auf jeden Knoten eine Vereinfachungsregel anzuwenden. Durch Anwendung dieser Methode können durch einzelne Vereinfachungen wieder neue Anwendungen der Vereinfachungsregeln möglich sein. Die Methode gibt zurück, ob mindestens eine Regel angewandt wurde.
- **applyRule(-Regelname-)** – Eine Vereinfachungsregeln wird auf einen Knoten angewendet. Ob der Knoten erfolgreich vereinfacht wurde, wird zurückgeliefert

• Remove_Redundant_Edges – Klasse

Diese Klasse ermöglicht es alle redundanten Kanten aus dem Graphen zu entfernen. Folgende wichtige Methoden sind daher in ihr enthalten.

Methoden:

- **removeRedundantEdges** – Iteriert alle Kanten des Graphen und entfernt alle Kanten, die redundant sind. Es wird zurückgeliefert, ob mindestens eine Kante entfernt wurde
- **isEdgeRedundant** – Prüft mittels Breitensuche, ob eine bestimmte Kante redundant ist

• Split_Graph - Klasse

Diese Klasse ist dazu da, um eine **Graph** Instanz in mehrere **Graph**-Instanzen zu unterteilen, welche dann nur noch zusammenhängende Graphen repräsentieren. Wichtige Funktionen sind im Folgenden aufgelistet.

Methoden:

- **split_graph** – Teilt den Graphen vollständig in zusammenhängende Graphen. Eine **Graph** Liste wird zurückgeliefert
- **getConnectedVertices** – Liefert alle Knoten zurück, die mit einem Knoten zusammenhängen

- **Export_Solution – Klasse**

Es wird ebenfalls eine Klasse benötigt, welche die Lösung in das gewünschte Format bringt und in eine Datei speichert. Dies wird von dieser Klasse übernommen.

Methoden:

- *printSolution* – Diese Methode funktioniert wie oben beschrieben

- **Import_Graph – Klasse**

Diese Klasse hat die Aufgabe Graphen aus einer .txt Datei einzulesen.

Methoden:

- *readGraph* – Liest einen Graphen aus einer vorgegebenen Datei ein
- *Import_graph* – Lässt den Benutzer eine Datei auswählen und gibt diese als Argument an *readGraph* und gibt erhaltenen Graph zurück

- **Solve_Graph – Klasse**

Diese Klasse verbindet die meisten aller vorher genannten Klassen, um die optimale Teilmenge aus einem Graph zu bestimmen und um die Lösung entsprechend in eine Datei speichern.

Methoden:

- *buyOptimalClosure* – Erfüllt die oben beschriebene Aufgabe. Zunächst werden die Vereinfachungsregeln angewendet mit **Shorten_Rules**, woraufhin der Graph in Teilgraphen unterteilt wird mit **Split_graph**. Danach werden diese Teilgraphen abgearbeitet. Mit **Remove_Cycles** werden Zyklen vereinfacht, danach werden wieder die Vereinfachungsregeln abwechselnd mit **Remove_redundant_edges** angewendet. Zum Schluss jedes Teilgraph wird seine optimale Teilmenge mit **MinCut_MaxFlow** bestimmt

8 Quelltextauszüge

Im Folgenden befinden kommentiert die meisten der oben beschriebenen Methoden. Attribute und simple aber wichtige Methoden wie „Getter“ und „Setter“ sind hier nicht enthalten, da deren Umsetzung uninteressant ist.

Im Code wird sich „[...]“ häufig wiederfinden. Dies ist eine Markierung dafür, dass Code ausgelassen wurde, der nicht relevant für die Lösung der Aufgabe ist (GUI, Statistiken etc.).

Die Quelltextauszüge aus den Klassen befinden sich in folgender Reihenfolge:

1. **Graph**
2. **Vertex**
3. **Edge**
4. **Solve_Graph (Löst die Aufgabe durch Aufruf der folgenden Methoden)**
5. **Shorten_Rules**
6. **Remove_Cycles**
7. **Remove_Redundant_Edges**
8. **Split_Graph**
9. **MinCut_MaxFlow**

Graph – Klasse: Datenstruktur für gerichteten, gewichteten Graph

```
/*
 * Fügt dem Graphen einen Knoten mit einem Initialgewicht und einem
 * festgelegten Wiedererkennungswert hinzu. Falls der angegebene
 * Wiedererkennungswert bereits einem anderen Knoten zugehörig ist, wird eine
 * Fehlermeldung geworfen. Der hinzugefügte Knoten wird als Objekt
 * zurückgegeben.
 */
Vertex addVertex(float value, int number) throws IllegalArgumentException {
    if (vertices_by_number.containsKey(number))
        throw new IllegalArgumentException("Duplicate identifier");
    Vertex ver = new Vertex(value, number);
    vertices.add(ver);
    vertices_by_number.put(number, ver);
    if (max_number < number)
        max_number = number;
    return ver;
}

/*
 * Ein Knoten wird gekauft und aus dem Graphen entfernt und der Liste, die die
 * gekauften Knoten speichert, hinzugefügt. Abhängige Informationen (Kanten)
 * werden in Methode 'removeVertex' mitbehandelt.
 */
void buyVertex(Vertex vertex) {
    removeVertex(vertex);
    bought_vertices.add(vertex);
    [...]
}

/*
 * Entfernt einen Knoten aus dem Graphen. Alle Kanten verbunden mit dem Knoten
 * werden ebenfalls entfernt. Wurde der Knoten entfernt, ist sein
 * Wiedererkennungswert wieder nutzbar.
 */
void removeVertex(Vertex vertex) {
    // Ausgehende Kanten entfernen
    for (Edge edg : new ArrayList<Edge>(vertex.getOutgoingEdges()))
        removeEdge(edg);
    // Eingehende Kanten entfernen
    for (Edge edg : new ArrayList<Edge>(vertex.getIncomingEdges()))
        removeEdge(edg);
    // Knoten aus Listen entfernen
    vertices.remove(vertex);
    vertices_by_number.remove(vertex.getNumber());
    [...]
}
```

```
/*
 * Verschmilzt eine Liste von Knoten mit einem Zielknoten. Es wird die Summe
 * der Gewichte aller Knoten aus der Liste auf den Zielknoten addiert.
 * Ebenfalls werden alle Kanten, verbunden mit einem der Knoten der Liste, nun
 * mit dem Zielknoten verbunden.
 */
void mergeVertices(Vertex merge_target, ArrayList<Vertex> vertices) throws IllegalArgumentException {
    float values = 0;

    for (Vertex to_merge : vertices) {
        // Falls der Knoten mit sich selber verschmolzen werden soll, wird ein
        // Fehler ausgegeben
        if (to_merge == merge_target)
            throw new IllegalArgumentException("Zielknoten kann nicht mit sich selbst gemergt werden.");
        // Alte, eingehende Kanten von Knoten aus Liste auf Mergeknoten umrichten
        for (Edge edg : new ArrayList<Edge>(to_merge.getIncomingEdges())) {
            removeEdge(edg);
            if (edg.getSourceVertex() != merge_target)
                addEdge(edg.getSourceVertex(), merge_target);
        }
        // Alte, ausgehende Kanten von Knoten der Liste von Mergeknoten kommend,
        // erzeugen
        for (Edge edg : new ArrayList<Edge>(to_merge.getOutgoingEdges())) {
            removeEdge(edg);
            if (merge_target != edg.getTargetVertex())
                addEdge(merge_target, edg.getTargetVertex());
        }

        // Knotenwert von Knoten der Liste zum Wert aller Knoten der Liste
        // addieren
        values += to_merge.getValue();
        // Knoten wird zu Liste der gemergten Knoten des Mergeknotens hinzugefügt
        merge_target.getMergedVertices().add(to_merge);
        // Falls der zu mergende Knoten selber Produkt einer Verschmelzung war,
        // werden zu ihm gemergten Knoten in den Mergeknoten übertragen
        merge_target.getMergedVertices().addAll(to_merge.getMergedVertices());
        to_merge.getMergedVertices().clear(); // Speicherverbrauch Eingrenzung!
        // Entfernen des zu verschmelzenden Knotens aus dem Graphen
        removeVertex(to_merge);
        [...]
    }
    // Gesammelte Knotenwerte, werden auf Mergeknoten addiert
    merge_target.changeValue(merge_target.getValue() + values);
}
```

```
/*
 * Eine ungewichtete Kante ausgehend von "source_vertex" und gerichtet auf
 * "target_vertex" wird dem Graphen hinzugefügt. Falls einer der beiden Knoten
 * nicht im Graphen vorhanden ist oder wenn der Zielknoten dem Quellknoten
 * entspricht, wird ein Fehler ausgegeben. Besteht bereits eine Kante
 * ausgehend vom Quellknoten gerichtet zum Zielknoten, wird die Kante nicht
 * erstellt. Die Methode gibt die hinzugefügte Kante als Objekt zurück.
 */
Edge addEdge(Vertex source_vertex, Vertex target_vertex) throws IllegalArgumentException {
    if (!vertices.contains(source_vertex) || !vertices.contains(target_vertex))
        throw new IllegalArgumentException("[addEdge] One of the selected vertices does not exist in the graph!");

    if (source_vertex == target_vertex)
        throw new IllegalArgumentException("[addEdge] source vertex equals target vertex");

    if (source_vertex.outgoingVerticesContainsVertex(target_vertex)) {
        return null;
    }

    Edge edge = new Edge(source_vertex, target_vertex);
    edges.add(edge);

    target_vertex.addIncomingEdge(edge);
    source_vertex.addOutgoingEdge(edge);

    return edge;
}

/*
 * Eine Kante wird aus dem Graph entfernt.
 */
void removeEdge(Edge edg) {
    edg.getSourceVertex().removeOutgoingEdge(edg); // Aus Quellknoten entfernen
    edg.getTargetVertex().removeIncomingEdge(edg); // Aus Zielknoten entfernen
    edges.remove(edg); // Aus Kantenliste entfernen
}
```

```
/*
 * Es wird eine Liste mit allen Wiedererkennungswerten der gekauften Knoten
 * zurückgegeben.
 */
ArrayList<Integer> getBoughtVertices() {
    if (max_number < 0)
        max_number = 0;
    ArrayList<Integer> list = new ArrayList<Integer>(max_number);
    for (Vertex bought : bought_vertices) {
        list.add(bought.getNumber());
        for (Vertex merged : bought.getMergedVertices())
            list.add(merged.getNumber());
    }
    return list;
}
```

Vertex – Klasse: Datenstruktur für einen Knoten

```
/*
 * Eine Kante, die diesen Knoten als Quellknoten hat, wird in die zugehörigen
 * Listen eingetragen. Diese Methode wird und darf nur über die addEdge
 * Methode aus der Graph-Klasse aufgerufen werden.
 */
void addOutgoingEdge(Edge edge) {
    outgoing_edges.add(edge);
    outgoing_vertices.put(edge.getTargetVertex(), edge);
}

/*
 * Entfernt eine Kante, die diesen Knoten als Quellknoten hat, aus den
 * zugehörigen Listen. Diese Methode wird darf nur über die Methode removeEdge
 * aus der Graph-Klasse aufgerufen werden.
 */
void removeOutgoingEdge(Edge edge) {
    outgoing_edges.remove(edge);
    outgoing_vertices.remove(edge.getTargetVertex());
}
```



```
/*
 * Eine Kante, gerichtet auf diesen Knoten ,wird in die zugehörigen Listen
 * gespeichert. Diese Methode wird und darf nur über die addEdge Methode aus
 * der DGraph-Klasse aufgerufen werden.
 */
void addIncomingEdge(Edge edge) {
    incoming_edges.add(edge);
    incoming_vertices.put(edge.getSourceVertex(), edge);
}

/*
 * Eine Kante gerichtet auf diesen Knoten wird aus den zugehörigen Listen
 * entfernt. Diese Methode wird und darf nur über die removeEdge Methode aus
 * der DGraph-Klasse aufgerufen werden.
 */
void removeIncomingEdge(Edge edge) {
    incoming_edges.remove(edge);
    incoming_vertices.remove(edge.getSourceVertex());
}
```

Edge – Klasse: Datenstruktur für eine gerichtete, optional auch gewichtete Kante

Da diese Klasse nur aus „Gettern und Setter“-Methoden besteht, wird hier kein Quelltext angegeben.

Solve_Graph - Klasse: Anwendung aller Verfahren - Lösung der Aufgabe

```
/*
 * Kombiniert alle implementierten Verfahren wie in der Doku genannten
 * Reihenfolge, um die maximale Teilmenge zu bestimmen. Die Lösung wird
 * ausgegeben.
 */
public static void buyOptimalClosure(Graph graph, boolean update_gui_during_algorithm) {
    [...]
    // Vereinfachungsregeln werden einmal angewendet um Anzahl der zu
    // erstellenden Teilgraphen zu verringern
    Shorten_Rules.apply_rules(graph);
    [...]
    // Der Graph wird in einzelne zusammenhängende Graphen unterteilt
    Stack<Graph> split_graphs = Split_Graph.splitGraph(graph);
    [...]
    for (Graph split_graph : split_graphs) {
        // Alle Zyklen werden aus den Teilgraphen entfernt
        Remove_Cycles.removeAllCycles(split_graph);
        // Vereinfachungsregeln und redundante Kante abwechselnd auf Teilgraphen
        // anwenden
        while (true) {
            // Alle Vereinfachungsregeln solange auf Teilgraphen anwenden bis kein
            // Knoten mehr entfernt wird
            Shorten_Rules.apply_rules(split_graph);
            // Alle redundanten Kanten werden aus dem verbleibenden Teilgraph
            // entfernt
            // Wenn mindestens eine Kante entfernt wurde, wird wieder versucht die
            // Vereinfachungsregeln anzuwenden
            // andernfalls wird die Schleife beendet
            if (!Remove_Redundant_Edges.removeRedundantEdges(split_graph))
                break;
        }
        // Die optimale Teilmenge des verbleibenden Teilgraphen wird ermittelt
        MinCut_MaxFlow.derive_max_closure(split_graph);
        // Alle zu kaufenden Knoten des Teilgraphen werden auch im Originalgraphen
        // gekauft
        for (Integer to_buy : split_graph.getBoughtVertices())
            graph.buyVertex(graph.getVertex(to_buy));
    }
    [...]
    // Lösungsausgabe
    Export_Solution.printSolution(graph);
}
```

Shorten_Rules - Klasse: Anwendung der Vereinfachungsregeln (1a,1b,2a,2b)

```
/*
 * Iteriert den Graph unter Anwendung der Vereinfachungsregeln, solange bis
 * keine Regel während einer gesamten Iteration mehr erfolgreich angewendet
 * werden konnte. Gibt zurück, ob mindestens ein Knoten vereinfacht wurde.
 */
static boolean apply_rules(Graph g) {
    int initial_vertex_count = g.getVertices().size();
    // Liste der zu untersuchenden Knoten für die nächste Iteration.
    // Nach der ersten Iteration besteht diese nur noch aus Knoten, die
    // an Knoten angrenzten, auf welche erfolgreich eine Regel angewendet wurde
    HashSet<Vertex> iteration_list = new HashSet<Vertex>(g.getVertices());
    while (apply_rules_one_iteration(g, iteration_list));
    return initial_vertex_count != g.getVertices().size();
}

/*
 * Iteriert einen Graphen einmal und versucht auf jeden Knoten eine
 * Vereinfachungsregel anzuwenden. Rückgabe, ob mindestens eine Regel
 * erfolgreich angewandt wurde.
 */
private static boolean apply_rules_one_iteration(Graph graph, HashSet<Vertex> iteration_list) {

    boolean change = false;

    // Liste in der alle Knoten gespeichert werden auf die die
    // Vereinfachungsregeln für die nächste Iteration angewendet werden sollen
    HashSet<Vertex> affectedVertices = new HashSet<Vertex>();

    for (Vertex vertex : iteration_list) {

        // Es ist möglich, dass ein zu iterierender Knoten nicht mehr im Graphen
        // existiert, da dieser eventuell schon durch eine
        // Regel vereinfacht wurde. Dieser Knoten kann deswegen auch nicht weiter
        // vereinfacht werden
        if (!graph.getVertices().contains(vertex))
            continue;

        /*
         * Vereinfachungsregeln: Wurde eine Regel erfolgreich angewandt, wird
         * markiert, dass eine Veränderung stattgefunden hat. Beim erfolgreichen
         * Anwenden einer Regel auf einen Knoten, wird keine weitere Regel auf
         * diesen angewendet, da dieser nicht mehr im Graphen existieren könnte
         * (z.B. wurde gekauft). Alle Knoten bei denen eine Vereinfachungsregel
         * anwendbar sein könnte dadurch, dass ein Knoten vereinfacht wurde,
         * werden in einer Liste gespeichert.
         */
    }
}
```

```
    */

    // Regel 1a anwenden
    if (applyRule1a(graph, vertex, affectedVertices)) {
        change = true;
        continue;
    }

    // Regel 1b anwenden
    if (applyRule1b(graph, vertex, affectedVertices)) {
        change = true;
        continue;
    }

    // Regel 2a anwenden
    if (applyRule2a(graph, vertex, affectedVertices)) {
        change = true;
        continue;
    }

    // Regel 2b anwenden
    if (applyRule2b(graph, vertex, affectedVertices)) {
        change = true;
        continue;
    }
}

iteration_list = new HashSet<Vertex>(affectedVertices);

return change;
}

/*
 * Regel 1a: Kaufen eines positiven Knoten mit keiner ausgehenden Kante
 */
private static boolean applyRule1a(Graph graph, Vertex vertex, HashSet<Vertex> affectedVertices) {
    // Überprüfung, ob Knoten positiv ist und keine ausgehenden Kanten besitzt
    if (vertex.getValue() >= 0 && vertex.getOutgoingEdges().size() == 0) {
        // Merken für nächste Iteration
        fillListWithConnectedVertices(graph, vertex, affectedVertices);
        // Kaufen und entfernen
        graph.buyVertex(vertex);
        [...]
        return true;
    }
    return false;
}
```

```
/*
 * Regel 1b: Entfernen eines negativen Knoten mit keiner eingehenden Kante
 */
private static boolean applyRule1b(Graph graph, Vertex vertex, HashSet<Vertex> affectedVertices) {
    // Überprüfung, ob Knoten negativ ist und keine eingehenden Kanten besitzt
    if (vertex.getValue() <= 0 && vertex.getIncomingEdges().size() == 0) {
        // Merken für nächste Iteration
        fillListWithConnectedVertices(graph, vertex, affectedVertices);
        // Entfernen ohne kaufen
        graph.removeVertex(vertex);
        [...]
        return true;
    }
    return false;
}

/*
 * Regel 2a: Zusammenfügen eines positiven Knotens mit nur einer ausgehenden
 * Kante mit dem Zielknoten
 */
private static boolean applyRule2a(Graph graph, Vertex vertex, HashSet<Vertex> affectedVertices) {
    // Überprüfung, ob Knoten positiv ist und exakt eine ausgehende Kante
    // besitzt
    if (vertex.getValue() >= 0 && vertex.getOutgoingEdges().size() == 1) {
        // Merken für nächste Iteration
        fillListWithConnectedVertices(graph, vertex, affectedVertices);
        fillListWithConnectedVertices(graph, vertex.getOutgoingEdges().iterator().next().getTargetVertex(),
                                     affectedVertices);
        // Zusammenfügen
        ArrayList<Vertex> merge_list = new ArrayList<Vertex>();
        merge_list.add(vertex.getOutgoingEdges().iterator().next().getTargetVertex());
        graph.mergeVertices(vertex, merge_list);
        [...]
        return true;
    }
    return false;
}
```

```
/*
 * Regel 2b: Zusammenfügen eines negativen Knotens mit nur einer eingehenden
 * Kante mit dessen Quellknoten
 */
private static boolean applyRule2b(Graph graph, Vertex vertex, HashSet<Vertex> affectedVertices) {
    // Überprüfung, ob Knoten negativ ist und exakt eine eingehende Kante
    // besitzt
    if (vertex.getValue() <= 0 && vertex.getIncomingEdges().size() == 1) {
        // Merken für nächste Iteration
        fillListWithConnectedVertices(graph, vertex, affectedVertices);
        fillListWithConnectedVertices(graph, vertex.getIncomingEdges().iterator().next().getSourceVertex(),
            affectedVertices);
        // Zusammenfügen
        ArrayList<Vertex> merge_list = new ArrayList<Vertex>();
        merge_list.add(vertex.getIncomingEdges().iterator().next().getSourceVertex());
        graph.mergeVertices(vertex, merge_list);
        [...]
        return true;
    }
    return false;
}

/*
 * Fügt einer Liste alle Knoten hinzu, die an einem Knoten angrenzen
 */
private static void fillListWithConnectedVertices(Graph graph, Vertex vertex, HashSet<Vertex> connected_list) {
    // Alle ausgehenden Knoten werden der Liste hinzugefügt
    for (Edge edge : vertex.getOutgoingEdges())
        connected_list.add(edge.getTargetVertex());
    // Alle eingehenden Knoten werden der Liste hinzugefügt
    for (Edge edge : vertex.getIncomingEdges())
        connected_list.add(edge.getSourceVertex());
}
```

Remove_Cycles - Klasse: Zusammenfassen aller Zyklen im Graph

```
/*
 * Entfernt alle Zyklen aus einem Graphen
 */
static void removeAllCycles(Graph graph) {
    HashSet<Vertex> whiteSet = new HashSet<Vertex>(graph.getVertices());
    HashSet<Vertex> blackSet = new HashSet<Vertex>();
    // Entfernt solange einen Zyklus aus dem Graphen wie dieser noch Zyklen
    // enthält
    while (removeCycleIteration(graph, whiteSet, blackSet));
}
```

```

/*
 * Findet und entfernt einen Zyklus aus einem Graphen. Liefert zurück, ob ein
 * Zyklus entfernt werden konnte. Knoten, die als nicht zu einem Zyklus
 * gehörend identifiziert wurden, werden gespeichert. Knoten, die noch nicht
 * untersucht wurden, werden gespeichert.
 */

private static boolean removeCycleIteration(Graph graph, HashSet<Vertex> stillWhite, HashSet<Vertex> ardyBlack) {
    // Alle unerforschten Knoten
    HashSet<Vertex> whiteSet = stillWhite;
    // Alle Knoten, die gerade erforscht werden
    HashSet<Vertex> graySet = new HashSet<Vertex>();
    // Alle Knoten, die nicht zu einem Zyklus gehören
    HashSet<Vertex> blackSet = ardyBlack;
    // Speichert welcher Knoten von welchem Knoten aufgerufen wurde (für
    // Backtracking)
    HashMap<Vertex, Vertex> parentMap = new HashMap<Vertex, Vertex>();

    // Der Algorithmus läuft solange bis alle Knoten erforscht wurden
    while (whiteSet.size() > 0) {
        // Ein Knoten aus der unerforschten Liste wird gewählt
        Vertex current = whiteSet.iterator().next();
        // Tiefensuche probiert einen Zyklus zu finden
        Vertex cycle_end = dfs(null, current, whiteSet, graySet, blackSet, parentMap);
        // cycle_end ist null, wenn kein Zyklus gefunden wurde
        if (cycle_end != null) {
            // Alle Knoten des Zyklus werden zum Knoten, der als Ende des Zyklus
            // markiert wurde zusammengefügt
            removeCycleFromGraphWithMap(graph, cycle_end, parentMap);
            // Alle Knoten, die sich noch in der grauen Liste befinden, werden
            // wieder in die weiße Liste verschoben
            for (Vertex gray : new ArrayList<Vertex>(graySet)) {
                if (graph.getVertices().contains(gray))
                    moveVertex(gray, graySet, whiteSet);
            }
            [...]
            // Falls ein Zyklus gefunden wurde wird abgebrochen und dies
            // entsprechend zurückgegeben
            return true;
        }
    }
    // Rückgabe, dass kein Zyklus gefunden wurde
    return false;
}

```



```
/*
 * Fasst einen Zyklus mittels Backtracking zusammen
 */
private static void removeCycleFromGraphWithMap(Graph graph, Vertex cycle_end, HashMap<Vertex, Vertex> parentMap) {
    // Liste der Knoten, die verschmolzen werden. Initialgröße ist die
    // Graphengröße, da der Zyklus nicht größer als der Graph sein kann
    ArrayList<Vertex> merge_list = new ArrayList<Vertex>(graph.getVertices().size());

    // Backtracking
    Vertex child = parentMap.get(cycle_end);
    while (child != cycle_end) {
        merge_list.add(child);
        child = parentMap.get(child);
    }
    // Liste mit den Zyklusnoten wird als Argument an die Methode zum
    // Verschmelzen dieser geliefert
    graph.mergeVertices(cycle_end, merge_list);
}
```

```

/*
 * Rekursive Tiefensuche nach einem Zyklus.
 */
private static Vertex dfs(Vertex parent, Vertex current_vertex, HashSet<Vertex> whiteSet, HashSet<Vertex> graySet,
    HashSet<Vertex> blackSet, HashMap<Vertex, Vertex> parentMap) {
    // Eintrag in die Backtrackingliste
    parentMap.put(current_vertex, parent);
    // Knoten wird in die Liste der zu erforschenden Knoten verschoben
    moveVertex(current_vertex, whiteSet, graySet);
    for (Edge edge : current_vertex.getOutgoingEdges()) {
        // Die Zielknoten aller ausgehenden Kanten werden untersucht
        Vertex neighbor = edge.getTargetVertex();
        // Falls der gefundene Knoten sich in der schwarzen Liste befindet, wurde
        // dieser bereits erforscht und ist daher uninteressant
        if (blackSet.contains(neighbor)) {
            continue;
        }
        // Falls sich der gefundene Knoten schon in der Liste der zu erforschenden
        // Knoten befindet (graue Liste), heißt es, dass ein Zyklus existiert
        if (graySet.contains(neighbor)) {
            parentMap.put(neighbor, current_vertex);
            return neighbor;
        }
        // Hier befindet sich der angrenzende Knoten in der weißen Liste.
        // Die Tiefensuche wird von diesem aus weiter fortgeführt.
        Vertex cycle_end = dfs(current_vertex, neighbor, whiteSet, graySet, blackSet, parentMap);
        if (cycle_end != null)
            return cycle_end;
    }
    // An dieser Stelle gehört der untersuchte Knoten nicht zu einem
    // Zyklus und wird daher in die schwarze Liste verschoben
    moveVertex(current_vertex, graySet, blackSet);
    return null;
}

/*
 * Verschiebt einen Knoten aus einer Liste in eine andere Liste. Der Knoten
 * wird aus einer Liste entfernt und einer anderen Liste hinzugefügt.
 */
private static void moveVertex(Vertex vertex, HashSet<Vertex> sourceSet, HashSet<Vertex> destinationSet) {
    sourceSet.remove(vertex);
    destinationSet.add(vertex);
}

```

Remove_Redundant_Edges - Klasse: Entfernt alle redundanten Kante aus einem Graph

```
/*
 * Entfernt alle redundanten Kanten des Graphen. Es wird zurückgeliefert, ob
 * mindestens eine redundante Kante entfernt wurde.
 */
static boolean removeRedundantEdges(Graph graph) {
    boolean removed_edge = false;
    // Alle Kanten werden iteriert
    for (Edge edge : new ArrayList<Edge>(graph.getEdges())) {
        // Liefert die Methode isEdgeRedundant wahr, dann ist die Kante redundant
        // und wird entfernt
        if (isEdgeRedundant(edge)) {
            graph.removeEdge(edge);
            [...]
            removed_edge = true;
        }
    }
    return removed_edge;
}
```

```
/*
 * Liefert ob eine Kante redundant ist. Es wird der Quellknoten der Kante
 * untersucht. Von hier wird eine Breitensuche über alle anderen ausgehenden
 * Kanten durchgeführt und geschaut, ob der Zielknoten der zu untersuchenden
 * Kante erreicht wird.
 */
private static boolean isEdgeRedundant(Edge suspected_edge) {
    HashSet<Vertex> visited_vertices = new HashSet<Vertex>();
    HashSet<Vertex> iteration_list = new HashSet<Vertex>();
    Stack<Vertex> in_queue = new Stack<Vertex>();

    visited_vertices.add(suspected_edge.getSourceVertex());
    // Festlegung der Startknoten der Breitensuche
    for (Edge edge : suspected_edge.getSourceVertex().getOutgoingEdges()) {
        if (edge == suspected_edge)
            continue; // Die zu untersuchende Kante wird ausgelassen
        iteration_list.add(edge.getTargetVertex());
    }

    /*
     * Breitensuche solange bis entweder keine neuen Knoten mehr gefunden werden
     * oder bis der Zielknoten der zu untersuchenden Kante gefunden wird
     */
    while (!iteration_list.isEmpty()) {
        for (Vertex vertex : iteration_list) {
            if (visited_vertices.contains(vertex))
                continue;
            visited_vertices.add(vertex);
            for (Edge edge : vertex.getOutgoingEdges()) {
                // Ist der gefundene Knoten der Zielknoten der zu untersuchenden
                // Kante, ist die zu untersuchende Kante redundant
                if (edge.getTargetVertex() == suspected_edge.getTargetVertex())
                    return true;
                in_queue.push(edge.getTargetVertex());
            }
        }
        iteration_list.clear();
        iteration_list.addAll(in_queue);
        in_queue.clear();
    }
    // Hier wurde nicht der Zielknoten der zu untersuchenden Kante nicht
    // gefunden
    // Die Kante ist daher nicht redundant
    return false;
}
```

Split_Graph - Klasse: Aufteilung eines Graphs in zusammenhängende Graphen

```
/*
 * Ein Graph wird in einzelne zusammenhängende Graphen unterteilt.
 */
static Stack<Graph> splitGraph(Graph graph) {
    Stack<Graph> graphs = new Stack<Graph>();

    HashSet<Vertex> unexplored_vertices = new HashSet<Vertex>(graph.getVertices());

    while (unexplored_vertices.size() > 0) {
        Graph part_graph = new Graph();
        // Ein noch unerforschter Knoten wird ausgewählt
        Vertex source = unexplored_vertices.iterator().next();
        // Alle Knoten, die mit dem gewählten Knoten zusammenhängen, werden
        // ermittelt
        HashSet<Vertex> connected_vertices = getConnectedVertices(source);
        // Aus allen gefundenen zusammenhängenden Knoten wird ein Teilgraph
        // gebildet
        for (Vertex vertex : connected_vertices) {
            unexplored_vertices.remove(vertex);
            part_graph.addVertex(vertex.getValue(), vertex.getNumber());
        }
        for (Vertex vertex : connected_vertices) {
            for (Edge edge : vertex.getOutgoingEdges())
                part_graph.addEdge(part_graph.getVertex(edge.getSourceVertex().getNumber()),
                                   part_graph.getVertex(edge.getTargetVertex().getNumber()));
        }

        [...]
        graphs.push(part_graph);
    }
    return graphs;
}
```

```
/*
 * Liefert alle Knoten zurück, die mit einem Knoten zusammenhängen. Dies
 * geschieht mittels Breitensuche.
 */
private static HashSet<Vertex> getConnectedVertices(Vertex source) {
    HashSet<Vertex> visited = new HashSet<Vertex>();

    HashSet<Vertex> iteration_list = new HashSet<Vertex>();
    Stack<Vertex> in_queue = new Stack<Vertex>();

    iteration_list.add(source);
    visited.add(source);

    while (!iteration_list.isEmpty()) {

        for (Vertex vertex : iteration_list) {

            for (Edge edge : vertex.getOutgoingEdges()) {
                Vertex outgoing = edge.getTargetVertex();
                if (visited.contains(outgoing))
                    continue;
                visited.add(outgoing);
                in_queue.push(outgoing);
            }

            for (Edge edge : vertex.getIncomingEdges()) {
                Vertex incoming = edge.getSourceVertex();
                if (visited.contains(incoming))
                    continue;
                visited.add(incoming);
                in_queue.push(incoming);
            }

            iteration_list.clear();
            iteration_list.addAll(in_queue);
            in_queue.clear();
        }

        return visited;
    }
}
```

MinCut_MaxFlow - Klasse: Bestimmung der optimalen Teilmenge

```
/*
 * Die optimale Teilmenge wird in einem Graphen bestimmt.
 */
static void derive_max_closure(Graph graph) {
    // Überführung von Closure-Problem zu Max-Flow-Problem
    Vertex[] source_and_target = closure_problem_to_max_flow(graph);
    // Max-Flow Problem lösen
    get_residual_graph(graph, source_and_target[0], source_and_target[1]);
    // Knoten auf Seite des Quellknotens nach minimalen Schnitt finden und
    // kaufen
    buyClosure(graph, source_and_target[0]);
    // Die hinzugefügten Knoten, die nur nötig waren um die Aufgabe zu lösen,
    // werden wieder aus dem Graph entfernt
    graph.removeVertex(source_and_target[0]);
    graph.removeVertex(source_and_target[1]);
}
```

```
/*
 * Überführung in ein Max-Flow Problem. Die hinzugefügten Knoten, werden
 * zurückgeliefert.
 */
private static Vertex[] closure_problem_to_max_flow(Graph graph) {
    ArrayList<Vertex> old_list = new ArrayList<Vertex>(graph.getVertices());
    // Quellknoten wird hinzugefügt
    Vertex source = graph.addVertex(0);
    // Zielknoten wird hinzugefügt
    Vertex target = graph.addVertex(0);
    for (Vertex vertex : old_list) {

        // Kapazität aller ursprünglichen Kanten des Graphen wird auf unendlich
        // gesetzt
        for (Edge edge : vertex.getIncomingEdges())
            edge.setCapacityInfinity();
        for (Edge edge : vertex.getOutgoingEdges())
            edge.setCapacityInfinity();

        float vertex_abs_value = Math.abs(vertex.getValue());
        // Alle positiven Knoten erhalten eine Kante, mit der Kapazität gleich dem
        // Knotengewicht, kommend von dem Quellknoten
        if (vertex.getValue() >= 0)
            graph.addEdge(source, vertex, vertex_abs_value, false, true);
        // Alle negativen Knoten erhalten eine Kante, mit Kapazität gleich dem
        // Absolutwert des Knotengewichtes, gerichtet zum Zielknoten
        else
            graph.addEdge(vertex, target, vertex_abs_value, false, true);
    }
    // Der Quell- und Zielknoten wird zurückgegeben
    return new Vertex[] { source, target };
}
```



```
/*
 * Anwendung des Edmond-Karp Algorithmus, um Residualgraph zu bilden.
 */
private static void get_residual_graph(Graph graph, Vertex source, Vertex target) {

    // Erzeugen von Gegenkanten mit der Initialkapazität 0
    for (Edge edge : new ArrayList<Edge>(graph.getEdges())) {
        graph.addEdge(edge.getTargetVertex(), edge.getSourceVertex(), 0, false, true);
    }

    // Es wird ein kürzester möglicher Weg vom Quellknoten zum Zielknoten
    // gesucht
    ArrayList<Vertex> shortest_path = new ArrayList<Vertex>(shortest_path(source, target));
    // Ist die Liste des kürzesten Weges leer, dann existiert kein erlaubter Weg
    // mehr und der Algorithmus terminiert
    while (!shortest_path.isEmpty()) {

        float min_capacity = Integer.MAX_VALUE;

        // Es wird die geringste Kapazität aller Kanten des gefundenen kürzesten
        // Weges ermittelt
        for (int i = 0; i < shortest_path.size() - 1; i++) {
            Vertex edge_source = shortest_path.get(i);
            Vertex edge_target = shortest_path.get(i + 1);
            Edge path_edge = edge_source.getOutgoingEdge(edge_target);
            if (!path_edge.isCapacityInfinite() && path_edge.getCapacity() < min_capacity)
                min_capacity = path_edge.getCapacity();
        }

        // Die geringste Kapazität wird der Kapazität aller Kanten des Weges
        // subtrahiert und zu den jeweiligen Gegenkanten addiert
        for (int i = 0; i < shortest_path.size() - 1; i++) {

            Vertex edge_source = shortest_path.get(i);
            Vertex edge_target = shortest_path.get(i + 1);
            Edge out = edge_source.getOutgoingEdge(edge_target);

            edge_source = shortest_path.get(i + 1);
            edge_target = shortest_path.get(i);
            Edge invers = edge_source.getOutgoingEdge(edge_target);

            if (!out.isCapacityInfinite())
                out.setCapacity(out.getCapacity() - min_capacity);
            if (!invers.isCapacityInfinite())
                invers.setCapacity(invers.getCapacity() + min_capacity);
        }

        // Ein kürzester Weg wird nach der Änderung der Kapazitäten erneut
        // bestimmt
    }
}
```

```

        shortest_path = new ArrayList<Vertex>(shortest_path(source, target));
    }
    /*
     * Ist das Abbruchskriterium der while-Schleife erfüllt, existiert kein
     * möglicher Weg mehr von Quell- zum Zielknoten. Damit wurde der
     * Residualgraph vollständig erstellt.
     */
}

/*
 * Es wird eine Liste von Knoten gebildet, die vom Quellknoten des
 * Residualgraph aus erreichbar sind.
 */
private static HashSet<Vertex> getVerticesByMinCut(Vertex source) {
    // Speichert alle erreichbaren Knoten
    HashSet<Vertex> visited_vertices = new HashSet<Vertex>();
    Stack<Vertex> next_vertices = new Stack<Vertex>();
    ArrayList<Vertex> iteration_list = new ArrayList<Vertex>();
    iteration_list.add(source);
    visited_vertices.add(source);

    /*
     * Eine Breitensuche wird ausgeführt. Es werden jedoch Knoten nur über
     * Kanten besucht, die eine Restkapazität von > 0 besitzen.
     */
    while (!iteration_list.isEmpty()) {

        for (Vertex vertex : iteration_list) {
            for (Edge edge : vertex.getOutgoingEdges()) {
                if (visited_vertices.contains(edge.getTargetVertex()))
                    continue;
                if (!(edge.isCapacityInfinite() || edge.getCapacity() > 0))
                    continue;
                next_vertices.push(edge.getTargetVertex());
                visited_vertices.add(edge.getTargetVertex());
            }
        }

        iteration_list.clear();
        iteration_list.addAll(next_vertices);
        next_vertices.clear();
    }
    // Da Quellknoten nicht im ursprünglichen Graphen existiert, kann dieser
    // auch nicht gekauft werden
    visited_vertices.remove(source);
    return visited_vertices;
}

```

```
/*
 * Kauft alle Knoten, die sich in der Liste befinden, die über die Methode
 * getVerticesByMinCut bestimmt werden.
 */
private static void buyClosure(Graph graph, Vertex source) {
    for (Vertex vertex : new ArrayList<Vertex>(getVerticesByMinCut(source)))
        graph.buyVertex(vertex);
}

/*
 * Findet einen kürzesten Weg zwischen zwei Knoten. Falls der Startknoten
 * gleich dem Zielknoten ist, wird eine leere Liste zurückgegeben. Wurde kein
 * kürzester Weg gefunden, wird ebenfalls eine leere Liste zurückgegeben. Beim
 * Finden eines Weges wird eine Liste, welche die Knoten des gelaufenen Weges
 * enthält, zurückgegeben.
 */
private static Stack<Vertex> shortest_path(Vertex source, Vertex target) {
    Stack<Vertex> path = new Stack<Vertex>();
    if (source == target)
        return path;
    // Aufrufliste, die über die Breitensuche startend vom Quellknoten, entsteht
    HashMap<Vertex, Vertex> parentMapSource = new HashMap<Vertex, Vertex>();
    // Aufrufliste, die über die Breitensuche startend vom Zielknoten, entsteht
    HashMap<Vertex, Vertex> parentMapTarget = new HashMap<Vertex, Vertex>();
    // Breitensuchen startend vom Quell- und Zielknoten werden ausgeführt. Der
    // Knoten, wo sich die Suchen treffen, wird erhalten
    Vertex bfs_meetup = bfs(source, target, parentMapSource, parentMapTarget);
    // Haben sich die beiden Breitensuchen nicht getroffen, existiert kein Weg,
    // was dazu führt, dass eine leere Liste zurückgegeben wird
    if (bfs_meetup == null)
        return path;

    // Über Backtracking wird der Weg vom Quellknoten zum Treffknoten bestimmt
    Vertex child = parentMapSource.get(bfs_meetup);
    path.add(bfs_meetup);
    while (child != null) {
        path.add(0, child);
        child = parentMapSource.get(child);
    }

    // Der Weg wird durch die Knoten vom Treffknoten zum Zielknoten ergänzt
    child = parentMapTarget.get(bfs_meetup);
    while (child != null) {
        path.push(child);
        child = parentMapTarget.get(child);
    }

    return path;
}
```

```
/*
 * Methode findet mit einer bidirektionalen Breitensuche den kürzesten Weg
 * zwischen zwei Knoten. Die Methode gibt den Knoten zurück an welchem sich
 * die beiden Breitensuche getroffen haben. Ist der Knoten 'null' wurde kein
 * Weg zwischen zwei Knoten gefunden.
 */
private static Vertex bfs(Vertex source, Vertex target, HashMap<Vertex, Vertex> parentMapSource,
    HashMap<Vertex, Vertex> parentMapTarget) {
    // Listen von Knoten, die die Breitensuche während eines Schrittes absuchen
    ArrayList<Vertex> iteration_list_source = new ArrayList<Vertex>();
    ArrayList<Vertex> iteration_list_target = new ArrayList<Vertex>();
    // Quell- und Zielknoten werden im ersten Schritt iteriert
    iteration_list_source.add(source);
    iteration_list_target.add(target);
    parentMapSource.put(source, null);
    parentMapTarget.put(target, null);

    while (true) {
        // Eine Iteration der Breitensuche startend vom Quellknoten
        Vertex bfs_meetup = bfsSourceToTargetIteration(iteration_list_source, parentMapSource, parentMapTarget);
        // Wurde ein Knoten gefunden, den die Suche startend vom Zielknoten
        // gefunden hat, wird dieser zurückgeliefert
        if (bfs_meetup != null)
            return bfs_meetup;
        // Hat die Suche startend vom Quellknoten keinen neuen, noch nicht
        // gefundenen Knoten gefunden, existiert kein Weg
        if (iteration_list_source.isEmpty())
            break;
        // Eine Iteration der Breitensuche startend vom Zielknoten
        bfs_meetup = bfsTargetToSourceIteration(iteration_list_target, parentMapSource, parentMapTarget);
        // Wurde ein Knoten gefunden, den die Suche startend vom
        // Quellknotengefunden hat, wird dieser zurückgeliefert
        if (bfs_meetup != null)
            return bfs_meetup;
        // Hat die Suche startend vom Zielknoten keinen neuen, noch nicht
        // gefundenen Knoten gefunden, existiert kein Weg
        if (iteration_list_target.isEmpty())
            break;
    }

    return null;
}
```

```
/*
 * Breitensuche startend von Quellknoten. Iteriert alle Knoten, die sich in
 * einer Liste befinden. Die Liste wird am Ende mit den Knoten gefüllt, die
 * während der Suche gefunden wurden. Findet diese Suche einen Knoten, der
 * bereits in der Breitensuche startend vom Zielknoten gefunden wurde, wird
 * dieser zurückgeliefert.
 */
private static Vertex bfsSourceToTargetIteration(ArrayList<Vertex> iteration_list,
    HashMap<Vertex, Vertex> parentMapSource, HashMap<Vertex, Vertex> parentMapTarget) {

    // Speichert alle Knoten, die während der Iteration gefunden werden
    Stack<Vertex> in_queue = new Stack<Vertex>();

    for (Vertex vertex : iteration_list) {
        // Alle Zielknoten der ausgehenden, nicht ausgelasteten Kanten, die noch
        // nicht besucht wurden, werden besucht
        for (Edge edge : vertex.getOutgoingEdges()) {
            // Überprüfung, ob die Kante noch nicht ausgelastet ist
            if (!edge.isCapacityInfinite() && edge.getCapacity() <= 0)
                continue;
            Vertex outgoing = edge.getTargetVertex();
            if (parentMapSource.containsKey(outgoing))
                continue;
            parentMapSource.put(outgoing, vertex);
            in_queue.push(outgoing);
            // Überprüfung, ob Knoten bereits von Breitensuche, startend vom
            // Zielknoten, gefunden wurde
            if (parentMapTarget.containsKey(outgoing))
                return outgoing;
        }
    }

    // Die Iterationliste wird mit den Knoten gefüllt, die während der Iteration
    // gefunden wurden
    iteration_list.clear();
    iteration_list.addAll(in_queue);
    in_queue.clear();

    return null;
}
```

```
/*
 * Breitensuche startend von Zielknoten. Iteriert alle Knoten, die sich in
 * einer Liste befinden. Die Liste wird am Ende mit den Knoten gefüllt, die
 * während der Suche gefunden wurden. Findet diese Suche einen Knoten, der
 * bereits in der Breitensuche startend vom Quellknoten gefunden wurde, wird
 * dieser zurückgeliefert.
 */
private static Vertex bfsTargetToSourceIteration(ArrayList<Vertex> iteration_list,
    HashMap<Vertex, Vertex> parentMapSource, HashMap<Vertex, Vertex> parentMapTarget) {

    // Speichert alle Knoten, die während der Iteration gefunden werden
    Stack<Vertex> in_queue = new Stack<Vertex>();

    for (Vertex vertex : iteration_list) {

        // Alle Quellknoten der eingehenden, nicht ausgelasteten Kanten, die noch
        // nicht besucht wurden, werden besucht
        for (Edge edge : vertex.getIncomingEdges()) {
            // Überprüfung, ob die Kante noch nicht ausgelastet ist
            if (!edge.isCapacityInfinite() && edge.getCapacity() <= 0)
                continue;
            Vertex incoming = edge.getSourceVertex();
            if (parentMapTarget.containsKey(incoming))
                continue;
            parentMapTarget.put(incoming, vertex);
            in_queue.push(incoming);
            // Überprüfung, ob Knoten bereits von Breitensuche startend vom
            // Zielknoten gefunden wurde
            if (parentMapSource.containsKey(incoming))
                return incoming;
        }

    }

    // Die Iterationliste wird mit den Knoten gefüllt, die während der Iteration
    // gefunden wurden
    iteration_list.clear();
    iteration_list.addAll(in_queue);
    in_queue.clear();

    return null;
}
```