Edmonds und Karp oder Dinic
Ein Laufzeitvergleich

Jan Niklas Hollenbeck
und
Marco Leeske

Hochschule Darmstadt

Abstract. In dieser Arbeit wird das Problem zur Findung des maximalen Flusses in Netzwerken beleuchtet. Für diese Flussprobleme gibt es unterschiedliche Algorithmen, welche auf dem von Ford und Fulkerson basieren. Die vorhandene Literatur geht vor allem auf die theoretische Grundlage und Funktion der einzelnen Algorithmen ein, aber bietet keinen zufriedenstellenden praktischen Vergleich zwischen diesen. Mit dieser Arbeit soll diese Lücke gefüllt werden und damit als Entscheidungshilfe für die Nutzung in der Praxis dienen. Basierend auf dem Algorithmus von Ford und Fulkerson untersuchen wir die beiden optimierten Algorithmen von Edmonds und Karp sowie Dinic. Ein Laufzeitvergleich wird mit Hilfe eines Programmes, welches anhand von Datensätzen die Algorithmen testet, realisiert. Die Test Daten werden so gewählt, dass man die Unterschiede der Algorithmen erkennen, ihre Laufzeit praktisch testen und die jeweiligen Vor- und Nachteile der Algorithmen aufzeigen kann. Anschließend werden die gesammelten Resultate der Laufzeittests verglichen, wodurch der theoretische Vorteil des Algorithmus von Dinic praktisch nachgewiesen wird. Trotzdem bleibt die Frage, welcher Algorithmus bei unterschiedlichen Ausgangssituationen und Erwartungen den Vorzug erhält, dies kommt unter anderem auf den Anwendungsfall und persönliche Anforderungen an.

1 Einleitung

Auf den folgenden Seiten behandeln wir das Flussproblem, welches ein mathematisches Problem zur Findung des maximalen Flusses in Netzwerken beschreibt. Solche Probleme des realen Lebens, beispielsweise in Kanal- oder Verkehrsleitsystemen, werden als gerichtete Graphen modelliert und mittels Algorithmen gelöst. Zur Lösung des Flussproblems gibt es unterschiedliche Algorithmen, welche sich in Laufzeit und Funktion unterscheiden. Diese basieren auf dem Algorithmus von Ford und Fulkerson der den Grundstein für Weiterentwicklungen gelegt hat. Die vorhandene Literatur geht vor allem auf die theoretische Grundlage und Funktion der einzelnen Algorithmen ein, aber bietet keinen zufriedenstellenden praktischen Vergleich zwischen diesen. Mit dieser Arbeit soll diese Lücke gefüllt werden und damit als Entscheidungshilfe für die Nutzung in der Praxis dienen. Basierend auf dem Algorithmus von Ford und Fulkerson untersuchen wir die

beiden optimierten Algorithmen von Edmonds und Karp sowie Dinic. Zwischen diesen werden Laufzeitvergleiche durchgeführt. Diese werden mit Hilfe eines Programmes, welches anhand von Datensätzen die Algorithmen testet, realisiert. Die Test Daten werden so gewählt, dass die Unterschiede der Algorithmen aufgezeigt, ihre Laufzeit praktisch geprüft und die jeweiligen Vor- und Nachteile der Algorithmen ersichtlich werden. Anschließend werden die gesammelten Resultate der Laufzeittests verglichen. Durch die Laufzeittests konnte der theoretische Vorteil des Algorithmus von Dinic praktisch nachgewiesen werden. Die Frage, welcher Algorithmus bei unterschiedlichen Ausgangssituationen und Erwartungen den Vorzug erhält, bleibt weiterhin bestehen, denn dies kommt unter anderem auf den Anwendungsfall und persönliche Anforderungen an.

2 Einführung

Das Flussproblem beschreibt ein mathematisches Problem in Netzwerken.
Flussprobleme können in Netzwerken mithilfe von Graphen modelliert und mittels Algorithmen gelöst oder vereinfacht werden. In den folgenden Zeilen werden die für diese Arbeit nötigen Voraussetzungen erläutert.

2.1 Algorithmen allgemein

Ein Algorithmus ist eine konkrete und eindeutige Handlungsvorschrift, um Probleme oder Klassen von Problemen zu lösen. Beispiele für einfachste Algorithmen können Gebrauchsanweisungen, Rezepte, Bauanleitungen oder Hashfunktionen sein. Wir begegnen Algorithmen im täglichen Leben wie auch bei mathematischen oder informationstechnischen Anwendungen. Algorithmen sind keine neuzeitliche Erfindung, bereits im 9. Jahrhundert beschreibt der arabische Mathematiker Al-Chwarismi (Namensgeber des Algorithmus) Algorithmen. Aus unserem heutigen Leben sind Algorithmen nicht mehr wegzudenken, Navigationssysteme zeigen uns den kürzesten Weg, Smartphones schlagen uns die nächsten zu schreibenden Worte vor oder unsere Texte werden auf Rechtschreibfehler geprüft. Das sind nur wenige von unzähligen Anwendungen, welche auf Algorithmen beruhen. Ein Algorithmus gibt die Vorgehensweise vor, wie Eingabedaten in Einzelschritten in Ausgabedaten umgewandelt werden, um ein bestimmtes Problem lösen zu können. Man spricht im Allgemeinen von Algorithmen, wenn folgende Eigenschaften erfüllt sind:

1. Ausführbarkeit
   Jeder der Einzelschritte eines Algorithmus muss ausführbar sein.

2. Endlichkeit / Finitheit
   Der Algorithmus bzw. dessen Beschreibung muss endlich sein.

3. Eindeutigkeit
   Algorithmen dürfen keine widersprüchliche Beschreibung haben, diese muss eindeutig sein.

4. Terminierung
   Ein Algorithmus muss nach endlich vielen Schritten ein Ergebnis liefern.

5. Determiniertheit
   Bei gleichen Voraussetzungen muss ein Algorithmus stets zum gleichen Ergebnis kommen.

6. Determinismus
   Der Folgeschritt muss immer bestimmt sein. Ein Algorithmus darf zu jedem Zeitpunkt nur maximal einen möglichen Schritt zu Fortsetzung haben.

   [Czernik, 2016]

2.2 Netzwerke

Hinter dem Begriff Netzwerk verbirgt sich ein System, das in unserem Fall mittels Knoten und Kanten dargestellt wird. In dieser Arbeit werden Netzwerke betrachtet, welche sich als mathematische Graphen modellieren lassen. Mithilfe solcher Netzwerke können Problemstellungen aus unserem Alltag so beschrieben werden, dass sie durch Anwendung geeigneter Algorithmen vereinfacht oder sogar gelöst werden können. Hier im speziellen werden wir uns dem (s, t)-Fluss in einem Netzwerk (G, u, s, t) widmen, wobei G einem kantenbewerteten, gerichteten Graph mit den oberen Kapazitäten u entspricht. Ein Knoten s wird als Quelle, sowie ein Knoten t als Senke bezeichnet. Die zwischen Quelle und Senke liegenden Knoten und Kanten können als Zwischenstationen aufgefasst werden. Überdies wird jeder Kante, einer Verbindung von zwei Knoten im Netzwerk, eine Kapazität u (> 0) zugewiesen. Sie gibt an, wie viel maximal durch die Kante fließen kann.

[Reintjes, 2016]

2.3 Gerichtete Graphen

Bei gerichteten, orientierten Graphen bzw. Digraphen werden die Kanten als Pfeile anstelle von Linien dargestellt. Die Pfeile beschreiben die Flussrichtung der Kanten wobei verdeutlicht wird, dass jede der Kanten nur in eine Richtung durchlaufen werden kann.

Der Graph selbst wird als $G = (V, E)$ mit einer Menge V von Knoten und einer Menge geordneter Knotenpaare $E \subseteq V \times V$ von Kanten dargestellt.

Kanten werden als $e = (a, b)$ mit a als Start- und b als Endknoten bezeichnet. Zwei Kanten e_1 und e_2 mit $e_1 = (a, b)$ und $e_2 = (b, a)$ heißen gegenläufig oder antiparallel.

Der Knoten s zeigt den Startpunkt des Flusses. Alle durch das Netzwerk zu transportierenden Mengen starten ihren Fluss an diesem Knoten, mit dem Ziel, den Endknoten t zu erreichen.

In Figure 1 unter 2.3 sieht man die Quelle auf der linken Seite, gekennzeichnet durch "s" und die Senke auf den rechten Seite dargestellt als "t" Des Weiteren sind die Kapazitäten "u" an jeder Kante angegeben.

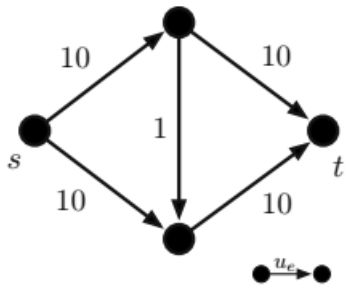


Fig. 1. Bild eines (s, t)-Netzwerkes als gerichteter Graph [Büsing, 2010]

2.4 Algorithmus von Ford und Fulkerson

Der Ford-Fulkerson Algorithmus ist der erste effiziente Algorithmus der in einem Netzwerk (G, u, s, t) den maximalen Fluss f errechnet. Der eigentliche Ablauf des Algorithmus ist sehr einfach: Man sucht einen beliebigen Pfad mit positiven Kapazitäten an allen Kanten von s nach t und bestimmt den maximal möglichen Fluss (geringste Kapazität auf dem Weg von s nach t). Anschließend merkt man sich den Fluss, ändert die Kapazitäten an den Kanten entsprechend des ersten Durchlaufs ab und iteriert diesen Schritt bis kein Weg mehr zu finden ist. Die erhaltenen Flüsse werden aufaddiert und das Ergebnis entspricht dem maximalen Fluss.

Ablauf:

Input: Netzwerk (G, u, s, t). Output: Maximaler Fluss f.

Schritt 1: Setzen Sie $f(e) = 0$ für alle Kanten $e \subseteq E$.

Schritt 2: Bestimmen Sie G^f und $u^f(e)$.

Schritt 3: Konstruieren Sie einen einfachen (s,t)-Weg p in G^f. Falls keiner existiert: STOPP.

Schritt 4: Verändern Sie den Fluss f entlang des Wegs p um $\gamma := min_{e \subseteq p} u^f(e)$.

Schritt 5: Gehen Sie zu Schritt 2

[Büsing, 2010]

2.5 Die Breitensuche

Im Folgenden wird die Breitensuche behandelt, da sie zum Verständnis der Algorithmen von Edmonds und Karp und Dinic benötigt wird. Die Breitensuche (breadth-first search) ist ein Suchalgorithmus für Graphen, der zunächst alle von Ursprung ausgehenden Knoten markiert, bevor die Folgeknoten untersucht werden (siehe Figur 2). Mit ihm ist es möglich, den kürzesten Pfad zwischen zwei Knoten zu finden. In unserem Anwendungsfall wird die Quelle s als Startknoten definiert und von ihr ausgehend alle Pfade zur Senke t gesucht. Der kürzeste Weg wird dann zurückgeliefert.

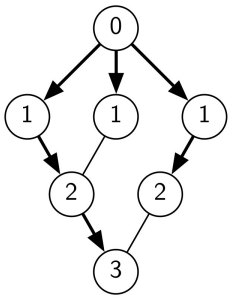


Fig. 2. Die Ebenen werden nacheinander abgearbeitet, wenn ein Knoten bereits besucht wurde, muss dieser nicht noch einmal markiert werden.

2.6 Algorithmus von Edmonds und Karp

Der Algorithmus von Edmonds und Karp 2.6 ist eine Weiterentwicklung des Ford und Fulkerson Algorithmus der 1972 publiziert wurde. Er unterscheidet sich zum Ford Fulkerson durch seine zusätzliche Breitensuche 2.5, welche immer den

kürzesten Weg von s nach t, gemessen an der Anzahl von Kanten, liefert. Entlang dieser Kanten wird der Fluss erhöht. Im Vergleich zum Ablauf in 2.4 ändert sich Schritt Nr. 3 von einem einfachen (s,t)-Weg p in G^f zu dem Kürzesten. Sobald von der Breitensuche kein (s,t)-Weg mehr gefunden wird, ist der Maximalfluss erreicht.

2.7 Algorithmus von Dinic

Dinic's Algorithmus ist dem von Edmonds und Karp sehr ähnlich. Auch hier wird im Gegensatz zu Ford und Fulkerson eine Breitensuche 2.5 durchgeführt. Der Unterschied zu Edmonds und Karp zeigt sich im Umgang mit den gefundenen Wegen. Bei Dinic wird nicht nur der kürzeste (s,t)-Weg p in G^f durchlaufen und berechnet, stattdessen wird ein blockierender (s,t)-Fluss aus allen kürzesten (s,t)-Wegen zusammengesetzt, anschließend abgearbeitet und auf den Fluss addiert. Sobald keine weiteren Wege mehr gefunden werden, ist der maximale Fluss erreicht.

3 Testablauf und Prüfungskriterien

Wir führen im Bezug auf die Mathematische Laufzeit eine Evaluierung der genannten Maximal-Fluss Algorithmen durch. Außerdem beschäftigen wir uns mit der praktischen Implementierung der Algorithmen. Zuletzt wird ein Laufzeittest der Algorithmen von Edmonds und Karp sowie Dinic anhand von Testdaten durchgeführt. Für die Laufzeittests wurde ein Java Programm an unser Bedürfnisse angepasst, welches den maximalen Fluss errechnet. Für jede Berechnung des Flusses wird die Zeit gemessen. Da bei wenigen Durchläufen eine genaue Zeitmessung unmöglich ist, werden jeweils einhundert Messungen erstellt. Diese bestehen aus einer Million Durchläufe der Algorithmen, von denen der Mittelwert errechnet wird. Erst bei dieser Anzahl an Durchläufen konnte ein aussagekräftiges Ergebnis erzielt werden. Die erhaltenen Ergebnisse werden zur einfacheren Analyse in eine .csv Datei (Comma-separated values) geschrieben. Die Messungen der unterschiedlichen Paare sollen zeigen, dass Dinic allgemein schneller ist. Der Vergleich innerhalb der Paare soll zeigen, dass der Algorithmus von Dinic bei der Erhöhung der Kantenanzahl einen größer werdenden prozentualen Vorteil zu verzeichnen hat. Die Messungen der Tests werden außerdem in einem Zweistichproben-t-Test auf einen signifikanten Unterschied getestet (siehe Anhang **??**). Es wird auf $H_0 : \mu = \mu_1$ geprüft, die Stichproben unterscheiden sich nicht. Des Weiteren wird auf $H_1 : \mu > \mu_1$ geprüft, ob die Edmonds und Karp Stichprobe größer ist als Dinic und damit langsamer.

4 Implementierung und Test

Es wurden beide Algorithmen in unser Programm implementiert und wir konnten feststellen, dass beide Algorithmen ohne Probleme implementierbar sind. Allerdings ist Dinic durch seine zusätzliche Berechnung des blockierenden Flusses komplexer, daher ist der Implementierungsaufwand größer.

4.1 Laufzeitvergleich

Die Laufzeiten in der Theorie (Fig. 3) setzen sich aus den inneren und äußeren Schleifendurchläufen zusammen. Die Laufzeit des Ford Fulkerson ist vom gewählten Weg bzw. von der maximalen Kapazität C einer Kante, der Anzahl der Knoten n und Kanten m abhängig. Durch den Einfluss der Kapazität hat der Algorithmus von Ford und Fulkerson einen Nachteil gegenüber den anderen. Deshalb vergleichen wir in unserem Laufzeittest nur die Algorithmen von Edmonds und Karp sowie Dinic.

Algorithmus	Laufzeit	Bemerkungen
Ford-Fulkerson	$\mathcal{O}(nmC)$	bei ganzzahligen Kapazitäten
Edmonds-Karp	$\mathcal{O}(nm^2)$	Erhöhung längs kürzester Wege
Dinic	$\mathcal{O}(n^2m)$	Benutzung blockierender Flüsse

Fig. 3. Allgemeine Laufzeiten in der Theorie [Noltemeier, 2009]

Unser Test wurde mit jeweils drei Testpaaren durchgeführt, die sich in ihrer Knotenanzahl unterscheiden. Die einzelnen Paare unterscheiden sich nur in der Anzahl ihrer Kanten. Bei allen Graphen konnte beobachtet werden, dass die Durchschnittswerte der Laufzeit von Dinic unter den von Edmonds und Karp liegen (Siehe Figur 4-6). Außerdem werden die Durchschnittszeiten für komplexere Graphen größer, dies kann sowohl bei den Graphenpaaren als auch bei den Testgruppen beobachtet werden. Dies deckt sich mit unseren Erwartungen hinsichtlich der theoretischen Laufzeit.

Die Paartests bezüglich des Verhältnisses bei mehr kanten, lieferten keine eindeutigen Ergebnisse. Beim ersten Testgraphen 1b (Fig. 4.2) wird der Dinic Algorithmus trotz steigender Kantenanzahl langsamer. Dies liegt vermutlich an schlecht gewählten zusätzlichen Kanten und der geringen Knotenanzahl des Graphen. Die weiteren Verhältnisse verändern sich wie erwartet, deshalb nehmen wir an, dass ein Zusammenhang besteht. Ohne weitere Tests kann dies jedoch nicht eindeutig bestätigt werden.

Der t-Test zeigt bei allen Vergleichen von Edmonds und Karp mit Dinic, dass die Laufzeiten signifikant sind und die Hypothese H_1 mit mehr als 99,99% angenommen wird. (Siehe Figur 4-6).

[Hemmerich, 2017]

	Test 1 mit 6 Knoten			
	9 Kanten		12 kanten	
	Edmonds Karp	Dinic	Edmonds Karp	Dinic
Summe (in ns)	118635	82415	153214	121022
Durchschnitt (in ns)	1186,35	824,15	1532,14	1210,22
Verhältnis Dinic zu Edmonds Karp	0,69		0,79	
t-Wert	29,2442257177558		16,0452959418458	

Fig. 4. Die Ergebnisse des 1. Tests

	Test 2 mit 8 Knoten			
	16 Kanten		20 kanten	
	Edmonds Karp	Dinic	Edmonds Karp	Dinic
Summe (in ns)	330642	201353	388427	216326
Durchschnitt (in ns)	3306,42	2013,53	3884,27	2163,26
Verhältnis Dinic zu Edmonds Karp	0,61		0,56	
t-Wert	36,6829158187241		48,8506373058474	

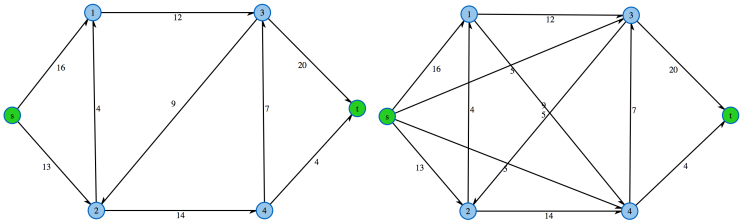
Fig. 5. Die Ergebnisse des 3. Tests

	Test 3 mit 11 Knoten			
	22 Kanten		27 Kanten	
	Edmonds Karp	Dinic	Edmonds Karp	Dinic
Summe (in ns)	543458	348911	609461	287072
Durchschnitt (in ns)	5434,58	3489,11	6094,61	2870,72
Verhältnis Dinic zu Edmonds Karp	0,64		0,47	
t-Wert	33,0874385632811		19,1231900801111	

Fig. 6. Die Ergebnisse des 3. Tests

4.2 Verwendete Graphen

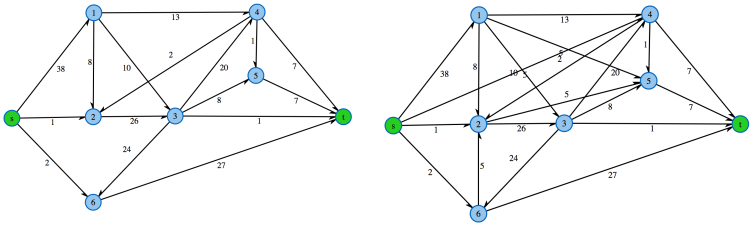
Hier eine bildliche Darstellung der in unserem Test genutzten Graphen.



(a) Graph 1a,
6 Knoten und 9 Kanten

(b) Graph 1b,
6 Knoten und 12 Kanten

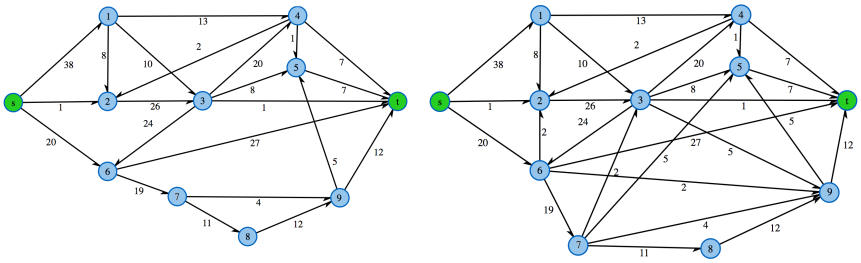
Fig. 7. Die Graphen des 1. Laufzeittests



(a) Graph 2a,
8 Knoten und 16 Kanten

(b) Graph 2b,
8 Knoten und 20 Kanten

Fig. 8. Die Graphen des 2. Laufzeittests



(a) Graph 3a,
11 Knoten und 22 Kanten

(b) Graph 3b, 11 Knoten und 27 Kanten

Fig. 9. Die Graphen des 3. Laufzeittests

[Fischer, 2015]

5 Related Work

Wie schon in der Einleitung 1 erwähnt wurde, konnten wir keine, für uns befriedigende, vorangegangene Arbeit mit einem direkten, praktischen Laufzeitvergleich der beiden Algorithmen von Edmonds und Karp sowie dem von Dinic finden.

6 Zusammenfassung

Anhand der unter Abschnitt 4 durchgeführten Laufzeittests, zeichnet sich der Vorteil des Dinic Algorithmus deutlich ab. Über je mehr Kanten ein Graph verfügt, desto vorteilhafter ist die Nutzung des Dinic Algorithmus. Fairerweise sollte aber gesagt sein, dass der Vorteil erst bei sehr großen Graphen spürbar zum tragen kommt. Der Zeitvorteil des Dinic Algorithmus bewegt sich bei den von uns gewählten Graphen im Nanosekundenbereich, aber die prozentuale Überlegenheit bei steigender Kantenanzahl war deutlich zu erkennen. Dem Algorithmus von Dinic den Vorzug zu geben, ist unserer Ansicht nach erst bei großen Projekten oder sehr großen Graphen lohnenswert. Für die Abwasser oder Kanalberechnung eines Hauses beispielsweise, liegt der zeitliche Vorteil in einem nicht merklichen Bereich und ist daher zu Vernachlässigen. Sollte aber die Routenberechnung einer Strecke durch das Straßennetz eines ganzen Landes das Ziel sein, bietet der Algoithmus von Dinic deutliche Zeitvorteile. Demnach ist die Entscheidung, welcher Algorithmus gewählt wird, vom Anwendungsfall und den zeitlichen Voraussetzungen abhängig.

Bibliography

Christina Büsing. *Graphen- und Netzwerkoptimierung*, volume 1. March 2010.

Agnieszka Czernik. Was ist ein algorithmus – definition und beispiele, 10 2016. URL `https://www.datenschutzbeauftragter-info.de/was-ist-ein-algorithmus-definition-und-beispiele/`.

Quirin Fischer. Was ist ein algorithmus – definition und beispiele, 10 2015. URL `https://www-m9.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_de.html`.

Wanja Hemmerich. t-test, 2017. URL `http://matheguru.com/stochastik/267-t-test.html`.

Sven Oliver Krumke , Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*, volume 2. July 2009.

Christian Reintjes. Eine mathematische optimierungsmodell zur statischen anordnung von fachwerktraegern. pages 17–21, April 2016.

7 Anhänge

7.1 Angang 1:

Komplette Testergebnisse in tabellarischer Form

Tabelle 1

	Test 1 mit 6 Knoten				Test 2 mit 8 Knoten				Test 3 mit 11 Knoten			
	9 Kanten		12 kanten		16 Kanten		20 kanten		22 Kanten		27 Kanten	
	Edmonds Karp	Dinic	Edmonds Karp	Dinic	Edmonds Karp	Dinic	Edmonds Karp	Dinic	Edmonds Karp	Dinic	Edmonds Karp	Dinic
	2014	895	2346	1274	4273	1993	4781	2142	5900	3369	6300	2678
	1159	835	1537	1171	3123	1959	3686	2166	5079	3359	5526	2717
	1179	838	1468	1233	3312	2005	3806	2127	5432	3875	5525	2649
	1208	924	1625	1249	3272	1940	3636	2135	6215	5157	5648	2652
	1548	842	1475	1186	3184	1981	3725	2122	5190	3340	5553	2708
	1152	839	2296	1459	3328	1964	3696	2034	5671	3898	5614	2644
	1072	782	1656	1408	3170	2080	3596	2193	5902	3793	5576	2656
	1083	784	1649	1586	3879	2259	3862	2037	6383	4059	5618	2744
	1100	832	1899	1162	3637	2559	3953	2076	5239	3423	5523	2738
	1125	793	1582	1450	4159	2469	3979	2099	5608	3394	5627	2690
	1122	798	1623	1156	3461	2083	3637	2135	5185	3248	5764	2648
	1156	787	1428	1146	3404	2099	4034	2265	5021	3292	5598	2706
	1062	782	1535	1179	3288	2022	3814	2086	5044	3241	5708	2676
	1073	827	1406	1170	3223	1922	3834	2705	5030	3292	5605	2838
	1088	793	1577	1206	3307	2011	5219	2380	5088	3644	5580	2629
	1247	835	1521	1159	3194	1975	5413	2495	5165	3332	5664	2693
	1238	824	1498	1179	3206	2118	4160	2373	5298	3315	5769	2684
	1228	996	1502	1185	3441	1954	4426	2339	5322	3436	5614	2633
	1249	832	1484	1195	3322	1911	3819	2068	5185	3638	5639	2694
	1249	819	1540	1209	3435	2056	4009	2335	6082	3379	5637	2640
	1218	779	1491	1192	3202	1983	3732	2023	5343	3258	5815	2647
	1417	1036	1519	1195	3268	2064	3595	2050	5075	3262	5714	2746
	1421	869	1466	1181	3346	1949	3727	2017	5143	3265	5668	2703
	1260	796	1486	1211	3282	1951	3582	2022	5077	3254	6112	2693
	1161	844	1505	1163	3279	2021	3714	2094	5207	3475	5772	2669
	1149	821	1421	1130	3282	1910	3591	2006	5284	3337	5641	2724
	1139	822	1420	1131	3249	2000	3611	2106	5129	3310	5596	2648
	1197	832	1424	1128	3440	1983	3669	2025	5158	3566	5606	2765
	1152	843	1445	1181	4197	2189	3559	2037	7280	3563	5668	2698
	1174	823	1464	1175	5694	2389	3727	2022	5278	3395	5657	2709
	1148	820	1435	1218	3316	1978	3581	2037	5323	3428	5737	2782
	1176	790	1478	1191	3387	2884	3673	2299	5309	3472	5669	2711
	1112	805	1480	1186	3323	2006	3594	2368	5216	3372	5880	2725
	1149	835	1486	1236	3300	2027	4687	2160	5444	3629	5802	2803
	1231	851	1500	1151	3304	2013	3787	2036	5350	3634	5718	2687
	1243	811	1489	1184	3199	1907	3673	2170	5401	3514	5843	2726
	1205	813	1462	1160	3352	2131	3839	2070	5458	3301	5695	2739
	1124	786	1462	1162	3519	1909	4748	2224	5279	3344	5662	2694
	1124	818	1488	1207	3223	1908	4100	2084	5352	3617	5718	2707
	1118	797	1463	1131	3343	1941	3772	2250	6196	3668	5619	2689
	1151	856	1444	1158	3286	2056	4147	2172	5223	3305	5645	2676
	1238	834	1418	1153	3249	2044	3912	2154	5037	3405	5733	2661
	1211	793	1420	1169	3181	1887	3942	2142	5903	3399	5786	2816
	1127	794	1500	1215	3158	1991	3703	2021	5174	3386	5824	2733
	1117	795	1499	1191	3279	2373	3623	2195	5245	3654	5684	2648
	1121	845	1472	1164	3367	2082	3904	2098	5877	3835	5706	2663
	1205	791	1471	1152	3544	2039	3636	2045	5143	3435	5702	2666
	1227	841	1423	1183	3155	2034	3779	2092	5275	3309	5990	2697
	1231	812	1452	1200	3203	1960	3658	2050	5496	3430	5640	2651
	1141	829	1510	1166	3399	1979	3751	2144	5237	3428	5682	2760
	1168	803	1421	1131	3145	1975	3718	2180	5268	3382	5643	2664
	1123	797	1419	1132	3173	2060	3759	2161	5046	3257	5680	2738
	1109	822	1474	1149	3310	1967	3876	2065	5015	3239	5922	2759
	1194	808	1525	1225	3285	1883	3981	2084	5035	3254	5773	2706
	1213	835	1511	1507	3146	1952	3814	2140	5205	3964	5573	2641
	1145	809	1522	1203	3145	1891	3745	2050	5368	3393	5997	2667
	1169	808	1495	1149	3093	1966	3724	2151	5495	4843	5611	2708
	1143	820	1513	1274	3251	1933	3753	2035	5879	3409	5718	2643
	1123	791	1567	1229	3100	1938	3736	2129	5208	3391	5655	2640
	1141	792	1527	1211	3208	2100	3745	2056	5285	3317	5590	2672
	1252	813	1503	1230	3248	1978	3738	2031	6382	4547	5799	2668
	1127	807	1503	1158	3165	1950	3740	2038	5168	3378	5696	2653
	1186	790	1537	1252	3244	2012	4022	2575	5156	3315	5712	2744
	1155	799	1533	1209	3206	1957	4137	2146	7008	4045	5630	2664
	1116	790	1467	1194	3170	1931	3688	2221	5120	3556	5743	2671
	1142	806	1429	1142	3242	1884	4034	2108	7411	3478	5633	2660
	1195	814	1422	1126	3060	1899	3620	2008	5909	3344	5662	2691
	1229	878	1466	1172	3107	1988	3577	2088	5362	3348	5725	2685
	1141	826	1482	1177	3226	2097	3682	2024	5257	3308	5665	2655
	1167	807	1454	1133	3133	1910	3601	2020	5303	3357	5620	2715
	1111	795	1420	1136	3430	2141	4018	2163	5337	3588	5761	2649
	1149	979	1420	1137	3244	2112	3759	2044	5240	3325	5632	2643
	1206	858	1505	1185	3188	1993	3751	2129	5281	3337	5702	2633
	1207	843	1470	1125	3232	1890	3776	2050	5312	3290	5758	2741
	1125	788	1408	1145	3080	1884	3959	2339	5178	3301	5595	2631
	1430	881	1416	1212	3185	1963	4394	2167	5153	3489	5788	2706
	1244	833	1511	1180	3150	1895	3886	2047	5254	3656	5616	2675
	1110	864	1517	1212	3197	2039	3697	2089	5284	3376	5710	2708
	1357	869	1536	1179	3233	2129	3755	2141	5333	3867	5859	2765
	1220	819	1475	1179	3159	1997	3824	2335	5915	3595	5680	2712
	1190	785	1491	1196	3610	2114	3849	2157	6937	3327	5749	2708
	1144	795	2153	1648	3084	1901	3788	2212	5643	3347	5623	2702
	1093	791	2230	1551	3085	1898	4102	2091	5183	3262	5617	2744
	1136	822	2015	1377	3177	1959	4080	2801	5350	3306	5738	2775
	1396	818	1476	1155	3164	1885	4077	2163	5312	3321	5579	2729
	1232	841	1442	1298	3063	1942	3875	2261	5233	3832	5772	4576
	1177	794	1802	1287	3283	1968	4029	2170	6743	4589	7360	3089
	1127	782	1423	1177	3123	1931	3960	2177	5238	3348	7569	4596
	1073	786	1610	1139	3282	1987	3923	2175	5105	3313	10953	3695
	1100	831	1425	1137	3110	1884	3814	2692	5039	3295	18703	6309

	1105	829	1554	1203	3091	1944	3947	2174	5037	3255	10564	4651
	1177	820	1594	1146	3230	1940	3793	2278	5077	3234	9251	5053
	1180	829	1427	1182	3361	1987	3908	2184	5528	3296	10143	4893
	1179	835	1505	1205	3279	2047	3723	2131	5303	3296	6430	2988
	1153	808	1457	1258	3319	2018	3756	2153	5214	3463	6263	3225
	1216	815	1635	1221	3196	1895	3874	2024	5689	3683	6546	3056
	1106	785	1464	1375	3116	1981	4258	2633	5253	3283	6440	3018
	1090	795	1457	1184	3126	1889	3931	2089	5199	3262	6376	2959
	1124	842	1439	1145	3069	1932	3836	2050	5118	3259	6172	2961
	1201	830	1527	1219	3175	1989	4294	2372	5139	3257	6393	2885
Summe	118635	82415	153214	121022	330642	201353	388427	216326	543458	348911	609461	287072
Durchschnitt	1186,35	824,15	1532,14	1210,22	3306,42	2013,53	3884,27	2163,26	5434,58	3489,11	6094,61	2870,72
Verhältnis	0,69469380874		0,789888652473012		0,608975871183939		0,556928328875181		0,64202017451210 6		0,471026037761235	

7.2 Angang 2:

Code des Java Testprogramms

File - /Users/Kyodu/Documents/wai1/Wissenschaftliches-arbeiten/MaximumFlow/src/Dinics.java

```java
/**
 * Created by Kyodu on 25.03.17.
 */

import java.util.*;

public class Dinics {

        static class Edge {
            int t, rev, cap, f;

            public Edge(int t, int rev, int cap) {
                this.t = t;
                this.rev = rev;
                this.cap = cap;
            }
        }

        private List<Edge>[] graph;

        public Dinics(){

            /*graph = createGraph(6); //erster test wenig
            addEdge(graph, 0, 1, 16);
            addEdge(graph, 0, 2, 13);
            addEdge(graph, 1, 3, 12);
            addEdge(graph, 2, 1, 4);
            addEdge(graph, 2, 4, 14);
            addEdge(graph, 3, 2, 9);
            addEdge(graph, 3, 5, 20);
            addEdge(graph, 4, 3, 7);
            addEdge(graph, 4, 5, 4);
*/

            /*graph = createGraph(6); //erster test viel
            addEdge(graph, 0, 1, 16);
            addEdge(graph, 0, 2, 13);
            addEdge(graph, 1, 3, 12);
            addEdge(graph, 2, 1, 4);
            addEdge(graph, 2, 4, 14);
            addEdge(graph, 3, 2, 9);
            addEdge(graph, 3, 5, 20);
            addEdge(graph, 4, 3, 7);
            addEdge(graph, 4, 5, 4);
            //zusätzliche Kanten
            addEdge(graph, 0, 3, 5);
            addEdge(graph, 0, 4, 5);
            addEdge(graph, 1, 4, 5);
            */

```

```
 51
 52              /*graph = createGraph(8); //zweiter Test kurz
 53              addEdge(graph, 0, 1, 38);
 54              addEdge(graph, 0, 2, 1);
 55              addEdge(graph, 0, 6, 2);
 56              addEdge(graph, 1, 4, 13);
 57              addEdge(graph, 1, 2, 8);
 58              addEdge(graph, 1, 3, 10);
 59              addEdge(graph, 2, 3, 26);
 60              addEdge(graph, 3, 4, 20);
 61              addEdge(graph, 3, 5, 8);
 62              addEdge(graph, 3, 6, 24);
 63              addEdge(graph, 3, 7, 1);
 64              addEdge(graph, 4, 5, 1);
 65              addEdge(graph, 4, 2, 2);
 66              addEdge(graph, 4, 7, 7);
 67              addEdge(graph, 5, 7, 7);
 68              addEdge(graph, 6, 7, 27);
 69 */
 70              /*graph = createGraph(8); //zweiter Test lang
 71              addEdge(graph, 0, 1, 38);
 72              addEdge(graph, 0, 2, 1);
 73              addEdge(graph, 0, 6, 2);
 74              addEdge(graph, 1, 4, 13);
 75              addEdge(graph, 1, 2, 8);
 76              addEdge(graph, 1, 3, 10);
 77              addEdge(graph, 2, 3, 26);
 78              addEdge(graph, 3, 4, 20);
 79              addEdge(graph, 3, 5, 8);
 80              addEdge(graph, 3, 6, 24);
 81              addEdge(graph, 3, 7, 1);
 82              addEdge(graph, 4, 5, 1);
 83              addEdge(graph, 4, 2, 2);
 84              addEdge(graph, 4, 7, 7);
 85              addEdge(graph, 5, 7, 7);
 86              addEdge(graph, 6, 7, 27);
 87              //zusätzliche kanten
 88              addEdge(graph, 0, 4, 5);
 89              addEdge(graph, 1, 5, 5);
 90              addEdge(graph, 2, 5, 5);
 91              addEdge(graph, 6, 2, 5);*/
 92
 93              /*graph = createGraph(11); //dritter test kurz
 94              addEdge(graph, 0, 1, 38);
 95              addEdge(graph, 0, 2, 1);
 96              addEdge(graph, 0, 6, 20);
 97              addEdge(graph, 1, 4, 13);
 98              addEdge(graph, 1, 2, 8);
 99              addEdge(graph, 1, 3, 10);
100              addEdge(graph, 2, 3, 26);
```

```
101             addEdge(graph, 3, 4, 20);
102             addEdge(graph, 3, 5, 8);
103             addEdge(graph, 3, 6, 24);
104             addEdge(graph, 3, 10, 1);
105             addEdge(graph, 4, 5, 1);
106             addEdge(graph, 4, 2, 2);
107             addEdge(graph, 4, 10, 7);
108             addEdge(graph, 5, 10, 7);
109             addEdge(graph, 6, 10, 27);
110             addEdge(graph, 6, 7, 19);
111             addEdge(graph, 7, 8, 11);
112             addEdge(graph, 7, 9, 4);
113             addEdge(graph, 8, 9, 12);
114             addEdge(graph, 9, 5, 5);
115             addEdge(graph, 9, 10, 12);*/
116
117             graph = createGraph(11); //dritter test kurz
118             addEdge(graph, 0, 1, 38);
119             addEdge(graph, 0, 2, 1);
120             addEdge(graph, 0, 6, 20);
121             addEdge(graph, 1, 4, 13);
122             addEdge(graph, 1, 2, 8);
123             addEdge(graph, 1, 3, 10);
124             addEdge(graph, 2, 3, 26);
125             addEdge(graph, 3, 4, 20);
126             addEdge(graph, 3, 5, 8);
127             addEdge(graph, 3, 6, 24);
128             addEdge(graph, 3, 10, 1);
129             addEdge(graph, 4, 5, 1);
130             addEdge(graph, 4, 2, 2);
131             addEdge(graph, 4, 10, 7);
132             addEdge(graph, 5, 10, 7);
133             addEdge(graph, 6, 10, 27);
134             addEdge(graph, 6, 7, 19);
135             addEdge(graph, 7, 8, 11);
136             addEdge(graph, 7, 9, 4);
137             addEdge(graph, 8, 9, 12);
138             addEdge(graph, 9, 5, 5);
139             addEdge(graph, 9, 10, 12);
140             //zusätzliche kanten
141             addEdge(graph, 6, 2, 2);
142             addEdge(graph, 6, 9, 2);
143             addEdge(graph, 7, 3, 2);
144             addEdge(graph, 7, 5, 5);
145             addEdge(graph, 3, 9, 5);
146
147
148         }
149
150         public int Maxflow(){
```

```
151             //return maxFlow(graph, 0, 5); //erster test
152             //return maxFlow(graph, 0, 7); //zweiter test
153             return maxFlow(graph, 0, 10); //dritter test
154 
155         }
156 
157         public static List<Edge>[] createGraph(int nodes)
    {
158             List<Edge>[] graph = new List[nodes];
159             for (int i = 0; i < nodes; i++)
160                 graph[i] = new ArrayList<>();
161             return graph;
162         }
163 
164         public static void addEdge(List<Edge>[] graph, int
     s, int t, int cap) {
165             graph[s].add(new Edge(t, graph[t].size(), cap)
    );
166             graph[t].add(new Edge(s, graph[s].size() - 1,
    0));
167         }
168 
169         static boolean dinicBfs(List<Edge>[] graph, int
    src, int dest, int[] dist) {
170             Arrays.fill(dist, -1);
171             dist[src] = 0;
172             int[] Q = new int[graph.length];
173             int sizeQ = 0;
174             Q[sizeQ++] = src;
175             for (int i = 0; i < sizeQ; i++) {
176                 int u = Q[i];
177                 for (Edge e : graph[u]) {
178                     if (dist[e.t] < 0 && e.f < e.cap) {
179                         dist[e.t] = dist[u] + 1;
180                         Q[sizeQ++] = e.t;
181                     }
182                 }
183             }
184             return dist[dest] >= 0;
185         }
186 
187         static int dinicDfs(List<Edge>[] graph, int[] ptr,
     int[] dist, int dest, int u, int f) {
188             if (u == dest)
189                 return f;
190             for (; ptr[u] < graph[u].size(); ++ptr[u]) {
191                 Edge e = graph[u].get(ptr[u]);
192                 if (dist[e.t] == dist[u] + 1 && e.f < e.
    cap) {
193                     int df = dinicDfs(graph, ptr, dist,
```

```
193 dest, e.t, Math.min(f, e.cap - e.f));
194                     if (df > 0) {
195                         e.f += df;
196                         graph[e.t].get(e.rev).f -= df;
197                         return df;
198                     }
199                 }
200             }
201             return 0;
202         }
203 
204         public static int maxFlow(List<Edge>[] graph, int
    src, int dest) {
205             int flow = 0;
206             int[] dist = new int[graph.length];
207             while (dinicBfs(graph, src, dest, dist)) {
208                 int[] ptr = new int[graph.length];
209                 while (true) {
210                     int df = dinicDfs(graph, ptr, dist,
    dest, src, Integer.MAX_VALUE);
211                     if (df == 0)
212                         break;
213                     flow += df;
214                 }
215             }
216             return flow;
217         }
218 
219 }
```

File - /Users/Kyodu/Documents/wai1/Wissenschaftliches-arbeiten/MaximumFlow/src/EdmondsKarp.java

```java
/**
 * Created by Kyodu on 25.03.17.
 */
// Java program for implementation of Ford Fulkerson algorithm
import java.util.*;
import java.lang.*;
import java.io.*;
import java.util.LinkedList;

public class EdmondsKarp {
    /*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */


        private final int V;
        public EdmondsKarp(int notes){
            V = notes; //Number of vertices in graph
        }
        /* Returns true if there is a path from source 's' to sink
          't' in residual graph. Also fills parent[] to store the
            path */
        boolean bfs(int rGraph[][], int s, int t, int parent[])
        {
            // Create a visited array and mark all vertices as not
            // visited
            boolean visited[] = new boolean[V];
            for(int i=0; i<V; ++i)
                visited[i]=false;

            // Create a queue, enqueue source vertex and mark
            // source vertex as visited
            LinkedList<Integer> queue = new LinkedList<Integer>();
            queue.add(s);
            visited[s] = true;
            parent[s]=-1;

            // Standard BFS Loop
            while (queue.size()!=0)
            {
```

```
43                  int u = queue.poll();
44
45                  for (int v=0; v<V; v++)
46                  {
47                      if (visited[v]==false && rGraph[u][v] >
    0)
48                      {
49                          queue.add(v);
50                          parent[v] = u;
51                          visited[v] = true;
52                      }
53                  }
54              }
55
56              // If we reached sink in BFS starting from
   source, then
57              // return true, else false
58              return (visited[t] == true);
59          }
60
61          // Returns tne maximum flow from s to t in the
   given graph
62          int fordFulkerson(int graph[][], int s, int t)
63          {
64              int u, v;
65
66              // Create a residual graph and fill the
   residual graph
67              // with given capacities in the original graph
   as
68              // residual capacities in residual graph
69
70              // Residual graph where rGraph[i][j] indicates
71              // residual capacity of edge from i to j (if
   there
72              // is an edge. If rGraph[i][j] is 0, then there
    is
73              // not)
74              int rGraph[][] = new int[V][V];
75
76              for (u = 0; u < V; u++)
77                  for (v = 0; v < V; v++)
78                      rGraph[u][v] = graph[u][v];
79
80              // This array is filled by BFS and to store
   path
81              int parent[] = new int[V];
82
83              int max_flow = 0;  // There is no flow
   initially
```

```
 84
 85              // Augment the flow while there is path from
    source
 86              // to sink
 87              while (bfs(rGraph, s, t, parent))
 88              {
 89                  // Find minimum residual capacity of the
    edges
 90                  // along the path filled by BFS. Or we can
     say
 91                  // find the maximum flow through the path
    found.
 92                  int path_flow = Integer.MAX_VALUE;
 93                  for (v=t; v!=s; v=parent[v])
 94                  {
 95                      u = parent[v];
 96                      path_flow = Math.min(path_flow, rGraph
    [u][v]);
 97                  }
 98
 99                  // update residual capacities of the edges
     and
100                  // reverse edges along the path
101                  for (v=t; v != s; v=parent[v])
102                  {
103                      u = parent[v];
104                      rGraph[u][v] -= path_flow;
105                      rGraph[v][u] += path_flow;
106                  }
107
108                  // Add path flow to overall flow
109                  max_flow += path_flow;
110              }
111
112              // Return the overall flow
113              return max_flow;
114          }
115
116 }
117
```

File - /Users/Kyodu/Documents/wai1/Wissenschaftliches-arbeiten/MaximumFlow/src/MaximumFlow.java

```java
/**
 * Created by Kyodu on 25.03.17.
 */

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.io.*;


public class MaximumFlow {

    public static void main(String...arg)throws IOException
{

        int[][] graph;
        int numberOfNodes;
        int source;
        int sink;
        int maxFlowD =0;
        int maxFlowK =0;
        long timeStart;
        long timeEnd;
        String filename = "log3_viel.csv";
        FileWriter fw = new FileWriter(filename, true); //
the true will append the new data

       /* Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the number of nodes");
        numberOfNodes = scanner.nextInt();
        graph = new int[numberOfNodes + 1][numberOfNodes +
1];

        /*System.out.println("Enter the graph matrix");
        for (int sourceVertex = 0; sourceVertex <=
numberOfNodes; sourceVertex++)
        {
            for (int destinationVertex = 0;
destinationVertex <= numberOfNodes; destinationVertex++)
            {
                graph[sourceVertex][destinationVertex] =
scanner.nextInt();
            }
        }

        System.out.println("Enter the source of the graph
");
        source= scanner.nextInt();

        System.out.println("Enter the sink of the graph");
```

```
44          sink = scanner.nextInt();
45  */
46          /*FordFulkerson fordFulkerson = new FordFulkerson(
    numberOfNodes);
47          long timeStart = System.nanoTime();
48          maxFlow = fordFulkerson.fordFulkerson(graph, source
    , sink);
49          long timeEnd = System.nanoTime();
50          System.out.println("The Max Flow is " + maxFlow);
51          System.out.println("Verlaufszeit der Schleife
    : " + (timeEnd - timeStart) + " Nanosekunden.");
52          */
53
54
55          // Usage example
56          // Driver program to test above functions
57          // Let us create a graph shown in the above example
58
59
60          /*int graph2[][] =new int[][]{ //test 1 wenig
61                  {0, 16, 13, 0, 0, 0}, //0
62                  {0, 0, 0, 12, 0, 0}, //1
63                  {0, 4, 0, 0, 14, 0},  //2
64                  {0, 0, 9, 0, 0, 20},  //3
65                  {0, 0, 0, 7, 0, 4},   //4
66                  {0, 0, 0, 0, 0, 0}    //5
67
68
69          };*/
70
71         /* int graph2[][] =new int[][] { //test 1 viel
72                  {0, 16, 13, 5, 5, 0}, //0
73                  {0, 0, 0, 12, 5, 0}, //1
74                  {0, 4, 0, 0, 14, 0},  //2
75                  {0, 0, 9, 0, 0, 20},  //3
76                  {0, 0, 0, 7, 0, 4},   //4
77                  {0, 0, 0, 0, 0, 0}    //5
78  };*/
79
80          /*int graph2[][] =new int[][] { //zweiter test
    wenig
81                  {0, 38, 1, 0, 0, 0, 2 ,0}, //0
82                  {0, 0, 8, 10, 13, 0, 0 ,0}, //1
83                  {0, 0, 0, 26, 0, 0, 0 ,0}, //2
84                  {0, 0, 0, 0, 20, 8, 24, 1}, //3
85                  {0, 0, 2, 0, 0, 1, 0 ,7}, //4
86                  {0, 0, 0, 0, 0, 0, 0 ,7}, //5
87                  {0, 0, 0, 0, 0, 0, 0 ,27}, //6
88                  {0, 0, 0, 0, 0, 0, 0 ,0}, //7
89          };*/
```

```
 90
 91         /*int graph2[][] =new int[][] { //zweiter test
    viel
 92                 {0, 38, 1, 0, 5, 0, 2 ,0}, //0
 93                 {0, 0, 8, 10, 13, 5, 0 ,0}, //1
 94                 {0, 0, 0, 26, 0, 5, 0 ,0}, //2
 95                 {0, 0, 0, 0, 20, 8, 24, 1}, //3
 96                 {0, 0, 2, 0, 0, 1, 0 ,7}, //4
 97                 {0, 0, 0, 0, 0, 0, 0 ,7}, //5
 98                 {0, 0, 5, 0, 0, 0, 0 ,27}, //6
 99                 {0, 0, 0, 0, 0, 0, 0 ,0}, //7
100         };*/
101 /*
102         int graph2[][] =new int[][]{ //dritter test wenig
103                 {0, 38, 1, 0, 0, 0, 20, 0, 0, 0, 0}, //0
104                 {0, 0, 8, 10, 13, 0, 0, 0, 0, 0, 0}, //1
105                 {0, 0, 0, 26, 0, 0, 0, 0, 0, 0, 0}, //2
106                 {0, 0, 0, 0, 20, 8, 24, 0, 0, 0, 1}, //3
107                 {0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 7}, //4
108                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7}, //5
109                 {0, 0, 0, 0, 0, 0, 0, 19, 0, 0, 27}, //6
110                 {0, 0, 0, 0, 0, 0, 0, 0, 11, 4, 0}, //7
111                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0}, //8
112                 {0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 12}, //9
113                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} //10
114         };*/
115
116         int graph2[][] =new int[][]{ //dritter test mit
    viel
117                 {0, 38, 1, 0, 0, 0, 20, 0, 0, 0, 0}, //0
118                 {0, 0, 8, 10, 13, 0, 0, 0, 0, 0, 0}, //1
119                 {0, 0, 0, 26, 0, 0, 0, 0, 0, 0, 0}, //2
120                 {0, 0, 0, 0, 20, 8, 24, 0, 0, 5, 1}, //3
121                 {0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 7}, //4
122                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7}, //5
123                 {0, 0, 2, 0, 0, 0, 0, 19, 0, 2, 27}, //6
124                 {0, 0, 0, 2, 0, 5, 0, 0, 11, 4, 0}, //7
125                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0}, //8
126                 {0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 12}, //9
127                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} //10
128         };
129
130         //EdmondsKarp m = new EdmondsKarp(6); //erster
    test
131         //EdmondsKarp m = new EdmondsKarp(8); //zweiter
    test
132         EdmondsKarp m = new EdmondsKarp(11); //dritter
    test
133
134         long tmp =0;
```

```
135         try {
136             fw.write("Edmonds Karp" + "," + "Dinic \n");//
    appends the string to the file
137         } catch (IOException ioe) {
138             System.err.println("IOException: " + ioe.
    getMessage());
139         }
140         int faktor =1000000; //zum versuch 10000000
141         for(int j= 0 ;j<100; j++) {
142             long TimeK =0;
143             long TimeD =0;
144             for (int i = 0; i < faktor; i++) {
145                 timeStart = System.nanoTime();
146                 //maxFlowK = m.fordFulkerson(graph2, 0, 5
    ); //erster test
147                 //maxFlowK = m.fordFulkerson(graph2, 0, 7
    ); //zweiter test
148                 maxFlowK = m.fordFulkerson(graph2, 0, 10);
     //dritter test
149                 timeEnd = System.nanoTime();
150                 TimeK = TimeK + (timeEnd - timeStart);
151 
152             }
153 
154             for (int i = 0; i < faktor; i++) {
155                 Dinics d = new Dinics();
156                 timeStart = System.nanoTime();
157                 maxFlowD = d.Maxflow();
158                 timeEnd = System.nanoTime();
159                 TimeD = TimeD + (timeEnd - timeStart);
160 
161             }
162             try {
163                 fw.write(TimeK/faktor + "," + TimeD/faktor
    +"\n");//appends the string to the file
164 
165             } catch (IOException ioe2) {
166                 System.err.println("IOException: " + ioe2.
    getMessage());
167             }
168             System.out.println("Durchlauf :" + j + " Max
    Flow Karp: " + maxFlowK + " Max Flow Dinic: "+ maxFlowD);
169         }
170         fw.close();
171         //scanner.close();
172     }
173 }
174 
175 
```