Appendix to
Random Sampling Of Finite Graphs With Constraints

Evelina Andersson

Contents

A Triangle free graphs

A.1 Generated graphs based on edges

Colours, $p = 1$

size #colours	100	200	300	400	500
7	8				
8	68				
9	23				
10	1	12			
11		66			
12		22	13		
13			70		
14			17	22	
15				67	1
16				11	58
17					41

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
7	23				
8	63				
9	14				
10		33			
11		65			
12		2	16		
13			75	1	
14			8	45	
15			1	51	16
16				3	72
17					12

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
6	36				
7	62				
8	2	69			
9		31	44		
10			56	47	
11				50	49
12				3	51

Colours, $p = \frac{1}{n}$

size #colours	100	200	300	400	500
3	92	79	79	62	51
4	8	21	21	38	49

Edges, $p = 1$

size	100	200	300	400	500
median	1852	5590	10600	16662	23650
$\bar{x}$	1852.02	5590.88	10602.78	16666.22	23652.92
σ^2	788.48	1781.04	3134.29	3510.70	5465.65
σ	28.08	42.20	55.98	59.25	73.93

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	1733	5282	10068	15875	22597
$\bar{x}$	1733.94	5282.58	10070.84	15875.76	22588.64
σ^2	555.19	1496.67	2588.38	2882.12	4119.34
σ	23.56	38.69	50.88	53.69	64.18

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	1128	3216	5935	9157	12859
$\bar{x}$	1127.80	3217.64	5937.78	9156.68	12856.34
σ^2	567.56	1369.89	3177.62	4932.38	4732.13
σ	23.82	37.01	56.37	70.23	69.79

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	196	389	590	803	996
$\bar{x}$	196.94	392.42	591.50	797.92	994.22
σ^2	356.16	681.88	1195.51	1594.90	2630.54
σ	18.87	26.11	34.58	39.94	51.29

A.2 Generated graphs based on vertices

Colours, $p = 1$

size #colours	100	200	300	400	500
2	100	100	100	100	100

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
5	34	3			
6	52	30	5	2	
7	14	40	17	12	5
8		25	58	34	24
9		2	19	39	34
10			1	13	29
11					8

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
5	53				
6	47	10			
7		86	24		
8		4	76	51	2
9				49	90
10					8

Colours, $p = \frac{1}{n}$

size #colours	100	200	300	400	500
2	25	6	2		
3	75	94	98	99	99
4				1	1

Edges, $p = 1$

size	100	200	300	400	500
median	3699	17335	34630	61375	86635
$\bar{x}$	3297.68	14639.34	29028.74	52573	80740.76
σ^2	2040095.57	31352972.04	214922389.71	673291298.46	1324826020.67
σ	1428.32	5599.37	14660.23	25947.86	36398.16

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	1783	6171	12732	21922	31984
$\bar{x}$	1800.16	6153.88	12894.76	22126	32860.54
σ^2	22810.08	385716.35	1606396.75	7853050.67	15018890.65
σ	151.03	621.06	1267.44	2802.33	3875.42

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	790	2248	4126	6359	8886
$\bar{x}$	790.36	2248.50	4126.74	6353.64	8889.08
σ^2	582.09	1401.08	4303.89	5492.80	7388.88
σ	24.13	37.43	65.60	74.11	85.96

Edges,$p = \frac{1}{n}$

size	100	200	300	400	500
median	98	198	298	398	496
$\bar{x}$	100.44	199.66	297.30	398.74	498.14
σ^2	234.39	366.71	662.49	769.59	1483.01
σ	15.31	19.15	25.74	27.74	38.51

B Four cycle free graphs

B.1 Generated graphs based on edges

Colours, $p = 1$

size #colours	100	200	300	400	500
5	11				
6	85	14			
7	4	83	44	1	
8		3	56	88	50
9				11	48
10					2

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
5	19				
6	81	11			
7		87	53	1	
8		2	47	92	67
9				7	33

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
5	76				
6	24	88	11		
7		12	88	86	43
8			1	14	57

Colours, $p = \frac{1}{n}$

size #colours	100	200	300	400	500
3	95	84	67	69	56
4	5	16	33	31	44

Edges, $p = 1$

size	100	200	300	400	500
median	756	1971	3444	5128	6966
$\bar{x}$	755.72	1971.28	3445.18	5125.06	6966.18
σ^2	40.57	88.20	187.32	194.06	298.756
σ	6.37	9.39	13.69	13.93	17.28

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	743	1940	3398	5052	6874
$\bar{x}$	742.36	1940.20	3397.6	5051.44	6871.86
σ^2	39.06	82.63	169.37	233.06	263.70
σ	6.25	9.09	13.01	15.27	16.24

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	648	1674	2904	4293	5813
$\bar{x}$	648.82	1672.64	2903.04	4291.76	5812.38
σ^2	75.72	146.66	254.91	572.95	621.96
σ	8.70	12.11	15.97	23.94	24.94

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	198	390	596	796	994
$\bar{x}$	195	393.50	598	793.64	996.68
σ^2	387.35	918.41	1019.64	1385.89	2095.78
σ	19.68	30.31	31.93	37.23	45.78

Triangles, $p = 1$

size	100	200	300	400	500
median	58	124.50	171.50	204	226
$\bar{x}$	57.73	125.27	171.95	203.730	226.02
σ^2	16.18	50.60	83.95	107.94	108.52
σ	4.02	7.11	9.16	10.39	10.42

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	55	119.50	165	193	219
$\bar{x}$	54.58	120.46	164.89	194.37	218.80
σ^2	18.12	37.60	73.19	128.86	119.60
σ	4.26	6.13	8.56	11.35	10.94

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	36	76	103	120	131
$\bar{x}$	35.84	75.85	101.56	119.57	131.23
σ^2	14.66	41.10	57.34	68.25	76.60
σ	3.83	6.41	7.57	8.26	8.75

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	1	1	1	1	1
$\bar{x}$	1.31	1.17	0.99	0.89	0.81
σ^2	1.41	1.29	1.08	0.85	0.70
σ	1.19	1.14	1.04	0.85	0.84

B.2 Generated graphs based on vertices

Colours, $p = 1$

size #colours	100	200	300	400	500
3	23	13	13	11	6
4	70	66	52	58	55
5	7	19	33	28	32
6		2	2	3	6
7					1

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
4	25				
5	72	71	31	17	5
6	3	29	66	75	68
7			3	8	27

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
4	9				
5	90	37			
6	1	62	90	48	2
7		1	10	52	97
8					1

Colours, $p = \frac{1}{n}$

size #colours	100	200	300	400	500
2	30	7	3		1
3	70	93	97	100	99

Edges, $p = 1$

size	100	200	300	400	500
median	334	687	1032	1375	1743
$\bar{x}$	346.74	719.48	1112.96	1441.06	1842.24
σ^2	2137.83	15304.70	46292.24	63123.43	119563.98
σ	46.24	123.71	215.16	251.24	345.78

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	513	1127	1827	2459	3244
$\bar{x}$	509.52	1155	1857.54	2473.76	3300.72
σ^2	3281.71	17715.19	61995.54	120683.82	204673.74
σ	57.29	133.10	248.99	347.40	452.41

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	562	1459	2545	3788	5140
$\bar{x}$	562.10	1459.50	2548.92	3786.86	5140.26
σ^2	156.47	396.64	528.44	842.12	1424.46
σ	12.51	19.92	22.99	29.02	37.74

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	98	200	298	406	502
$\bar{x}$	98.24	197.48	296.04	402.36	501.76
σ^2	175.78	433.18	489.05	698.78	946.53
σ	13.26	20.81	22.11	26.43	30.77

Triangles, $p = 1$

size	100	200	300	400	500
median	49	99	144	149.50	126
$\bar{x}$	49.43	94.71	122.87	148.76	153.97
σ^2	1.94	118.27	940.78	2514.49	5257.81
σ	1.39	10.88	30.67	50.14	72.51

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	51	105	143	175	179
$\bar{x}$	50.93	104.80	141.64	175.04	182.84
σ^2	12.17	39.66	276.17	574.50	1102.24
σ	3.49	6.30	16.62	23.97	33.20

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	27	59	83	100.50	117
$\bar{x}$	26.77	58.40	83.57	101.81	116.62
σ^2	16.66	31.35	45.66	75.10	83.55
σ	4.08	5.60	6.76	8.67	9.14

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	0	0	0	0	0
$\bar{x}$	0.15	0.16	0.14	0.17	0.12
σ^2	0.17	0.16	0.20	0.16	0.11
σ	0.41	0.39	0.45	0.40	0.33

C Tetrahedron free graphs

C.1 Generated graphs based on edges

Colours, $p = 1$

size #colours	100	200	300	400	500
11	26				
12	56				
13	18				
14					
15		7			
16		43			
17		42			
18		8	1		
19			20		
20			64		
21			15	1	
22				35	
23				55	
24				9	25
25					53
26					20
27					2

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
10	1				
11	32				
12	53				
13	14				
14					
15		19			
16		53			
17		27			
18		1	3		
19			53		
20			40	1	
21			2	7	
22				59	
23				33	3
24					45
25					43
26					9

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
8	26				
9	71				
10	3	3			
11		70			
12		26	11		
13		1	74	1	
14			14	53	
15			1	45	31
16				1	64
17					5

Colours, $p = \frac{1}{n}$

size #colours	100	200	300	400	500
3	84	74	71	65	54
4	16	26	29	35	46

Edges, $p = 1$

size	100	200	300	400	500
median	2978	9273	17929	28565	40993
$\bar{x}$	2982.30	9270.74	17928.74	28571.18	40997.64
σ^2	572.31	1091.69	2987.12	3826.00	4374.41
σ	23.92	33.04	54.65	61.86	66.14

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2842	8917	17306	27636	39724
$\bar{x}$	2844.98	8915.1	17302.78	27633.46	39727.80
σ^2	416.28	1169.16	2827.30	2644.59	5172.32
σ	20.40	34.19	53.17	51.43	71.92

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	1715	5015	9370	14562	20530
$\bar{x}$	1715.66	5012.90	9364.02	14570.38	20534.04
σ^2	1345.22	4023.30	8813.86	19467.61	27033.70
σ	36.68	63.43	93.89	139.53	164.42

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	198	400	596	800	996
$\bar{x}$	196.10	397.72	594.64	793.50	998.14
σ^2	447.55	698.10	1368.07	1547.51	2145.15
σ	21.16	26.42	36.99	39.34	46.32

Triangles, $p = 1$

size	100	200	300	400	500
median	3060.50	10967	20032	28961	37754
$\bar{x}$	3069.54	10961.05	20040.81	28962.93	37742.30
σ^2	3917.75	11004.29	29359.69	53866.87	77745.85
σ	62.59	104.90	171.35	232.09	278.83

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2705.50	9914.50	18309.50	26620.50	34871
$\bar{x}$	2713.26	9918.16	18303.93	26617.89	34865.29
σ^2	2584.78	11621.51	33357.17	57706.44	56163.32
σ	50.84	107.80	182.64	240.22	236.99

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	721	2191	3649	4941	6105
$\bar{x}$	719.23	2190.30	3644.23	4949.91	6094.20
σ^2	1959.82	6990.43	15652.02	27188.97	23987.96
σ	44.27	83.61	125.11	164.89	154.88

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	1	1	1	1	1
$\bar{x}$	1.23	1.42	1.12	0.98	0.74
σ^2	1.29	1.60	1.00	0.97	0.60
σ	1.14	1.26	1.00	0.98	0.77

C.2 Generated graphs based on vertices

Colours, $p = 1$

size #colours	100	200	300	400	500
3	100	100	100	100	100

Colours, $p = \frac{1}{2}$

size #colours	100	200	300	400	500
8	2				
9	22				
10	68				
11	8	2			
12		24			
13		58	4		
14		15	26		
15		1	52	6	
16			15	37	3
17			2	40	22
18			1	16	38
19				1	31
20					6

Colours, $p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
6	64				
7	35	5			
8	1	85	4		
9		10	89	10	
10			7	86	37
11				4	62
12					1

$p = \frac{1}{n}$

size #colours	100	200	300	400	500
2	30	5	1		
3	70	95	99	100	100

Edges, $p = 1$

size	100	200	300	400	500
median	5454	21097	48548	87163	132836
$\bar{x}$	5150.84	19751.26	45072.02	85056.46	125343.86
σ^2	1574629.02	29523395.20	133919426.87	211508154.09	1118065558.81
σ	1254.84	5433.54	11572.36	14543.32	33437.49

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2656	8615	17186	27860	40167
$\bar{x}$	2667.56	8634.42	17195.40	27855.64	40506.52
σ^2	4190.75	41881.96	302442.30	1288118.09	3162116.74
σ	64.74	204.65	549.95	1134.95	1778.23

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	980	2796	5133	7931	11113
$\bar{x}$	983.24	2808	5139.70	7938.72	11125.54
σ^2	1066.33	5340.44	7485.20	11774.43	20204.51
σ	32.65	73.08	86.52	108.51	142.14

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	100	196	302	404	498
$\bar{x}$	100.20	196.72	301.60	403.50	496.78
σ^2	215.47	348.97	524.69	994.13	1015.02
σ	14.68	18.68	22.91	31.53	31.86

Triangles, $p = 1$

size	100	200	300	400	500
median	16858	117999	439824	1049156.50	1621016
$\bar{x}$	18301	131261.85	435298.63	1005264.53	1613333.18
σ^2	111311138.71	7877313655.38	70486127878.66	261904085453.16	935684361245.14
σ	10550.41	88754.23	265492.24	511765.65	967307.79

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2495	10571	22506	35635	49936.50
$\bar{x}$	2525.90	10569.78	22655.98	36086.46	51289.45
σ^2	25765.48	364530.21	2713419.07	12933355.20	32936532.78
σ	160.52	603.76	1647.25	3596.30	5739.04

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	150	424	656	871	1059
$\bar{x}$	151.22	424.99	660.89	871.43	1057.86
σ^2	316.05	1271.93	1912.77	2241.28	2894.02
σ	17.78	35.66	43.74	47.34	53.80

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	0	0	0	0	0
$\bar{x}$	0.19	0.16	0.16	0.13	0.08
σ^2	0.18	0.14	0.16	0.11	0.07
σ	0.42	0.37	0.39	0.34	0.27

D Octahedron free graphs

D.1 Generated graphs based on edges

$p = 1$

size #colours	100	200	300	400	500
11	11				
12	61				
13	27				
14	1				
15		1			
16		50			
17		47			
18		2	3		
19			24		
20			63		
21			10		
22				40	
23				50	
24				10	12
25					67
26					19
27					2

$p = \frac{1}{2}$

size #colours	100	200	300	400	500
11	8				
12	71				
13	21				
14					
15		1			
16		60			
17		38			
18		1	2		
19			42		
20			45		
21			11	5	
22				50	
23				42	1
24				3	44
25					50
26					5

$p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
8	10				
9	69				
10	21				
11		30			
12		66			
13		4	37		
14			63	10	
15				74	2
16				16	62
17					35
18					1

$p = \frac{1}{n}$

size #colours	100	200	300	400	500
3	89	72	67	57	64
4	11	28	33	43	36

Edges, $p = 1$

size	100	200	300	400	500
median	2852	8900	17268	27636	39748
$\bar{x}$	2852.16	8900.14	17271.06	27633.92	39747.56
σ^2	203.37	486.16	1228.08	2356.60	2658.55
σ	14.26	22.049	1228.08	48.54	51.56

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2788	8722	19234	27133	39066
$\bar{x}$	2786.80	8721.54	19237.52	27136.64	39066.90
σ^2	263.27	638.29	17684.05	2466.98	3517.85
σ	16.23	25.26	132.98	49.67	59.31

Edges,$p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	1832	5354	9998	15442	21735
$\bar{x}$	1832.54	5360.74	9989.06	15465.50	21717.26
σ^2	2055.83	7392.21	14490.95	34632.11	35671.89
σ	45.34	85.98	120.38	186.10	188.87

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	198	396	605	794	990
$\bar{x}$	199.54	396.20	604.28	798.04	988.72
σ^2	352.15	959.72	862.22	1425.21	2140.00
σ	18.77	30.98	29.36	37.75	46.26

Triangles,$p = 1$

size	100	200	300	400	500
median	3006.50	10866.50	20056.50	29258.50	38325.50
$\bar{x}$	3006.61	10865.01	20057.85	29268.78	38358.50
σ^2	894.54	6139.67	18813.28	33926.74	52729.04
σ	29.91	6139.67	137.16	184.19	229.63

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2863	10380.50	16960	28138.50	36941
$\bar{x}$	2860.76	10384.58	16956.86	28142.86	36937.75
σ^2	871.58	5317.98	1669.60	50720.34	47842.84
σ	29.52	72.92	40.86	225.21	218.73

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	995	2992.50	4894	6479.50	7896.50
$\bar{x}$	992.40	3000.15	4916.26	6515.75	7933.17
σ^2	5358.63	21895.79	38400.23	59870.61	91233.19
σ	73.20	147.97	195.96	244.68	302.05

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	1	1	1	1	0
$\bar{x}$	1.13	1.34	1.15	0.96	0.70
σ^2	1.08	1.18	1.22	0.93	0.76
σ	1.08	1.08	1.10	0.96	0.87

D.2 Generated graphs based on vertices

$p = 1$

size #colours	100	200	300	400	500
5	5			1	
6	28	12	6	2	2
7	44	38	20	22	16
8	16	36	45	25	27
9	7	13	23	34	31
10		1	6	14	22
11				1	1
12				1	1

$p = \frac{1}{2}$

size #colours	100	200	300	400	500
9	6				
10	64				
11	30				
12		15			
13		62			
14		23	8		
15			68		
16			24	19	
17				70	10
18				11	58
19					30
20					2

$p = \frac{1}{\sqrt{n}}$

size #colours	100	200	300	400	500
5	2				
6	59				
7	36	5			
8	3	89	6		
9		6	88	13	
10			6	86	46
11				1	51
12					3

$p = \frac{1}{n}$

size #colours	100	200	300	400	500
2	24	6	1		
3	76	94	99	100	100

Edges, $p = 1$

size	100	200	300	400	500
median	2418	8785	18891	32664	45218
$\bar{x}$	2376	8598.02	18172.98	31287	46589.22
σ^2	413423.92	7496039.31	33443421.53	150561950.10	383501481.12
σ	642.98	2737.89	5783.03	12270.37	19583.19

Edges, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2457	7505	14370	22935	32645
$\bar{x}$	2456.32	7495.06	14395.72	22867.02	32594.96
σ^2	2580.58	33674.95	123367.92	314222.34	774531.00
σ	50.80	183.51	351.24	560.56	880.07

Edges, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	994	2810	5174	7964	11163
$\bar{x}$	991.74	2813.36	5176.60	7969.48	11162.16
σ^2	2244.82	4700.76	9809.98	10697.67	20248.62
σ	47.38	68.56	99.05	103.43	142.30

Edges, $p = \frac{1}{n}$

size	100	200	300	400	500
median	96	195	298	399	498
$\bar{x}$	97.68	198.26	295.16	400.44	499.46
σ^2	167.01	408.54	451.09	929.42	932.59
σ	12.92	20.21	21.24	30.49	30.54

Triangles, $p = 1$

size	100	200	300	400	500
median	2060.50	7541	15591	21876	31744
$\bar{x}$	2040.85	7532.73	15187.80	23515.72	32646.76
σ^2	393648.71	6320378.99	24187203.05	83921911.05	149156150.67
σ	627.41	2514.04	4918.05	9160.89	12212.95

Triangles, $p = \frac{1}{2}$

size	100	200	300	400	500
median	2257.50	7944	15412	23837	32962.50
$\bar{x}$	2261.34	7934.07	15444.51	23803.83	32770.87
σ^2	6981.26	110012.09	384769.42	979251.72	2004087.10
σ	83.55	331.68	620.30	989.57	1415.66

Triangles, $p = \frac{1}{\sqrt{n}}$

size	100	200	300	400	500
median	157.50	447.50	690.50	901	1084
$\bar{x}$	160.22	444.99	696.78	900.83	1084.03
σ^2	739.61	1650.07	2459.77	2545.23	3309.91
σ	27.20	40.62	49.60	50.45	57.53

Triangles, $p = \frac{1}{n}$

size	100	200	300	400	500
median	0	0	0	0	0
$\bar{x}$	0.14	0.23	0.09	0.12	0.11
σ^2	0.16	0.30	0.08	0.11	0.10
σ	0.40	0.55	0.29	0.33	0.31

E Triangle free graphs

E.1 Triangle free graphs generated according to edges

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.121 (1.011, 1.231) b = 0.4315 (0.4146, 0.4484) Goodness of fit: SSE: 0.01541 R-square: 0.9996 Adjusted R-square: 0.9995 RMSE: 0.07166	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.117 (0.7872, 1.446) b = 0.4306 (0.3798, 0.4813) Goodness of fit: SSE: 0.1364 R-square: 0.9967 Adjusted R-square: 0.9956 RMSE: 0.2132

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.338 (1.27, 1.406) b = 1.574 (1.565, 1.582) Goodness of fit: SSE: 1135 R-square: 1 Adjusted R-square: 1 RMSE: 19.45	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.337 (1.272, 1.402) b = 1.574 (1.566, 1.582) Goodness of fit: SSE: 1046 R-square: 1 Adjusted R-square: 1 RMSE: 18.67

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.062 (0.9338, 1.191) b = 0.4367 (0.4159, 0.4575) Goodness of fit: SSE: 0.02211 R-square: 0.9995 Adjusted R-square: 0.9993 RMSE: 0.08584	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.117 (0.7872, 1.446) b = 0.4306 (0.3798, 0.4813) Goodness of fit: SSE: 0.1364 R-square: 0.9967 Adjusted R-square: 0.9956 RMSE: 0.2132

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.191 (1.124, 1.259) b = 1.585 (1.576, 1.594) Goodness of fit: SSE: 1261 R-square: 1 Adjusted R-square: 1 RMSE: 20.5	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.185 (1.123, 1.247) b = 1.586 (1.577, 1.595) Goodness of fit: SSE: 1077 R-square: 1 Adjusted R-square: 1 RMSE: 18.95

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.364 (1.191, 1.538) b = 0.3422 (0.3202, 0.3642) Goodness of fit: SSE: 0.01523 R-square: 0.9989 Adjusted R-square: 0.9986 RMSE: 0.07124	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.259 (0.4148, 2.103) b = 0.3615 (0.2456, 0.4774) Goodness of fit: SSE: 0.4404 R-square: 0.9744 Adjusted R-square: 0.9659 RMSE: 0.3831

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.063 (1.021, 1.105) b = 1.513 (1.506, 1.519) Goodness of fit: SSE: 224.3 R-square: 1 Adjusted R-square: 1 RMSE: 8.646	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.058 (1.016, 1.099) b = 1.513 (1.507, 1.52) Goodness of fit: SSE: 219.8 R-square: 1 Adjusted R-square: 1 RMSE: 8.56

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.174 (1.53, 2.818) b = 0.07347 (0.0209, 0.126) Goodness of fit: SSE: 0.01346 R-square: 0.8702 Adjusted R-square: 0.827 RMSE: 0.06699	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.847 (1.636, 2.057) b = 1.012 (0.993, 1.031) Goodness of fit: SSE: 24.32 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 2.847	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.759 (1.368, 2.149) b = 1.021 (0.9834, 1.058) Goodness of fit: SSE: 91.59 R-square: 0.9998 Adjusted R-square: 0.9997 RMSE: 5.525

E.2 Triangle free graphs generated according to vertices

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2 (2, 2) b = 1.309e-10 (-1.001e-09, 1.263e-09) Goodness of fit: SSE: 2.453e-18 R-square: NaN Adjusted R-square: NaN RMSE: 9.043e-10	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2 (2, 2) b = 7.307e-11 (-5.035e-10, 6.497e-10) Goodness of fit: SSE: 6.363e-19 R-square: NaN Adjusted R-square: NaN RMSE: 4.606e-10

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.4648 (0.1015, 0.8282) b = 1.941 (1.813, 2.07) Goodness of fit: SSE: 1.975e+06 R-square: 0.9995 Adjusted R-square: 0.9993 RMSE: 811.4	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.504 (-0.5821, 3.591) b = 1.766 (1.538, 1.994) Goodness of fit: SSE: 9.048e+06 R-square: 0.998 Adjusted R-square: 0.9973 RMSE: 1737

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.566 (1.326, 1.807) b = 0.283 (0.2563, 0.3097) Goodness of fit: SSE: 0.01599 R-square: 0.9977 Adjusted R-square: 0.9969 RMSE: 0.073	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.698 (0.822, 2.574) b = 0.272 (0.1822, 0.3618) Goodness of fit: SSE: 0.1896 R-square: 0.9721 Adjusted R-square: 0.9628 RMSE: 0.2514

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.4029 (0.2826, 0.5231) b = 1.82 (1.771, 1.869) Goodness of fit: SSE: 5.534e+04 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 135.8	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.4854 (0.2208, 0.75) b = 1.786 (1.697, 1.876) Goodness of fit: SSE: 1.831e+05 R-square: 0.9997 Adjusted R-square: 0.9996 RMSE: 247

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.335 (1.148, 1.523) b = 0.3086 (0.2842, 0.333) Goodness of fit: SSE: 0.01266 R-square: 0.9984 Adjusted R-square: 0.9979 RMSE: 0.06496	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.175 (0.1158, 2.235) b = 0.3279 (0.1716, 0.4841) Goodness of fit: SSE: 0.4913 R-square: 0.9466 Adjusted R-square: 0.9288 RMSE: 0.4047

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.7863 (0.7692, 0.8034) b = 1.502 (1.498, 1.505) Goodness of fit: SSE: 32.37 R-square: 1 Adjusted R-square: 1 RMSE: 3.285	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.7887 (0.7742, 0.8031) b = 1.501 (1.498, 1.504) Goodness of fit: SSE: 22.97 R-square: 1 Adjusted R-square: 1 RMSE: 2.767

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.157 (1.728, 2.585) b = 0.05549 (0.02016, 0.09082) Goodness of fit: SSE: 0.004961 R-square: 0.8955 Adjusted R-square: 0.8607 RMSE: 0.04066	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9997 (0.9169, 1.082) b = 0.9994 (0.9855, 1.013) Goodness of fit: SSE: 3.271 R-square: 1 Adjusted R-square: 1 RMSE: 1.044	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9728 (0.9057, 1.04) b = 1.003 (0.9918, 1.015) Goodness of fit: SSE: 2.251 R-square: 1 Adjusted R-square: 1 RMSE: 0.8662

F Four Cycle free graphs

F.1 Four Cycle free graphs generated according to edges

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.081 (1.979, 2.183) b = 0.2266 (0.2181, 0.2351) Goodness of fit: SSE: 0.001602 R-square: 0.9996 Adjusted R-square: 0.9995 RMSE: 0.02311	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.281 (1.256, 3.305) b = 0.2127 (0.1341, 0.2914) Goodness of fit: SSE: 0.1415 R-square: 0.9646 Adjusted R-square: 0.9528 RMSE: 0.2172

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.327 (1.306, 1.349) b = 1.378 (1.376, 1.381) Goodness of fit: SSE: 13.26 R-square: 1 Adjusted R-square: 1 RMSE: 2.102	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.327 (1.291, 1.363) b = 1.378 (1.374, 1.383) Goodness of fit: SSE: 36.4 R-square: 1 Adjusted R-square: 1 RMSE: 3.483

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.428 (-1.433, 6.289) b = 0.7359 (0.4671, 1.005) Goodness of fit: SSE: 442.5 R-square: 0.9755 Adjusted R-square: 0.9673 RMSE: 12.15	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.396 (-1.325, 6.117) b = 0.738 (0.4756, 1.001) Goodness of fit: SSE: 420.4 R-square: 0.9767 Adjusted R-square: 0.9689 RMSE: 11.84

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.092 (1.754, 2.43) b = 0.2237 (0.1954, 0.2519) Goodness of fit: SSE: 0.01717 R-square: 0.9957 Adjusted R-square: 0.9943 RMSE: 0.07565	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.616 (1.107, 4.124) b = 0.1811 (0.07984, 0.2824) Goodness of fit: SSE: 0.222 R-square: 0.9207 Adjusted R-square: 0.8943 RMSE: 0.272

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.294 (1.272, 1.316) b = 1.38 (1.377, 1.383) Goodness of fit: SSE: 13.96 R-square: 1 Adjusted R-square: 1 RMSE: 2.157	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.292 (1.274, 1.31) b = 1.381 (1.378, 1.383) Goodness of fit: SSE: 9.182 R-square: 1 Adjusted R-square: 1 RMSE: 1.749

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.219 (-1.255, 5.692) b = 0.7444 (0.4799, 1.009) Goodness of fit: SSE: 391.5 R-square: 0.9768 Adjusted R-square: 0.9691 RMSE: 11.42	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.203 (-1.128, 5.533) b = 0.7453 (0.4898, 1.001) Goodness of fit: SSE: 363.3 R-square: 0.9784 Adjusted R-square: 0.9712 RMSE: 11

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.844 (1.418, 2.271) b = 0.2276 (0.1872, 0.268) Goodness of fit: SSE: 0.02852 R-square: 0.9916 Adjusted R-square: 0.9888 RMSE: 0.09751	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.371 (0.4993, 2.243) b = 0.2803 (0.1696, 0.3909) Goodness of fit: SSE: 0.2045 R-square: 0.9607 Adjusted R-square: 0.9476 RMSE: 0.2611

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.242 (1.223, 1.261) b = 1.36 (1.357, 1.362) Goodness of fit: SSE: 8.589 R-square: 1 Adjusted R-square: 1 RMSE: 1.692	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.244 (1.216, 1.273) b = 1.36 (1.356, 1.363) Goodness of fit: SSE: 18.56 R-square: 1 Adjusted R-square: 1 RMSE: 2.487

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.766 (-1.026, 4.557) b = 0.7 (0.4323, 0.9677) Goodness of fit: SSE: 159 R-square: 0.9726 Adjusted R-square: 0.9635 RMSE: 7.281	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.833 (-1.233, 4.899) b = 0.6942 (0.4109, 0.9775) Goodness of fit: SSE: 180.6 R-square: 0.9689 Adjusted R-square: 0.9585 RMSE: 7.76

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.174 (1.741, 2.608) b = 0.07255 (0.03717, 0.1079) Goodness of fit: SSE: 0.006041 R-square: 0.9357 Adjusted R-square: 0.9142 RMSE: 0.04487	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.863 (1.689, 2.037) b = 1.011 (0.9953, 1.026) Goodness of fit: SSE: 16.31 R-square: 1 Adjusted R-square: 0.9999 RMSE: 2.332	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.868 (1.624, 2.113) b = 1.01 (0.9883, 1.032) Goodness of fit: SSE: 32.1 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 3.271

triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 5.048 (1.988, 8.109) b = -0.2875 (-0.3998, -0.1751) Goodness of fit: SSE: 0.007449 R-square: 0.9555 Adjusted R-square: 0.9407 RMSE: 0.04983	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1 (1, 1) b = 0 (0, 0) Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

F.2 Four Cycle free graphs generated according to vertices

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.678 (2.185, 3.171) b = 0.0792 (0.04656, 0.1118) Goodness of fit: SSE: 0.008359 R-square: 0.9535 Adjusted R-square: 0.938 RMSE: 0.05279	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 4 (4, 4) b = 0 (0, 0) Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.227 (1.888, 4.566) b = 1.021 (0.9515, 1.09) Goodness of fit: SSE: 1077 R-square: 0.9992 Adjusted R-square: 0.999 RMSE: 18.95	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.04 (2.57, 3.51) b = 1.022 (0.9958, 1.047) Goodness of fit: SSE: 133.7 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 6.676

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.124 (-1.83, 8.078) b = 0.6363 (0.3669, 0.9057) Goodness of fit: SSE: 258.2 R-square: 0.9652 Adjusted R-square: 0.9536 RMSE: 9.277	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.248 (-21.68, 36.17) b = 0.4889 (-0.1948, 1.173) Goodness of fit: SSE: 1915 R-square: 0.7164 Adjusted R-square: 0.6218 RMSE: 25.27

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.251 (1.96, 2.542) b = 0.1626 (0.1398, 0.1853) Goodness of fit: SSE: 0.006838 R-square: 0.9946 Adjusted R-square: 0.9928 RMSE: 0.04774	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.576 (0.5116, 4.64) b = 0.139 (-0.002205, 0.2803) Goodness of fit: SSE: 0.2704 R-square: 0.7747 Adjusted R-square: 0.6996 RMSE: 0.3002

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.644 (1.05, 4.237) b = 1.146 (1.046, 1.246) Goodness of fit: SSE: 5826 R-square: 0.9988 Adjusted R-square: 0.9984 RMSE: 44.07	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.635 (1.525, 3.746) b = 1.144 (1.074, 1.214) Goodness of fit: SSE: 2777 R-square: 0.9994 Adjusted R-square: 0.9992 RMSE: 30.42

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.4 (-1.828, 6.629) b = 0.7066 (0.4084, 1.005) Goodness of fit: SSE: 390.9 R-square: 0.967 Adjusted R-square: 0.956 RMSE: 11.41	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.654 (-2.599, 7.906) b = 0.6886 (0.3533, 1.024) Goodness of fit: SSE: 500.1 R-square: 0.9564 Adjusted R-square: 0.9418 RMSE: 12.91

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.799 (1.421, 2.176) b = 0.2162 (0.1795, 0.253) Goodness of fit: SSE: 0.01991 R-square: 0.9922 Adjusted R-square: 0.9896 RMSE: 0.08147	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.899 (0.6078, 3.191) b = 0.2114 (0.09236, 0.3304) Goodness of fit: SSE: 0.2218 R-square: 0.9208 Adjusted R-square: 0.8944 RMSE: 0.2719

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.008 (0.987, 1.028) b = 1.374 (1.37, 1.377) Goodness of fit: SSE: 11.6 R-square: 1 Adjusted R-square: 1 RMSE: 1.966	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.001 (0.9677, 1.035) b = 1.375 (1.369, 1.38) Goodness of fit: SSE: 30.94 R-square: 1 Adjusted R-square: 1 RMSE: 3.212

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.7594 (-0.2369, 1.756) b = 0.8148 (0.5938, 1.036) Goodness of fit: SSE: 67.36 R-square: 0.9868 Adjusted R-square: 0.9824 RMSE: 4.739	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.7727 (-0.159, 1.704) b = 0.8114 (0.6083, 1.015) Goodness of fit: SSE: 56.86 R-square: 0.9887 Adjusted R-square: 0.9849 RMSE: 4.354

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.06 (1.502, 2.618) b = 0.06249 (0.01435, 0.1106) Goodness of fit: SSE: 0.009041 R-square: 0.8548 Adjusted R-square: 0.8065 RMSE: 0.0549	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.06 (1.502, 2.618) b = 0.06249 (0.01435, 0.1106) Goodness of fit: SSE: 0.009041 R-square: 0.8548 Adjusted R-square: 0.8065 RMSE: 0.0549

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.8894 (0.7487, 1.03) b = 1.02 (0.9932, 1.046) Goodness of fit: SSE: 11.75 R-square: 0.9999 Adjusted R-square: 0.9998 RMSE: 1.979	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9325 (0.7125, 1.153) b = 1.012 (0.973, 1.052) Goodness of fit: SSE: 26.61 R-square: 0.9997 Adjusted R-square: 0.9997 RMSE: 2.978

triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.2061 (-0.1963, 0.6085) b = -0.05962 (-0.4112, 0.292) Goodness of fit: SSE: 0.001346 R-square: 0.09069 Adjusted R-square: -0.2124 RMSE: 0.02118	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = -0 (-0, 0) b = 0.7667 Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

G Tetrahedron graphs

G.1 Tetrahedron free graphs generated according to edges

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.453 (1.325, 1.581) b = 0.4584 (0.4433, 0.4736) Goodness of fit: SSE: 0.02755 R-square: 0.9997 Adjusted R-square: 0.9997 RMSE: 0.09584	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.458 (1.2, 1.717) b = 0.4586 (0.4281, 0.489) Goodness of fit: SSE: 0.112 R-square: 0.999 Adjusted R-square: 0.9986 RMSE: 0.1933

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.722 (1.656, 1.788) b = 1.622 (1.615, 1.628) Goodness of fit: SSE: 1833 R-square: 1 Adjusted R-square: 1 RMSE: 24.72	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.723 (1.652, 1.794) b = 1.622 (1.615, 1.628) Goodness of fit: SSE: 2078 R-square: 1 Adjusted R-square: 1 RMSE: 26.32

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 8.843 (-3.401, 21.09) b = 1.348 (1.118, 1.577) Goodness of fit: SSE: 3.062e+06 R-square: 0.996 Adjusted R-square: 0.9947 RMSE: 1010	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 8.806 (-3.374, 20.99) b = 1.348 (1.119, 1.577) Goodness of fit: SSE: 3.053e+06 R-square: 0.996 Adjusted R-square: 0.9947 RMSE: 1009

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.423 (1.351, 1.494) b = 0.4585 (0.4499, 0.4671) Goodness of fit: SSE: 0.008591 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 0.05351	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.377 (0.855, 1.9) b = 0.4639 (0.3988, 0.529) Goodness of fit: SSE: 0.483 R-square: 0.9953 Adjusted R-square: 0.9937 RMSE: 0.4012

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.584 (1.514, 1.654) b = 1.63 (1.623, 1.637) Goodness of fit: SSE: 2231 R-square: 1 Adjusted R-square: 1 RMSE: 27.27	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.587 (1.51, 1.665) b = 1.63 (1.622, 1.638) Goodness of fit: SSE: 2732 R-square: 1 Adjusted R-square: 1 RMSE: 30.18

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.185 (-2.789, 17.16) b = 1.368 (1.138, 1.598) Goodness of fit: SSE: 2.544e+06 R-square: 0.9961 Adjusted R-square: 0.9948 RMSE: 920.9	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.168 (-2.831, 17.17) b = 1.369 (1.138, 1.6) Goodness of fit: SSE: 2.569e+06 R-square: 0.9961 Adjusted R-square: 0.9948 RMSE: 925.3

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.642 (1.586, 1.698) b = 0.3635 (0.3576, 0.3693) Goodness of fit: SSE: 0.001968 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 0.02561	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.667 (0.8249, 2.509) b = 0.36 (0.2726, 0.4473) Goodness of fit: SSE: 0.4313 R-square: 0.9852 Adjusted R-square: 0.9803 RMSE: 0.3792

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.444 (1.413, 1.475) b = 1.539 (1.535, 1.542) Goodness of fit: SSE: 157.6 R-square: 1 Adjusted R-square: 1 RMSE: 7.249	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.45 (1.407, 1.494) b = 1.538 (1.533, 1.543) Goodness of fit: SSE: 310.8 R-square: 1 Adjusted R-square: 1 RMSE: 10.18

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 5.104 (-2.664, 12.87) b = 1.144 (0.8903, 1.397) Goodness of fit: SSE: 1.354e+05 R-square: 0.9926 Adjusted R-square: 0.9901 RMSE: 212.4	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 5.081 (-2.527, 12.69) b = 1.145 (0.8953, 1.394) Goodness of fit: SSE: 1.31e+05 R-square: 0.9928 Adjusted R-square: 0.9905 RMSE: 209

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.486 (2.077, 2.894) b = 0.0511 (0.0219, 0.0803) Goodness of fit: SSE: 0.004299 R-square: 0.9128 Adjusted R-square: 0.8838 RMSE: 0.03786	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.89 (1.733, 2.047) b = 1.009 (0.9946, 1.022) Goodness of fit: SSE: 13.03 R-square: 1 Adjusted R-square: 1 RMSE: 2.084	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.983 (1.809, 2.158) b = 1.001 (0.9862, 1.016) Goodness of fit: SSE: 14.8 R-square: 1 Adjusted R-square: 1 RMSE: 2.221

triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 4.163 (-5.829, 14.16) b = -0.2411 (-0.6832, 0.2009) Goodness of fit: SSE: 0.1282 R-square: 0.5137 Adjusted R-square: 0.3516 RMSE: 0.2067	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1 (1, 1) b = 0 (0, 0) Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

G.2 Tetrahedron free graphs generated according to vertices

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.6403 (-0.2847, 1.565) b = 1.962 (1.725, 2.199) Goodness of fit: SSE: 1.623e+07 R-square: 0.9983 Adjusted R-square: 0.9978 RMSE: 2326	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.6419 (0.29, 0.9939) b = 1.97 (1.88, 2.06) Goodness of fit: SSE: 2.568e+06 R-square: 0.9998 Adjusted R-square: 0.9997 RMSE: 925.2

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.3463 (-0.6897, 1.382) b = 2.473 (1.985, 2.961) Goodness of fit: SSE: 6.864e+09 R-square: 0.9961 Adjusted R-square: 0.9948 RMSE: 4.783e+04	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.4259 (-1.353, 2.205) b = 2.442 (1.76, 3.123) Goodness of fit: SSE: 1.41e+10 R-square: 0.9923 Adjusted R-square: 0.9897 RMSE: 6.855e+04

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.713 (1.589, 1.836) b = 0.3798 (0.3674, 0.3923) Goodness of fit: SSE: 0.01139 R-square: 0.9997 Adjusted R-square: 0.9996 RMSE: 0.06163	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.844 (1.448, 2.241) b = 0.3682 (0.331, 0.4053) Goodness of fit: SSE: 0.1042 R-square: 0.9975 Adjusted R-square: 0.9966 RMSE: 0.1864

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.165 (1.089, 1.241) b = 1.683 (1.672, 1.693) Goodness of fit: SSE: 4699 R-square: 1 Adjusted R-square: 1 RMSE: 39.58	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.251 (1.009, 1.494) b = 1.67 (1.638, 1.702) Goodness of fit: SSE: 4.196e+04 R-square: 1 Adjusted R-square: 0.9999 RMSE: 118.3

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.565 (-0.1269, 3.257) b = 1.674 (1.496, 1.852) Goodness of fit: SSE: 2.148e+06 R-square: 0.9986 Adjusted R-square: 0.9981 RMSE: 846.2	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.863 (-0.4422, 4.167) b = 1.642 (1.439, 1.846) Goodness of fit: SSE: 2.794e+06 R-square: 0.9981 Adjusted R-square: 0.9974 RMSE: 965

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.497 (1.349, 1.645) b = 0.3157 (0.2986, 0.3329) Goodness of fit: SSE: 0.008489 R-square: 0.9992 Adjusted R-square: 0.999 RMSE: 0.0532	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.126 (0.8419, 1.409) b = 0.366 (0.3224, 0.4095) Goodness of fit: SSE: 0.05209 R-square: 0.9965 Adjusted R-square: 0.9953 RMSE: 0.1318

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9506 (0.898, 1.003) b = 1.507 (1.498, 1.516) Goodness of fit: SSE: 325.6 R-square: 1 Adjusted R-square: 1 RMSE: 10.42	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9395 (0.9061, 0.973) b = 1.509 (1.503, 1.515) Goodness of fit: SSE: 134.1 R-square: 1 Adjusted R-square: 1 RMSE: 6.685

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.616 (-0.6537, 3.886) b = 1.047 (0.8123, 1.281) Goodness of fit: SSE: 4087 R-square: 0.992 Adjusted R-square: 0.9894 RMSE: 36.91	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.552 (-0.5786, 3.683) b = 1.053 (0.8242, 1.283) Goodness of fit: SSE: 3863 R-square: 0.9925 Adjusted R-square: 0.99 RMSE: 35.89

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.061 (1.418, 2.704) b = 0.06302 (0.007588, 0.1185) Goodness of fit: SSE: 0.01207 R-square: 0.819 Adjusted R-square: 0.7587 RMSE: 0.06342	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.007 (0.7333, 1.281) b = 0.9987 (0.9532, 1.044) Goodness of fit: SSE: 35.63 R-square: 0.9996 Adjusted R-square: 0.9995 RMSE: 3.446	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9838 (0.7026, 1.265) b = 1.003 (0.9551, 1.051) Goodness of fit: SSE: 39.23 R-square: 0.9996 Adjusted R-square: 0.9995 RMSE: 3.616

triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.017 (-1.15, 3.184) b = -0.3549 (-0.7534, 0.04367) Goodness of fit: SSE: 0.001857 R-square: 0.7317 Adjusted R-square: 0.6423 RMSE: 0.02488	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = -0 (-0, 0) b = 0.1008 Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

H Octahedron graphs

H.1 Octahedron free graphs generated according to edges

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.503 (1.385, 1.621) b = 0.4529 (0.4394, 0.4663) Goodness of fit: SSE: 0.02195 R-square: 0.9998 Adjusted R-square: 0.9997 RMSE: 0.08553	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.367 (0.9387, 1.795) b = 0.469 (0.4152, 0.5227) Goodness of fit: SSE: 0.342 R-square: 0.9969 Adjusted R-square: 0.9959 RMSE: 0.3376

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.558 (1.517, 1.6) b = 1.633 (1.628, 1.637) Goodness of fit: SSE: 807.8 R-square: 1 Adjusted R-square: 1 RMSE: 16.41	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.557 (1.515, 1.6) b = 1.633 (1.628, 1.637) Goodness of fit: SSE: 854.7 R-square: 1 Adjusted R-square: 1 RMSE: 16.88

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.721 (-2.72, 18.16) b = 1.372 (1.148, 1.596) Goodness of fit: SSE: 2.905e+06 R-square: 0.9963 Adjusted R-square: 0.9951 RMSE: 984	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.767 (-2.781, 18.32) b = 1.371 (1.146, 1.596) Goodness of fit: SSE: 2.93e+06 R-square: 0.9963 Adjusted R-square: 0.9951 RMSE: 988.2

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.584 (1.477, 1.692) b = 0.4416 (0.43, 0.4533) Goodness of fit: SSE: 0.01626 R-square: 0.9998 Adjusted R-square: 0.9998 RMSE: 0.07362	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.435 (0.9227, 1.947) b = 0.4586 (0.3973, 0.5199) Goodness of fit: SSE: 0.4399 R-square: 0.9958 Adjusted R-square: 0.9944 RMSE: 0.3829

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.504 (1.461, 1.547) b = 1.636 (1.631, 1.64) Goodness of fit: SSE: 909.3 R-square: 1 Adjusted R-square: 1 RMSE: 17.41	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.506 (1.463, 1.548) b = 1.636 (1.631, 1.64) Goodness of fit: SSE: 885.3 R-square: 1 Adjusted R-square: 1 RMSE: 17.18

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.118 (-2.49, 16.73) b = 1.379 (1.155, 1.602) Goodness of fit: SSE: 2.655e+06 R-square: 0.9964 Adjusted R-square: 0.9952 RMSE: 940.8	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 7.103 (-2.456, 16.66) b = 1.379 (1.157, 1.602) Goodness of fit: SSE: 2.638e+06 R-square: 0.9964 Adjusted R-square: 0.9952 RMSE: 937.8

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.72 (1.657, 1.783) b = 0.3624 (0.356, 0.3687) Goodness of fit: SSE: 0.002482 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 0.02876	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.884 (1.09, 2.677) b = 0.3467 (0.2738, 0.4196) Goodness of fit: SSE: 0.3345 R-square: 0.9891 Adjusted R-square: 0.9855 RMSE: 0.3339

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.657 (1.546, 1.768) b = 1.526 (1.515, 1.537) Goodness of fit: SSE: 1770 R-square: 1 Adjusted R-square: 1 RMSE: 24.29	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.641 (1.511, 1.77) b = 1.527 (1.514, 1.54) Goodness of fit: SSE: 2461 R-square: 1 Adjusted R-square: 1 RMSE: 28.64

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 8.928 (-5.871, 23.73) b = 1.097 (0.8202, 1.373) Goodness of fit: SSE: 2.96e+05 R-square: 0.9903 Adjusted R-square: 0.987 RMSE: 314.1	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 8.991 (-5.718, 23.7) b = 1.095 (0.8219, 1.367) Goodness of fit: SSE: 2.863e+05 R-square: 0.9905 Adjusted R-square: 0.9873 RMSE: 308.9

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.447 (1.916, 2.977) b = 0.0538 (0.01525, 0.09235) Goodness of fit: SSE: 0.007469 R-square: 0.871 Adjusted R-square: 0.8279 RMSE: 0.0499	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.89 (1.733, 2.047) b = 1.009 (0.9946, 1.022) Goodness of fit: SSE: 13.03 R-square: 1 Adjusted R-square: 1 RMSE: 2.084	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.983 (1.809, 2.158) b = 1.001 (0.9862, 1.016) Goodness of fit: SSE: 14.8 R-square: 1 Adjusted R-square: 1 RMSE: 2.221

triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.285 (-5.388, 11.96) b = -0.2051 (-0.6891, 0.2789) Goodness of fit: SSE: 0.1401 R-square: 0.3932 Adjusted R-square: 0.191 RMSE: 0.2161	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 6.786 (-38.93, 52.5) b = -0.3878 (-1.653, 0.8774) Goodness of fit: SSE: 0.586 R-square: 0.2675 Adjusted R-square: 0.02333 RMSE: 0.442

H.2 Octahedron free graphs generated according to vertices

colour, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.617 (3.31, 3.924) b = 0.1399 (0.1249, 0.1549) Goodness of fit: SSE: 0.00604 R-square: 0.9968 Adjusted R-square: 0.9957 RMSE: 0.04487	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3.309 (2.208, 4.411) b = 0.158 (0.09939, 0.2165) Goodness of fit: SSE: 0.09344 R-square: 0.9626 Adjusted R-square: 0.9502 RMSE: 0.1765

edges, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.5168 (0.3582, 0.6755) b = 1.836 (1.786, 1.887) Goodness of fit: SSE: 1.155e+05 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 196.2	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.032 (-0.5376, 2.602) b = 1.722 (1.472, 1.972) Goodness of fit: SSE: 3.152e+06 R-square: 0.9974 Adjusted R-square: 0.9966 RMSE: 1025

triangles, $p(n) = 1$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.931 (0.08344, 3.779) b = 1.568 (1.41, 1.726) Goodness of fit: SSE: 7.855e+05 R-square: 0.9987 Adjusted R-square: 0.9983 RMSE: 511.7	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.392 (-1.374, 6.158) b = 1.527 (1.267, 1.787) Goodness of fit: SSE: 2.08e+06 R-square: 0.9962 Adjusted R-square: 0.9949 RMSE: 832.6

colour, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.935 (1.819, 2.052) b = 0.3612 (0.3508, 0.3716) Goodness of fit: SSE: 0.008407 R-square: 0.9998 Adjusted R-square: 0.9997 RMSE: 0.05294	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.844 (1.448, 2.241) b = 0.3682 (0.331, 0.4053) Goodness of fit: SSE: 0.1042 R-square: 0.9975 Adjusted R-square: 0.9966 RMSE: 0.1864

edges, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.552 (1.442, 1.662) b = 1.602 (1.59, 1.613) Goodness of fit: SSE: 4060 R-square: 1 Adjusted R-square: 1 RMSE: 36.79	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.534 (1.361, 1.708) b = 1.604 (1.585, 1.622) Goodness of fit: SSE: 1.025e+04 R-square: 1 Adjusted R-square: 1 RMSE: 58.46

triangles, $p(n) = \frac{1}{2}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.469 (0.424, 4.514) b = 1.529 (1.392, 1.666) Goodness of fit: SSE: 6.253e+05 R-square: 0.9989 Adjusted R-square: 0.9986 RMSE: 456.5	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.34 (0.5554, 4.125) b = 1.538 (1.412, 1.664) Goodness of fit: SSE: 5.285e+05 R-square: 0.9991 Adjusted R-square: 0.9988 RMSE: 419.7

colour, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.542 (1.44, 1.643) b = 0.3099 (0.2985, 0.3213) Goodness of fit: SSE: 0.00376 R-square: 0.9996 Adjusted R-square: 0.9995 RMSE: 0.0354	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.126 (0.8419, 1.409) b = 0.366 (0.3224, 0.4095) Goodness of fit: SSE: 0.05209 R-square: 0.9965 Adjusted R-square: 0.9953 RMSE: 0.1318

edges, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9706 (0.9455, 0.9957) b = 1.504 (1.5, 1.509) Goodness of fit: SSE: 71.95 R-square: 1 Adjusted R-square: 1 RMSE: 4.897	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9733 (0.9277, 1.019) b = 1.504 (1.496, 1.512) Goodness of fit: SSE: 236.6 R-square: 1 Adjusted R-square: 1 RMSE: 8.881

triangles, $p(n) = \frac{1}{\sqrt{(n)}}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.96 (-1.059, 4.978) b = 1.02 (0.7628, 1.278) Goodness of fit: SSE: 5442 R-square: 0.9898 Adjusted R-square: 0.9864 RMSE: 42.59	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 1.911 (-1.025, 4.848) b = 1.024 (0.7675, 1.281) Goodness of fit: SSE: 5372 R-square: 0.99 Adjusted R-square: 0.9866 RMSE: 42.32

colour, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 2.206 (1.746, 2.667) b = 0.05143 (0.01431, 0.08854) Goodness of fit: SSE: 0.005491 R-square: 0.8695 Adjusted R-square: 0.826 RMSE: 0.04278	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 3 (3, 3) b = 0 (-9.57e-16, 9.57e-16) Goodness of fit: SSE: 3.944e-30 R-square: NaN Adjusted R-square: NaN RMSE: 1.147e-15

edges, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.9097 (0.7708, 1.048) b = 1.015 (0.9897, 1.041) Goodness of fit: SSE: 10.91 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 1.907	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.8845 (0.7735, 0.9954) b = 1.02 (0.9986, 1.04) Goodness of fit: SSE: 7.294 R-square: 0.9999 Adjusted R-square: 0.9999 RMSE: 1.559

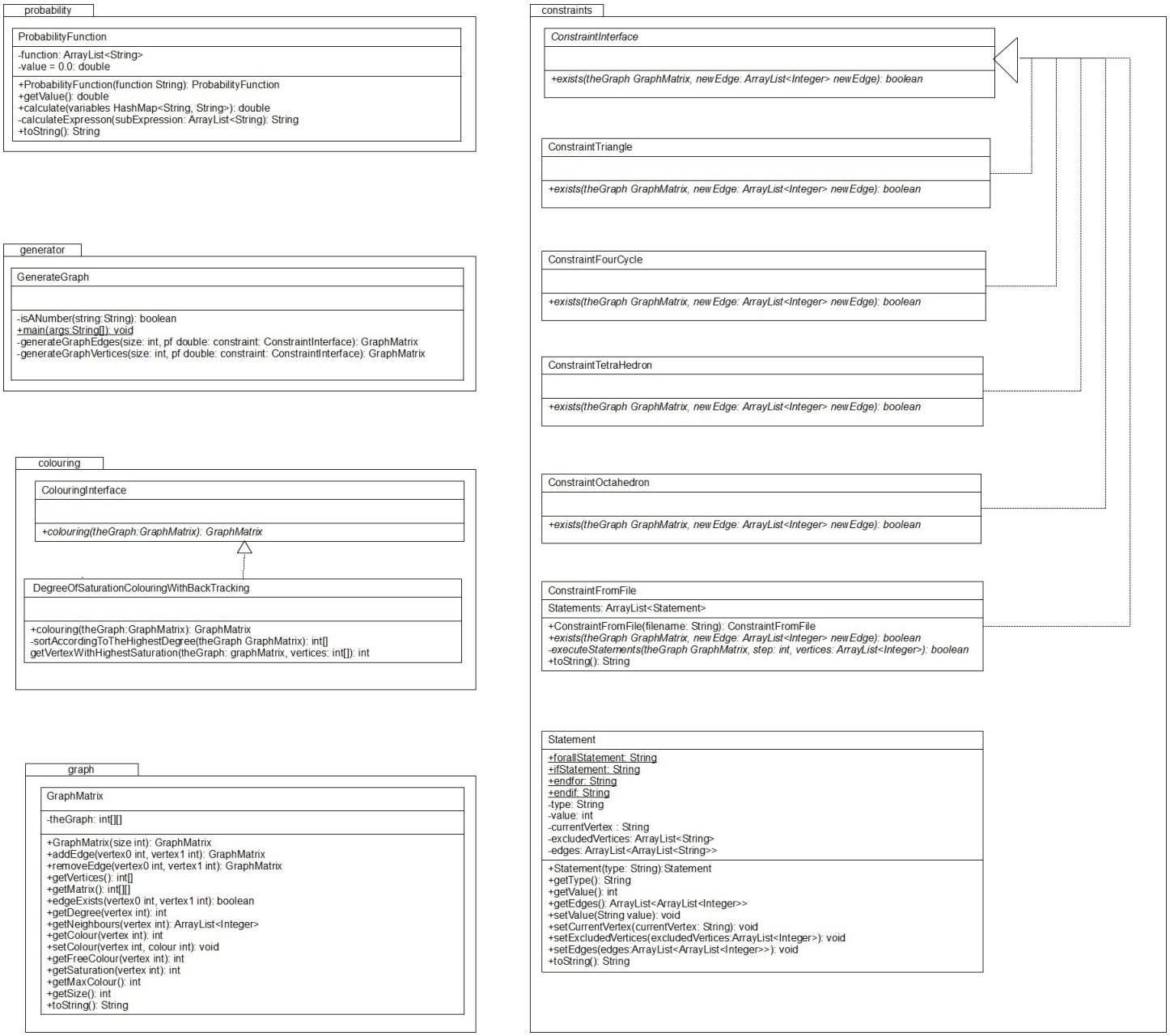
triangles, $p(n) = \frac{1}{n}$:

Mean	Median
$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = 0.4841 (-2.105, 3.073) b = -0.2269 (-1.21, 0.7563) Goodness of fit: SSE: 0.009967 R-square: 0.161 Adjusted R-square: -0.1187 RMSE: 0.05764	$f(n) = a * n^b$ Coefficients (with 95% confidence bounds): a = -0 (-0, 0) b = 0.7629 Goodness of fit: SSE: 0 R-square: NaN Adjusted R-square: NaN RMSE: 0

I Implementation

This appendix contains the source code to the program.

I.1 class overview



I.2 package graph

I.2.1 GraphMatrix.java

```
package graph;
import java.util.*;
import java.io.*;

/**
 * This class represents an undirected graph <tt>G = (V,E)</tt> and the
 * edges in the graph are stored in an adjacent matrix.
 */
public class GraphMatrix
{
    private int[][] theGraph = null;

    /**
     * Returns a graph <tt>G = (V, E)</tt> consisting of n vertices and an
     * empty set of edges, <tt>E</tt>.
     *
     * @param  size the number of vertices in the graph
     */
    public GraphMatrix(int size)
    {
      theGraph = new int[size + 1][size + 1];
    }

    /**
     * Inserts the edge <tt>e = {vertex0, vertex1}</tt> to the graph
     * <tt>G = (V, E)</tt>.
     *
     * @param  vertex0 - the first vertex in the edge e
     * @param  vertex1 - the second vertex in the edge e
     * @return a graph in which e has been inserted.
     */
    public GraphMatrix addEdge(int vertex0, int vertex1)
    {
      if (vertex0 != vertex1) {
          theGraph[vertex0][vertex1] = 1;
          theGraph[vertex1][vertex0] = 1;
      }
```

```
    return this;
  }

  /**
   * Removes the edge <tt>e = {vertex0, vertex1}</tt> from the graph
   * <tt>G = (V, E)</tt>.
   *
   * @param  vertex0 - the first vertex in the edge e
   * @param  vertex1 - the second vertex in the edge e
   * @return a graph in which e has been removed
   */
  public GraphMatrix removeEdge(int vertex0, int vertex1)
  {
    if (vertex0 != vertex1) {
        theGraph[vertex0][vertex1] = 0;
        theGraph[vertex1][vertex0] = 0;
    }

    return this;
  }

  /**
   * Returns an array of the vertices in the graph.
   *
   * @return  an array of the vertices in the graph
   */
  public int[] getVertices() {
    int[] vertices = new int[size() - 1];

    for(int i = 1; i < size(); i++) {
        vertices[i - 1] = i;
    }

    return vertices;
  }

   /**
   * Returns the adjacent matrix which contains the edges in the graph.
   *
   * @return  an array of the vertices in the graph
   */
  public int[][] getMatrix() {
```

```
    return theGraph;
  }

  /**
   * Returns true if the edge <tt>e = {vertex0, vertex1} </tt> eixsts in
   * the graph <tt>G = (V, E)</tt>.
   *
   * @param  vertex0 - the first vertex in the edge e
   * @param  vertex1 - the second vertex in the edge e
   * @return true if the edge e exists in the graph and false otherwise.
   */
  public boolean edgeExists(int vertex0, int vertex1)
  {
    if(vertex0 == vertex1) {
        return false;
    }
    else {
        return theGraph[vertex0][vertex1] != 0;
    }
  }

  /**
   * Returns the degree of the given vertex, that is how many edges the
   * vertex has.
   *
   * @param  vertex - the given vertex
   * @return returns the degree of vertex.
   */
  public int getDegree(int vertex) {
    return getNeighbours(vertex).size();
  }

  /**
   * Returns the neighbours of the given vertex, that an arraylist of
   * the vertices that are adjancent to the given vertex.
   *
   * @param  vertex - the given vertex
   * @return returns an arraylist constaining the neighbours of the
   * given vertex.
   */
  public ArrayList<Integer> getNeighbours(int vertex)
  {
```

```
  ArrayList<Integer> neighbours = new ArrayList<Integer>();

  for(int i = 1; i < theGraph.length; i++) {
      if(theGraph[vertex][i] != 0) {
        neighbours.add(i);
      }
  }

  return neighbours;
}

/**
 * Returns the colour of the given vertex.
 *
 * @param  vertex - the given vertex
 * @return returns the colour of the given vertex.
 */
public int getColour(int vertex)
{
  return theGraph[vertex][0];
}

/**
 * Assigns the given colour to the given vertex.
 *
 * @param  vertex - the given vertex
 * @param  colour - the given colour
 */
public void setColour(int vertex, int colour)
{
  if(getColour(vertex) == 0) {
      theGraph[vertex][0] = colour;
  }
}

/**
 * Returns the lowest colour that can be used for colour the vertex.
 * A colour is free to use for colouring if none of the neighbours of
 * the given vertex has been coloured by the colour.
 *
 * @param  vertex - the given vertex
 * @return the lowest colour that can be used for colour the vertex
```

```
 */
public int getFreeColour(int vertex)
{
  int colour = 1;
  HashSet<Integer> colours = new HashSet<Integer>();

  for(int i=1; i < theGraph.length; i++) {
      if(theGraph[vertex][i] != 0) {
        colours.add(getColour(i));
      }
  }

  while(colours.contains(colour)) {
      colour++;
  }

  return colour;
}

/**
 * Returns the saturation of the given vertex, that is number of
 * colours in found in the neighbourhood
 *
 * @param  vertex - the given vertex
 * @return Returns the saturation of the given vertex.
 */
public int getSaturation(int vertex)
{
  HashSet<Integer> colours = new HashSet<Integer>();

  for(int i=1; i < theGraph.length; i++) {
      if(theGraph[vertex][i] != 0) {
        colours.add(getColour(i));
      }
  }

  return colours.size();
}

/**
 * Returns the highest colour that has been used to colour the graph.
 *
```

```
 * @return Returns the highest colour that has been used to colour
 * the graph.
 */
public int getMaxColour()
{
  int colour = getColour(1);

  for(int i=2; i < theGraph.length; i++) {
      if(colour < getColour(i)) {
        colour = getColour(i);
      }
  }

  return colour;
}

/**
 * Returns the size of the graph including an extra coloum/row for
 * storing the colours given to each vertex.
 *
 * @return Returns the size of the graph including an extra coloum/row
 * for storing the colours given to each vertex.
 */
public int size()
{
  return theGraph.length;
}


/**
 * Converts the graph to a string and returns it.
 *
 * @return  Converts the graph to a string and returns it.
 */
public String toString()
{
  StringBuffer string = new StringBuffer();

  string.append("------------------------------------\n");

  for(int i=1; i < theGraph.length;i++) {
      string.append(i + ":\n----\ncolour: " + theGraph[i][0] +
```

```
                    "\nedges: ");

            for(int j=1;j < theGraph[i].length;j++) {
              if(theGraph[i][j] != 0) {
                  string.append(j + " ");
              }
            }
            string.append("\n-------------------------------------\n");
        }

        return string.toString();
    }
}
```

I.3 package probability

I.3.1 ProbabilityFunction.java

```
package probability;
import java.util.*;

/**
 * This class handles a probability function, the function can contain
 * integers, the variable n (used as the number of vertices in the
 * graph), addition(<tt>+</tt>), substraction(<tt>-</tt>),
 * multiplication(<tt>*</tt>), division(<tt>/</tt>) and
 * parentheses(<tt>()</tt>).
 */

public class ProbabilityFunction{
    ArrayList<String> function = new ArrayList<String>();
    double value = 0.0;

    public ProbabilityFunction(String function) {
      String[] data = function.split(" ");

      for(int i = 0; i < data.length; i++) {
          this.function.add(data[i]);
      }
    }

     /**
     * This method returns the value of the function.
     *
     * @return the value of the function
     */
    public double getValue() {
      return value;
    }

    /**
     * This calculates the value of the function.
     * @param variables the variables and their values.
     * @return the value of the function
     */
    public double calculate(HashMap<String, String>  variables) {
```

```
ArrayList<String> result = new ArrayList<String>();

for(int i = 0; i < function.size(); i++) {
    result.add(function.get(i));
}

for(int i = 0; i < result.size(); i++) {
    String data = result.get(i);

    if(variables.get(data) != null) {
      result.set(i, variables.get(data));
    }
}


for(int i = 0; i < result.size(); i++) {
    String data = result.get(i);

    if(data.equals("sqrt") && result.get(i+1).equals("(")) {
      ArrayList<String> subExpression =
          new ArrayList<String>();

        for(int j = i+2; j < result.size(); j++) {
          if(!result.get(j).equals(")")) {
            subExpression.add(result.get(j));
          }
          else {
            j = result.size();
          }
      }

      int subExpressionSize = subExpression.size();

      for(int j = 0; j < subExpressionSize + 3; j++) {
          result.remove(i);
      }

      if(result.size() <= i) {
          result.add(Math.sqrt
                    (Double.parseDouble
                   (calculateSubExpression
                    (subExpression))) + "");
```

```
        } else {
            result.set(i,
                     Math.sqrt
                     (Double.parseDouble
                    (calculateSubExpression
                     (subExpression))) + "");
        }

        i = i - subExpressionSize;
      }
    }

    value = Double.parseDouble(calculateSubExpression(result));

    return value;
  }

   /**
   * This method calculates the value of an expression.
   * @param subExpression the expression to be calculated
   * @return the value of the expression
   */
  private String calculateSubExpression(ArrayList<String>subExpression) {
    ArrayList<String> result = subExpression;

    for(int i = 0; i < result.size(); i++) {
        String data = result.get(i);

        if(data.equals("*")) {
          double factor =
              Double.parseDouble(result.get(i-1)) * Double.parseDouble(result.get(i+1));

          result.set(i, factor + "");
          result.remove(i-1);
          result.remove(i);
          i = i - 2;
        } else if(data.equals("/")) {
          double qout = Double.parseDouble(result.get(i-1)) / Double.parseDouble(result.get(i+1));

          result.set(i, qout + "");
          result.remove(i-1);
          result.remove(i);
```

```
            i= i - 2;
        }
    }

    for(int i = 0; i < result.size(); i++) {
        String data = result.get(i);

        if(data.equals("+")) {
            double sum = Double.parseDouble(result.get(i-1)) + Double.parseDouble(result.get(i+1));

            result.set(i, sum + "");
            result.remove(i-1);
            result.remove(i);
            i = i - 2;

        } else if(data.equals("-")) {
            double diff = Double.parseDouble(result.get(i-1)) - Double.parseDouble(result.get(i+1));

            result.set(i, diff + "");
            result.remove(i-1);
            result.remove(i);
            i = i - 2;
        }
    }

    return result.get(0);
}

 /**
 * This methods returns a string consisting of the probability
 * function.
 * @return a string consisting o the probability function.
 */
public String toString() {
    String outString = "p(n) =";

    for(int i = 0; i < function.size(); i++) {
        outString += " " + function.get(i);
    }

    return outString;
}
```

I.4 package constraints

I.4.1 ConstraintInterface.java

```
package constraints;
import graph.*;
import java.util.*;

/**
 * This interface handles the method used for examing if a constraint
 * exists in the graph.
 */
public interface ConstraintInterface{
    /**
     * This method examining if some structure exists in the graph
     * after a new edge has been
     * added to it.
     * @param theGraph The graph to be examined.
     * @param newEdge the edge to be added to the graph
     * @return true if the substructure exists in the graph
     *         and false otherwise.
     */
    public boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge);
}
```

I.4.2 ConstraintTriangle.java

```
package constraints;
import graph.*;
import java.util.*;

/**
 * This class implements the interface <tt>ConstraintInterface</tt>
 * and is used for examing if the constraint <tt>triangle</tt> will
 * exist in the graph after the new edge has been added to the graph.
 */

public class ConstraintTriangle implements ConstraintInterface{
    /**
     * This method examines if a triangle exists in the graph after
     * a new edge has been added to it.
     * @param theGraph The graph to be examined.
     * @param newEdge the edge to be added to the graph
     * @return true if the substructure exists in the graph and false
     *         otherwise.
     */
    public boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge){
      int e0 = newEdge.get(0);
      int e1 = newEdge.get(1);

      for(int v = 1; v < theGraph.size(); v++) {
          if(v != e0 && v != e1 &&
             theGraph.edgeExists(e0, v) && theGraph.edgeExists(e1, v)) {
            return true;
          }
      }

       return false;
    }
}
```

I.4.3 ConstraintFourCycle.java

```
package constraints;
import graph.*;
import java.util.*;

/**
 * This class implements the interface <tt>ConstraintInterface</tt>
 * and is used for examing if the constraint <tt>four cycle</tt> will
 * exist in the graph after the new edge has been added to the graph.
 */
public class ConstraintFourCycle implements ConstraintInterface{
    /**
     * This method examines if a four cycle exists in the graph after
     * a new edge has been added to it.
     * @param theGraph The graph to be examined.
     * @param newEdge the edge to be added to the graph
     * @return true if the substructure exists in the graph and false
     *         otherwise.
     */
    public  boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge) {
      int e0 = newEdge.get(0);
      int e1 = newEdge.get(1);

      for(int v0 = 1; v0 < theGraph.size(); v0++) {
          for(int v1 = 1; v1 < theGraph.size(); v1++) {
            if(v0 != v1 &&
               !newEdge.contains(v0) && !newEdge.contains(v1) &&
               theGraph.edgeExists(e0, v0) && theGraph.edgeExists(e1, v1) && theGraph.edgeExists(v0, v1)) {
                return true;
            }
          }
      }

      return false;
    }
}
```

I.4.4 ConstraintTetrahedron.java

```
package constraints;
import graph.*;
import java.util.*;

/**
 * This class implements the interface <tt>ConstraintInterface</tt>
 * and is used for examing if the constraint <tt>tetrahedron</tt>
 * will exist in the graph after the new edge has been added to the
 * graph.
 */
public class ConstraintTetrahedron implements ConstraintInterface{
    /**
     * This method examines if a tetrahedron exists in the graph
     * after a new edge has been added to it.
     * @param theGraph The graph to be examined.
     * @param newEdge the edge to be added to the graph
     * @return true if the substructure exists in the graph and false
     *         otherwise.
     */
    public  boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge) {
      int e0 = newEdge.get(0);
      int e1 = newEdge.get(1);

      for(int v0 = 1; v0 < theGraph.size(); v0++) {
          if(theGraph.edgeExists(e0, v0) && theGraph.edgeExists(e1, v0)) {
            for(int v1 = 1; v1 < theGraph.size(); v1++) {
                if(v1 != v0 && e0 != v0 && e1 != v1) {
                  if(theGraph.edgeExists(e0, v1) && theGraph.edgeExists(e1, v1)) {
                      if(theGraph.edgeExists(v0, v1)) {
                        return true;
                      }
                  }
                }
            }
          }
      }

      return false;
    }
}
```

I.4.5 ConstraintOctahedron.java

```
package constraints;
import graph.*;
import java.util.*;

/**
 * This class implements the interface <tt>ConstraintInterface</tt>
 * and is used for examing if the constraint <tt>octahedron</tt>
 * will exist in the graph after the new edge has been added to the
 * graph.
 */
public class ConstraintOctahedron implements ConstraintInterface{
    /**
     * This method examines if an octahedron exists in the graph
     * after a new edge has been added to it.
     * @param theGraph The graph to be examined.
     * @param newEdge the edge to be added to the graph
     * @return true if the substructure exists in the graph and false
     * otherwise.
     */

    public boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge) {
      int e0 = newEdge.get(0);
      int e1 = newEdge.get(1);

      for(int v0 = 1; v0 < theGraph.size(); v0++) {
          for(int v1 = 1; v1 < theGraph.size(); v1++) {
            if(v0 != v1 && v0 != e0 &&
               !newEdge.contains(v0) && !newEdge.contains(v1) &&
               theGraph.edgeExists(e0, v0) && theGraph.edgeExists(e0, v1) &&
               theGraph.edgeExists(v0, v1) && theGraph.edgeExists(e1, v0)) {
                for(int v2 = 1; v2 < theGraph.size(); v2++) {
                  if(v2 != v0 && v2 != v1 &&
                     !newEdge.contains(v2) &&
                     theGraph.edgeExists(e0, v2) && theGraph.edgeExists(v1, v2) && theGraph.edgeExists(v2, e1)) {
                       for(int v3 = 1; v3 < theGraph.size(); v3++)
                           {
                         if(v3 != v0 && v3 != v1 && v3 != v2 &&
                          !newEdge.contains(v3)) {
                           if(theGraph.edgeExists(e1, v3) && theGraph.edgeExists(v0, v3) &&
                              theGraph.edgeExists(v1, v3) && theGraph.edgeExists(v2, v3)) {
```

```
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }

    return false;
  }
}
```

I.4.6 ConstraintFromFile.java

```
package constraints;
import java.util.*;
import java.io.*;
import graph.*;

/**
 * This class implements the interface <tt>ConstraintInterface</tt>
 * and is used for reading a file which contain a desciption of a
 * a constraint written in pseudo code. The class can also
 * examine if the constraint will exist in the graph after the new
 *  edge has been added to the
 * graph.
 */
public class ConstraintFromFile implements ConstraintInterface{
    ArrayList<Statement> statements = new ArrayList<Statement>();

    /**
     * This Constructor reads in the pseudocode needed to examine a
     * constraint from an inputfile and stores it.
     * @param filename the file to be read
     */
    public ConstraintFromFile(String filename) {
      try {
          BufferedReader reader =  new BufferedReader(new FileReader(filename));
          String line = reader.readLine();

          while(line != null) {
            if(line.trim().startsWith("for_all")) {
                Statement theStatement = new Statement(Statement.forallStatement);
                String[] split = line.trim().split(" ");
                theStatement.setCurrentVertex(split[2]);
                ArrayList<String> excludedVertices = new ArrayList<String>();
                String[] splitVertices = split[6].split(",");

                excludedVertices.add(splitVertices[0].substring(1, splitVertices[0].length()));

                for(int i = 1; i < splitVertices.length-1; i++) {
                  excludedVertices.add(splitVertices[i]);
                }
```

```
        excludedVertices.add
          (splitVertices[splitVertices.length-1].substring(0, splitVertices[0].length()-1));

        theStatement.setExcludedVertices(excludedVertices);
        statements.add(theStatement);
    }
    else if(line.trim().startsWith("if")) {
        Statement theStatement = new Statement(Statement.ifStatement);
        line = line.trim().substring(4, line.trim().length()-6);
        String[] splitExpressions = line.split("and");
        ArrayList<ArrayList<String>> edges = new ArrayList<ArrayList<String>>();

        for(int i = 0; i < splitExpressions.length; i++) {
          String[] currentEdge = splitExpressions[i].trim().split(" ")[0].split(",");
          ArrayList<String> edge = new ArrayList<String>();

          edge.add(currentEdge[0].substring(1,currentEdge[0].length()));
          edge.add(currentEdge[1].substring(0,currentEdge[1].length()-1));
          edges.add(edge);
        }

        theStatement.setEdges(edges);
        statements.add(theStatement);
    }
    else if(line.trim().startsWith("return")) {
        Statement theStatement = new Statement(Statement.returnStatement);

        theStatement.setValue(line.trim().split(" ")[1]);
        statements.add(theStatement);

    }
    else if(line.trim().startsWith("end for")) {
        Statement theStatement = new Statement(Statement.endfor);

        statements.add(theStatement);
    }
    else if(line.trim().startsWith("end if")) {
        Statement theStatement = new Statement(Statement.endif);

        statements.add(theStatement);
    }
```

```
        line = reader.readLine();
      }

      reader.close();
    } catch(Exception e) {
      System.err.println("Error while reading the file " + filename + ".");
    }
  }

  /**
   * This method examines if a constraint written in a given file will
   * exists in the graph after a new edge has been added to it.
   * @param theGraph The graph to be examined.
   * @param newEdge the edge to be added to the graph
   * @return true if the substructure exists in the graph and false
   *         otherwise.
   */
  public boolean exists(GraphMatrix theGraph, ArrayList<Integer> newEdge) {
    ArrayList<Integer> vertices = new ArrayList<Integer>();

    vertices.add(newEdge.get(0));
    vertices.add(newEdge.get(1));

    return executeStatements(theGraph, 0, vertices);
  }

   /**
   * This method executes the statement read from the file and examines if the constraint
   * exists in the graph.
   * @param theGraph The graph to be examined.
   * @param step location of the statement to be executed
   * @return true if the substructure exists in the graph and false
   *         otherwise.
   */
  public boolean executeStatements(GraphMatrix theGraph, int step, ArrayList<Integer> vertices) {
    Statement currentStatement = statements.get(step);
    boolean result = false;

    if(currentStatement.getType().equals(Statement.forallStatement)) {
      for(int v = 1; v < theGraph.size(); v++) {
        if(!vertices.contains(v)) {
          vertices.add(v);
```

```
            result = result || executeStatements(theGraph, step + 1, vertices);
            if(result) {
              return result;
            }

            vertices.remove(vertices.size() - 1);
        }
      }
    }
    else if(currentStatement.getType().equals(Statement.ifStatement)) {
      boolean allExists = true;

      for(ArrayList<String> edge: currentStatement.getEdges()) {
        ArrayList<Integer> testEdge = new ArrayList<Integer>();

        for(String vertex : edge) {
            if(vertex.equals("e0")) {
              testEdge.add(vertices.get(0));
            } else if(vertex.equals("e1")) {
              testEdge.add(vertices.get(1));
            }
            else {
              testEdge.add(vertices.get(2 + Integer.parseInt(vertex.substring(1, vertex.length()))));
            }
        }

        if(!theGraph.edgeExists(testEdge.get(0), testEdge.get(1))) {
            allExists = false;
        }
      }

      if(allExists) {
        boolean nextResult = false;

        if(step + 1 < statements.size()) {
            nextResult = executeStatements(theGraph, step + 1, vertices);
        }

        if(allExists && nextResult) {
            return true;
        }
      }
```

```
        }
        else if(currentStatement.getType().equals(Statement.returnStatement)) {
            return currentStatement.getValue() == 1;
        }
        else if((currentStatement.getType().equals(Statement.endfor) ||
                 currentStatement.getType().equals(Statement.endif)) &&
                step + 1 < statements.size()) {
            return result || executeStatements(theGraph, step + 1, vertices);
        }

        return result;
    }

    /**
     * This method returns a string which contains the pseudocode read from the file.
     * @return a string with the pseudocode
     */
    public String toString() {
        String outString = "";
        String tab = "";

        for(Statement statement: statements) {
            if (statement.getType().equals(Statement.endfor) || statement.getType().equals(Statement.endif)) {
                tab = tab.substring(0, tab.length()-1);
            }

            outString += tab + statement + "\n";

            if(statement.getType().equals(Statement.forallStatement) ||
                statement.getType().equals(Statement.ifStatement)) {
                tab += "\t";
            }
        }

        return outString;
    }
}
```

I.4.7 Statement.java

```
package constraints;
import java.util.*;

/**
 * This class stores information about the statements in a given
 * input file.
 */
public class Statement {
    public static String forallStatement = "for_all";
    public static String ifStatement = "if";
    public static String endfor = "endfor";
    public static String endif = "endif";
    public static String returnStatement = "return";
    private String type = "";
    private int value = -1;
    private String currentVertex = "";
    private ArrayList<String> excludedVertices = null;
    private ArrayList<ArrayList<String>> edges = null;

    /**
     * This Constructor creates a statement.
     * @param type the type of the statement
     */
    public Statement(String type) {
      this.type = type;
    }

    /**
     * This method returns the type of the statement.
     * @return the type of the statement
     */
    public String getType() {
      return type;
    }

    /**
     * This method returns the value of the statement (if it exists).
     * @return the value of the statement (if it exists).
     */
    public int getValue() {
```

```
  return value;
}

/**
 * This method returns a list of edges used by an if-statement to
 * test if some edges exists in the graph.
 * @return a list of edges
 */
public ArrayList<ArrayList<String>> getEdges() {
  return edges;
}

/**
 * This method sets a value to a statement.
 * @param value the value to be set
 */
public void setValue(String value) {
  if(value.equals("true")) {
      this.value = 1;
  } else {
      this.value = 0;
  }
}

/**
 * This method sets which vertex a for all loop is working with
 * @param currentVertex the vertex a for all loop is working with
 */
public void setCurrentVertex(String currentVertex) {
  this.currentVertex = currentVertex;
}

/**
 * This method sets which vertices should be avoided by the for all loop
 * @param excludedVertices the vertices to be avoided
 */
public void setExcludedVertices(ArrayList<String> excludedVertices) {
  this.excludedVertices = excludedVertices;
}

/**
 * This method sets edges to be examined by an if-statement.
```

```
 * @param edges the edges to be examined
 */
public void setEdges(ArrayList<ArrayList<String>> edges) {
  this.edges = edges;
}

/**
 * This method returns a string containing the statement.
 * @return a string containing the statement
 */
public String toString() {
  String outString = type;

  if(value != -1) {
      outString += " " + value;
  }

  if(excludedVertices != null) {
      outString += " ( " + currentVertex + " in V \\ {";
      for(int i = 0; i < excludedVertices.size(); i++) {
        if(i > 0) {
            outString += ",";
        }

        outString += excludedVertices.get(i);
      }
      outString += "} ) do";
  }

  if(edges != null) {
      outString += " ( {";

      for(int i = 0; i < edges.size(); i++) {
        if(i > 0) {
            outString += " and {";
        }

        outString +=
            edges.get(i).get(0) + "," +
            edges.get(i).get(1) + "} in E";
      }
```

```
            outString += " ) then";
        }

        return outString;
    }
}
```

I.5 package Colouring

I.5.1 ColouringInterface.java

```
package colouring;
import graph.*;
/**
 * This interface handles the method used for colouring.
 *
 */

public interface ColouringInterface {
    /**
     * This method colours the given graph.
     *
     *@param theGraph - the graph to be coloured
     *@return the coloured graph
     */
    public GraphMatrix colouring(GraphMatrix theGraph);
}
```

I.5.2 DegreeOfSaturationColouringWithBacktracking.java

```
package colouring;
import graph.*;
import java.util.*;

/**
 * This class implements the interface <tt>ColouringInterface</tt> and
 * the method in this class colours the graph according to the BSC-
 * algorithm.
 */
public class DegreeOfSaturationColouringWithBackTracking implements ColouringInterface {
    /**
     * This method colours the given graph <tt>theGraph</tt> and returns
     * it.
     * @param theGraph the graph to be coloured
     * @return the coloured graph
     */
    public GraphMatrix colouring(GraphMatrix theGraph) {
      int[] A = sortAccordingToTheHighestDegree(theGraph, theGraph.getVertices());
      int[] Fopt = new int[A.length];
      int[] colours = new int[A.length + 1];
      colours[0] = 0;
      int start = 0;
      int optColourNumber = A.length + 1;
      int x = A[0];
      int i = start;
      ArrayList<Integer> U = new ArrayList<Integer>();
      U.add(1);

      while(start >= 0) {
          boolean back = false;

          for(i = start; i < A.length; i++) {
            if(i > start) {
                x = getVertexWithHighestSaturation(theGraph, A);
                U = theGraph.getFreeColours(x, optColourNumber);
            }

            if(U.size() > 0) {
                theGraph.setColour(x, U.get(0));
                colours[i+1] = Math.max(U.get(0), colours[i]);
```

```
  } else {
      start = i - 1;
      back = true;
      break;
  }
}

if(back) {
  if(start >= 0) {
      x = A[start];
      theGraph.uncolour(x);
  }
}
else {
  for(int c=0; c < A.length; c++) {
      Fopt[c] = theGraph.getColour(A[c]);
  }

  optColourNumber = colours[A.length];

  for(int c=0; c < A.length; c++) {
      if(Fopt[c] == optColourNumber) {
        i = c;
        c = A.length;
      }
  }

  start = i - 1;

  if(start < 0) {
      break;
  }

  for(int c=start; c < A.length; c++) {
      theGraph.uncolour(A[c]);
  }

  x = A[start];
  U = theGraph.getFreeColours(x, optColourNumber);

}
```

```
  }

  for(int c=0; c < A.length; c++) {
      theGraph.setColour(A[c], Fopt[c]);
  }

  return theGraph;
}

/**
 * This method sorts the vertices in the graph according to their
 degrees.
 * @param theGraph the graph to be coloured
 * @param vertices the vertices to be sorted
 * @return the vertices sorted in according to their degree and in
 a descending order.
 */
private int[] sortAccordingToTheHighestDegree(GraphMatrix theGraph, int[] vertices) {
    for(int i = 0; i < vertices.length; i++) {
      for(int j = i; j < vertices.length; j++) {
           if(theGraph.getDegree(vertices[i]) <
          theGraph.getDegree(vertices[j])) {
           int temp = vertices[i];
           vertices[i] = vertices[j];
           vertices[j] = temp;
         }
      }
    }

  return vertices;
}

/**
 * This method returns the vertex with the highest saturation.
 * @param theGraph the graph to be coloured
 * @param vertices the vertices in the graph
 * @return the vertex with the highest saturation
 */
private int getVertexWithHighestSaturation(GraphMatrix theGraph, int[] vertices) {
  int v = -1;
  int vSaturation = -1;
```

```
        for(int i = 0; i < vertices.length; i++){

            if(theGraph.getColour(vertices[i]) == 0) {
              if(v == -1) {
                  v = i;
                  vSaturation = theGraph.getSaturation(vertices[i]);
              } else
                  if(vSaturation < theGraph.getSaturation(vertices[v]))
                  {
                    v = i;
                    vSaturation = theGraph.getSaturation(vertices[v]);
                  }
            }
        }

        return vertices[v];
    }
}
```

I.6 package generator

I.6.1 GenerateGraph.java

```
package generator;
import graph.*;
import colouring.*;
import constraints.*;
import probability.*;
import java.util.*;
import java.io.*;

/**
 * This class contains the main-method and methods used for
 * generating a random graph with a constraint.
 */

public class GenerateGraph {
    /**
     * This method checks if the given string only contains an integer.
     * @param string the string to be examined.
     * @return true if the string only contains an integer and false
     * otherwise.
     */
    private static boolean isANumber(String string)
    {
      try  {
          int d = Integer.parseInt(string);
      }
      catch(NumberFormatException e)  {
          return false;
      }
      return true;
    }

    /**
     * The main method for the program.
     */
    public static void main(String[] args) {
      try {
          HashMap<String,String> variables = new HashMap<String,String>();
          int size = 0;
```

```
ProbabilityFunction pf = null;
String function = "";
ConstraintInterface theConstraint = null;
GraphMatrix theGraph = null;
String method = "";
String outputfile = "";
String constraint = "";

if (args.length == 0) {
  System.out.println("Usage: java -jar G_n_p_generator.jar -help for more help and options");
  System.exit(0);
}

for(int a =0; a < args.length; a++) {
  if(args[a].equals("-help")) {
      System.out.println("-----------------------------------------------------------------");
      System.out.println("   RANDOM SAMPLING OF STRUCTURES WITH FORBIDDEN CONSTRAINTS");
      System.out.println("                          VERSION 1.0");
      System.out.println("-----------------------------------------------------------------");
      System.out.println("-help                                - show options\n");
      System.out.println("-size <integer>                      - the size of the graph\n");
      System.out.println("-constraint <substructure>           - the structure that is not");
      System.out.println("                                       allowed in the graph, can");
      System.out.println("                                       either be triangle,");
      System.out.println("                                       four_cycle, tetrahedron,");
      System.out.println("                                       octahedron or a given file\n");
      System.out.println("-probability <probability function> - the probability used to");
      System.out.println("                                       determine the existence");
      System.out.println("                                       of each edge in the graph,");
      System.out.println("                                       should be on the form");
      System.out.println("                                       p ( n ) = expression\n");
      System.out.println("-method <edges/vertices>             - the method used for");
      System.out.println("                                       generating the graph,");
      System.out.println("                                       can either be based on");
      System.out.println("                                       edges or vertices\n");
      System.out.println("-o <filename>                        - the output-file in which");
      System.out.println("                                       the graph and its properties");
      System.out.println("                                       is written to");
      System.out.println("-----------------------------------------------------------------");
      System.exit(0);
  }
  else if(args[a].equals("-size")) {
```

```
    if(isANumber(args[a+1])) {
      size = Integer.parseInt(args[a+1]);
      variables.put("n", size + "");
      a++;
    } else {
      System.out.println("The size must be an integer.");
      System.exit(0);
    }
}
else if(args[a].equals("-probability")) {
    for(int f = a + 6; f < args.length; f++) {
      if(args[f].length() > 1 && args[f].startsWith("-")) {
          a = f - 1;
          f = args.length;
      } else {
          function += args[f] + " ";
      }
    }

    pf = new ProbabilityFunction(function);
}
else if(args[a].equals("-constraint")) {
    constraint = args[a+1];

    if(args[a+1].equals("triangle")) {
      theConstraint = new ConstraintTriangle();
    }
    else if(args[a+1].equals("four_cycle")) {
      theConstraint = new ConstraintFourCycle();
    }
    else if(args[a+1].equals("tetrahedron")) {
      theConstraint = new ConstraintTetrahedron();
    }
    else if(args[a+1].equals("octahedron")) {
      theConstraint = new ConstraintOctahedron();
    } else {
      theConstraint = new ConstraintFromFile(args[a+1]);
    }
    a++;
} else if(args[a].equals("-method")) {
    method = args[a+1];
    a++;
```

```
  } else if(args[a].equals("-o")) {
      outputfile = args[a+1];
      a++;
  }
}

boolean missingArgument = false;

if(size == 0) {
  System.out.println("The size is missing or has the value 0.");
  missingArgument = true;
}

if(pf == null) {
  System.out.println("The probability function is missing.");
  missingArgument = true;
}

if(theConstraint == null) {
  System.out.println("The forbidden constraint is missing.");
  missingArgument = true;
}
if(method.equals("")) {
  System.out.println("The generating method is missing.");
  missingArgument = true;
} if(outputfile.equals("")) {
  System.out.println("The outputfile is missing.");
  missingArgument = true;
}

if(missingArgument) {
  System.exit(0);
}

if(method.equals("edges")) {
  theGraph = generateGraphEdges(size, pf.calculate(variables), theConstraint);
} else if(method.equals("vertices")) {
  theGraph = generateGraphVertices(size, pf.calculate(variables), theConstraint);
} else {
  System.out.println("The generating method" + " " + method + " is unknown.");
  System.exit(0);
}
```

```
        ColouringInterface colouring = new DegreeOfSaturationColouringWithBackTracking();

        theGraph = colouring.colouring(theGraph);

        BufferedWriter writer =
          new BufferedWriter(new FileWriter(outputfile));

        writer.write("-----------------------------------------------------------------");
        writer.write("\n   RANDOM SAMPLING OF STRUCTURES WITH FORBIDDEN CONSTRAINTS");
        writer.write("\n                          VERSION 1.0");
        writer.write("\n-----------------------------------------------------------------");
        writer.write("\ngraph size          : " + size);
        writer.write("\nforbidden structure : " + constraint);
        writer.write("\nprobability function: p ( n ) = " + function);
        writer.write("\ngenerating method   : " + method);
        writer.write("\n-----------------------------------------------------------");
        writer.write("\nDSATUR              : " + theGraph.getMaxColour());
        writer.write("\n#edges              : " + numberOfEdges(theGraph));
        writer.write("\n#triangles          : " + countTriangles(theGraph));
        writer.write("\n-----------------------------------------------------------");
        writer.write("\ngenerated graph:\n" + theGraph);

        writer.close();

      } catch(Exception e) {
        System.out.println("Error during writing to file" +  e.toString());
      }
    }

/**
     * This method creates a random graph according to edges.
     * @param size the number of vertices in the graph
     * @param pf the probability function
     * @param theConstraint method for searching after the forbidden structure
     * @return a random graph
     */
    private static GraphMatrix generateGraphEdges(int size, double pf, ConstraintInterface theConstraint) {
            GraphMatrix newGraph = new GraphMatrix(size);
      Random randomizer = new Random();
      int numberOfEdges = 0;
      ArrayList<ArrayList<Integer>> edges = new ArrayList<ArrayList<Integer>>();
```

```
  for(int i = 1; i < size; i++) {
      for(int j = i + 1; j <= size; j++) {
        ArrayList<Integer> edge = new ArrayList<Integer>();

        edge.add(i);
        edge.add(j);
        edges.add(edge);
      }
  }

  while(edges.size() > 0) {
      int i = (int)(randomizer.nextDouble() * edges.size());
      ArrayList<Integer> edge = edges.get(i);

      if(randomizer.nextDouble() <= pf) {
        newGraph.addEdge(edge.get(0), edge.get(1));
        if(theConstraint.exists(newGraph, edge)) {
            newGraph.removeEdge(edge.get(0), edge.get(1));
        }
      }

      edges.remove(i);
  }

  return newGraph;
}

/**
 * This method creates a graph according to vertices.
 * @param size the number of vertices in the graph
 * @param pf the probability function
 * @param theConstraint method for searching after the forbidden structure
 * @return a random graph
 */
private static GraphMatrix generateGraphVertices(int size, double pf, ConstraintInterface theConstraint) {
        GraphMatrix newGraph = new GraphMatrix(size);
  Random randomizer = new Random();
  ArrayList<Integer> vertices = new ArrayList<Integer>();

  for(int i = 1; i <= size; i++) {
      vertices.add(i);
```

```
  }

  while(vertices.size() > 0) {
      int i = (int)(randomizer.nextDouble() * vertices.size());
      int v = vertices.get(i);

      System.out.print(v + " ");

      for(int j = v + 1; j <= size; j++) {
        if(randomizer.nextDouble() <=  pf) {
           newGraph.addEdge(v,j);

           ArrayList<Integer> newEdge = new ArrayList<Integer>();
           newEdge.add(v);
           newEdge.add(j);

           if(theConstraint.exists(newGraph, newEdge)) {
               newGraph.removeEdge(v,j);
           }
        }
        }

      vertices.remove(i);
  }

  return newGraph;
}


/**
 * This method returns the number of triangles in the graph
 * @param theGraph the graph to be examined
 * @return the number of triangles
 */
static private int countTriangles(GraphMatrix theGraph) {
    HashSet<ArrayList<Integer>> T = new HashSet<ArrayList<Integer>>();

  for(int i = 1; i < theGraph.size(); i++) {
      ArrayList<Integer> visited = new ArrayList<Integer>();
      visited.add(i);
      T.addAll(findAllTriangles(theGraph, visited));
  }
```

```
    return T.size();
  }

  /**
   * This method returns found in the graph when the search starts at a given vertex
   * @param theGraph the graph to be examined
   * @param visited contains the vertices that have been visited sofar
   * @return a set of the triangles found in the graph
   */
  static private HashSet<ArrayList<Integer>> findAllTriangles(GraphMatrix theGraph, ArrayList<Integer> visited) {
    HashSet<ArrayList<Integer>> T = new HashSet<ArrayList<Integer>>();

    if(visited.size() <= 3) {
        ArrayList<Integer> neighbours = theGraph.getNeighbours(visited.get(visited.size()-1));

        for (Integer neighbour : neighbours) {
          if(visited.get(0) == neighbour && visited.size() == 3) {
              ArrayList<Integer> newRelation = new ArrayList<Integer>();
              newRelation.addAll(visited);
              Collections.sort(newRelation);
              T.add(newRelation);
          }

          if (visited.size() < 3 && !visited.contains(neighbour)){
              visited.add(neighbour);
              T.addAll(findAllTriangles(theGraph, visited));
              visited.remove(visited.size()-1);
          }
        }
    }

    return T;
  }

  /**
   * This method returns the number of edges in the graph
   * @param theGraph the graph to be examined
   * @return the number of edges in the graph
   */
  static private int numberOfEdges(GraphMatrix theGraph) {
    int[][] matrix = theGraph.getMatrix();
```

```
        int numberOfEdges = 0;

        for(int i = 1; i < matrix.length; i++) {
            for(int j = i + 1; j < matrix.length; j++) {
              if (matrix[i][j] == 1) {
                  numberOfEdges++;
              }
            }
        }

        return numberOfEdges;
    }
}
```

I.7 build.xml

```
<project name="G_n_p_generator" default="jar" basedir=".">

<property name="src" location="src"/>
<property name="build" location="bin"/>

<target name="all" depends="jar">
   <tstamp/>
   <echo message="Done!"/>
</target>

<target name="clean">
   <delete dir="${build}"/>
</target>

<target name="compile" depends="init">
   <javac srcdir="${src}" destdir="${build}/generator/classes" includeAntRuntime="no" />
</target>


<target name="jar" depends="compile">
   <jar destfile="G_n_p_generator.jar" basedir="${build}/generator/classes">
       <manifest>
           <attribute name="Main-Class" value="generator.GenerateGraph"/>
       </manifest>
   </jar>
</target>

<target name="init" depends="clean">
  <tstamp/>
  <mkdir dir="${build}"/>
  <mkdir dir="${build}/generator"/>
  <mkdir dir="${build}/generator/classes"/>
</target>

</project>
```

J Manual

J.1 README.txt

```
CONTENT

This package includes a module for generating random graph without a forbidden constraint.

This software requires Java 6 (JDK 1.6.0+). (You must have installed it separately.
Check that the command "java -version" works and gives 1.6.0+.)

INSTALLATION

Place the gz-file in a directory and unzip it
> gtar -xzvf G_n_p_generator.tar.gz
> cd G_n_p_generator
> ant jar

To check that the installation completed successfully, run the following
command:

> java -jar G_n_p_generator

You should now get the usage printed. If you see this, you can start
using the software.

QUICKSTART

You can run the random graph generator software like this:

> java -jar G_n_p_generator.jar -size 4 -constraint triangle -method edges
-probability p ( n ) = 1 / 2 -o  testtriangle.graph

This will result in a graph containing 4 vertices and no triangles generated
according to edges and with the probability 1/2.

> java -jar G_n_p_generator.jar -size 10 -constraint four_cycle -method vertices
-probability p ( n ) = 1 / sqrt ( n )
-o  testfourcycle.graph

This will result in a graph containing 10 vertices and no four cycles generated
according to vertices and with the probability 1/sqrt(n).
```

```
> java -jar G_n_p_generator.jar -size 10 -constraint triangle.txt -method vertices
-probability p ( n ) = 1 / n  -o  testfourcycle.graph

This will result in a graph containing 10 vertices and no triangles  generated
according to vertices and with the probability 1/n. Note that in this case has
the constraint in the file triangle.txt been used.
```

J.2 Input files

J.2.1 triangle.txt

```
// check for triangle
for_all ( v0 in V \ {e0,e1} ) do
      if ( {e0,v0} in E and {e1,v0} in E ) then
         return true
      end if
end for
return false
```

J.2.2 fourcycle.txt

```
// check for fourcycle
for_all ( v0 in V \ {e0,e1} ) do
   for_all ( v1 in V \ {v0,e0,e1} ) do
      if ( {e0,v0} in E and {e1,v1} in E and {v0,v1} in E ) then
         return true
      end if
   end for
end for

return false
```

J.2.3 tetrahedron.txt

```
// check for tetrahedron
for_all ( v0 in V \ {e0,e1} ) do
   for_all ( v1 in V \ {v0,e0,e1} ) do
      if ( {e0,v0} in E and {e1,v0} in E and {e0,v1} in E and {e1,v1} in E and {v0,v1} in E ) then
         return true
      end if
   end for
end for

return false
```

J.2.4 octahedron.txt

```
// check for octahedron
for_all ( v0 in V \ {e0,e1} ) do
   for_all ( v1 in V \ {e0,e1,v0} ) do
      if ( {e0,v0} in E and {e0,v1} in E and {v0,v1} in E and {e1,v0} in E ) then
         for_all ( v2 in V \ {e0,e1,v0,v1} ) do
              if ( {e0,v2} in E and {v1,v2} in E and {v2,e1} in E ) then
                 for_all ( v3 in V \ {e0,e1,v0,v1,v2} ) do
                    if ( {e1,v3} in E and {v0,v3} in E and {v1,v3} in E and {v2,v3} in E ) then
                       return true
                    end if
                 end for
             end if
         end for
      end if
   end for
end for
return false
```