Fluxos em Redes
Análise Experimental

Juliana Barby Simão
Apoio financeiro da FAPESP
Processo 04/00580-8

Marcelo Hashimoto
Apoio financeiro da FAPESP
Processo 04/00581-4

Orientador: José Coelho de Pina

Sumário

1 Introdução

Neste documento apresentamos os resultados dos testes experimentais feitos com os algoritmos que implementamos para resolver problemas sobre fluxos em redes. A intenção deste relatório é mostrar o desempenho de nossas implementações na prática, comparar a eficiência prática com a teórica e conjecturar a respeito dos resultados obtidos. Todos os testes foram realizados em uma máquina com processador AMD Athlon(tm) 2000 MHz, com cerca de 2 GB de memória RAM e 5 GB de *swap*, executando Linux. Além disso, o tempo foi sempre medido em segundos.

2 Fluxos máximos

Os algoritmos implementados para resolver o problema do fluxo máximo foram testados utilizando-se dois conjuntos diferentes de instâncias. O primeiro, cujas instâncias serão denotadas por `ak`, foi obtido a partir de um gerador desenvolvido por Cherkassky e Goldberg [1] e consiste de redes especialmente projetadas para prejudicar a eficiência do método dos caminhos de aumento. O segundo, cujas instâncias serão denotadas por `Bsp`, foi obtido a partir de uma biblioteca de instâncias para problemas combinatórios mantida por Schmidt [2].

2.1 Caminhos de aumento de comprimento mínimo

O algoritmo dos caminhos de aumento de comprimento mínimo executa $O(nm)$ iterações e a implementação feita tem consumo de tempo $O(nm(n+m))$. As tabelas abaixo mostram os resultados obtidos para os dois conjuntos de instâncias utilizados.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	7	0.000007
ak2	22	31	1000000	11	0.000013
ak3	30	43	1000000	15	0.000021
ak4	50	73	1000000	25	0.000051
ak5	102	151	1000000	51	0.000198
ak6	202	301	1000000	101	0.000754
ak7	302	451	1000000	151	0.001691
ak8	1002	1501	1000000	501	0.022347
ak9	3002	4501	1000000	1501	0.296530
ak10	5002	7501	1000000	2501	2.221487

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	9	00.000012
Bsp2	20	100	100	5	00.000011
Bsp3	30	200	100	10	00.000034
Bsp4	50	500	100	20	00.000126
Bsp5	100	2000	1000	30	00.000717
Bsp6	200	5000	1000	51	00.004381
Bsp7	300	10000	1000	81	00.092229
Bsp8	1000	25000	1000	81	00.484416
Bsp9	3000	100000	1000	109	03.398092
Bsp10	5000	500000	1000	282	49.805928

Podemos observar que, embora o crescimento de n seja o mesmo em ambos os conjuntos de instâncias, o consumo de tempo do algoritmo cresceu muito mais para o segundo.

Como a complexidade do algoritmo não depende do valor de U e sabemos que as instâncias do primeiro conjunto foram especialmente projetadas para prejudicar o método dos caminhos de aumento, podemos conjecturar que o responsável pela disparidade nos resultados foi o crescimento de m no segundo conjunto, ou seja, podemos conjecturar que o algoritmo tem desempenho pior em redes com muitos arcos, também chamadas de redes *densas*.

Visto que o crescimento de m é quadrático na complexidade do algoritmo, esse resultado não é exatamente uma surpresa. Em uma análise mais superficial, podemos considerar que a grande quantidade de arcos resulta em uma grande quantidade de caminhos de aumento, aumentando drasticamente o número de iterações necessárias para o algoritmo terminar.

2.2 Fluxos bloqueadores de aumento

O algoritmo dos fluxos bloqueadores de aumento executa $O(n)$ iterações e a implementação feita tem consumo de tempo total $O(n^2m)$.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	4	0.000013
ak2	22	31	1000000	7	0.000025
ak3	30	43	1000000	10	0.000043
ak4	50	73	1000000	18	0.000119
ak5	102	151	1000000	37	0.000438
ak6	202	301	1000000	75	0.001745
ak7	302	451	1000000	112	0.003781
ak8	1002	1501	1000000	375	0.057970
ak9	3002	4501	1000000	1125	1.560307
ak10	5002	7501	1000000	1875	6.818910

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	4	00.000021
Bsp2	20	100	100	2	00.000016
Bsp3	30	200	100	3	00.000052
Bsp4	50	500	100	3	00.000150
Bsp5	100	2000	1000	3	00.001049
Bsp6	200	5000	1000	3	00.008650
Bsp7	300	10000	1000	4	00.080186
Bsp8	1000	25000	1000	3	00.373921
Bsp9	3000	100000	1000	3	01.802990
Bsp10	5000	500000	1000	3	19.992204

O bom desempenho do algoritmo dos fluxos bloqueadores no segundo conjunto de instâncias condiz com o fato de m crescer apenas linearmente na complexidade desse algoritmo e demonstra que a observação feita para o algoritmo dos caminhos de aumento de comprimento mínimo está provavelmente correta: a grande quantidade de arcos aumenta o número total de caminhos de aumento. Como o algoritmo dos fluxos bloqueadores considera vários caminhos de aumento em uma única iteração, ele obtém melhor desempenho em redes densas.

Por outro lado, um aumento de fluxo através de um fluxo bloqueador consome mais tempo do que um aumento através de um caminho, ou seja, o algoritmo dos caminhos de aumento de comprimento mínimo leva vantagem se existem poucos caminhos de aumento. Espera-se, portanto, que o algoritmo dos fluxos bloqueadores seja pior para redes *esparsas*, com poucos arcos. Isso realmente aconteceu para o primeiro conjunto de instâncias.

2.3 Caminhos de maior aumento

O algoritmo dos caminhos de maior aumento executa $O(m \log mU)$ iterações e a implementação feita tem consumo de tempo $O(m \log mU(n + m \log n))$.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	7	0.000014
ak2	22	31	1000000	9	0.000021
ak3	30	43	1000000	11	0.000031
ak4	50	73	1000000	16	0.000063
ak5	102	151	1000000	29	0.000208
ak6	202	301	1000000	54	0.000732
ak7	302	451	1000000	79	0.001552
ak8	1002	1501	1000000	254	0.018466
ak9	3002	4501	1000000	754	0.220338
ak10	5002	7501	1000000	1254	1.484394

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	9	00.000028
Bsp2	20	100	100	5	00.000025
Bsp3	30	200	100	7	00.000064
Bsp4	50	500	100	12	00.000285
Bsp5	100	2000	1000	15	00.001207
Bsp6	200	5000	1000	24	00.005339
Bsp7	300	10000	1000	24	00.052927
Bsp8	1000	25000	1000	31	00.221724
Bsp9	3000	100000	1000	45	01.710145
Bsp10	5000	500000	1000	129	27.555617

Como primeira observação, cabe mencionar que o crescimento de uma função logaritmica é muito baixo e que, em ambos os conjuntos de instâncias, o valor de U, embora seja alto, não cresce. Levando em consideração esses dois fatos, a complexidade do algoritmo dos caminhos de maior aumento é melhor que a dos caminhos de aumento de comprimento mínimo.

Portanto, espera-se que o desempenho seja melhor que o do algoritmo dos caminhos de aumento de comprimento mínimo para ambos os conjuntos de instâncias, como realmente ocorreu. Também espera-se que o comportamento não difira muito do algoritmo dos caminhos de aumento de comprimento mínimo quando comparado ao algoritmo dos fluxos bloqueadores, pelos motivos mencionados anteriormente. Isso também ocorreu.

2.4 Capacity scaling

O algoritmo capacity scaling executa $O(m \log U)$ iterações e a implementação feita tem consumo de tempo $O(m \log U(n+m))$.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	26	0.000008
ak2	22	31	1000000	28	0.000012
ak3	30	43	1000000	30	0.000016
ak4	50	73	1000000	35	0.000030
ak5	102	151	1000000	48	0.000095
ak6	202	301	1000000	73	0.000317
ak7	302	451	1000000	98	0.000687
ak8	1002	1501	1000000	273	0.009094
ak9	3002	4501	1000000	773	0.182068
ak10	5002	7501	1000000	1273	1.347071

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	15	00.000014
Bsp2	20	100	100	11	00.000009
Bsp3	30	200	100	14	00.000026
Bsp4	50	500	100	22	00.000133
Bsp5	100	2000	1000	27	00.000631
Bsp6	200	5000	1000	40	00.003421
Bsp7	300	10000	1000	68	00.080504
Bsp8	1000	25000	1000	58	00.359640
Bsp9	3000	100000	1000	72	02.272817
Bsp10	5000	500000	1000	224	37.972411

A alta similaridade com o comportamento do algoritmo dos caminhos de maior aumento reflete a baixa diferença entre as complexidades. O algoritmo capacity scaling leva uma evidente vantagem na complexidade, mas essa vantagem é apenas logaritmica.

Isso pôde ser bem observado no desempenho de ambos os algoritmos no primeiro conjunto de instâncias: o algoritmo capacity scaling obteve melhor desempenho, mas por muito pouco. Entretanto, para o segundo conjunto de instâncias obtivemos um resultado completamente inesperado: o algoritmo dos caminhos de maior aumento obteve desempenho bem melhor.

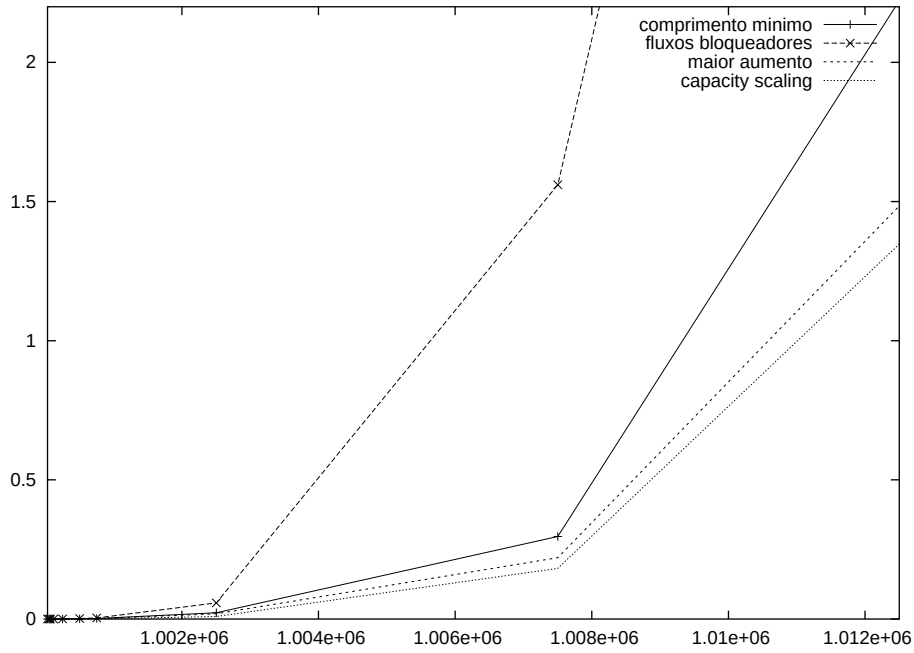
Esse resultado é ainda mais interessante quando observamos que parte da diferença entre as complexidades é justamente uma maior dependência de m pelo algoritmo dos caminhos de maior aumento, o que a princípio deveria piorar seu desempenho no segundo conjunto de instâncias. A causa do resultado inesperado foi provavelmente a estrutura específica das redes envolvidas, e bem demonstra a importância dos testes práticos na análise.

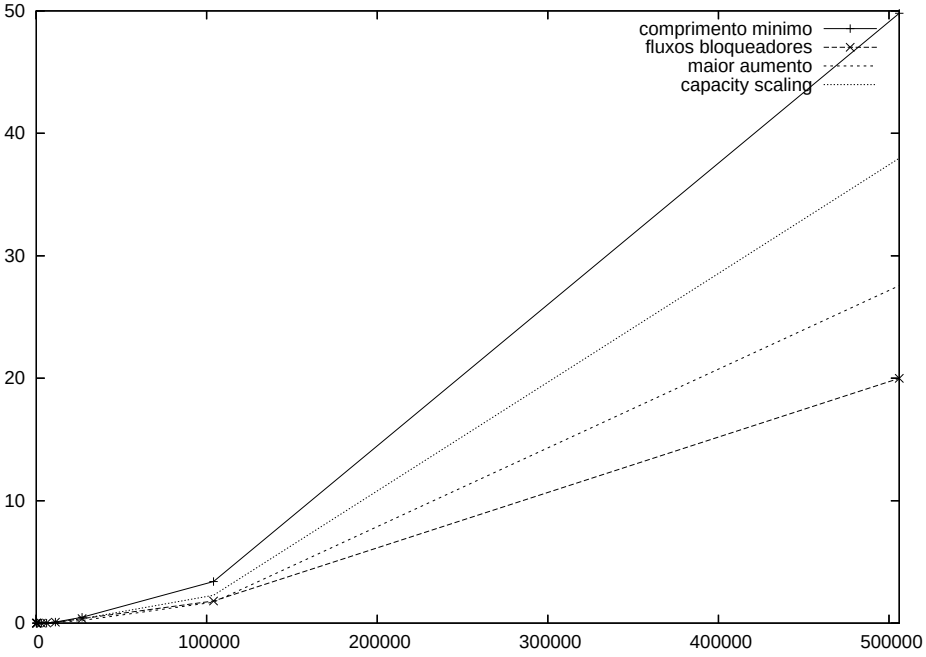
2.5 Análise do método dos caminhos de aumento

Recapitulando o que foi mencionado nas seções anteriores, as implementações do método dos caminhos de aumento apresentam as seguintes complexidades:

implementação	iterações	tempo
comprimento mínimo	$O(nm)$	$O(nm(n+m))$
fluxos bloqueadores	$O(n)$	$O(n^2m)$
maior aumento	$O(m \log mU)$	$O(m \log mU(n + m \log n))$
capacity scaling	$O(m \log U)$	$O(m \log U(n+m))$

Os gráficos abaixo mostram o consumo de tempo para os dois conjuntos de instâncias.





O algoritmo dos fluxos bloqueadores se revelou o mais eficiente para as redes densas e o pior para redes esparsas. Dentre os três algoritmos restantes, o algoritmo dos caminhos de aumento de comprimento mínimo obteve o pior desempenho. A causa disso foi provavelmente o fato de que o valor de U é praticamente constante em ambos os conjuntos de instâncias.

Isso acabou favorecendo o algoritmo dos caminhos de maior aumento e o algoritmo capacity scaling porque o crescimento de n e m é menor na complexidade destes.

Um resultado muito interessante que pode ser inferido destes experimentos é a confirmação prática de que existem situações em que os algoritmos fracamente polinomiais obtém melhor desempenho que os algoritmos fortemente polinomiais.

2.6 Fila de vértices ativos

O algoritmo da fila de vértices ativos executa $O(nm+n^3)$ iterações. Duas implementações diferentes foram feitas: uma inicializa os rótulos dos vértices da maneira normal (n) e a outra inicializa os rótulos de acordo com as distâncias entre os vértices e o sorvedouro (d). Ambas as implementações têm consumo de tempo $O(nm + n^3)$.

instância	n	m	U	iterações (n)	tempo (n)	iterações (d)	iterações (d)
ak1	14	19	1000000	63	0.000005	31	0.000004
ak2	22	31	1000000	159	0.000011	70	0.000006
ak3	30	43	1000000	284	0.000017	129	0.000008
ak4	50	73	1000000	927	0.000045	370	0.000015
ak5	102	151	1000000	3529	0.000154	1560	0.000048
ak6	202	301	1000000	13901	0.000601	6260	0.000175
ak7	302	451	1000000	31413	0.001324	14035	0.000395
ak8	1002	1501	1000000	329600	0.014950	156260	0.004389
ak9	3002	4501	1000000	2611392	0.136961	1406260	0.046488
ak10	5002	7501	1000000	7027587	0.516628	3906260	0.149258

instância	n	m	U	iterações (n)	tempo (n)	iterações (d)	iterações (d)
Bsp1	10	50	96	159	0000.000020	140	0000.000017
Bsp2	20	100	100	56	0000.000009	17	0000.000004
Bsp3	30	200	100	135	0000.000018	35	0000.000005
Bsp4	50	500	100	5389	0000.000804	5148	0000.000787
Bsp5	100	2000	1000	23491	0000.006528	22909	0000.006316
Bsp6	200	5000	1000	90384	0000.049883	89154	0000.050484
Bsp7	300	10000	1000	203650	0000.476450	202886	0000.462124
Bsp8	1000	25000	1000	2173179	0010.215220	2168071	0010.190467
Bsp9	3000	100000	1000	19398030	0181.432353	19382609	0180.098988
Bsp10	5000	500000	1000	54761096	1869.725449	54772210	1658.850363

Uma observação imediata ao se comparar as duas tabelas é a diferença brutal do consumo de tempo para os dois conjuntos de instâncias. Para a versão que inicializa os rótulos de acordo com as distâncias, o consumo de tempo chegou a ser aproximadamente dez mil vezes

maior no segundo conjunto. Esse resultado é ainda mais interessante diante do fato de que a diferença de desempenho não foi tão alta no que se refere a número de iterações: esse número foi aproximadamente apenas dezoito vezes maior no pior caso.

Uma boa conjectura para a razão dessa discrepância entre o número de iterações e o consumo de tempo é o tempo consumido por cada iteração. Sabemos que cada iteração do algoritmo consome tempo constante, mas os resultados demonstram que essa constante é alta o suficiente para provocar uma diferença significativa para instâncias grandes. Os testes experimentais deram, portanto, uma perspectiva mais realista da complexidade teórica.

Cabe observar também a diferença de desempenho entre as duas implementações do algoritmo. Para o primeiro conjunto de instâncias, a segunda versão obteve um desempenho aproximadamente três vezes melhor. Essa diferença é bem menor para as instâncias maiores do segundo conjunto, mas ainda assim demonstrou ser significativa.

2.7 Vértices ativos de maior rótulo

O algoritmo dos vértices de maior rótulo executa $O(nm + n^2\sqrt{m} + n^3/\sqrt{m})$ iterações e a implementação feita tem consumo de tempo $O(nm + n^2\sqrt{m} + n^3/\sqrt{m})$.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	50	0.000005
ak2	22	31	1000000	110	0.000008
ak3	30	43	1000000	194	0.000012
ak4	50	73	1000000	509	0.000024
ak5	102	151	1000000	2030	0.000086
ak6	202	301	1000000	7805	0.000305
ak7	302	451	1000000	17330	0.000657
ak8	1002	1501	1000000	189005	0.008398
ak9	3002	4501	1000000	1692005	0.076058
ak10	5002	7501	1000000	4695005	0.231620

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	175	0000.000022
Bsp2	20	100	100	44	0000.000007
Bsp3	30	200	100	166	0000.000020
Bsp4	50	500	100	5525	0000.000850
Bsp5	100	2000	1000	35045	0000.007182
Bsp6	200	5000	1000	116184	0000.044795
Bsp7	300	10000	1000	259424	0000.482500
Bsp8	1000	25000	1000	2733003	0007.586690
Bsp9	3000	100000	1000	23479197	0120.242250
Bsp10	5000	500000	1000	64019081	1485.265751

Como esperado, o algoritmo dos vértices de maior rótulo teve um comportamento parecido com o do algoritmo da fila de vértices ativos. O desempenho foi melhor do que o de ambas as versões da fila para o segundo conjunto de instâncias e melhor do que o da primeira versão para o primeiro conjunto. Isso demonstra que, apesar da maior dependência de m, o algoritmo dos vértices de maior rótulo funciona muito bem na prática para grafos densos.

Também é importante observar que inicializar os rótulos de acordo com as distâncias foi o suficiente para o algoritmo da fila de vértices ativos obter um desempenho superior no primeiro conjunto de instâncias. A diferença de implementação não alterou a complexidade teórica, mas provocou uma melhoria considerável no desempenho prático.

2.8 Excess scaling

O algoritmo excess scaling executa $O(nm + n^2 \log mU)$ iterações e a implementação feita tem consumo de tempo $O(nm + n^2 \log mU)$.

instância	n	m	U	iterações	tempo
ak1	14	19	1000000	53	0.000008
ak2	22	31	1000000	131	0.000014
ak3	30	43	1000000	272	0.000024
ak4	50	73	1000000	807	0.000049
ak5	102	151	1000000	3431	0.000172
ak6	202	301	1000000	12822	0.000586
ak7	302	451	1000000	29240	0.001292
ak8	1002	1501	1000000	290640	0.013809
ak9	3002	4501	1000000	2840196	0.145101
ak10	5002	7501	1000000	7586856	0.646149

instância	n	m	U	iterações	tempo
Bsp1	10	50	96	182	0000.000025
Bsp2	20	100	100	60	0000.000011
Bsp3	30	200	100	145	0000.000022
Bsp4	50	500	100	4882	0000.000830
Bsp5	100	2000	1000	19588	0000.006291
Bsp6	200	5000	1000	74226	0000.043363
Bsp7	300	10000	1000	175319	0000.479489
Bsp8	1000	25000	1000	1960037	0007.377310
Bsp9	3000	100000	1000	10369120	0115.854204
Bsp10	5000	500000	1000	33025544	1477.053552

O desempenho do algoritmo excess scaling foi praticamente igual ao do algoritmo dos vértices de maior rótulo para o segundo conjunto de instâncias. Entretanto, ele obteve o

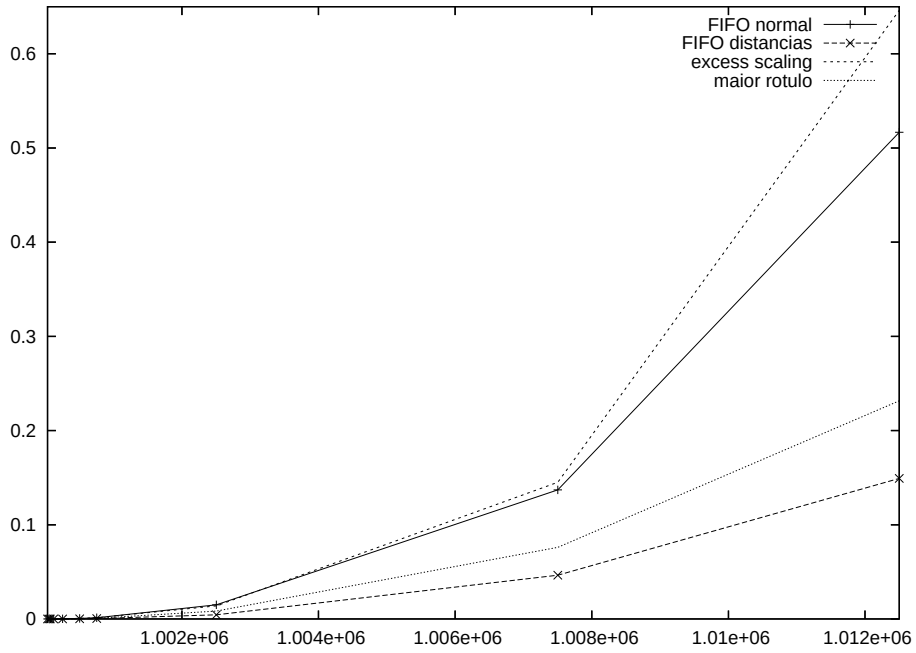
pior de todos os desempenhos para o primeiro conjunto. Uma conjectura razoável para esse fato é a dependência de U. O valor de U, apesar de constante, é mil vezes maior no primeiro conjunto de instâncias. Dependendo da estrutura da rede, isso pode fazer muita diferença.

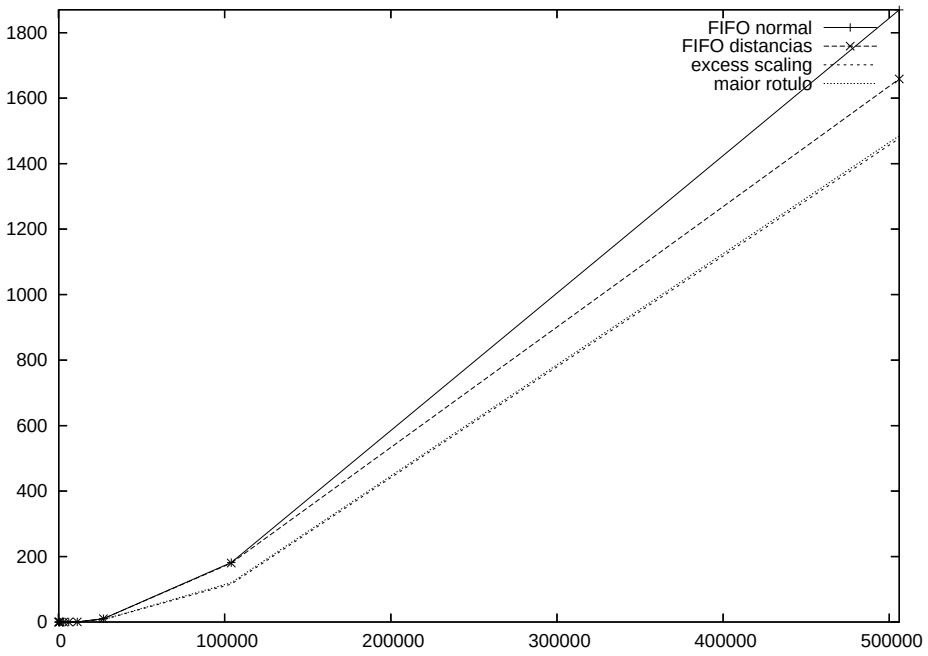
2.9 Análise do método do pré-fluxo

Recapitulando o que foi mencionado nas seções anteriores, as implementações do método do pré-fluxo apresentam as seguintes complexidades:

implementação	iterações	tempo
FIFO pré-fluxo	$O(nm + n^3)$	$O(nm + n^3)$
maior rótulo	$O(nm + n^2\sqrt{m} + n^3/\sqrt{m})$	$O(nm + n^2\sqrt{m} + n^3/\sqrt{m})$
excess scaling	$O(nm + n^2 \log mU)$	$O(nm + n^2 \log mU)$

Os gráficos abaixo mostram o consumo de tempo para os dois conjuntos de instâncias.



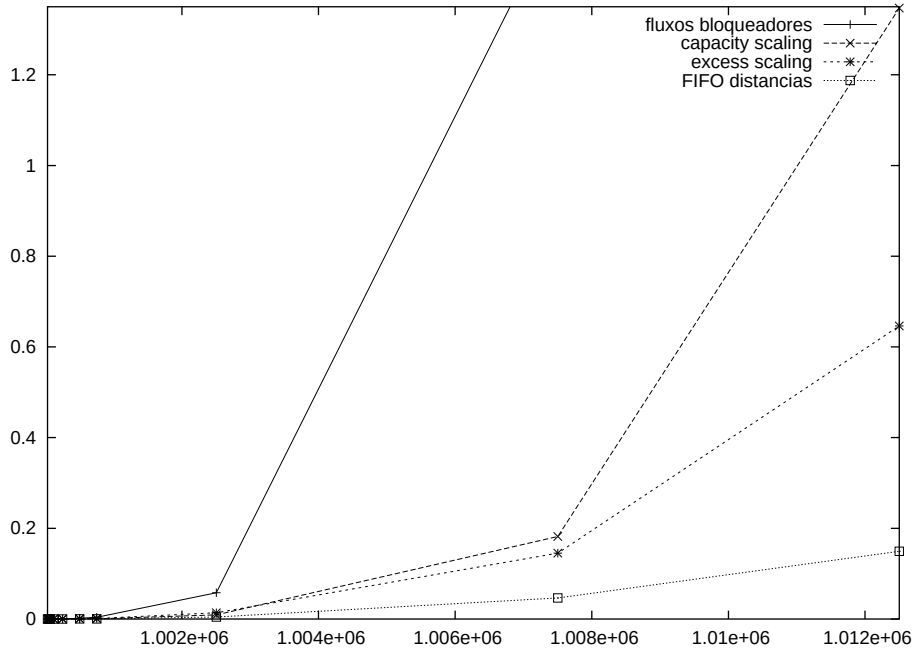


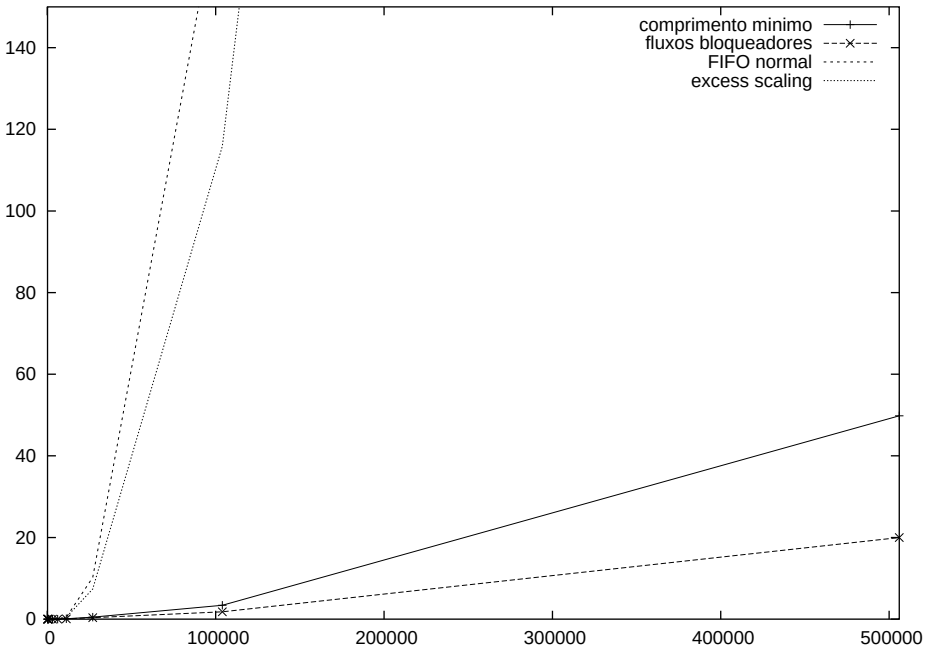
Uma observação inicial interessante é que a diferença entre as duas versões do algoritmo da fila de vértices ativos foi significativa. Ainda que para o segundo conjunto de instâncias ambas as versões obtiveram os piores desempenhos, para o primeiro conjunto a diferença de implementação fez com que uma versão fosse a melhor e a outra a segunda pior.

O algoritmo excess scaling obteve desempenhos bem diferentes nos dois conjuntos de instâncias: foi o melhor algoritmo para um e o pior para outro. Por outro lado, o algoritmo dos vértices de maior rótulo, embora não tenha sido o melhor em nenhum dos conjuntos, obteve desempenho razoável em ambos. Pode ser visto como uma vantagem o fato de o algoritmo dos vértices de maior rótulo ter essa maior estabilidade no desempenho, pois muitas vezes o formato da instância de um problema é imprevisível, e algoritmos que obtém bom desempenho apenas para tipos específicos de instâncias não são os mais indicados.

2.10 Estudo comparativo

Os gráficos abaixo comparam para cada conjunto de instâncias as implementações de cada método que obtiveram o melhor e o pior desempenho.





Podemos observar que, para o primeiro conjunto de instâncias, o método dos caminhos de aumento obteve desempenho pior que o método do pré-fluxo independentemente da implementação. Visto que esse conjunto de instâncias foi construído de forma a prejudicar o método dos caminhos de aumento, esse resultado era esperado. Realmente surpreendente

foi o resultado para o segundo conjunto de instâncias: não só ocorreu o completo oposto, o método dos caminhos de aumento teve vantagem, como a diferença foi enorme.

Essa análise comparativa demonstra muito bem como os desempenhos teórico e prático podem divergir muito dependendo da instância. Fatores como a própria estrutura da rede e a constante de tempo envolvida em algumas operações não podem ser desconsiderados.

3 Fluxos de custo mínimo

Os algoritmos que implementamos para resolver o problema do fluxo de custo mínimo foram testados com um conjunto de instâncias obtido da biblioteca de Schmidt [2].

3.1 Método do cancelamento de circuitos

O método do cancelamento de circuitos executa $O(mUC)$ iterações e a implementação feita tem consumo de tempo $O(nm^2UC)$. As tabelas abaixo mostram os resultados obtidos.

instância	n	m	U	C	iterações	tempo
bsp1	50	250	234	100	42	0000.007983
bsp2	100	1000	234	100	46	0000.072269
bsp3	400	5000	234	100	67	0021.030239
bsp4	600	20000	2000	100	59	0317.491182
bsp5	1000	40000	2000	100	59	1232.744774
bsp6	2000	100000	2000	100	54	5763.164254
bsp7	4000	200000	2000	100		
bsp8	8000	500000	2000	100		

Para as duas últimas instâncias, o tempo de execução foi tão alto que não pôde ser medido. Esse resultado não é inesperado. Uma vez que a complexidade do algoritmo depende quadraticamente de m e tal valor foi multiplicado por 2 da instância `bsp6` para a instância `bsp7`, então estima-se que o tempo de processamento do algoritmo, quando o mesmo recebe a instância `bsp7` como parâmetro, seja multiplicado por 4. Ou seja, estima-se que o algoritmo demore cerca de 6 horas para devolver uma resposta quando executado com a instância `bsp7`. Pelo mesmo tipo de argumento, é possível estimar que o tempo de processamento do algoritmo quando a instância `bsp8` é passada como parâmetro é de cerca de 40 horas.

Ao observarmos o número de iterações, entretanto, podemos perceber que os resultados merecem uma análise mais cuidadosa: esse número não cresce muito e até chega a diminuir conforme a instância aumenta. Isso indica que a pseudo-polinomialidade é menos prejudicial do que aparenta ser para as instâncias utilizadas. Com isso podemos concluir que o verdadeiro responsável pelo consumo de tempo foi o processo de encontrar circuitos negativos.

3.2 Método dos caminhos de viabilidade

O método dos caminhos de viabilidade executa $O(nmU)$ iterações e a implementação feita tem consumo de tempo $O(nmU(n + m \log n))$.

instância	n	m	U	C	iterações	tempo
bsp1	50	250	234	100	77	00.001381
bsp2	100	1000	234	100	61	00.003701
bsp3	400	5000	234	100	89	00.124253
bsp4	600	20000	2000	100	41	00.423681
bsp5	1000	40000	2000	100	39	00.911701
bsp6	2000	100000	2000	100	45	02.794692
bsp7	4000	200000	2000	100	47	06.616300
bsp8	8000	500000	2000	100	28	11.850933

Um aspecto muito interessante dos resultados obtidos é o fato de que o número de iterações segue um comportamento completamente instável em relação ao tamanho das instâncias. O menor número de iterações ocorreu para a maior das instâncias.

Para o consumo de tempo pode-se observar um crescimento mais evidente. Mas cabe observar que o método dos caminhos de viabilidade obteve um desempenho excelente, quinhentas vezes melhor que o do método do cancelamento de circuitos. Esse fato é certamente surpreendente diante do fato de que o algoritmo é pseudo-polinomial.

3.3 Path scaling

O algoritmo path scaling executa $O(nm + (n + m) \log U)$ iterações e a implementação feita tem consumo de tempo $O((nm + (n + m) \log U)(n + m \log n))$.

instância	n	m	U	C	iterações	tempo
bsp1	50	250	234	100	123	000.002799
bsp2	100	1000	234	100	228	000.016966
bsp3	400	5000	234	100	1010	001.462366
bsp4	600	20000	2000	100	174	002.010709
bsp5	1000	40000	2000	100	188	004.688266
bsp6	2000	100000	2000	100	387	025.342203
bsp7	4000	200000	2000	100	719	105.331842
bsp8	8000	500000	2000	100	1335	595.883327

Assim como a versão pseudo-polinomial do método dos caminhos de viabilidade, o algoritmo path scaling teve um comportamento instável no que se refere ao número de iterações. Mas para o path scaling um crescimento mais evidente pode ser observado.

Foi realmente inesperado o fato de que esse algoritmo, que possui complexidade polinomial, obteve desempenho muito pior na prática que a versão pseudo-polinomial, principalmente porque a diferença foi muito alta: a versão pseudo-polinomial chegou a ser cinqüenta vezes melhor. Uma possível razão para essa diferença é o fato de que o algoritmo path scaling considera um limitante superior para o aumento do fluxo através de um caminho de viabilidade, enquanto a outra versão aumenta o máximo possível.

3.4 Cost scaling

O algoritmo cost scaling executa $O(n^2 m \log nC)$ iterações e a implementação feita tem consumo de tempo $O(n^2 m \log nC)$.

instância	n	m	U	C	iterações	tempo
bsp1	50	250	234	100	10240	000.001136
bsp2	100	1000	234	100	21506	000.003983
bsp3	400	5000	234	100	144855	000.166666
bsp4	600	20000	2000	100	226997	001.944109
bsp5	1000	40000	2000	100	391414	004.681587
bsp6	2000	100000	2000	100	902338	015.513802
bsp7	4000	200000	2000	100	2024795	039.561442
bsp8	8000	500000	2000	100	4467808	124.902662

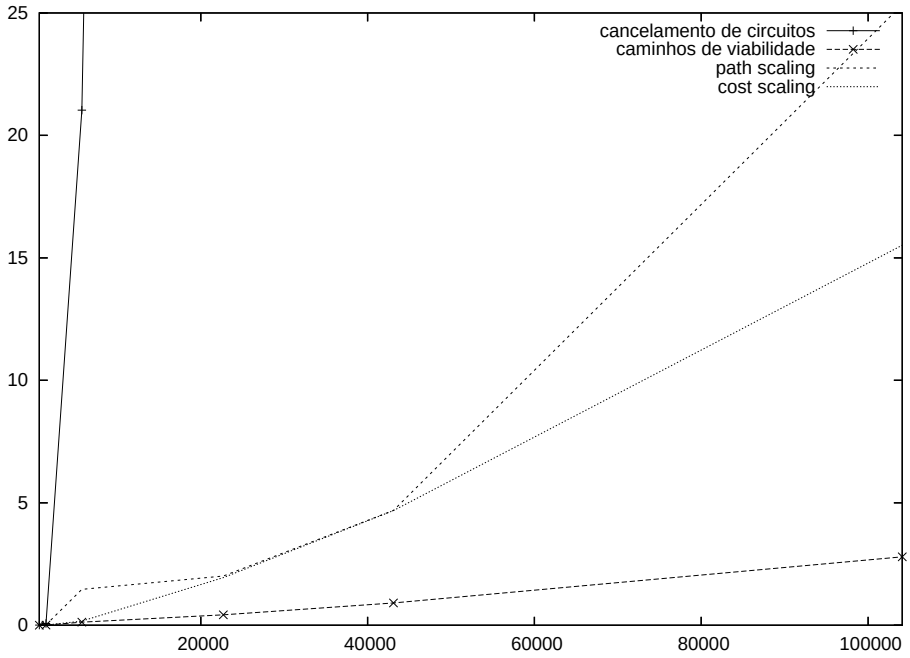
O desempenho do algoritmo cost scaling não foi tão bom quanto o do método dos caminhos de viabilidade, mas foi melhor que o dos outros dois algoritmos. A complexidade teórica do cost scaling é claramente superior à do método dos caminhos de viabilidade, principalmente se considerarmos que o valor de C nas instâncias é baixo e não cresce. Como este algoritmo é bastante similar ao método do pré-fluxo para o problema do fluxo máximo, podemos conjecturar de maneira análoga que um fator que piorou seu desempenho foi o tempo consumido por cada uma das operações de tempo constante do algoritmo.

3.5 Estudo comparativo

Recapitulando, os algoritmos implementados para resolver o problema do fluxo de custo mínimo apresentam as seguintes complexidades:

implementação	iterações	tempo
cancelamento de circuitos	$O(mUC)$	$O(nm^2UC)$
caminhos de viabilidade	$O(nmU)$	$O(nmU(n + m \log n))$
path scaling	$O(nm + (n + m) \log U)$	$O((nm + (n + m) \log U)(n + m \log n))$
cost scaling	$O(n^2 m \log nC)$	$O(n^2 m \log nC)$

O gráfico abaixo compara as quatro implementações:



A análise comparativa dos quatro algoritmos produziu muitos resultados interessantes. O algoritmo que obteve melhor desempenho foi um algoritmo cuja complexidade teórica é pseudo-polinomial, o que demonstrou que algoritmos polinomiais nem sempre são os melhores algoritmos na prática. Por outro lado, o que obteve pior desempenho também é pseudo-polinomial, o que bem demonstra a instabilidade desse tipo de algoritmo.

Referências

[1] A.V. Goldberg, *Network otimization library*, `http://www.avglab.com/andrew/soft.html`.

[2] B. Schmidt, *Test problems for max flow, min cost flow, mst, shortest path, and edge connectivity*, `http://www.math.uni-augsburg.de/~schmidtb/bschmidt/OR_Testdata/ORTestdata.html`.