Relatório Laboratório 2

Vinícius Luiz Ferreira Farias

December 2023

1 Implementação

Foi feito um algoritmo branch-and-cut para resolver instâncias do problema de Steiner utilizando o Gurobi 10.0. Para tanto, foram utilizadas callbacks de solução (MIPSOL) para descartar soluções fora do poliedro de Steiner(poliedro formado pelo todo das restrições do problema) adicionando restrições violadas. Além disso, foram utilizados user cuts e heurísticas primais para tornar o modelo mais eficiente nas callbacks de nodo (MIPNODE).

1.1 Lazy Constraints

As restrições lazy foram geradas através da candidata a solução inteira encontrada em MIPSOL. Mais especificamente, é criada uma árvore de gomory-hu através do algoritmo gomory_hu_tree() da biblioteca networkX, encontrados vértices de steiner que não estão conectados e gerada uma nova restrição para o modelo através do corte mínimo entre eles.

1.2 User Cuts

Para encontrar relaxações lineares mais precisas (maior valor), foram utilizados User Cuts nas callbacks MIPNODE. Eles foram gerados utilizando a solução fracionária encontrada na heurística gulosa para encontrar partições de steiner. Se violada a equação de partição gerada ($x(\Delta(P)) \geq k - 1$) ela é adicionada ao modelo.

1.3 Primal heuristic

Foi utilizada a heurística primal de árvore geradora máxima descrita nos slides de aula para encontrar soluções inteiras antecipadamente na solução de uma instância. A geração destas soluções foi utilizada nas chamadas MIPNODE.

2 Testes

Foi utilizado para os testes um computador com processador Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz 8GB RAM e sistema operacional Fedora Linux 37. Foi

utilizada a linguagem python (3.11.4) para escrever e interpretar o código do programa. Além disso, foi feito uso do gurobi 10.0.0.

Foram inicialmente feitos testes para os testes 1 a 10 do testset B da *SteinLib Testdata Library* sem User Cuts e Primal Heuristics e em seguida com, a fim de testar o ganho de eficiencia proveniente da adição desses protocolos. Foram obtidas as seguintes tabelas:

resultados testset B10

	|V|	|E|	|T|	Sol	Time	isOpt
b01	50	63	9	82.0	21.65	yes
b02	50	63	13	83.0	94.53	yes
b03	50	63	25	138.0	48.12	yes
b04	50	100	9	59.0	4.63	yes
b05	50	100	13	61.0	51.49	yes
b06	50	100	25	125.0	600.04	no
b07	75	94	13	111.0	26.72	yes
b08	75	94	19	104.0	120.99	yes
b09	75	94	38	220.0	64.01	yes
b10	75	150	13	86.0	39.57	yes

Figure 1: B10 com código inicial

resultados testset B10

	|V|	|E|	|T|	Sol	Time	isOpt
b01	50	63	9	82.0	1.31	yes
b02	50	63	13	83.0	2.27	yes
b03	50	63	25	138.0	0.72	yes
b04	50	100	9	59.0	8.65	yes
b05	50	100	13	61.0	21.09	yes
b06	50	100	25	122.0	15.35	yes
b07	75	94	13	111.0	5.95	yes
b08	75	94	19	104.0	3.82	yes
b09	75	94	38	220.0	1.44	yes
b10	75	150	13	86.0	18.59	yes

Figure 2: B10 com código otimizado

Sendo a coluna $|V|$ o número de vértices, $|E|$ o número de arestas, $|T|$ a quantidade de vértices de Steiner, Sol o valor da melhor solução encontrada, Time o tempo de execução e isOpt o indicador de se a solução é ótima.
Fica evidente através das tabelas 1 e 2 que os User Cuts e Primal Heuristics aumentam drasticamente a eficiência da otimização. Em seguida, foram feitos os testes para os testsets B e C da SteinLib, utilizando este modelo completo e obtendo os seguintes resultados:

resultados testset B

	|V|	|E|	|T|	Sol	Time	isOpt
b01	50	63	9	82.0	1.26	yes
b02	50	63	13	83.0	2.18	yes
b03	50	63	25	138.0	0.72	yes
b04	50	100	9	59.0	8.47	yes
b05	50	100	13	61.0	21.25	yes
b06	50	100	25	122.0	15.50	yes
b07	75	94	13	111.0	5.78	yes
b08	75	94	19	104.0	3.62	yes
b09	75	94	38	220.0	1.37	yes
b10	75	150	13	86.0	18.16	yes
b11	75	150	19	88.0	19.86	yes
b12	75	150	38	174.0	21.53	yes
b13	100	125	17	165.0	291.13	yes
b14	100	125	25	235.0	317.32	yes
b15	100	125	50	318.0	604.34	no
b16	100	200	17	127.0	443.51	yes
b17	100	200	25	131.0	496.34	yes
b18	100	200	50	219.0	601.21	no

Figure 3: B com código otimizado

resultados testset C

	|V|	|E|	|T|	Sol	Time	isOpt
c01	500	625	5	85.0	576.60	yes
c02	500	625	10	180.0	619.16	no
c03	500	625	83	821.0	602.85	no
c04	500	625	125	1153.0	609.99	no
c05	500	625	250	1655.0	601.52	no
c06	500	1000	5	61.0	604.34	no
c07	500	1000	10	148.0	600.25	no
c08	500	1000	83	618.0	601.15	no
c09	500	1000	125	849.0	617.67	no
c10	500	1000	250	1244.0	600.44	no
c11	500	2500	5	32.0	444.89	yes
c12	500	2500	10	62.0	600.15	no
c13	500	2500	83	628.0	616.48	no
c14	500	2500	125	542.0	602.44	no
c15	500	2500	250	849.0	742.57	no
c16	500	12500	5	13.0	610.40	no
c17	500	12500	10	19.0	601.33	no
c18	500	12500	83	381.0	629.33	no
c19	500	12500	125	403.0	602.47	no
c20	500	12500	250	735.0	620.58	no

Figure 4: C com código otimizado

Com as tabelas 3 e 4 é possível perceber que o modelo criado teve facilidade para o início do testSet B, enquanto que teve dificuldades para o final, não conseguindo finalizar os testes b15 e b18 a tempo. Teve também dificuldades no testset C, conseguindo finalizar apenas c01 e c11.
O principal motivo para isto esteve na duração das user Callbacks, consumindo em torno de 85% do tempo de execução dos testes. Possivelmente diminuiria bastante trocando a linguagem de programação por alguma mais eficiente que o python, como o C++. Seria válido também testar outras rotinas durante as callbacks, como a de cortes mínimos para encontrar partições de steiner e alguma heurística primal mais eficiente.