Combinatória Poliédrica – Laboratório 2
Branch-and-Cut

Guilherme Guidotti Brandt (235970)

1 Implementação

Organizei o código do programa da seguinte forma:

- `main.py` – Arquivo de entrada
- `steiner/` – Código relacionado ao problema
  - `basic.py` – Funções auxiliares de validação e custo de solução.
  - `solver.py` – Implementação da heurística e modelo.
  - `steinlib.py` – Leitura de arquivos no formato da SteinLib.

1.1 Inicialização do modelo

Inicializei o modelo com restrições de corte de Steiner para os conjuntos unitários contendo os terminais:

$$\sum_{e \in \delta(\{v\})} x_e \geq 1 \qquad \forall v \in Z$$

São conjuntos fáceis de gerar e que sempre definem corte de Steiner.

Para definir um cut-off inicial razoável, utilizei uma heurística simples, gerando uma árvore geradora mínima do grafo e removendo todas folhas que não são terminais da árvore até que todas as folhas sejam terminais. Isso dá uma árvore de Steiner minimal com custo razoavelmente baixo, e é barato (e a implementação é bastante simples, especiamente dado que a biblioteca NetworkX, utilizada para a árvore de Gomory-Hu, também tem uma função de árvore geradora mínima).

Uma melhoria possível seria construir a árvore geradora mínima sobre a união das árvores de caminhos mínimos entre os terminais, ao invés do grafo todo; não implementei essa variação da heurística para poder priorizar outras partes do problema, visto que a heurística simples serviu razoavelmente bem.

1.2 Lazy Constraints

Implementei a separação das restrições de corte com a árvore de Gomory-Hu, para as soluções fracionárias (quando `where` é `MIPNODE` e o estado do modelo é `OPTIMAL` no callback do Gurobi).

A princípio, usei a mesma rotina para a separação das restrições violadas nas soluções inteiras (`where` = `MIPSOL`), mas o programa ficou muito lento e tinha dificuldade em resolver mesmo instâncias pequenas. Substituí a rotina de separação para as soluções inteiras por uma que simplesmente olha para as componentes conexas do grafo correspondente e gera restrições de corte de Steiner para cada componente que contém algum terminal (mas que não contenha todos). Isso melhorou o desempenho consideravelmente.

Para economizar chamadas à função de Gomory-Hu, adicionei também um pool de restrições para a função de separação. Quando calcula a árvore de Gomory-Hu, a função adiciona todas as restrições que puder no pool; nas chamadas subsequentes, a função simplesmente retira uma restrição do pool e devolve, uma por uma, até que o pool fique vazio (e nesse caso a árvore de Gomory-Hu é computada novamente, repetindo o mesmo processo). Isso também diminuiu o gargalo do callback.

1.3 User cuts

Implementei a heurística gulosa para restrição de partição de Steiner violada descrita no livro, utilizando o callback do Gurobi (`where` = `MIPNODE` e estado `OPTIMAL`). Aproveitei a estrutura de Union-Find da biblioteca NetworkX para representar a partição (mantendo V_0 em separado) durante a heurística.

2 Resultados computacionais

Executei os testes em uma máquina com um processador Intel i7-8650U de 1.9Ghz e 8 núcleos, com 16GiB de RAM, com o sistema operacional Ubuntu 22.04. Utilizei a versão 3.10.12 do Python, a versão 10.0.3 do Gurobi, e a versão 3.2.1 da biblioteca NetworkX.

Limitei o tempo de execução definindo o parâmetero `TimeLimit` para 600 segundos.

A tabela 1 apresenta os resultados do programa para o conjunto de testes B, e a tabela 2 apresenta os resultados para o conjunto C. Onde o programa encontra uma solução ótima, o valor da melhor solução é apresentado em destaque.

As tabelas 3 e 4 aparesentam mais detalhes sobre a execução do algoritmo, indicando os valores obtidos para as heurísticas, o tempo total gasto pelo Gurobi na execução do callback, e o melhor limitante inferior obtido, quando o programa não encontrou a solução ótima ou não conseguiu provar a otimalidade dentro do tempo limite.

Todos os valores indicados nas tabelas foram extraídos a partir dos logs gerados pelo Gurobi.

Instância	$\|V\|$	$\|E\|$	$\|T\|$	Melhor solução	Tempo
b01	50	63	9	**82**	0,12s
b02	50	63	13	**83**	0,15s
b03	50	63	25	**138**	0,05s
b04	50	100	9	**59**	0,32s
b05	50	100	13	**61**	0,64s
b06	50	100	25	**122**	5,67s
b07	75	94	13	**111**	1,69s
b08	75	94	19	**104**	1,75s
b09	75	94	38	**220**	0,77s
b10	75	150	13	**86**	4,25s
b11	75	150	19	**88**	9,36s
b12	75	150	38	**174**	194,86s
b13	100	125	17	**165**	28,88s
b14	100	125	25	**235**	98,22s
b15	100	125	50	**318**	200,97s
b16	100	200	17	**127**	30,25s
b17	100	200	25	**131**	70,48s
b18	100	200	50	222	604,31s

Tabela 1: Resultados computacionais do grupo B de instâncias.

Instância	$\|V\|$	$\|E\|$	$\|T\|$	Valor	Tempo
c01	500	625	5	**85**	161.74s
c02	500	625	10	**144**	604.85s
c03	500	625	83	–	619.78s
c04	500	625	125	–	617.63s
c05	500	625	250	–	620.74s
c06	500	1000	5	**55**	148.85s
c07	500	1000	10	**102**	604.58s
c08	500	1000	83	–	601.40s
c09	500	1000	125	–	609.09s
c10	500	1000	250	–	623.26s
c11	500	2500	5	**32**	168.44s
c12	500	2500	10	**46**	603.92s
c13	500	2500	83	–	632.07s
c14	500	2500	125	–	612.91s
c15	500	2500	250	–	607.83s
c16	500	12500	5	**11**	404.00s
c17	500	12500	10	20	603.70s
c18	500	12500	83	–	653.64s
c19	500	12500	125	–	600.08s
c20	500	12500	250	–	673.05s

Tabela 2: Resultados computacionais do grupo C de instâncias.

Instância	Melhor	Heurística	Limitante	Tempo	Callback
b01	82	89	–	0,12s	0,09s
b02	83	96	–	0,15s	0,12s
b03	138	144	–	0,05s	0,03s
b04	59	68	–	0,32s	0,26s
b05	61	68	–	0,64s	0,54s
b06	122	130	–	5,67s	3,92s
b07	111	127	–	1,69s	1,54s
b08	104	111	–	1,75s	1,57s
b09	220	220	–	0,77s	0,61s
b10	86	95	–	4,25s	3,83s
b11	88	124	–	9,36s	6,85s
b12	174	189	–	194,86s	156,95s
b13	165	189	–	28,88s	24,38s
b14	235	255	–	98,22s	83,39s
b15	318	333	–	200,97s	165,68s
b16	127	166	–	30,25s	24,78s
b17	131	151	–	70,48s	59,73s
b18	222	230	216	604,31s	513.67s

Tabela 3: Resultados computacionais do grupo B de instâncias.

Instância	Valor	Heurística	Limitante	Tempo	Callback
c01	85	118	–	161.74s	157.30s
c02	144	194	–	604.85s	530.96s
c03	–	847	680	619.78s	597.25s
c04	–	1155	995	617.63s	598.18s
c05	–	1637	1544	620.74s	595.11s
c06	55	116	–	148.85s	143.74s
c07	102	186	100	604.58s	556.63s
c08	–	648	463	601.40s	561.39s
c09	–	835	635	609.09s	556.96s
c10	–	1163	1060	623.26s	516.96s
c11	32	47	–	168.44s	161.82s
c12	46	69	44	603.92s	539.79s
c13	–	321	230	632.07s	566.54s
c14	–	402	300	612.91s	532.20s
c15	–	616	535	607.83s	551.09s
c16	11	38	–	404.00s	393.35s
c17	20	39	17	603.70s	571.11s
c18	–	181	108	653.64s	496.52s
c19	–	232	145	600.08s	218.82s
c20	–	341	260	673.05s	221.02s

Tabela 4: Resultados computacionais do grupo C de instâncias.

O algoritmo se sai bem no conjunto de testes B, resolvendo a maior parte das instâncias em menos de três minutos. A única exceção é a instância `b18`, com 100 vértices e 200 arestas, na para a qual o programa atingiu o tempo limite.

No conjunto de testes C, o programa encontra soluções apenas para instâncias onde a árvore tem tamanho 5 ou 10, e consegue provar a otimalidade para quatro (de sete) delas. Das que o programa encontra uma solução mas não consegue provar sua otimalidade, apenas uma (a maior delas, `c17`) não atingiu de fato o valor ótimo.