INSTITUTO DE COMPUTAÇÃO - UNICAMP
MC658 - Análise de Algoritmos III

Docente: Prof. Cid Carvalho de Souza
PED: Natanael Ramos

4º Trabalho Prático

Grupo 03:

José Ribeiro - RA : 176665
Felipe Lopes De Mello - RA : 171119

IC/UNICAMP
2019

1 Breve descrição do problema

O problema NP-Completo tratado neste trabalho é o da árvore geradora de custo mínimo com restrição de grau nos vértices (DCMSTP do inglês). Para sua resolução, duas abordagens heurísticas são testadas: uma dada por relaxação lagrangiana e outra por algoritmos genéticos. Com a primeira, devido a realização da relaxação, um limitante dual é obtido, e por tabela também um primal é encontrado a cada iteração do método (onde uma heurística gulosa, com pitadas de busca local, utiliza informações da solução dual como forma de obtenção do primal). Já com a segunda, um método metaheurístico é testado onde apenas um limitante primal é encontrado.

2 Heurística Lagrangiana

2.1 Breve Descrição

Para a implementação da heurística lagrangiana, boa parte do algoritmo utilizado na relaxação lagrangiana, responsável por encontrar soluções duais, é reaproveitado. A citar, durante o cálculo da árvore geradora de custo mínimo com pesos nas arestas modificadas, aproveitamos para calcular a árvore geradora com restrições de grau. Para tanto, utilizamos a mesma ordenação imposta pelos pesos das arestas modificadas pela adição dos multiplicadores de lagrange. Assim, ao fim de cada iteração do método (Relaxação Lagrangiana), obtemos tanto um limitante dual quanto um primal, a um custo computacional praticamente irrisório em comparação a um código que computa apenas limitantes duais.

2.1.1 KruskalX

Para computarmos os limitantes duais, utilizamos simplesmente um algoritmo de kruskal (usado na construção de árvores geradoras mínimas) que utiliza como critério de ordenação o peso das arestas do grafo completo de entrada G = (V, E) somado com os multiplicadores de lagrange relacionados com a dada aresta. Já para o cálculo dos primais, precisamos de uma estratégia que agregue informações do dual para construção do primal. Tal estratégia escolhida é chamada *Kruskalx*, uma implementação direta do algoritmo descrito por [1]. Com ela, um esquema muito próximo ao kruskal é utilizado, com a única exceção de que ao encontrar uma aresta $e = (i, j) \in E$

passível de ser inserida na solução primal (ou seja e conecta duas árvores de G desconexas), duas propriedades são verificadas:

1. Caso a aresta $e = (i, j) \in E$ seja inserida na solução sendo construída, o grau de i e o grau de j não devem ter ultrapassado os limitantes de grau estabelecidos para os vértices i, j
2. Caso a aresta $e = (i, j) \in E$ seja inserida na solução primal sendo construída, a floresta obtida após a inserção não pode conter árvores saturadas, a menos que a floresta gerada seja uma árvore geradora para o grafo G.

Com a primeira propriedade, conseguimos garantir que a solução primal sempre respeitará as restrições de grau impostas pelo problema. Já com a segunda, conseguimos garantir que o algoritmo sempre produzirá uma árvore geradora. Esta última garantia decorre do fato de que, se existisse alguma árvore saturada, então o limite de grau de cada um de seus vértices teria sido atingida, e portanto nenhuma outra aresta poderia conectar esta árvore com alguma outra árvore ou vértice do grafo, impedindo assim com que uma árvore geradora fosse gerada. Assim, com a hipótese inicial de que o grafo G é completo, e com as duas propriedades acima sendo garantidas a cada inserção de aresta, temos que uma árvore com restrição nos vértices sempre é viável, e portanto kruskalx é um bom algoritmo para uso na heurística lagrangiana.

2.2 Redução do grafo original

Visando melhorar os resultados obtidos em instâncias maiores pela heurística, a mesma variante descrita por [1], onde um grafo restrito G' = (V, E') é utilizado para a geração dos limitantes duais e primais. Para a criação deste grafo, realizamos uma primeira execução do algoritmo kruskalx, assumindo que os multiplicadores de lagrange são nulos. Após o término da execução deste algoritmo, obtemos a n-th e última aresta utilizada pelo algoritmo (assumindo aqui a ordenação crescente para o peso das arestas). Assim, G' é dado por:

$$G' = (V, \{e_1, e_2, \ldots, e_n, e_{n+1}, ..., e_{k*n}\})$$

sendo aqui $e_1 \leq e_2 \leq e_3 \leq \cdots \leq e_{k*n}$, e k uma constante escolhida de acordo com o problema. Com esta estratégia, um tempo menor de execução necessário para a ordenação é utilizado, contudo a hipótese de grafo completo deixa de ser verdade, sendo necessário com que a constante k seja escolhida de forma sábia (ou seja, um trade-off entre tempo e garantia de viabilidade).

2.2.1 Improvement Procedure

Partindo também da sugestão posta por [1], após a obtenção de uma solução primal T na $k-th$ iteração, sendo T melhor do que todas as soluções primais anteriores, realizamos uma tentativa de melhora dessa mesma solução encontrada (ou seja, realizamos uma busca local sobre T). Sua execução consiste em ordenar decrescentemente as arestas da árvore T e em seguida remover cada aresta $e \in T$, de forma a desconectá-la, ou seja $T = T_1 \cup T_2 \cup \{e\}$ e $T_1 \cap T_2 = \emptyset$. Em seguida, após desconectá-la, é necessário verificar a existência de uma outra aresta f pertencente ao grafo (restrito) que possui peso menor que a removida e que reconecta as duas árvores geradas (respeitando as duas propriedades acima) de forma a produzir $T' = T_1 \cup T_2 \cup \{f\}$ com garantia de custo $c(T') \leq c(T)$. Tal aresta f pode possivelmente não existir. Por fim, a melhor árvore obtida neste procedimento (podendo ser a original T), deve passar pela função $testViability()$ para constatar que a mesma respeita os limites de grau nos vértices e que de fato é uma árvore geradora do grafo G. Só então, após ser validada como uma solução primal viável, é que o melhor primal será atualizado.

2.3 Análise de Complexidade Assintótica

Para a execução do kruskalx que já calcula os limitantes duais, primais e subgradiente, temos uma complexidade dada majoritariamente pela ordenação e iteração sobre as arestas. Para a ordenação, o custo é $O(|E|log(|E|)) = O(|V| * |V|log(|V|))$. Já para a iteração sobre as arestas, considerando apenas um algoritmo de kruskal simples teríamos complexidade $O(|E|log(|E|) = O(|V| * |V|log(|V|))$. Contudo, para que valha a segunda propriedade citada nos parágrafos anteriores, temos que adicionar uma verificação de grau sobre cada vértice para cada aresta passível de entrar na árvore primal sendo calculada. Assim, com esta nova iteração, no pior caso temos um custo total de $O(|V| * |E| * log(|E|)) = O(|V| * |V| * |V| * log(|V|))$. Este caso ocorre quando a restrição de grau dos $|V| - 1$ vértices é igual a 1 e somente um vértice possui restrição de grau igual a $|V| - 1$, somado ao fato de que todas as arestas do grafo possuem peso 1, com exceção das incidentes no vértice com limite de grau $|V| - 1$, que neste caso possui peso 2. Já para a execução do improve procedure, no total $|V| - 1$ arestas são substituídas e para cada aresta substituída, uma arestas dentre as $|E|$ arestas é encontrada. Neste caso temos uma complexidade igual a $\mathrm{O}(|V| * |E|) = \mathrm{O}(|V| * |V| * |V|)$. Assim sendo, temos que a complexidade é dominada pela execução do *kruskalx* e assim a heurística lagrangiana leve tempo $O(|V| * |V| * |V| * log(|V|))$.

2.4 Atualização dos multiplicadores de lagrange e Parâmetros Utilizados

Utilizamos dois critérios de parada para o método: um por tempo e outro por optimalidade. Além disso, ao atualizarmos os multiplicadores de lagrange na $i-th$ iteração, somamos o valor dos multiplicadores atuais com o valor do subgradiente calculado na respectiva iteração i e escalado por um fator *step*. Para o valor deste step, utilizamos a seguinte fórmula: $step = \alpha * ((1+\beta) * bestPrimal - dual_i)/||subgradiente_i||^2$, sendo $dual_i$ o limitante dual encontrado na $i-th$ iteração, α uma variável inicialmente setada como 2.0 e escalada por um fator de 1/1.5 a cada 500 iterações sem que haja atualização do melhor limitante primal. Para *beta*, utilizamos um valor igual a 0 por ser o que produziu os melhores limitantes duais. Por fim, para a constante k do grafo restrito, utilizamos o valor 4, pois foi o único que possibilitou produzir soluções viáveis para qualquer uma das instâncias.

3 Algoritmo Genético

3.1 Breve Descrição

A meta-heurística escolhida para resolver o problema DCMSTP foi o algoritmo genético. Esta meta-heurística consiste em começar com um conjunto de soluções viáveis inicial, chamado de população, e realizar operações que alteram a população com o objetivo de melhorar as soluções de cada geração. A geração da população inicial é descrita na subseção 3.2.2. As operações utilizadas são chamadas de *crossover* e mutação.

Cada 'indivíduo' da população é representada como um vetor de arestas e um inteiro que é o valor de sua solução (*fitness*). Uma aresta é representada pelo seus vértices incidentes e seu peso.

No *crossover*, dois indivíduos distintos são escolhidos aleatoriamente e aquele que tiver o melhor *fitness* (menor valor) é selecionado para ser o primeiro pai. Este processo é repetido para a determinação do segundo pai. Após esta seleção, o material genético (conjunto de arestas) de ambos é recombinado gerando um filho. A subseção 3.2.3 detalha como essa recombinação é realizada.

Após o *crossover*, uma parte dos indivíduos é selecionado para a mutação. A mutação é baseada em uma modificação aleatória de um alelo do indivíduo. Para o DCMSTP, isso significa que uma aresta será retirada e outra será adicionada, como será descrito em 3.2.4.

Finalmente, após a computação de todos os filhos, esta nova geração substituirá os pais que possuírem as piores soluções (os menos aptos). Este processo é repetido até que o tempo limite de execução seja atingido. As técnicas adotadas em todo processo seguiram sugestões da referência [2].

Os parâmetros utilizados no algoritmo projetado foram escolhidos após a realização de testes com as instâncias dadas visando a otimização da solução obtida. São eles:

- Tamanho da população = max{150000/n, 10}, onde n é o número de vértices. Valor é inversamente proporcional a n, pois quanto maior o número de vértices, maior será a complexidade do algoritmo (descrita na seção 3.3). Por outro lado, é desejado que a população inicial tenha o maior tamanho possível para haver uma maior variabilidade genética. Assim, por exemplo, se n = 100, temos uma população de tamanho 1500. O máximo com 10 é para garantia de funcionamento do algoritmo para qualquer instância, embora n >15000 resulta em uma complexidade intratável.
- Probabilidade de *crossover* = 80%, ou seja 80% de uma nova geração será composta pelos filhos da geração anterior. Valor é um *tradeoff* entre variabilidade genética e otimização da solução. Enquanto um alto valor aumenta o primeiro, diminui o segundo.
- Probabilidade de mutação = 0,2%. A mutação tem o objetivo de aumentar a variabilidade genética. Como essa função se mostrou computacionalmente custosa para o DCMSTP, um valor baixo foi optado.

3.2 Procedimentos Auxiliares

3.2.1 RandomKruskalX

O procedimento RandomKruskalX é uma adaptação do algoritmo KruskalX descrito em [1]. Em suma, KruskalX retorna uma árvore geradora mínima que respeita as restrições de grau. A adaptação de RandomKruskalX é que as arestas são escolhidas de modo aleatório, ou seja, apenas uma árvore geradora que respeita as restrições de grau é retornada. Este procedimento é uma importante sub-rotina da função que inicializa a população.

3.2.2 Inicialização da População

Uma vez projetado o procedimento RandomKruskalX se torna fácil construir uma população inicial. A árvore resultante de RandomKruskalX é

atribuída para cada indivíduo. Além disso, como existem muitas possibilidades de árvores geradoras é necessário que pelo menos 1 dos indivíduos possua uma boa solução inicial. Portanto, um desses indivíduos terá sua árvore construída a partir do KruskalX, ou seja, com as arestas de menor custo possível.

Ademais, um outro modo de inicialização da população foi implementado com o objetivo de melhorar as soluções dos indivíduos da população inicial. É a inicialização heurística descrita em [2] que consiste em uma estratégia gulosa com variabilidade genética. Para isso, o primeiro indivíduo é construído começando pelas arestas de menor peso. Os demais são construídos a partir de uma ordem de arestas igual a ordenada crescentemente com k permutações aleatórias entre arestas. Naturalmente, quanto maior o valor de k, mais aleatória será a solução. Ao aumentar o valor de k gradativamente para cada indivíduo uma população inicial com um bom conjunto de soluções e variabilidade é obtido. O melhor valor de k determinado foi:

$$k = \alpha * i * n/P$$

Onde α é uma constante que aumenta a aleatoriedade e cujo valor foi definido como 1.5, P é o tamanho da população, i é o i-ésimo indivíduo e n o número de vértices. Apesar de diversos testes com diferentes valores de α, a primeira inicialização descrita obteve melhores resultados na prática e, portanto, foi a utilizada nos testes finais.

3.2.3 Crossover

Dados dois pais, ou seja, duas árvores geradoras mínimas com restrição de grau, o objetivo do *crossover* é combinar ambas soluções, formando o filho. Para isso, a estratégia implementada seguiu 3 passos.

O primeiro passo consiste em encontrar todas as arestas em comum de ambos os pais. Estas arestas irão fazer parte da nova árvore gerada. Deste modo, assumindo que ambos os pais possuem boas soluções, as arestas presentes nas duas árvores provavelmente são muito boas e, por isso, são mantidas. Assim, as melhores características dos pais são passadas para a próxima geração.

Seguramente este passo não é o suficiente para a construção de uma árvore completa. Se o fosse, ambos os pais seriam iguais e a próxima geração seria igual. Portanto, o segundo passo é a adição de arestas aleatórias presentes em pelo menos um dos pais. Isto é baseado na mesma ideia anterior, a passagem de características da solução anterior para a nova.

Por último, como após o segundo passo ainda existe a possibilidade de que a árvore esteja incompleta, é necessário completar esta árvore com arestas selecionadas aleatoriamente (respeitando as restrição de grau e não formando ciclos).

3.2.4 Mutação

A mutação é um método que busca aumentar a variabilidade genética da população, alterando a solução de um indivíduo. A técnica produzida para este trabalho foi a remoção de uma aresta aleatória, formando duas árvores disjuntas no grafo. Depois, uma estratégia gulosa é a adotada: dentre todas as arestas que conectam as árvores e respeitam as restrições de grau, adicionar à arvore a aresta de menor peso.

3.3 Análise de Complexidade Assintótica

A estrutura principal do algoritmo consiste em um laço denotado por *Geração* no qual sua condição de parada é o tempo limite estabelecido pelo usuário. Dentro de *Geração* temos dois laços disjuntos. Um para a criação dos filhos e o outro para a substituição da pior parte da população original pelos filhos. Naturalmente, ambos os laços possuem a mesma complexidade que é igual a Θ(probabilidade de *crossover* * tamanho da população).

No primeiro destes laços, temos um *crossover* e, no pior caso, também uma mutação. O procedimento *crossover* descrito na subseção 3.2.3 primeiro varre as arestas de ambos os pais, tanto no primeiro passo como no segundo, e depois, no pior caso, todas as arestas. Como resultado, temos O(n + n + m), equivalente à $O(n^2)$, pois o grafo é completo. A mutação definida em 3.2.4 também verifica todas as arestas no pior caso, assim temos $O(n^2)$. Em suma, o primeiro laço possui complexidade assintótica $O(n^2) + O(n^2)$, resultando em $O(n^2)$.

No segundo laço temos apenas atribuições, o que é considerado como tempo constante e, portanto, assintoticamente menor que o primeiro laço. Deste modo, a complexidade total é dominada pela complexidade de *Geração* vezes a complexidade do primeiro laço ($O(n^2)$). É importante notar, contudo, que as constantes multiplicativas omitas são bem altas.

4 Otimizações Realizadas

4.1 Resumo das otimizações realizadas

Para esta etapa, uma planilha no google sheets foi criada com o objetivo de mantenimento do histórico das melhores soluções encontradas para cada versão do código. O link para planilha pode ser encontrado em link.

5 Resultados do Experimento

5.1 Dados da máquina

O experimento foi realizado em uma máquina com 144 GiB de RAM e Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz × 72, rodando Ubuntu 18.04.1. O programa foi compilado com g++ em sua versão *7.4.0*, estando ativadas as flags '-std=c++11' e '-O3'.

5.2 Análise dos resultados obtidos

Os resultados do algoritmo que implementa a heurística lagrangiana e o algoritmo genético são demonstrados na tabela 1. Em 18 das 40 instâncias testadas, a heurística lagrangiana foi capaz de determinar a solução ótima e, em todas elas, realizou isso em menos de 20 segundos. Esses resultados demonstram que o algoritmo implementado é eficiente, em termos de tempo computacional, e uma boa capacidade em determinar a solução ótima. Isto é bastante expressivo considerando o tamanho das instâncias nas quais os valores ótimos foram obtidos. Na maior delas (a instância tb3ct900_1), por exemplo, temos um grafo completo de 900 vértices cuja solução foi obtida em cerca de 12 segundos.

Analisando os resultados da meta-heurística, o algoritmo genético, como os critérios de parada possuem um limite por tempo de execução, já que a complexidade por iteração é de $O(n^2)$ como descrito na seção 3.3 e o tempo de execução limite é de 5 minutos. Este fator dificulta a análise do momento no qual o algoritmo encontrou a solução ótima. Além disso, como a meta-heurística não calcula limitantes duais e, portanto, não é possível determinar quais soluções são ótimas. Entretanto, utilizando os resultados da heurística lagrangiana, é possível inferir que 5 das 40 instâncias tiveram suas soluções ótimas determinadas.

Comparando ambas, é possível notar uma clara vantagem da heurística lagrangiana. Além de ser capaz de calcular um limitante dual, o que permite

deduzir se a solução é ótima, seu tempo de execução pode ser menor do que o tempo limite dado pelo usuário. Outro fator positivo para a heurística lagrangiana é a obtenção de soluções melhores para as instâncias do tipo tbXctX00 do que o algoritmo genético.

Por outro lado, a meta-heurística implementada obteve soluções melhores para as instâncias do tipo tbXchX00. Esse último conjunto de instâncias apresentam todas as restrições de grau iguais a 2, ou seja, o problema se reduz a um ciclo hamiltoniano de peso mínimo. A distinção das soluções encontradas pelos algoritmos para cada instância nos permite recomendar o uso do algoritmo genético quando a maior parte das restrições de grau é baixa (em torno de 2) e o uso da heurística lagrangiana para as demais instâncias.

Dois principais fatores limitaram a qualidade do algoritmo genético. O primeiro deles é o tempo necessário para que uma geração seja produzida. Como consequência, uma população menor do que o ideal teve que ser utilizada, pois caso contrário, um número muito pequeno de gerações seria produzido. Este fator reduziu a variabilidade genética. O segundo fator é a própria questão da aleatoriedade do código, fator que impossibilitou com que boas soluções pudessem ser obtidas em todas as execuções do código.

References

[1] R. Andrade, A. Lucena and N. Maculan. Using lagrangian dual information to generate degree constrained spanning trees. Discrete Applied Mathematics, 154(5):703–717, 2006.

[2] G. R. Raidl and B. A. Julstrom, "Edge sets: an effective evolutionary coding of spanning trees," in IEEE Transactions on Evolutionary Computation, vol. 7, no. 3, pp. 225-239, June 2003. doi: 10.1109/TEVC.2002.807275

Table 1: Resultados obtidos com as heurísticas

Instância	**Lim. Primal Lagrange**	**Lim. Dual Lagrange**	**Tempo Total Lagrange**	**Lim. Primal Metaheuristica**	**Tempo Total Metaheuristica**
tb1ct100_1	3789.2993	3790	0.02	3790	300
tb1ct100_2	3828.9932	3829	0.0318	3829	300
tb1ct100_3	3915.0139	3916	0.0784	3916	300
tb1ct200_1	5315.0107	5316	1.2058	5316	300
tb1ct200_2	5646.9331	5647	0.1915	5647	300
tb1ct200_3	5697.9517	5698	0.2974	5711	300
tb1ct300_1	6475.001	6477	300	6483	300
tb1ct300_2	6802.3926	6811	300	6814	300
tb1ct300_3	6429.0103	6430	0.8749	6431	300
tb1ct400_1	7413.0112	7414	2.1786	7423	300
tb1ct400_2	7775.4717	7783	300	7809	300
tb1ct400_3	7601.4629	7605	300	7627	300
tb2ct500_1	8271.0107	8272	14.4774	8276	300
tb2ct500_2	8391.4824	8395	300	8399	300
tb2ct500_3	8501.9482	8502	3.9	8532	300
tb2ct600_1	9036.0107	9037	2.885	9040	300
tb2ct700_1	9785.0107	9786	9.1529	9790	300
tb2ct800_1	10332.0127	10333	15.5205	10347	300
tb3ct900_1	10917.0117	10918	12.1115	10929	300
tb3ct1000_1	11407.001	11408	300	11419	300
tb3ct2000_1	15669.999	15672	300	15707	300
tb3ct2000_2	16238.5479	16263	300	16376	300
tb3ct2000_3	16667.7969	16677	300	16773	300
tb3ct2000_4	16367.8701	16376	300	16508	300
tb3ct2000_5	16519.2168	16538	300	16638	300
tb8ch100_0	4239.5703	4383	300	4346	300
tb8ch100_1	3871.9705	3884	300	3883	300
tb8ch100_2	4363.6875	4616	300	4508	300
tb8ch200_0	5551.542	5743	300	5751	300
tb8ch200_1	5601.3999	5935	300	5916	300
tb8ch200_2	5932.7979	6116	300	6208	300
tb8ch300_0	6706.0527	7235	300	7118	300
tb8ch300_1	6673.645	7258	300	7228	300
tb8ch300_2	6845.0303	7643	300	7516	300
tb8ch400_0	7715.2344	8403	300	8337	300
tb8ch400_1	7873.5825	8692	300	8587	300
tb8ch400_2	7526.7075	8239	300	8347	300
tb8ch500_0	8454.3184	9430	300	9322	300
tb8ch500_1	8468.791	9524	300	9305	300
tb8ch500_2	8335.5479	9537	300	9237	300