Modelagem e Otimização Algorítmica – Segunda Avaliação
Prof. Ademir Aparecido Constantino

Desenvolvimento de um Algoritmo Heurístico para Solução do Problema de Cobertura de Conjuntos

Lucas Wolschick[1]

[1]Departamento de Informática — Universidade Estadual de Maringá.
Maringá, PR — Brasil

ra123658@uem.br

Resumo. *O Problema de Cobertura de Conjuntos é um problema tradicional da área de otimização combinatória. Neste trabalho, um processo que utiliza algoritmos heurísticos para aproximar soluções a instâncias do problema é proposto, implementado e avaliado. O desenvolvimento e questões de projeto do sistema são relatados e expostos. Por meio do processo, foi possível obter soluções melhores ou iguais que as conhecidas para as instâncias-teste fornecidas em quatro de oito casos, e nos demais, a diferença ficou abaixo de 1% em dois casos e abaixo de 6% em quatro casos.*

1. Introdução

O Problema de Cobertura de Conjuntos (em inglês, *Set Cover Problem* — SCP) é um problema de otimização tradicional nas áreas de otimização combinatória e de pesquisa operacional. Nele, é fornecido uma lista de itens e uma lista de subconjuntos desses itens, cada subconjunto associado a um custo, e o objetivo é encontrar um conjunto desses subconjuntos contendo todos os itens de modo a minimizar o custo total (dado pela soma dos custos dos subconjuntos selecionados).

Várias aplicações práticas que surgem no dia a dia podem ser reduzidas a instâncias desse problema, tais como os problemas de roteirização de veículos, de escalonamento de tarefas, de atribuição de escalas, de manufatura de circuitos impressos, entre outros.

Esse problema é **NP-**Difícil. Ou seja, não se conhece nenhum algoritmo exato que resolve o problema de cobertura de conjuntos em tempo polinomial, e caso $\mathbf{P} \neq \mathbf{NP}$, esse algoritmo nem existirá. Entretanto, mesmo frente à classe de complexidade do problema, ele permanece extremamente aplicável para estas situações na vida real. Esses problemas nem sempre exigem a melhor solução possível, pois em vários casos uma solução próxima à ótima já é boa o suficiente.

Uma abordagem que permite aplicar o Problema de Cobertura de Conjuntos de forma prática para modelar problemas reais é pelo uso de algoritmos inexatos, como algoritmos heurísticos e algoritmos aproximativos, para obter soluções próximas aos ótimos de cada instância. Embora esses algoritmos não garantam a descoberta da solução ótima no espaço de soluções, eles geralmente fornecem soluções boas o suficiente para serem aplicadas.

Uma das possibilidades no desenvolvimento de sistemas que buscam soluções imprecisas é via algoritmos heurísticos. Uma heurística pode ser informalmente descrita como uma intuição ou informação que ajude a resolver algum problema; um algoritmo heurístico é um algoritmo que utiliza alguma(s) heurística(s) durante o processo de obtenção da solução. Muitas vezes, algoritmos heurísticos não possuem prova formal de possibilidade de obtenção da solução ótima (nem de que conseguem chegar a uma distância de *k* vezes do valor da solução ótima, para k > 1); sua discussão e análise é geralmente feita em torno de testes empíricos que põem o comportamento do algoritmo em prova para várias instâncias do problema.

Existem dois principais tipos de algoritmos heurísticos: algoritmos heurísticos construtivos, que buscam construir uma solução para o problema de acordo com algum critério guloso, e algoritmos heurísticos melhorativos, que tomam uma solução existente e viável para uma instância do problema e realizam uma busca na vizinhança daquela solução, buscando melhorá-la de acordo com algum critério guloso. A vizinhança de uma solução é definida como o conjunto de soluções de alguma forma adjacentes a uma solução original, no sentido de serem resultantes da realização de alguma alteração bem-definida sobre ela.

Este documento relata o desenvolvimento de um trabalho cujo objetivo é a exploração de algoritmos heurísticos visando implementar e validar um método para a obtenção de soluções para o Problema da Cobertura de Conjuntos, tendo em vista a aplicabilidade do problema em situações do mundo real. Foram implementadas duas propostas construtivas e uma proposta melhorativa, cujas trajetórias e relatos de desenvolvimento são descritos no restante do texto.

Através dos algoritmos desenvolvidos, foi possível obter quatro soluções melhores que as descritas nas soluções das instâncias apresentadas, e nas demais foi possível permanecer a uma margem de 6% das melhores soluções conhecidas. No final do artigo é realizada uma análise dos resultados encontrados.

O relatório está organizado em cinco seções. Na primeira seção, o contexto do trabalho é apresentado e introduzido. Na seção 2, o Problema da Cobertura de Conjuntos é descrito formalmente e uma instância-exemplo do problema é dada. A seção 3 apresenta os algoritmos heurísticos desenvolvidos e realiza uma discussão sobre o seu processo de elaboração e sobre o seu funcionamento. Os resultados da execução dos algoritmos são apresentados e discutidos na seção 4, e na seção 5 é feita uma retomada e conclusão do relatório.

2. Descrição do Problema

Em termos gerais, uma instância do Problema de Cobertura de Conjuntos é composta por um conjunto de elementos (estes chamados *linhas*) e um conjunto de subconjuntos dos elementos (esses chamados *colunas*), onde cada subconjunto fornecido é associado a um custo. Se uma linha *i* pertence a uma coluna *j*, dizemos que a coluna *j cobre* a linha *i*. Um conjunto de colunas que, quando unidas, formam o conjunto original de elementos, é chamado de *cobertura* das linhas. O objetivo do problema é encontrar uma cobertura para a instância que possua custo mínimo dentre todas as coberturas, onde o custo de uma cobertura é dado pela soma do custo de cada coluna incluída na cobertura.

Formalmente, o problema de cobertura de conjuntos pode ser definido como um problema de Programação Linear Binária, da seguinte forma:

$$min \quad \sum_{j=1}^{n} c_j x_j$$

$$suj.\ a \quad \sum_{j=1}^{n} a_{ij} x_j \geq 1, \text{ para } i=1, 2, \ldots, m$$

onde M = {1, 2, …, *m*} é o conjunto das linhas,

N = {1, 2, …, n} é o conjunto das colunas,

c_j é o custo da coluna *j*,

a_{ij} é 1 se a linha *i* é coberta pela coluna *j*, ou 0 caso contrário,

A = $(a_{ij})_{m \times n}$ é a matriz de cobertura das colunas,

x_j é 1 se a coluna *j* pertence à solução, ou 0 caso contrário.

Um exemplo de uma instância deste problema é ilustrado na Figura 1, no formato matricial. Na figura são mostrados a matriz de coberturas e os custos de cada coluna (alinhados às suas respectivas colunas na matriz de coberturas). Uma solução viável para esse problema é dada pela primeira, segunda e terceira colunas, que em conjunto cobrem todas as linhas do problema (a analogia de linhas e colunas surge desta matriz de coberturas).

Minimizar ($c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7$)

Sujeito à

$$\begin{vmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{vmatrix} \geq \begin{vmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{vmatrix}$$

Figura 1 – Exemplo de instância do problema de cobertura de conjuntos

No restante deste relatório, é relatado o processo do desenvolvimento de um algoritmo heurístico para a obtenção de soluções próximas ao ótimo para o Problema da Cobertura de Conjuntos.

3. Descrição do Algoritmo

3.1. Visão geral

O algoritmo implementado é baseado no trabalho de Jacobs e Brusco (1995), que deram uma descrição de um procedimento para realização de uma busca em vizinhança no espaço de soluções. O método implementado é dividido em duas etapas: uma etapa construtiva, para a qual são apresentadas duas propostas, e uma etapa melhorativa, que utiliza a busca em vizinhança para melhorar as soluções encontradas na etapa construtiva. A implementação do algoritmo foi realizada na linguagem de programação Python.

Para representar as instâncias do problema de cobertura de conjuntos, foi utilizado no trabalho uma estrutura análoga a uma lista de adjacências em grafos: para cada coluna do problema, armazenou-se uma lista de linhas cobertas por aquela coluna. Essa representação foi adotada em vista da esparsidade das instâncias-exemplo fornecidas na especificação deste trabalho (com densidade entre 5% e 10%). Para reduzir o tempo de processamento, nos algoritmos melhorativos, foi utilizada uma estrutura de dados auxiliar associando a cada linha as colunas que a cobrem. Para representar as soluções incumbentes, foram utilizadas listas ordenadas.

A estrutura do processamento realizado pelo programa desenvolvido é a seguinte: primeiro, o conteúdo dos arquivos é lido para a memória e em seguida convertido para uma representação interna mais imediata. Em seguida, uma solução é gerada utilizando-se um dos algoritmos construtivos descritos posteriormente nesta seção. A solução gerada é então iterativamente melhorada através do algoritmo melhorativo detalhado abaixo, e por fim a solução encontrada é convertida para uma representação externa e mostrada ao usuário.

3.2. A etapa construtiva

A primeira etapa de processamento do programa é a construção da primeira solução, realizada de modo iterativo e guloso: a cada iteração, uma coluna ainda não incluída na solução é escolhida para ser adicionada, de acordo com algum critério de prioridade. Novas iterações são realizadas enquanto houver linhas não cobertas pelas colunas adicionadas ao conjunto solução. Quando todas as linhas estiverem contempladas, é feita uma última passagem sobre as colunas adicionadas buscando remover colunas redundantes. O conjunto resultante é retornado e fornecido para a próxima etapa.

Para este trabalho, foram avaliadas duas abordagens seguindo esta descrição geral. A primeira abordagem seleciona, para cada coluna candidata e em cada passo da iteração, uma função f do conjunto F e a aplica com os valores c_j e k_j, onde c_j é o custo da coluna j e k_j é o número de linhas cobertas pela coluna j que ainda não foram cobertas por nenhuma coluna na solução. A coluna com menor valor $f(c_j, k_j)$ é selecionada para ser inserida. Colunas que não cobrem nenhuma linha das restantes ($k_j = 0$) não são consideradas para inserção. O conjunto F possui as seguintes funções heurísticas:

1. $f(c_j, k_j) = U[0, 1]$ (valor aleatório entre 0 e 1);
2. $f(c_j, k_j) = c_j/k_j$;
3. $f(c_j, k_j) = c_j/k_j^2$;
4. $f(c_j, k_j) = c_j^{0,5}/k_j^2$;
5. $f(c_j, k_j) = c_j/log_2(k_j + 1)$;
6. $f(c_j, k_j) = c_j/(k_j\ log_2(k_j + 1))$; e
7. $f(c_j, k_j) = c_j/(k_j\ ln(k_j + 1))$.

As funções e a seleção aleatória são baseadas no trabalho de Vasco e Wilson (1984), que sugerem realizar o mesmo. O objetivo é evitar encontrar uma solução que esteja presa num mínimo local ao usar a mesma heurística. As funções que usam logaritmos foram modificadas para evitar uma divisão por zero nos casos em que as colunas cobriam apenas uma linha.

A segunda abordagem construtiva usa apenas a função 1 em F, e atribui a cada coluna um número pseudo-aleatório entre zero e um, e as colunas são selecionadas para serem incluídas na solução do menor valor ao maior valor, até que todas as linhas estejam contempladas. A intenção é que este método possa fornecer outros pontos de partida para os algoritmos melhorativos de modo que eles não fiquem presos nos mesmos mínimos locais.

O código que implementa as duas abordagens construtivas está na função `constroi_solucao`, no arquivo `resolve.py`. A função começa construindo uma lista das linhas que ainda não foram cobertas, uma lista vazia que conterá as colunas da solução e uma estrutura auxiliar do tipo conjunto que armazena as colunas que ainda não foram inseridas. O laço principal itera enquanto houver linhas não cobertas, e seleciona uma lista de colunas candidatas para serem ranqueadas segundo o critério desejado (passado como o argumento *f*). Das colunas candidatas, a com menor valor fornecido por *f* é inserida na solução e marcada como utilizada, e a lista de linhas não cobertas é devidamente atualizada.

3.3. A etapa melhorativa

Explicação da etapa melhorativa. Comentar sobre estruturas de dados utilizadas e sobre desafios de implementação.

A segunda etapa do processamento é a etapa melhorativa. Nesta etapa, a solução gerada pela etapa construtiva é melhorada por meio de uma busca na vizinhança local da solução. A estratégia adotada nesta implementação é semelhante ao algoritmo de busca em vizinhança proposto por Jacobs e Brusco (1995), apresentado em sala de aula, que usa aleatoriedade na obtenção de soluções vizinhas.

O algoritmo que gera uma solução vizinha, implementado na função `jacobs_brusco` no arquivo `jacobs_brusco.py`, recebe uma solução já existente *S* e dois parâmetros, ρ_1 e ρ_2. Primeiro, um subconjunto de colunas é removido de *S* de forma aleatória. A proporção de colunas removidas de *S* é dada por ρ_1, com $\rho_1 = 1$ removendo todas as colunas e $\rho_1 = 0$ removendo nenhuma. Em seguida, é definido um conjunto *U* contendo as linhas que se tornaram descobertas com a remoção das colunas. Estruturas de dados auxiliares que mantém a quantidade de colunas que cobrem cada linha, e a quantidade de linhas em *U* cobertas por cada coluna candidata, são criadas para facilitar o processamento. A seguir, enquanto *U* não estiver vazio, uma coluna das colunas candidatas, definidas como aquelas cujo custo não supera o da coluna de maior custo na solução *S* original, multiplicado por ρ_2, é selecionada para ser inserida em *S*. A coluna é escolhida como a contendo o menor valor para uma dada heurística, que nessa implementação é a mesma função $f(c_j, k_j) = c_j/k_j$ usada na subseção 3.2, onde k_j é o número de linhas de *U* que a coluna *j* cobre. Ao fim do passo da iteração, as estruturas de dados auxiliares são atualizadas, e no término do laço é realizada outra passada para remover colunas redundantes da solução obtida.

Essa função é utilizada no algoritmo melhorativo para obter soluções vizinhas da incumbente (a melhor solução encontrada até um dado momento). Em cada passo, uma solução vizinha da incumbente é gerada. Se essa solução for melhor do que a incumbente, então ela se torna a incumbente (estratégia *first improvement*). Ao todo, são gerados 100 vizinhos para cada configuração de parâmetros ρ_1 e ρ_2, sobre os quais é feito uma busca em grade nos intervalos [0.1, 1.0] e [1.1, 3.0] respectivamente, com 10

pontos intermediários, sempre começando da solução construtiva originalmente encontrada para cada par (ρ_1, ρ_2). Ao todo, são geradas $10 \cdot 10 \cdot 100 = 10000$ soluções.

O algoritmo utiliza a estratégia *first improvement* com o critério de definição de vizinhança adotado no algoritmo gerador de soluções vizinhas. Foi considerada a adoção de uma estratégia *best improvement* para este método, mas determinou-se que o seu custo computacional seria excessivamente alto devido ao tamanho da vizinhança. Isso ocorre porque o espaço de vizinhança de uma solução removida de 20% de suas colunas ($\rho_1 = 0.2$), por exemplo, pode ser imenso: se a solução tem 20 colunas, existem $\binom{20}{16} = 4845$ soluções vizinhas possíveis; para 100 colunas, existem 5.36×10^{20} vizinhos. Uma alternativa seria remover uma ou duas colunas da solução ao invés de uma proporção fixa, mas mesmo essas duas propostas teriam custo $\binom{n}{1} = O(n)$ e $\binom{n}{2} = O(n^2)$, respectivamente, onde n é o número de colunas na solução, na estratégia *best improvement*.

Por fim, a solução incumbente é retornada após as iterações pré-definidas concluírem. O código que implementa essa estratégia se localiza na função `encontra_solucao` no arquivo `resolve.py`.

É importante notar que fora explorada uma implementação da estratégia *best improvement* no código, cujos resquícios se encontram no arquivo `swap_k_opt.py`. A função ali implementada recebe uma solução existente e um k, e define a vizinhança da solução como todas as soluções das quais k colunas foram removidas. Quando as colunas são removidas da solução, ela pode deixar de cobrir todas as linhas. Assim, para cada vizinho, são adicionadas colunas em ordem crescente da heurística $f(c_j, k_j) = c_j/k_j$. Das soluções geradas, a melhor delas é retornada. Entretanto, através deste método não foi possível obter soluções melhores do que as geradas pelo primeiro algoritmo construtivo (que usa a mesma heurística), então a estratégia não foi mais investigada posteriormente.

3.4. Comentários sobre o processo de desenvolvimento dos algoritmos

Durante o desenvolvimento dos algoritmos, foram realizados testes para verificar o desempenho e a qualidade do código, visando encontrar pontos de mudança para melhorar o custo das soluções geradas.

A implementação final itera sobre os parâmetros ρ_1 e ρ_2 dos maiores valores aos menores; observou-se que ao inverter e iterar dos menores para os maiores, algumas soluções melhores eram encontradas mais tarde, possivelmente devido às variações menores incorridas na solução incumbente para esses valores pequenos. Neste aspecto, foram anotados os melhores parâmetros para cada execução, e observaram-se dois fatos: primeiro, que para uma dada instância, geralmente havia um ou dois pares de valores que se destacaram e apareceram mais vezes; segundo, que para instâncias diferentes esses pares variavam. Assim, como não foi possível encontrar um par de parâmetros que funcionasse adequadamente para cada problema, foi mantida a busca em grade no código, e os melhores parâmetros foram omitidos do relatório por brevidade. Tentou-se reduzir o intervalo de valores e o número de passos em cada intervalo para reduzir a quantidade de soluções geradas, mas isso previsivelmente diminuiu a qualidade das soluções.

Em um dado momento, a busca em grade a partir da solução gerada na etapa construtiva foi alterada temporariamente para uma busca em grade a partir da solução incumbente. Notou-se que o custo médio dos resultados piorou e que as soluções ficaram presas em mínimos locais mais frequentemente. A mudança foi posteriormente revertida.

O algoritmo que gera uma solução vizinha foi otimizado diversas vezes ao longo de sua codificação. A descrição apresentada em sala de aula estava em um nível bem alto e sua tradução direta implicaria numa complexidade de tempo maior que quadrática. Assim, onde possível, foram adotadas listas ordenadas para armazenar colunas e linhas e foi adotada uma estrutura que conta quantas linhas cobrem uma dada coluna, para poupar uma varredura linear quando isso precisa ser consultado (na lógica da determinação do valor $\square_j$).

Uma estratégia *best improvement* foi tentada com um algoritmo análogo ao *k-opt* em Grafos, conforme descrito acima. Quando a função foi implementada, a estratégia construtiva era apenas utilizar a função $f(c_j, k_j) = c_j/k_j$ como critério guloso, e o algortimo "*k-opt*" também usava o mesmo critério. Como resultado, as soluções obtidas eram sempre as mesmas da etapa construtiva, então o algortimo não foi explorado posteriormente.

Durante o desenvolvimento, o algoritmo proposto conseguiu encontrar em momentos a melhor solução conhecida para o problema Teste_01 (com as colunas 58, 103, 156, 178, 179, 187, 194, 222 e 261 e custo 557.44) mas essa solução não emergiu na execução dos testes finais para as sementes testadas.

Os resultados obtidos, discutidos na seção seguinte, são os melhores dentre todas as tentativas realizadas de ajustar o algoritmo, que parece estar em um equilíbrio delicado: apenas uma das alterações sugeridas anteriormente já é suficiente para derrubar o desempenho do processo.

4. Resultados

Após a implementação, os algoritmos discutidos na seção 3 foram testados. Os resultados dos testes são apresentados nesta seção. Todos os testes foram realizados em um computador utilizando o sistema operacional Windows 10, com um processador Intel i5-11300H e 24 GB de memória RAM DDR4 3200 MHz. O código foi executado utilizando a versão 3.12.0 da linguagem de programação Python. Nesta seção são apresentados os tempos de execução por completude, mas eles não são medidas "puras": várias instâncias foram executadas de maneira concorrente, às vezes em números diferentes de execuções simultâneas, o que afetou o tempo de processamento. Ainda assim, os números podem ter utilidade como medidas "crus" do tamanho de cada instância.

Como os algoritmos testados utilizam aleatoriedade, para cada teste foi adotado um valor de semente (*seed*) para o gerador de números aleatórios fixa, e para cada experimento foram realizadas cinco execuções com as sementes 1, 2, 3, 4 e 5. As instâncias testadas foram as fornecidas em sala de aula: Teste_01.dat, Teste_02.dat, Teste_03.dat, Teste_04.dat, Wren_01.dat, Wren_02.dat, Wren_03.dat e Wren_04.dat.

Foram montadas oitenta execuções ao todo: para cada instância de entrada e para cada estratégia construtiva, foram realizadas cinco execuções, cada uma com uma das

sementes 1, 2, 3, 4 e 5. Para a análise individual de cada instância, foram tomados os valores médios do tempo de execução da etapa melhorativa e do melhor custo de cada semente, e os melhores custos.

A tabela 1 mostra os resultados gerais das instâncias para a estratégia construtiva usando as funções F. As colunas indicam tempo médio de execução, custo médio das soluções construtivas, melhor solução construtiva, custo médio das soluções melhoradas e melhor solução melhorada. As médias são em relação às execuções para cada uma das cinco sementes. A tabela 2 mostra os mesmos resultados gerais, mas desta vez para a estratégia construtiva aleatória. As colunas e as sementes usadas são as mesmas da tabela 1.

	Teste_01	Teste_02	Teste_03	Teste_04	Wren_01	Wren_02	Wren_03	Wren_04
Tempo médio	18.06	29.69	38.39	57.91	554.83	576.65	517.52	25344.55
Melhor custo construtivo	881.89	896.76	956.82	1885.15	10556.0	19229.0	20337.0	82504.0
Custo médio construtivo	1042.43	976.65	991.78	2092.85	10798.2	20485.8	20671.0	85375.2
Melhor custo melhorativo	557.86	543.84	515.3	1173.7	8142.0	13926.0	13764.0	61780.0
Custo médio melhorativo	557.86	545.04	515.76	1176.37	8154.0	13960.6	13854.2	61978.2

Tabela 1 – Resultados gerais para o algoritmo construtivo usando *F*.

	Teste_01	Teste_02	Teste_03	Teste_04	Wren_01	Wren_02	Wren_03	Wren_04
Tempo médio	17.95	26.24	32.16	43.70	471.72	461.91	411.00	25028.14
Melhor custo construtivo	940.22	873.32	979.79	2003.68	9638.0	19935.0	19827.0	82258.0
Custo médio construtivo	1016.65	949.57	1074.37	2174.83	10706.8	20925.6	21039.0	84112.8
Melhor custo melhorativo	557.86	537.89	515.3	1171.46	8142.0	13850.0	13835.0	61319.0
Custo médio melhorativo	557.86	543.55	516.67	1175.32	8171.6	13988.0	13920.8	61869.4

Tabela 2 – Resultados gerais para o algoritmo construtivo aleatório.

A tabela 3 mostra os resultados gerais das duas estratégias combinadas, tomando-se as melhores soluções para cada instância. O valor de referência na coluna "melhor solução" se refere ao melhor valor conhecido para cada instância segundo o material fornecido para o desenvolvimento do trabalho.

Considerando as tabelas, destacamos algumas observações. Primeiramente, a proposta deste trabalho conseguiu alcançar resultados muito próximos às melhores soluções conhecidas, ficando menos de 1% acima do melhor valor em seis dos oito casos; nos dois casos em que o algoritmo ficou acima de 1% do melhor valor, a diferença ainda não superou 6%. Além disso, dos seis casos que ficaram abaixo de 1%, um caso conseguiu chegar ao melhor valor conhecido e três casos ficaram abaixo do melhor valor conhecido.

Instância	Melhor solução	Solução construtiva	Gap	Solução melhorada	Gap
Teste_01	557.44	881.89	58.2%	557.86	0.08%
Teste_02	537.89	873.32	62.36%	537.89	0.00%
Teste_03	517.58	956.82	84.86%	515.3	**-0.44%**
Teste_04	1162.8	1885.15	62.12%	1171.46	0.74%
Wren_01	7856.0	9638.0	22.68%	8142.0	3.64%
Wren_02	13908.0	19229.0	38.26%	13850.0	**-0.42%**
Wren_03	13780.0	19827.0	43.88%	13764.0	**-0.12%**
Wren_04	58161.0	82258.0	41.43%	61319.0	5.43%

Tabela 3 – Resultados resumidos dos algoritmos implementados.

Segundo, a abordagem construtiva por valor aleatório foi competitiva com a abordagem da lista de funções, gerando melhores custos na etapa construtiva em sete das oito instâncias e com média superior à da lista em três das oito instâncias. Ademais, as soluções geradas na etapa construtiva com a estratégia aleatória serviram como base para melhoramentos que superaram as soluções melhoradas da outra abordagem em quatro dos oito casos. Sugere-se um estudo futuro mais aprofundado sobre as duas abordagens para atestar se a abordagem aleatória realmente tem alguma vantagem sobre a por lista de funções ou não.

Adicionalmente, percebe-se que os custos médios e os melhores custos ficaram relativamente próximos em todos os casos. O algoritmo construtivo aleatorizado apresentou uma variação maior entre o melhor valor e a média, o que pode sugerir que ele não seja tão consistente quanto o por lista de funções. Sobre o tempo de execução, considerando o viés discutido nesta seção, resguardamo-nos a apenas comentar o salto expressivo das demais instâncias para o Wren_04 (especialmente, ao comparar com o Wren_02, o segundo problema que levou mais tempo para ser processado, há uma razão de quarenta a sessenta vezes maior).

A lista 1 mostra as melhores soluções obtidas para cada uma das instâncias. Cada entrada da lista contém o nome da instância, o custo da solução e a lista de instâncias.

1. Teste_01: 557.86 – 10, 22, 58, 99, 103, 179, 190, 194, 222.

2. Teste_02: 537.89 – 7, 61, 62, 189, 251, 325, 386, 434, 471.
3. Teste_03: 515.3 – 103, 176, 265, 295, 341, 445, 588, 653, 655.
4. Teste_04: 1171.46 – 48, 176, 194, 197, 206, 211, 226, 244, 253, 287, 296, 316, 326, 359, 382, 443, 468, 473, 484.
5. Wren_01: 8142.0 – 1, 2, 75, 126, 146, 208, 240, 241, 249, 250, 303, 323, 324, 338, 345, 488.
6. Wren_02: 13850.0 – 564, 759, 959, 2420, 2763, 2790, 2993, 4537, 4979, 4990, 5050, 5071, 5074, 5089, 5134, 5147, 5328, 5363, 5389, 5399, 5409, 5426, 5427, 5448, 5449, 5459, 5460, 5477, 5481, 5482, 5484, 5503, 5522.
7. Wren_03: 13764.0 – 388, 646, 1150, 1161, 2227, 3166, 3296, 3371, 4220, 4452, 4529, 4551, 4629, 4634, 4654, 4686, 4695, 4790, 4839, 4862, 4881, 4882, 4900, 4901, 4919, 4920, 4936, 4937, 4964, 4966, 4969, 4984.
8. Wren_04: 61319.0 – 20, 51, 93, 131, 133, 139, 142, 162, 166, 212, 218, 228, 248, 277, 285, 302, 410, 442, 512, 533, 558, 599, 612, 647, 668, 685, 729, 759, 791, 826, 854, 888, 895, 913, 915, 934, 977, 1011, 1033, 1077, 1084, 1091, 1096, 1137, 1154, 1167, 1175, 1258, 1275, 1284, 1309, 1363, 1364, 1377, 1404, 1482, 1534, 1714, 1725, 1768, 1776, 1788, 1802, 1805, 1842, 1864, 1896, 1938, 2047, 2057, 2144, 2246, 2331, 2386, 2396, 2409, 2431, 2529, 2538, 2552, 2563, 2631, 2689, 2716, 2728, 2733, 2763, 2767, 2776, 2785, 2803, 2827, 2889, 2924, 2954, 3105, 3153, 3220, 3255, 3263, 3283, 3361, 3367, 3388, 3394, 3423, 3458, 3461, 3494, 3684, 3715, 3718, 3773, 3788, 3809, 3857, 3884, 3934, 4061, 4093, 4258, 4376, 4439, 4444, 4503.

Lista 1 – Melhores soluções encontradas nos experimentos executados.

As figuras 2 a 9 contém gráficos do custo da solução incumbente por tempo transcorrido (com o custo no eixo vertical e tempo no eixo horizontal). A progressão mostrada é da execução que obteve o melhor custo. Caso mais de uma execução teve o mesmo custo, uma delas foi aleatoriamente selecionada (a execução selecionada é indicada no rótulo).

É possível observar que, de imediato, o algoritmo melhorativo encontra soluções bem melhores do que as geradas pelos algoritmos construtivos. A maior parte das reduções nos custos ocorre próxima ao início da execução, enquanto o restante do tempo apenas alcança pequenas melhoras incrementais. Pela sequência de iteração dos parâmetros da função de geração de vizinhos, que começa com grandes variações e termina com pequenas variações, pode-se notar que grandes variações são ideais para obter avanços grandes, mas elas rapidamente ficam presas em mínimos locais; enquanto pequenas variações são boas para obter avanços incrementais. Nos testes Wren_01 a Wren_04, essa queda inicial é menos pronunciada e ocorrem reduções maior no meio do processamento.

Outra observação é que todas as instâncias exceto a Teste_01 se beneficiaram do tempo completo de processamento, e que cortar o tempo máximo de processamento em mais que 20% privaria o sistema das melhores soluções encontradas na execução.

Por fim, nota-se que as instâncias mais complexas possuem bem mais “patamares” de custo do que as instâncias mais simples. Isso pode significar que melhorias que “mudam de patamar” são mais difíceis de serem encontradas nesses problemas.

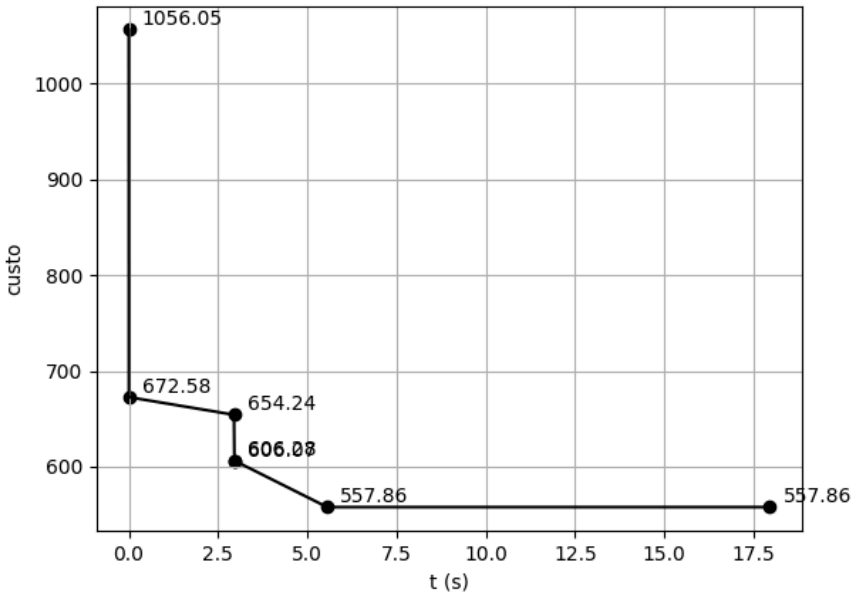

Figura 2 – Progressão do custo da instância Teste_01 (aleatório, semente=3).

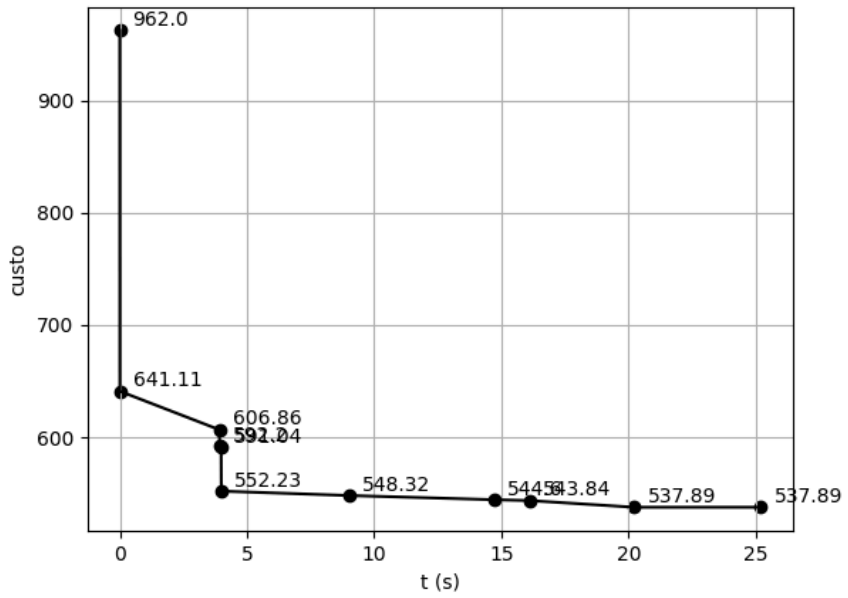

Figura 3 – Progressão do custo da instância Teste_02 (aleatório, semente=3).

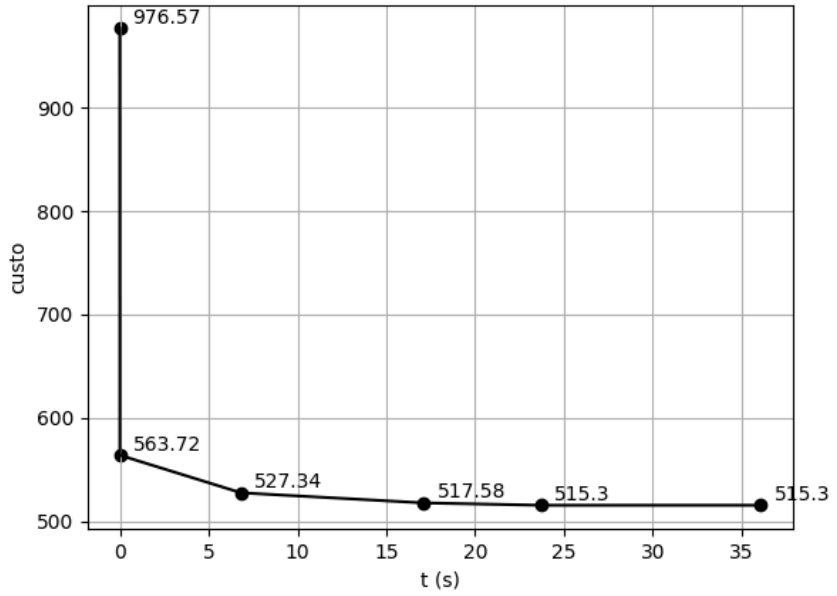

Figura 4 – Progressão do custo da instância Teste_03 (F, semente=5).

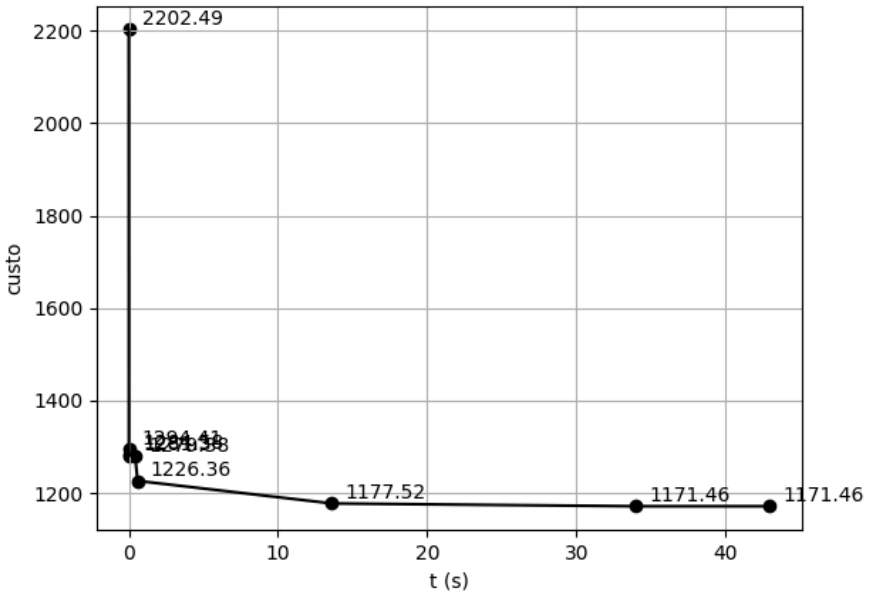

Figura 5 – Progressão do custo da instância Teste_04 (aleatório, semente=4).

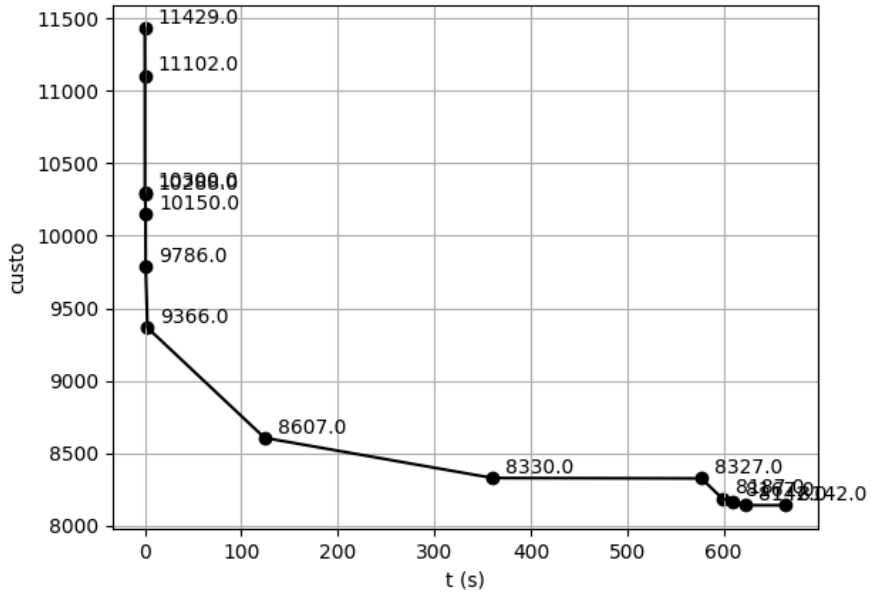

Figura 6 – Progressão do custo da instância Wren_01 (F, semente=1).

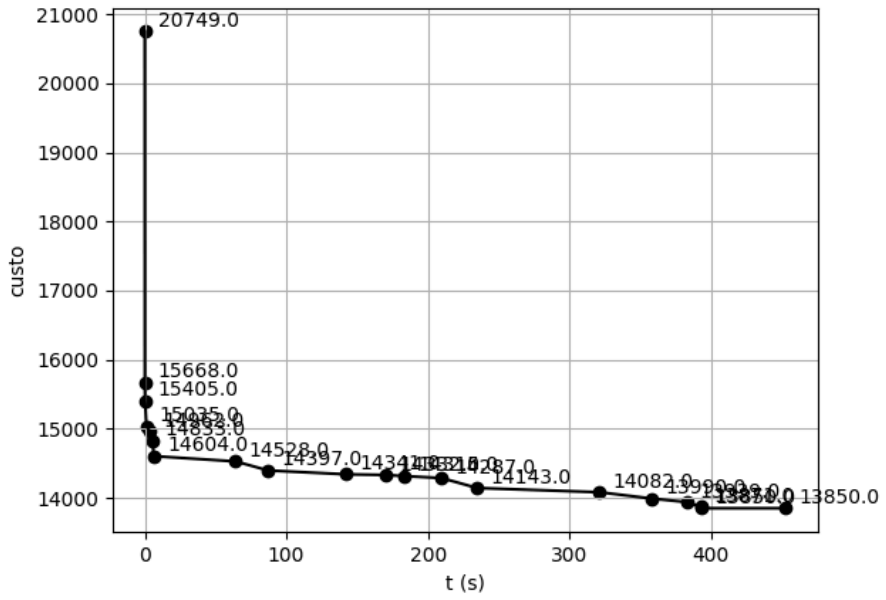

Figura 7 – Progressão do custo da instância Wren_02 (aleatório, semente=2).

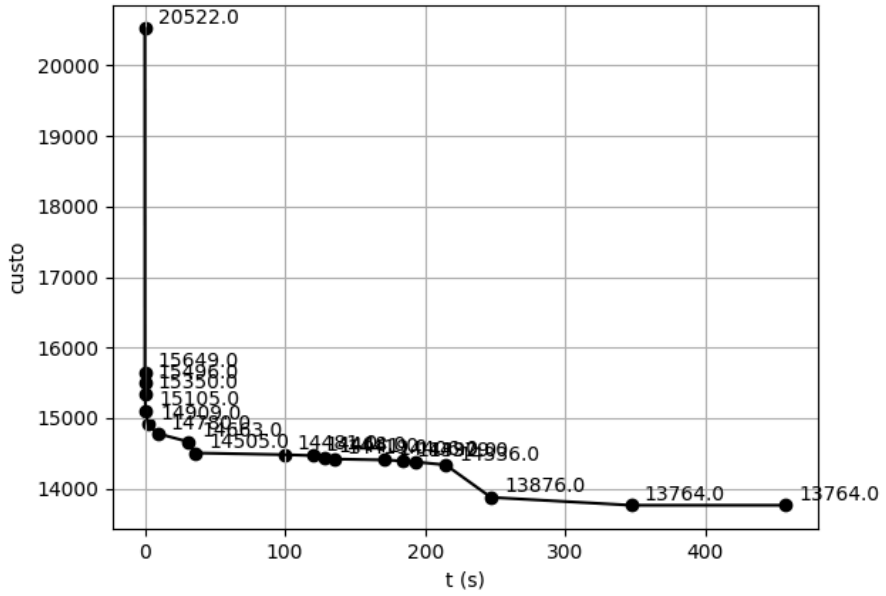

Figura 8 – Progressão do custo da instância Wren_03 (F, semente=3).

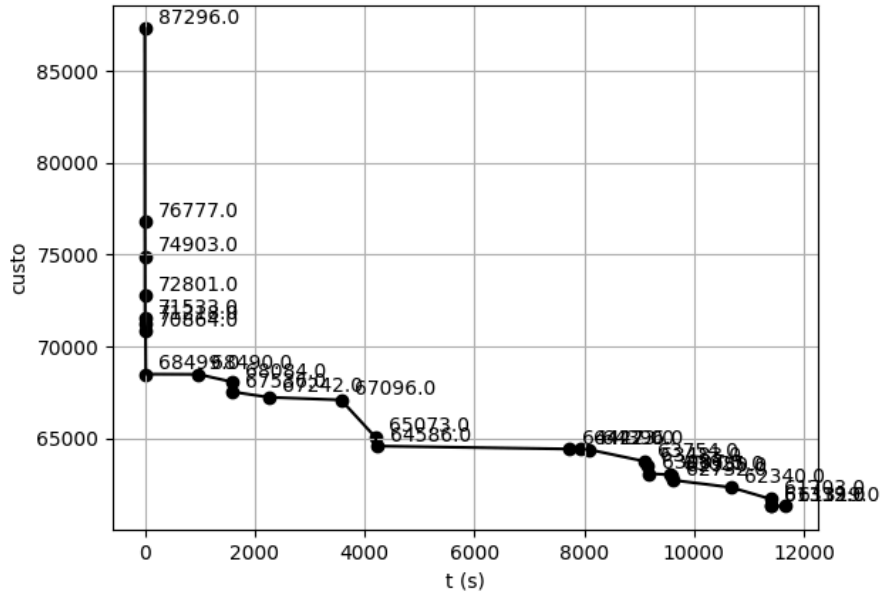

Figura 9 – Progressão do custo da instância Wren_04 (aleatório, semente=5).

De modo geral, os algoritmos propostos foram capazes de gerar soluções com custos adequados e próximos aos melhores custos conhecidos, em tempo de execução razoável para instâncias de pequeno e médio porte, usando uma abordagem simples, sem técnicas como meta-heurísticas.

5. Conclusões

O Problema de Cobertura de Conjuntos, ou *Set-Cover Problem* (SCP), é um problema clássico da área de otimização combinatória com várias aplicações em pesquisa operacional. Neste trabalho, foi apresentada uma proposta para um sistema que busca soluções otimizadas para instâncias do SCP usando algoritmos heurísticos construtivos e melhorativos.

Através da solução proposta, o sistema alcançou soluções dentro de 1% do melhor custo conhecido em seis das oito instâncias fornecidas e em quatro casos produziu soluções melhores ou iguais às melhores conhecidas. Foi proposto um algoritmo construtivo aleatório que demonstrou ser competitivo com técnicas mais tradicionais propostas na literatura, que em alguns casos levou a soluções melhores na etapa melhorativa. Além disso, foram expostas questões em aberto e levantados pontos que podem ser discutidos em trabalhos futuros.

Por meio do desenvolvimento do trabalho e do código, foi possível ver em prática os conceitos vistos em sala de aula sobre otimização, heurísticas e algoritmos heurísticos e estratégias de implementação, enriquecendo o conhecimento do autor deste trabalho.

Referências

CONSTANTINO, Ademir A. Notas de aula de heurísticas, meta-heurísticas, problema de cobertura de conjuntos. 2023.

JACOBS, L. W. and BRUSCO, M. J. A Local-Search Heuristic for Large Set-Covering Problems. Naval Research Logistic. Vol. 42, pp. 1129-1140, 1995.

VASKO, F. J. and WILSON, G. R. An Efficient Heuristic for Large Set Covering Problems. Naval Research Logistic Quarterly, V. 31, pp. 163-171, 1984.