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

Desenvolvimento de um Algoritmo Genético 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. *RESUMO*

1. Introdução

O Problema de Cobertura de Conjuntos (Set Cover Problem — SCP) é um problema clássico na área de otimização combinatória. Nele, é dado uma lista de elementos e uma lista de subconjuntos destes elementos, cada um com um custo, e o objetivo é encontrar um conjunto destes subconjuntos de modo que a união destes contenha todos os elementos de modo que a soma dos custos seja mínima.

O SCP é um problema NP-Difícil, e não se conhece nenhum algoritmo que solucione de forma exata todas as instâncias do SCP em tempo polinomial. Entretanto, mesmo frente à complexidade do problema, o SCP pode ser aplicado na modelagem de diversas situações do dia-a-dia que requerem otimização, como na roteirização de veículos, no escalonamento de tarefas, na manufatura de circuitos impressos, entre outros. Entretanto, para esses problemas, muitas vezes soluções próximas à ótima são suficientes, então uma opção para permitir o uso do SCP nessas situações é o uso de algoritmos que consigam produzir soluções aproximadas, embora não-garantidamente ótimas, para cada instância.

Uma classe de algoritmos inexatos que pode ser usada na resolução de problemas de otimização é a dos Algoritmos Heurísticos. Uma heurística é alguma informação ou intuição que ajuda a respeito de um problema; um algoritmo heurístico é um algoritmo que utiliza alguma heurística para resolver o problema. O funcionamento de um algoritmo heurístico geralmente se resume em duas etapas: a geração de uma solução inicial e uma posterior busca local sobre a vizinhança da solução gerada, buscando levá-la a um mínimo local. Uma vizinhança de uma solução é o conjunto das soluções que podem ser obtidas a partir da original através de perturbações bem definidas realizadas sobre os seus componentes.

Através do uso de algoritmos heurísticos, num trabalho anterior desta disciplina, das oito instâncias fornecidas para teste, foi possível obter quatro soluções melhores que as fornecidas, e nas demais foi possível atingir uma margem de erro de 6% em relação ao custo mínimo conhecido.

Entretanto, algoritmos heurísticos podem não ser bons o suficiente para determinados tipos de problemas; a sua simplicidade e a tendência de se prenderem em

soluções localizadas em mínimos locais os tornam inadequados para serem usados em determinadas instâncias do SCP, por exemplo. Tentativas de adequá-los à esses problemas, seja pela introdução de aleatoriedade ou pela expansão da vizinhança, podem prejudicar o desempenho e previsibilidade do algoritmo resultante. Às vezes, é difícil até pensar em uma heurística que pode ser utilizada para resolver um problema.

Algoritmos meta-heurísticos, ou meta-heurísticas, são propostas de modelos que podem ser usados para auxiliar na construção de algoritmos heurísticos. De modo geral, uma meta-heurística descreve, em alto nível, um processo genérico em termos de uma série de operações cujas definições ficam em aberto. No momento da implementação, as operações devem ser definidas considerando o problema a ser resolvido. Algoritmos meta-heurísticos tendem a ser inspirados em processos naturais, tais como o comportamento de uma colônia de formigas, a selação natural ou a cristalização de um composto químico.

Algoritmos genéticos são uma meta-heurística inspirada na seleção natural. Nela, um conjunto de soluções, nomeado população de indivíduos (ou cromossomos), é iterativamente alterada de acordo com uma sequência de etapas até que um critério de parada pré-estabelecido seja atingido. Primeiro, os indivíduos são avaliados de acordo com algum critério; depois, seguindo essa avaliação, os melhores indivíduos são selecionados para gerar descendentes. Esses descendentes são construídos a partir do cruzamento das características dos indivíduos genitores, e passam por mutações, que alteram as características resultantes de modo aleatório. Por fim, os indivíduos gerados e a população inicial são combinados segundo algum critério na etapa de atualização, e produz-se um novo conjunto de indivíduos, denominado geração, que será usado como população inicial na próxima iteração. As etapas de avaliação, seleção, cruzamento, mutação e atualização são também chamadas de operadores. O comportamento esperado é que ao longo das gerações, as melhores características sejam selecionadas e preservadas e eventualmente um conjunto de soluções boas possa ser gerado.

Este relatório documenta a implementação e avaliação de um algoritmo genético que resolve o Problema de Cobertura de Conjuntos. Além das etapas tradicionais de um algoritmo genético, é realizada a inclusão de uma etapa de melhoramento por busca local após a mutação dos novos indivíduos, de modo a acelerar a convergência do processo. Foram implementados vários operadores para cada etapa e foram testadas combinações de diferentes operadores e parâmetros no estudo da efetividade do algoritmo.

Através dos testes, a melhor combinação de operadores e argumentos obteve soluções melhores do que os melhores custos conhecidos em X casos, e as soluções restantes ficaram a uma margem de X%.

Este documento está organizado em cinco seções. Na primeira seção, o problema de cobertura de conjuntos, algoritmos heurísticos e meta-heurísticos e os algoritmos genéticos são introduzidos e o trabalho, apresentado. Na segunda seção, o problema de cobertura de conjuntos é apresentado de maneira formalizada. A terceira seção apresenta o algoritmo final e uma discussão dos seus detalhes de implementação. Os resultados obtidos pelo algoritmo proposto são detalhados e avaliados na seção quatro. Por fim, na seção cinco, é apresentada uma retomada e a conclusão do trabalho.

2. Descrição do Problema

Uma instância do Problema de Cobertura de Conjuntos é composta por um conjunto de elementos, as linhas, e um conjunto de subconjuntos dos elementos, as colunas, onde cada coluna é associada 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, em união, se igualam ao conjunto original de linhas, é denominado uma cobertura das linhas. O objetivo do problema é encontrar uma cobertura de custo mínimo para a instância sob estudo, 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:

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

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

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).

$$\text{Minimizar} \quad (\; c_1 + \; c_2 + \; c_3 + \; c_4 + \; c_5 + \; c_6 + \; c_7 \;)$$

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

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

3. Descrição do Algoritmo

3.1. Visão geral do algoritmo

O algoritmo implementado é um algoritmo genético como originalmente proposto na literatura, com o adicional de possuir uma etapa de melhoramento por busca local executada após a etapa de mutação. A implementação funciona da seguinte forma: após realizar a leitura do arquivo contendo uma entrada, os dados são extraídos e organizados em uma série de objetos que em conjunto formam uma instância.

A instância lida é utilizada na construção de um resolvedor, que é instanciado com uma série de parâmetros e operadores pré-especificados. O resolvedor é então executado — após gerar uma população inicial utilizando o operador de geração fornecido, o resolvedor entra em um laço que é executado até que o critério de parada seja atingido. Nesse laço, são realizadas as etapas de avaliação, seleção, cruzamento, mutação, busca local e atualização. O resolvedor pode ser configurado para que opcionalmente as etapas de mutação e busca local não sejam realizadas. Ao fim do processo de otimização, a melhor solução é retornada e exibida na tela ao usuário.

O programa é uma aplicação de linha de comando e foi implementado na linguagem Java, versão 17, e se aproveita do paralelismo disponível em máquinas *multicore* para acelerar o seu processamento, caso disponível, e de novos recursos adicionados à linguagem desde a versão 8, como *streams* e inferência de tipos por meio da palavra-chave *var*. Uma série de execuções com diferentes parâmetros podem ser realizadas por meio de invocações do programa com argumentos de linha de comando distintos. O restante desta seção entre em detalhes sobre as partes individuais do algoritmo implementado.

3.2. Leitura e representação das instâncias do problema

A leitura e representação das instâncias do problema é realizada pelo módulo *parser* contido no código-fonte do projeto. O arquivo da instância de entrada é passado como um argumento de linha de comando, lido para uma cadeia de caracteres e armazenado em memória, e as linhas são tratadas individualmente. As três primeiras linhas da entrada são utilizadas na extração do tamanho do problema e as demais são convertidas em instâncias do tipo Coluna.

A representação adotada para o problema é do tipo lista de listas, onde uma coluna possui um conjunto que contém todas as linhas que a coluna cobre. A estrutura de dados HashSet foi adotada para o problema pois as operações mais realizadas sobre as linhas cobertas por uma coluna foram a busca, iteração e diferença de conjuntos; todas operações realizadas de maneira eficiente nessa estrutura de dados. Optou-se por usar uma estrutura esparsa devido à baixa densidade das instâncias analisadas (em torno de 5% a 10%).

O conjunto de todas as colunas é armazenado numa instância do tipo Instancia, que contém, além delas, metadados sobre a instância, como o número total de colunas, o número total de linhas e a densidade. As colunas são armazenadas numa lista indexadas pelo seu número no arquivo de entrada. A classe Instancia fornece também uma representação invertida contendo as colunas que cobrem cada linha, que é computada dinamicamente apenas quando necessária, e posteriormente armazenada.

No interior dos objetos, os números das linhas e colunas são mantidos como aparecem no arquivo da instância do problema, diferentemente da implementação original do algoritmo heurístico em Python.

3.3. A etapa construtiva

Lida e armazenada a entrada em um formato adequado para o processamento, a primeira etapa do processo de otimização é a geração de uma população inicial para ser posteriormente evoluída. Nesta etapa é importante que a população inicial seja a mais diversificada possível, para que o algoritmo possa varrer um espaço de soluções maior, e evitar que as soluções fiquem presas em ótimos locais sem atingir o ótimo global da solução. Cada solução da população inicial é gerada de acordo com um processo pré-definido e o conjunto resultante de indivíduos é passado para o laço principal de evolução.

Neste trabalho, apenas um algoritmo de geração de população inicial foi testado, que é a estratégia de geração aleatória exposta no trabalho anterior. O método funciona da seguinte maneira: primeiro, são construídos um conjunto vazio de colunas que formará a solução e uma lista de linhas que ainda não foram cobertas (inicialmente, todas as linhas). Enquanto houver linhas que não foram cobertas, são elencadas colunas candidatas à serem inseridas na solução. São consideraras colunas candidatas aquelas que cobrem ao menos uma das linhas que ainda não foram cobertas. Das colunas candidatas, uma é escolhida aleatoriamente para ser inserida na solução, e uma nova iteração é realizada.

A intenção do método é gerar soluções diversas, não necessariamente ótimas em relação a um determinado critério guloso. Este método foi adotado pois constatou-se no último trabalho que, quando pareado com a busca local, as soluções obtidas foram muito próximas aos melhores resultados conhecidos.

Na implementação, utilizou-se otimizações para acelerar a escolha de colunas candidatas. O algoritmo mantém uma associação entre uma linha e o número de colunas que cobrem a dada linha, o número de linhas que ainda não foram cobertas e uma associação entre uma coluna e o número de linhas ainda descobertas que a dada coluna cobriria. Essas três estruturas são atualizadas a cada inserção.

Antes de retornar a solução construída com as colunas selecionadas, uma etapa de remoção de colunas redundantes é realizada numa tentativa de reduzir o custo da solução marginalmente. Esse método ordena as colunas da solução por custo de maneira decrescente e tenta remover as colunas nesta ordem, apenas removendo-as se a solução resultante permaneceria válida.

Nota-se que o processo gera apenas soluções viáveis. O algoritmo final não trabalha com soluções inviáveis em qualquer etapa do seu processamento. Isso simplifica a implemenação de várias operações e facilita a obtenção de uma solução ao fim da otimização.

A partir das colunas selecionadas, um objeto do tipo Solucao é construído e inserido na lista de indivíduos da população inicial.

3.4. Definição do critério de parada

Após a construção da população inicial, o algoritmo entra em um laço onde uma sucessão de gerações é criada. O laço para eventualmente, quando um critério de parada predeterminado é atingido.

Existem vários critérios de parada que podem ser escolhidos para um algoritmo genético; os mais comuns são atingir um número máximo de indivíduos gerados, atingir um número máximo de gerações criadas, ultrapassar um tempo limite de processamento ou parar quando melhorias não forem atingidas por um período longo de tempo.

Para este trabalho, adotou-se como critério de parada o número máximo de gerações criadas. Este número é passado como parâmetro na criação do objeto resolvedor do problema e pode ser configurado na linha de comando quando a aplicação é invocada.

3.5. Operadores de avaliação

O primeiro passo na criação de uma nova geração de indivíduos é a avaliação da população. Foi definida uma interface genérica para um método de avaliação que recebe uma lista de soluções e deve retornar um objeto que associa a cada solução um custo computado.

Para esse problema, apenas um operador de avaliação foi desenvolvido, que meramente associa a cada solução o seu custo, obtido pela soma dos custos das colunas contidas na solução. O programa trabalha com um problema de minimização, então valores de *fitness* menores são priorizados nas demais etapas.

3.6. Operadores de seleção

A próxima etapa na produção da próxima geração é a seleção de indivíduos para o cruzamento. O trabalho implementou dois operadores selecionadores que, em comum, recebem o resultado de uma avaliação e retornam dois dos indivíduos avaliados: um selecionador por classificação, e o selecionador por torneio.

O selecionador por classificação ordena os indivíduos da população por custo em ordem decrescente, e seleciona dois desses indivíduos com probabilidade proporcional ao seu índice na lista. Se o selecionador sorteia o mesmo indíviduo mais de uma vez, ele repete o sorteio para que dois indivíduos diferentes possam ser selecionados.

O selecionador por torneio, por sua vez, escolhe aleatoriamente k indivíduos da população e dentre os sorteados, o com menor custo é escolhido para gerar um descendente. Esse torneio ocorre ao mínimo duas vezes, uma para cada indivíduo-pai. Ambas as abordagens são baseadas em abordagens tradicionais em algoritmos genéticos.

3.7. Operadores de cruzamento

Selecionados dois indivíduos, eles são então passados para um operador de cruzamento para geração de um descendente. A interface para um operador de cruzamento recebe duas soluções e detalhes da instância do problema sendo analisada e deve retornar uma nova solução.

Este trabalho implementou apenas um cruzador, o por remoção de redundâncias, onde as colunas contidas nos dois indivíduos selecionados são juntadas em uma única solução. A solução então passa pelo mesmo algoritmo de remoção de redundâncias descrito na seção 3.3.

3.8. Operadores de mutação

Após a geração dos indivíduos descendentes, opcionalmente, o sistema aplica um operador de mutação à população gerada. O operador de mutação recebe uma lissta de soluções e a instância do problema e retorna uma lista de soluções mutadas de acordo com algum critério implementado no operador.

Este trabalho investigou dois operadores de mutação: um que remove apenas uma coluna do indivíduo, trocando-a por um conjunto aleatório de colunas que cubra as linhas descobertas pela colunas removida, com uma probabilidade p, e um mutador que realiza o mesmo processo, mas com a probabilidade de mutação sendo definida por coluna do indivíduo e não pelo indivíduo como um todo, podendo assim trocar mais de uma coluna.

3.9. A buca local

Um dos objetivos deste trabalho é investigar o uso de um algoritmo de busca e melhoria local em conjunto com o algoritmo genético. Assim, o sistema implementado possui um algoritmo que implementa a busca local sobre vizinhanças semelhantes às propostas em [Castro 2019].

O algoritmo de busca local possui dois parâmetros, sendo eles o número n de colunas a serem trocadas e o tipo de estratégia de busca (*best improvement* e *first improvement*); para cada coluna da solução, o algoritmo busca construir uma nova solução excluindo-se essa coluna e acrescentando n novas colunas ainda não utilizadas. O algoritmo verifica todas as combinações possíveis das colunas restantes tomadas n a n em busca de uma solução melhor; caso a estratégia seja *first improvement*, o algoritmo para na primeira melhoria, e caso a estratégia seja *best improvement*, o algoritmo primeiro avalia todas as combinações e em seguida escolhe a melhor.

Neste trabalho, foram utilizadas duas combinações destes parâmetros nos testes: *first improvement* com $n = 1$, e *best improvement* com $n = 1$. O algoritmo recebe a população gerada pela etapa de mutação (ou pela etapa de cruzamento, caso a mutação não tenha sido realizada) e retorna uma nova lista de indivíduos melhorados para a próxima etapa.

3.10. Operadores de atualização

Tendo sida gerada a população de novos indivíduos, os operadores de atualização misturam a população existente com a antiga para finalizar a construção da próxima geração. O operador recebe uma lista de indivíduos da população antiga e uma lista dos indivíduos gerados e retorna uma população final para a próxima geração.

Neste trabalho, foi implementado um operador de atualização elitista, isto é, que mantém uma fração dos melhores indivíduos da geração anterior na próxima geração. O operador possui um parâmetro, f, que representa o tamanho desta fração (e deve estar entre zero e um).

3.11. Desafios de implementação

A implementação apresentada neste trabalho teve alguns desafios complexos de implementação. O algoritmo heurístico desenvolvido no trabalho anterior utilizou a linguagem Python e possuía um tempo de execução grande. Para este trabalho, buscou-se utilizar a linguagem Java para desenvolvimento, tendo em mente o seu sistema de tipos mais robusto e uma organização mais sujeitável a oportunidades de paralelização.

A tradução das partes comuns do trabalho anterior de Python para Java ocorreu com tranquilidade. Entretanto, como o algoritmo heurístico proposto no trabalho anterior possuía um tempo de execução longo e era não-determinístico, optou-se por descartar esta parte do trabalho e tomar uma outra abordagem — a implementação do algoritmo de busca local descrito na seção 3.9.

Algoritmos genéticos são estocásticos e várias partes deles dependem da aleatoriedade para alcançarem um bom desempenho. Entretanto, gerenciar essa aleatoriedade para possibilitar a geração determinística de resultados, com base no fornecimento de uma semente aleatória, mostrou-se uma tarefa complexa de se realizar, frente ao uso extensivo de programação paralela no trabalho desenvolvido. Como a ordem de execução das *threads* é não-determinística, e devido à concorrência entre elas, não é possível utilizar um único gerador de números aleatórios globbal. Cada *thread* deve receber o seu próprio gerador, devidamente alimentado com uma *seed*, para utilizar em seus processos, mas isso também gera alguns desafios: como determinar essa *seed* para cada *thread* sem viciar o sistema e deixá-lo cair em padrões previsíveis de geração? Ao fim, a estratégia tomada foi atribuir a cada *thread* um identificador determinístico, e realizar o cálculo de uma função *hash* sobre a combinação desse identificador com a semente global do programa. Assim, a esperança é de que os geradores de números aleatórios não sejam tão correlacionados. Contudo, mesmo utilizando uma cadeia de geradores de números aleatórios alimentados por uma semente global, ainda foram encontrados alguns

A implementação eficiente do algoritmo de busca local foi um desafio. A aplicação executa quase instantaneamente sem a etapa da busca local, mas com a busca local, uma execução pode levar vários minutos. Parte disso é inevitável, visto que as duas estratégias abordadas na seção 3.9 têm complexidade $O(n)$ e $O(n^2)$, onde n é o número de colunas no problema, então o foco foi sobre a redução de fatores constantes.

No geral, a implementação de um sistema otimizador por algoritmo genético em Java foi uma experiência nova ao autor do trabalho e a codificação foi em partes agradável e desafiadora.

4. Resultados

A implementação descrita na seção 3 foi testada 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 processador é multi-núcleo e é capaz de executar oito *threads* em paralelo. O código foi implementado e executado com o Java OpenJDK 21.0.2.

As instâncias do Problema de Cobertura de Conjuntos testadas foram as fornecidas em sala de aula e as mesmas utilizadas no trabalho anterior, sendo elas

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. Cada teste foi realizado uma única vez e a semente utilizada por cada execução é relatada neste documento.

Para o teste, foram testadas todas as combinações dos seguintes operadores e parâmetros para cada instância de entrada:

1. **Tamanho da população:** 100, 1000;
2. **Número de gerações:** 10, 20;
3. **Avaliador:** de custo;
4. **Selecionador:** por ordenação e por torneio (k = 3);
5. **Cruzador:** por remoção de redundâncias;
6. **Mutador:** por substituição de colunas (p = 0.2);
7. **Busca local:** sem busca local, com busca local por *first improvement* (n = 1) e com busca local por *best improvement* (n=1);
8. **Atualizador:** elitista (f = 0.15).

Não optamos por testar o mutador por substituição de uma única coluna pois testes realizados ao longo do desenvolvimento indicaram que o mutador de várias colunas gerava resultados melhores ao fim. Usamos uma taxa de mutação alta em comparação com o recomendado na literatura, pois as instâncias estudadas eram esparsas, e o número de avaliações feitas sobre a representação adotada, em comparação com a representação binária tradicional, seria muito reduzida. Valores menores não estavam conseguindo preservar a diversidade das populações.

Ao todo, dezesseis (24) combinações de parâmetros e operadores, combinadas com cada instância, totalizaram cento e noventa e seis (196) testes propostos. Nem todos os testes propostos foram executados devido a questões de tempo de execução; Esta seção descreve os melhores resultados encontrados para cada instância e o tempo de execução do melhor resultado. Todos os testes foram executados com a semente 42 [Adams 1979].

4.1. Resultados sem o uso de busca local

A primeira porção da bateria de testes verificou as combinações de parâmetros e operadores especificadas na seção 4.2 sem o uso de busca local; nesta porção, foram realizados sessenta e quatro (64) testes. A tabela 1 exibe as configurações testadas em cada instância junto de seus identificadores; apenas o tamanho da população, número de gerações e o operador de seleção foram variados em relação ao disposto na seção anterior, com os demais valores sendo os únicos disponíveis.

A tabela 2 mostra os melhores resultados obtidos por cada instância, junto com o tempo de execução e configuração utilizada na obtenção deste resultado. Por melhor resultado entende-se o que possui o menor custo, ou em caso de empate de menor custo, o que possui menor tempo.

É possível observar que em alguns casos os melhores resultados obtidos foram com uma população menor, de tamanho 100, mas na maioria dos casos os melhores resultados foram obtidos com populações de 1000 indivíduos e 20 gerações. Isso indica que o algoritmo se beneficia de um tempo de execução maior e não entra em estagnação tão facilmente, mas para esses casos requer um tempo de execução bem mais elevado em relação a configurações menores.

O desvio padrão se manteve razoável em todos os casos ao fim da otimização, numa faixa de 5-15% da média dos custos finais. Isso também indica que o algoritmo conseguiu preservar a diversidade da população inicial, mesmo após várias iterações do processo.

	População	Gerações	Selecionador
1	100	10	Classificação
2	100	10	Torneio
3	100	20	Classificação
4	100	20	Torneio
5	1000	10	Classificação
6	1000	10	Torneio
7	1000	20	Classificação
8	1000	20	Torneio

Tabela 1 – Configurações testadas para cada instância.

	Menor custo final	Custo médio final	Desvio padrão	Tempo (s)	Config.
Teste_01	604.59	751.73	85.03	2.155	8
Teste_02	622.45	801.25	104.36	1.525	6
Teste_03	555.3	761.15	103.59	11.167	7
Teste_04	1294.95	1586.62	183.56	0.586	3
Wren_01	8330	9910.72	699.57	0.317	1
Wren_02	16073	18157.04	955.13	92.765	7
Wren_03	15997	18138.53	1062.72	82.485	8
Wren_04	68364	73779.13	4352.78	75.894	5

Tabela 2 – Melhores resultados obtidos para cada instância sem busca local.

A convergência do algoritmo sem busca local, entretanto, é lenta. A figura 1 contém um gráfico indicando a convergência do processo de otimização que levou ao melhor resultado para a instância Wren_02. A curva azul indica o custo da melhor solução na população, a curva laranja indica o custo médio e a curva vermelha indica o desvio padrão dos custos na população. Note que o custo da melhor solução esteve relativamente estável até a geração 16, quando ela começou a cair até chegar a um patamar estável em torno de 16073 na geração 18. Devido à falta de melhorias por várias gerações, em conjunto com o atualizador elitista, é possível observar também que o desvio padrão das soluções decaiu ao longo do tempo, mas ainda assim permaneceu em torno de 90% do seu valor máximo atingido.

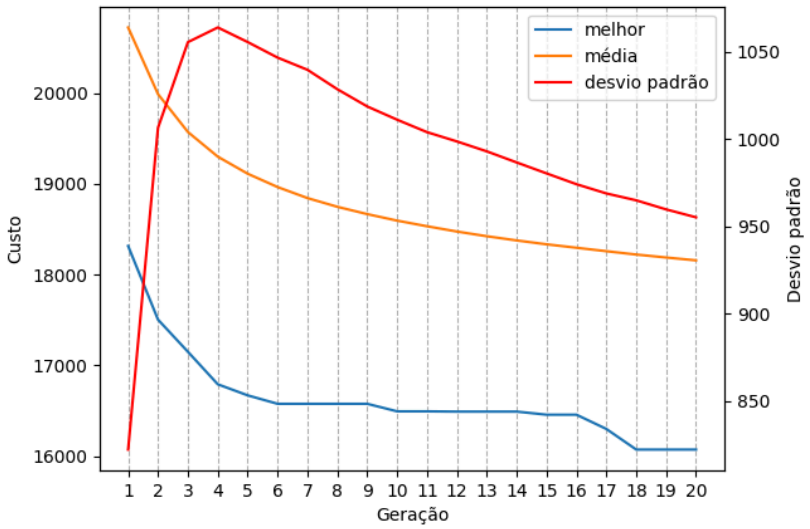

Figura 1 – Convergência da melhor configuração para Wren_02 sem busca local.

Os melhores resultados e a comparação deles com os valores fornecidos na especificação da atividade podem ser vistos na tabela 3. Não foi possível obter a melhor solução conhecida em nenhum dos casos, mas a pior diferença foi de 17.54% na instância Wren_04, o que não é tão grande.

Instância	Melhor solução	Solução obtida	Gap
Teste_01	557.44	604.59	8.46%
Teste_02	537.89	622.45	15.72%
Teste_03	517.58	555.3	7.29%
Teste_04	1162.8	1294.95	11.36%
Wren_01	7856.0	8330	6.03%
Wren_02	13908.0	16073	15.57%
Wren_03	13780.0	15997	16.09%
Wren_04	58161.0	68364	17.54%

Tabela 3 – Resultados resumidos do algoritmo genético sem busca local.

4.2. Resultados com o uso de busca local

A segunda porção da bateria de testes foi realizada sobre uma fração das cento e vinte e oito (128) combinações de configurações e instâncias restantes. Nesta etapa, foram testadas as duas opções de vizinhança: *best improvement* trocando-se uma coluna, e *first improvement* trocando-se duas colunas. Devido a restrições de tempo, nem todos os testes puderam ser realizados, particularmente todas as configurações da instância Wren_04 na estratégia *best improvement* e as configurações 2-8 na estratégia *first improvement*. Os resultados nesta seção são, portanto, parciais para essa instância.

A tabela 4 exibe os melhores resultados obtidos para cada instância utilizando-se a busca local *best improvement*, enquanto a tabela 5 exibe os melhores resultados com o *first improvement*.

	Menor custo final	Custo médio final	Desvio padrão	Tempo (s)	Config.
Teste_01	557.86	663.086	90.8502	26.187	7
Teste_02	537.89	665.482	128.761	20.043	6
Teste_03	515.3	643.706	144.596	27.499	6
Teste_04	1162.8	1412.6	253.435	75.831	5
Wren_01	7856.0	8668.52	926.571	48.108	6
Wren_02	14038.0	15000.2	1418.68	4104.37	7
Wren_03	13952.0	15481.1	2004.83	1916.61	6
Wren_04*	-	-	-	-	-

* Nenhuma configuração foi testada devido a limitações de tempo.

Tabela 4 – Melhores resultados obtidos para cada instância com busca local *best improvement*.

	Menor custo final	Custo médio final	Desvio padrão	Tempo (s)	Config.
Teste_01	557.44	662.552	93.6574	19.454	7
Teste_02	537.89	672.325	126.156	13.608	5
Teste_03	515.3	649.729	143.138	20.717	5
Teste_04	1171.43	1417.24	251.005	48.585	5
Wren_01	7856.0	8348.85	691.851	6.966	4
Wren_02	14034.0	15523.6	1859.99	1506.09	5
Wren_03	13905.0	15610.5	1960.63	1433.85	6
Wren_04*	59878.0	64344.88	7165.58	4903.77	1

* Apenas a configuração 1 foi testada devido a limitações de tempo.

Tabela 5 – Melhores resultados obtidos para cada instância com busca local *first improvement*.

Como mostram as tabelas, a estratégia *first improvement* aparentemente gerou melhores resultados tanto em termos de menores custos quanto em tempo de execução; a única instância que não foi superada pelo método *first improvement* foi a Teste_04, mas não existe nenhum impedimento técnico que impeça esta estratégia de obter resultados melhores. Em testes realizados durante o desenvolvimento com outras sementes a estratégia *first improvement* também atingiu esse mínimo (a semente e os resultados não foram salvos, infelizmente).

Outra observação que podemos realizar é a de que as configurações 5 e 6, com mil indivíduos na população e dez gerações, foi a mais favorável no conjunto de experimentos propostos; ela foi capaz de manter uma diversidade grande o suficiente para não estagnar o algoritmo genético em mínimos locais, mas ao mesmo tempo não

realizou uma quantidade demasiada grande de computações desnecessariamente. Essa observação se repete parcialmente no conjunto de testes realizados sem busca local, também.

Por outro lado, observa-se que o aumento no tempo de processamento das instâncias aumentou em uma ou duas ordens de magnitude em relação à busca local. Durante o desenvolvimento do trabalho, o algoritmo de busca local e alguns algoritmos auxiliares de construção de soluções tiveram de ser reescritos e otimizados para que o trabalho pudesse ser desenvolvido em tempo hábil, o que significa que os resultados desta subseção usam algoritmos mais eficientes que os da seção anterior. Contudo, a diferença ainda é gritante.

Alguns casos de teste rapidamente alcançaram os mínimos atingidos em uma ou duas gerações, mas os mais complexos apresentaram uma tendência de melhoria contínua; se fossem executados por mais gerações poderiam ser capazes de gerar resultados ainda melhores. A figura 2 mostra um gráfico de convergência que exemplifica o primeiro caso, da instância Wren_01 com a busca local por *best improvement*. Nota-se que a queda ocorre na segunda geração e lá permanece até o fim do processo de otimização.

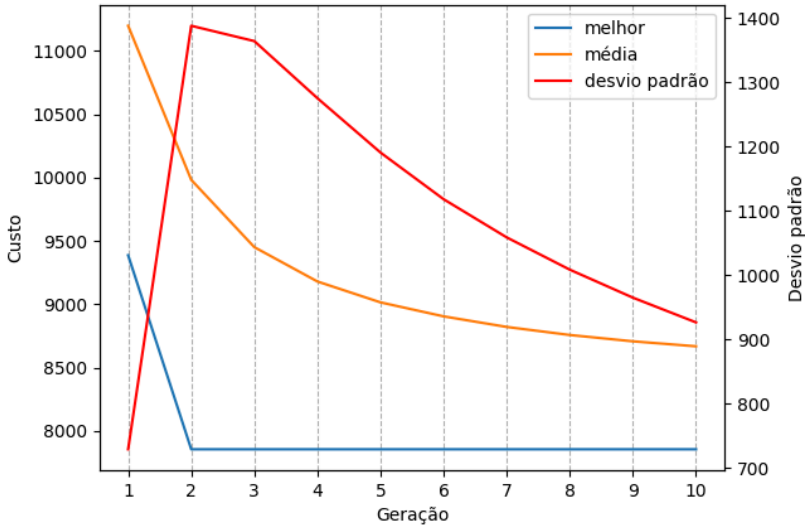

Figura 2 – Convergência da melhor configuração para Wren_01 com busca local *best improvement*.

A figura 3 mostra um gráfico de convergência do segundo caso, para a instância Wren_03, com o uso de busca local *first improvement*. A melhora é bem gradual mas ocorre até a geração 9, quando o conjunto de dados estabiliza em torno do mínimo alcançado.

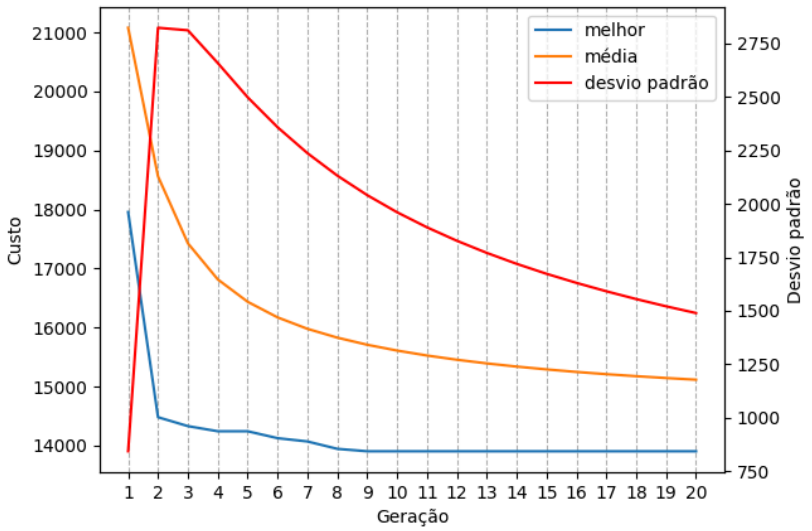

Figura 3 – Convergência da melhor configuração para Wren_03 com busca local *first improvement*.

De modo geral, o uso de busca local acelera a convergência em termos de gerações necessárias para atingir-se bons resultados mas piora o tempo de execução, devido à alta quantidade de manipulações que devem ser feitas sobre os indíviduos da população. A diferença na convergência é grande; a figura 4 mostra uma análise comparativa da convergência da instância Wren_04 com e sem busca local.

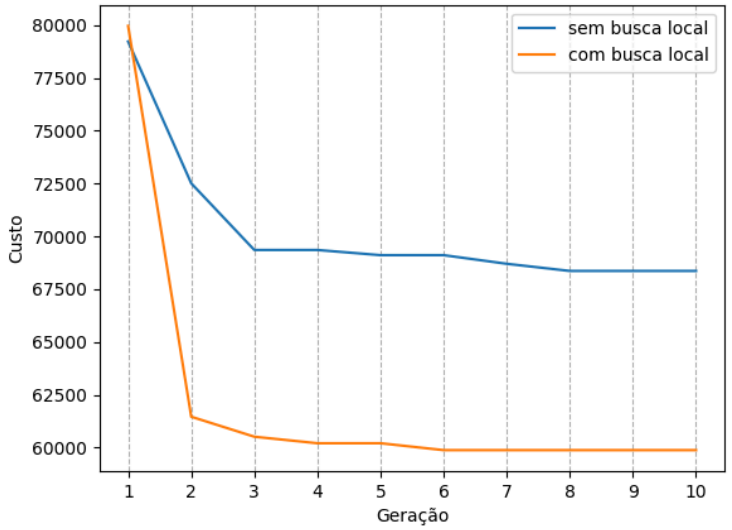

Figura 4 – Análise de convergência das melhores soluções obtidas para a instância Wren_04 (com e sem busca local).

O uso ou não de busca local num algoritmo genético deve ser analisado com consideração e em relação aos objetivos desejados e aos recursos disponíveis para execução. A tabela 6 apresenta de maneira resumida os melhores resultados encontrados com a busca local e a diferença percentual entre eles e os melhores valores fornecidos.

Instância	Melhor solução	Solução obtida	Gap
Teste_01	557.44	557.44	0.00%
Teste_02	537.89	537.89	0.00%
Teste_03	517.58	515.3	**-0.44%**
Teste_04	1162.8	1162.8	0.00%
Wren_01	7856.0	7856.0	0.00%
Wren_02	13908.0	14034.0	0.91%
Wren_03	13780.0	13905.0	0.91%
Wren_04	58161.0	59878.0	2.95%

Tabela 6 – Resultados resumidos do algoritmo genético com as buscas locais.

4.3. Resultados globais

A tabela 7 apresenta os melhores resultados obtidos no trabalho inteiro e as diferenças entre os melhores valores conhecidos fornecidos e os resultados obtidos. A diferença ficou menor que 4% em todos os casos, atingindo o patamar de 0% em quatro deles e um valor negativo, -0.44% no caso Teste_03. Em comparação com o trabalho anteriormente desenvolvido, foram obtidos resultados melhores para os casos Teste_01, Teste_04, Wren_01 e Wren_04 e resultados piores para os casos Wren_02 e Wren_03.

O uso de algoritmos genéticos, mesmo sem busca local, entretanto, proporcionou resultados melhores em todos os casos quando apenas comparados com o uso apenas de um algoritmo construtivo (vide tabela 8). Isso demonstra a capacidade desta meta-heurística em explorar um espaço maior de busca de maneira relativamente simples e pouco custosa.

Pela análise dos gráficos de convergência deste trabalho, somos tentados a pensar que se deixados executando por mais tempo, soluções melhores poderiam ser alcançadas. Neste trabalho não exploramos números de gerações ou populações maiores por limitações de tempo, mas a proposta deve ser investigada em trabalhos futuros.

Instância	Melhor solução fornecida	Solução s/ busca local	*Gap*	Solução c/ busca local	*Gap*
Teste_01	557.44	604.59	8.46%	557.44	0.00%
Teste_02	537.89	622.45	15.72%	537.89	0.00%
Teste_03	517.58	555.3	7.29%	515.3	**-0.44%**
Teste_04	1162.8	1294.95	11.36%	1162.8	0.00%
Wren_01	7856.0	8330.0	6.03%	7856.0	0.00%
Wren_02	13908.0	16073.0	15.57%	14034.0	0.91%
Wren_03	13780.0	15997.0	16.09%	13905.0	0.91%
Wren_04	58161.0	68364.0	17.54%	59878.0	2.95%

Tabela 7 – Tabela resumida dos resultados finais.

As soluções apresentadas na lista 1 foram as melhores encontradas para cada instância. As soluções vêm acompanhadas do nome da instância e do custo final da solução.

1. Teste_01: 557.44 – 58, 103, 156, 178, 179, 187, 194, 222, 261.
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: 1162.8 – 44, 56, 60, 74, 76, 176, 194, 197, 206, 211, 226, 244, 245, 264, 287, 326, 349, 418, 468.
5. Wren_01: 7856.0 – 1, 3, 75, 127, 145, 208, 241, 246, 249, 250, 303, 345, 357, 361, 456.
6. Wren_02: 14034.0 – 184, 423, 758, 1934, 2094, 2836, 2971, 3000, 3841, 4024, 4753, 4979, 4990, 5005, 5007, 5050, 5070, 5130, 5143, 5147, 5202, 5363, 5389, 5399, 5409, 5427, 5448, 5449, 5460, 5481, 5484, 5503, 5522.
7. Wren_03: 13905.0 – 446, 566, 653, 1164, 1401, 1572, 1690, 1980, 2224, 2733, 2808, 3371, 3776, 3918, 4551, 4578, 4634, 4654, 4691, 4695, 4791, 4843, 4862, 4881, 4900, 4901, 4919, 4920, 4936, 4964, 4968, 4983.
8. Wren_04: 59878.0 – 20, 34, 93, 133, 142, 147, 166, 182, 195, 218, 228, 248, 263, 271, 283, 285, 306, 330, 418, 462, 518, 525, 533, 549, 560, 599, 607, 645, 650, 675, 726, 737, 758, 759, 763, 841, 885, 894, 915, 937, 968, 994, 1014, 1074, 1093, 1156, 1167, 1168, 1169, 1228, 1244, 1275, 1299, 1324, 1364, 1377, 1393, 1480, 1552, 1615, 1720, 1740, 1777, 1787, 1868, 2047, 2087, 2198, 2246, 2323, 2360, 2405, 2428, 2441, 2484, 2492, 2529, 2538, 2561, 2627, 2630, 2640, 2655, 2660, 2681, 2687, 2693, 2706, 2723, 2728, 2735, 2741, 2757, 2762, 2769, 2776, 2799, 2804, 2814, 2833, 2970, 2973, 3038, 3105, 3120, 3131, 3331, 3361, 3367, 3392, 3423, 3454, 3528, 3718, 3743, 3788, 4093, 4201, 4256, 4377, 4439, 4444, 4455.

Lista 1 – Melhores soluções encontradas nos experimentos executados.

4.4. Apêndice: melhores resultados obtidos no trabalho anterior

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 8 – Resultados alcançados no trabalho anterior.

5. Conclusões

Neste trabalho exploramos a implementação de um algoritmo genético com busca local para solucionar o Problema da Cobertura de Conjuntos (*Set Cover Problem,* ou SCP), um problema tradicional de otimização combinatória. Puderam ser explorados vários conceitos vistos em sala de aula e a implementação serviu como um exercício no uso da linguagem Java e na exploração de técnicas de processamento paralelo.

O sistema proposto e aqui implementado conseguiu gerar resultados que se equiparam ou até superam os custos mínimos estabelecidos para as instâncias fornecidas. Por meio da combinação da flexibilidade do algoritmo genético com o uso da busca local, foi possível atingir diferenças percentuais menores que 2% para todas as instâncias, e menores ou iguais a 0% para quatro delas.

Como sugestões de trabalhos futuros, devem ser estudados a influência que um aumento no número de indivíduos ou no número de gerações tem sobre os resultados obtidos para as instâncias mais complexas, e o sistema deve ser aplicado em outras instâncias diferentes das fornecidas para que o seu desempenho possa ser avaliado.

Referências

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

JACOBS, Larry W.; BRUSCO, Michael J. Note: A local‐search heuristic for large set‐covering problems. **Naval Research Logistics (NRL)**, v. 42, n. 7, p. 1129-1140, 1995.

VASKO, Francis J. An efficient heuristic for large set covering problems. **Naval Research Logistics Quarterly**, v. 31, n. 1, p. 163-171, 1984.

CASTRO, Eduardo Meca. Two Neighbourhood-based Approaches for the Set Covering Problem. **U. Porto Journal of Engineering**, v. 5, n. 1, p. 1-15, 2019.

ADAMS, Douglas. The Hitchhiker's Guide to the Galaxy. Pan Books: Reino Unido, 1979.