PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ABORDAGENS HEURÍSTICAS, META-HEURÍSTICAS E DE PROGRAMAÇÃO LINEAR INTEIRA PARA O COVERING SALESMAN PROBLEM

João Augusto Ferreira de Moura

GOIÂNIA
2023

João Augusto Ferreira de Moura

ABORDAGENS HEURÍSTICAS, META-HEURÍSTICAS E DE PROGRAMAÇÃO LINEAR INTEIRA PARA O COVERING SALESMAN PROBLEM

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

GOIÂNIA
2023

João Augusto Ferreira de Moura

ABORDAGENS HEURÍSTICAS, META-HEURÍSTICAS E DE PROGRAMAÇÃO LINEAR INTEIRA PARA O COVERING SALESMAN PROBLEM

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do grau de Bacharel em Ciência da Computação e aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, em 20 de junho de 2023.

Profa. Ma. Ludmilla Reis Pinheiro dos Santos Coordenadora de Trabalho de Conclusão de Curso

Banca examinadora:

Orientador: Prof. Me. Alexandre Ribeiro

Profª. Dra. Maria José Pereira Dantas

Prof. Me. Max Gontijo de Oliveira

GOIÂNIA
2023

AGRADECIMENTOS

Agradeço, primeiramente, a Deus por ter me guiado e ter possibilitado que tudo acontecesse.

Aos meus pais e a minha tia que me deram suporte e estiveram presentes por toda a minha vida me motivando a continuar.

Ao meu professor, orientador e amigo Alexandre Ribeiro, por tudo que fez por mim em todos esses anos, agradeço por todas as broncas, conselhos, ensinamentos e apoio, e por ter sido o melhor professor que já tive.

A todos os meus amigos que conheci durante o curso. Em especial, a Lara Ferigatto, Jovânio Júnior e Victor Araújo. Vocês foram pessoas muito importantes para que eu conseguisse concluir essa etapa.

A todos os professores da universidade. Obrigado por todos os ensinamentos e por continuarem passando seus conhecimentos e experiências para todos.

RESUMO

Neste trabalho de conclusão de curso, foi abordada uma variação do Problema do Caixeiro Viajante, chamada *The Covering Salesman Problem*. Neste problema, são dados um conjunto de cidades, e uma matriz de distância entre elas. O objetivo é encontrar um ciclo de custo mínimo em um subconjunto de cidades, dentro do conjunto das *n* cidades dadas, de forma que cada cidade que não faça parte do ciclo esteja dentro de uma distância de cobertura *S* de alguma cidade que esteja no ciclo. Ao longo do trabalho foram propostos um algoritmo de aproximação baseado em programação linear inteira chamado Relaxação Lagrangiana, uma heurística Gulosa e duas meta-heurísticas chamadas Busca Tabu e *Ant Colony Optimization* para resolver o problema. A estratégia utilizada a partir destas abordagens foi dividir o problema em dois problemas clássicos: O Problema da Cobertura de Conjuntos e o Problema do Caixeiro Viajante. Desta forma, a Relaxação Lagrangiana, a Heurística Gulosa e a Busca Tabu foram utilizadas para resolver o Problema da Cobertura de Conjuntos primeiramente, com o objetivo de minimizar a quantidade de cidades que fazem parte do ciclo, para que então o Ant Colony Optimization resolva o Problema do Caixeiro Viajante, com as respostas dadas pelos algoritmos responsáveis por resolver a primeira parte, buscando o ciclo hamiltoniano de custo mínimo entre as cidades selecionadas. Para os experimentos computacionais, foram utilizados os resultados obtidos por um algoritmo da literatura baseado em busca local denominado LS2 para fins de comparação. Os resultados não se mostraram satisfatórios, com os algoritmos elaborados não conseguindo obter nenhuma resposta melhor do que a encontrada pelo LS2, o que mostra que a estratégia utilizada neste trabalho é ineficiente para resolver o *Covering Salesman Problem*. No entanto, pôde-se observar que, para praticamente todos os ciclos gerados pelos algoritmos elaborados, a quantidade de vértices presentes neles foi menor do que no algoritmo LS2, isso fez com que o problema deste trabalho fosse reelaborado, associando um custo para adicionar os vértices de cobertura ao ciclo. Com essa variação do CSP, os algoritmos que utilizaram a abordagem deste trabalho apresentaram resultados melhores que o algoritmo LS2 na maior parte dos casos em que o custo de adicionar novos vértices de cobertura ao ciclo é elevado.

Palavras-chave: *Covering Salesman Problem*. Caixeiro Viajante. Cobertura de Conjuntos.

ABSTRACT

In this term paper, a variation of the Traveling Salesman Problem, called The Covering Salesman Problem, was studied. In this problem, a set of cities and a matrix of distances between them are given. The objective is to find a minimum cost cycle in a subset of cities, within the set of n given cities, such that each city that is not part of the cycle is within coverage distance S of some city that is in the cycle. Throughout the work, an approximation algorithm based on integer linear programming called Lagrangian Relaxation, a Greedy heuristic and two meta-heuristics called Tabu Search and Ant Colony Optimization were proposed to solve the problem. The strategy used from these approaches was to divide the problem into two classic problems: The Set Coverage Problem and the Traveling Salesman Problem. In this way, the Lagrangian Relaxation, the Greedy Heuristic and the Tabu Search were used to solve the Set Coverage Problem firstly, with the objective of minimizing the number of cities that are part of the cycle, so that Ant Colony Optimization can solve the problem. Traveling Salesman Problem, with the answers given by the algorithms responsible for solving the first part, seeking the Hamiltonian cycle of minimum cost among the selected cities. For the computational experiments, the results obtained by an algorithm from the literature based on local search called LS2 were used for comparison purposes. The results were not satisfactory, with the developed algorithms not managing to obtain any better response than the one found by LS2, which shows that the strategy used in this work is inefficient to solve the Covering Salesman Problem. However, it was observed that, for practically all the cycles generated by the elaborated algorithms, the amount of vertices present in them was smaller than in the LS2 algorithm, this caused the problem of this work to be re-elaborated, associating a cost to add the cover vertices to the cycle. With this CSP variation, the algorithms that used the approach of this work presented better results than the LS2 algorithm in most cases where the cost of adding new coverage vertices to the cycle is high.

Keywords: *Covering Salesman Problem*. *Travelling Salesman Problem. Set Covering Problem.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

O Problema do Caixeiro Viajante *ou Travelling Salesman Problem* (TSP) é um dos problemas mais estudados na área de otimização combinatória e possui ampla importância atualmente. Ele envolve encontrar o percurso mais curto que um caixeiro viajante deve realizar para visitar um conjunto de cidades uma única vez e retornar à cidade inicial. Sua relevância deve-se ao fato de que a otimização de rotas é uma questão fundamental em diversas áreas, como transporte, logística, planejamento urbano e telecomunicações. (MATAI et al, 2010)

Uma variação do TSP interessante também, é o *Covering Salesman Problem* (CSP). Nessa variação, o objetivo é encontrar um percurso de custo mínimo entre um subconjunto dentro do conjunto total de cidades, mas garantindo que cada cidade que não faz parte do subconjunto de cidades no percurso, esteja a uma distância de no máximo *S* de pelo menos uma cidade que faz parte do percurso. Garantindo assim, a cobertura total das cidades. (CURRENT; SCHILLING, 1989)

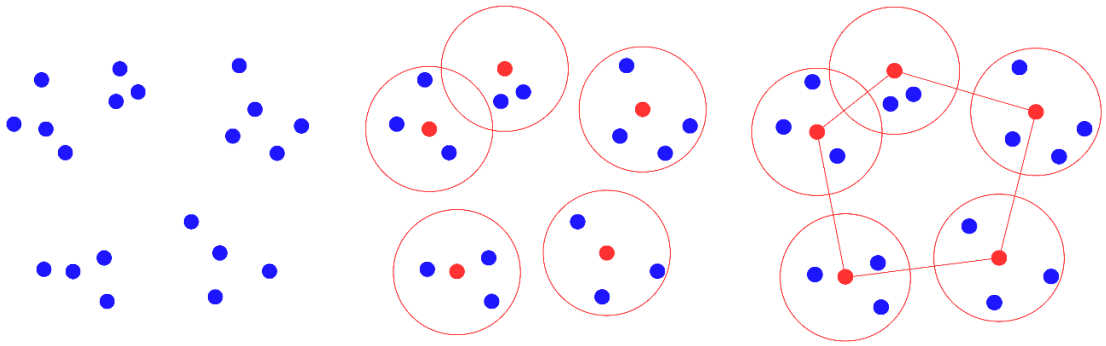
O CSP tem uma ampla gama de aplicações práticas. Na área de logística e transporte, por exemplo, ele pode ser utilizado para o planejamento de rotas de entregas, garantindo que todas as regiões sejam atendidas por um entregador, com uma distância máxima para fazer as entregas. Na área de planejamento urbano, o CSP pode ser aplicado para otimizar a coleta de lixo, levando em consideração a cobertura de cada rua ou bairro. Além disso, o CSP também é relevante na área de telecomunicações, para otimizar a instalação de torres de celular, por exemplo, garantindo a cobertura de determinadas áreas. (SALARI et al., 2015)

Para exemplificar uma aplicação do CSP mais detalhadamente, é possível imaginar uma empresa, que pretende utilizar um serviço de fretados para trazer seus funcionários para o local de trabalho. Um fretado deve sair da empresa, percorrer a cidade passando por pontos de ônibus espalhados por ela, pegar todos os passageiros, e retornar à empresa. No entanto, para selecionar os pontos de ônibus em que o fretado passará, a empresa deseja que nenhum de seus funcionários percorra uma distância maior do que *S* para chegar até ele. Para isso, a empresa utilizará como localização para construir pontos de ônibus as casas de seus funcionários. Desta forma, um subconjunto de casas será selecionado para

cobrir todo o conjunto total de casas, as casas que estão neste subconjunto serão tidas como pontos de ônibus. A condição de cobertura é que a casa de cada funcionário esteja a uma distância de no máximo *S* de um ponto de ônibus pelo qual o fretado passará. O objetivo então, é encontrar um percurso para o fretado, que sai da empresa, percorre a menor distância possível para pegar todos os seus funcionários, que estarão em alguns dos pontos de ônibus espalhados pela cidade, e os leva para a empresa, tal que nenhum funcionário percorra uma distância maior que *S* para chegar ao ponto de ônibus.

A Figura 1 mostra o exemplo detalhado acima, com a) contendo o conjunto das casas dos funcionários representadas por vértices azuis, b) mostra como as casas dos funcionários selecionados (representadas pela cor vermelha) pode cobrir os outros, se tornando pontos de ônibus e, com a restrição de cobertura satisfeita, é possível determinar um ciclo de distância mínima entre eles, como feito em c), ao final disso, seria adicionado um vértice fixo, que representaria a empresa, que também deve fazer parte do ciclo.

Figura 1 - exemplo de resolução do CSP



a) b) c)

Resolver o CSP exige um desafio computacional complexo, pois combina a busca pelo percurso mínimo com a cobertura total de pontos. Tanto o TSP quanto o CSP são desafiadores do ponto de vista computacional, pertencendo à classe de problemas NP-difíceis (CURRENT; SCHILLING, 1989). Isso significa que não existem algoritmos conhecidos que possam resolver esses problemas de forma exata e eficiente para todas as instâncias. Dessa forma, são necessárias abordagens

aproximadas de otimização para encontrar soluções eficientes para esses problemas.

Até então, o problema foi resolvido de forma exata para instâncias menores (CURRENT; SCHILLING, 1989), e de maneira aproximada utilizando algoritmos não exatos. Boa parte das implementações da literatura utilizam algoritmos baseados em busca local, como em GOLDEN et al. (2012), em que são apresentados os algoritmos LS1 e LS2, ou em VENKATESH et al. (2019), que apresenta um algoritmo de busca local *multi-start* para o CSP. Também são utilizados algoritmos aproximados que resolvem bem o TSP, por conta de o CSP ser considerado uma variação do mesmo, como em SALARI; NAJI-AZIMI, (2015) em que é apresentado um algoritmo híbrido baseado na meta-heurística *Ant Colony Optimization* e programação dinâmica. Dentre esses métodos, o algoritmo LS2 foi escolhido para fins de comparação, pois no artigo em que é apresentado (GOLDEN et al., 2012), é mostrada a quantidade de vértices de cobertura utilizados na melhor resposta encontrada para o CSP, e essa informação será relevante para este trabalho.

No entanto, todas as abordagens citadas resolvem o problema de forma direta e única, a não ser em Current e Schilling (1989), em que para resolver o CSP, o problema é dividido em dois problemas de otimização combinatória distintos, primeiramente resolvendo um *Set Covering Problem* para garantir a cobertura dos vértices, e em seguida resolvendo um TSP, para garantir a distância mínima entre eles. Porém essa abordagem foi resolvida apenas de forma exata, fazendo com que instâncias maiores do problema não possam ser executadas em tempo hábil.

1.1 Objetivos

O objetivo inicial deste trabalho de conclusão de curso era propor uma abordagem para resolver o *Covering Salesman Problem*, dividido-o em dois problemas distintos, o *Set Covering Problem* (SCP), resolvendo-o primeiramente, com algoritmos baseados em programação linear inteira (Relaxação Lagrangiana), métodos heurísticos (Heurística Gulosa) e meta-heurísticos (Busca Tabu), a fim de comparar o desempenho das três abordagens para o problema, e o *Travelling Salesman Problem* resolvendo-o com ACO em seguida, considerando a abordagem

de minimização de custos e sendo guiada pela primeira parte, responsável por resolver o SCP. E comparar esta abordagem com o algoritmo da literatura LS2.

O objetivo final é avaliar o problema CSP como multiobjetivo e propor uma solução mono-objetivo para comparação com a proposta do objetivo inicial, e com o LS2 novamente.

1.2 Organização Textual

O restante deste trabalho está textualmente organizado da seguinte forma: No capítulo 2 são apresentados conceitos preliminares sobre o que são problemas de otimização, programação linear inteira, abordagens heurísticas e meta-heurísticas, também são introduzidos quatro algoritmos utilizando estas abordagens; o capítulo 3 apresenta o *Covering Salesman Problem* com sua definição formal e formulação matemática, também são apresentadas as definições e definições formais do Problema do Caixeiro Viajante e Problema da Cobertura de Conjuntos; o capítulo 4 relaciona o Problema de Cobertura de Conjuntos e Problema do Caixeiro Viajante ao *Covering Salesman Problem*, e apresenta a proposta para resolvê-lo, logo depois mostra detalhadamente como cada algoritmo apresentado no capítulo 2 foi adaptado para resolver o problema; o capítulo 5 apresenta os resultados dos experimentos computacionais realizados, apresenta um exemplo que melhor se adequa à proposta e faz a reelaboração do problema; por fim o capítulo 6 conclui com algumas considerações finais.

2 CONCEITOS PRELIMINARES

Neste capítulo são apresentados fundamentos teóricos e conceitos para o entendimento deste trabalho. A seção 2.1 apresenta uma introdução aos problemas de otimização combinatória e de natureza inteira e seus desafios. As seções 2.2, 2.3 e 2.4 apresentam abordagens para resolver estes problemas, cada uma com suas vantagens e desvantagens. As seções restantes apresentam pseudocódigos para as abordagens apresentadas, sendo estas respectivamente: Relaxação Lagrangiana, Heurística Gulosa, Busca Tabu e, por fim, *Ant Colony Optimization*.

2.1 Problemas de Otimização Combinatória

Problemas de otimização combinatória podem ser resumidos como sendo a busca de um subconjunto de elementos que represente a melhor solução possível para o problema dentre um conjunto discreto de opções, satisfazendo também um conjunto de regras (PAPADIMITRIOU; STEIGLITZ, 1998).

CARVALHO (2001) descreve problemas de otimização combinatória como tendo três "ingredientes" principais: um conjunto de instâncias, um conjunto de soluções viáveis $sol(I)$ para cada instância I e uma função que atribui um valor à cada solução viável., sendo que quando o conjunto $Sol(I)$ é vazio, significa não haver solução viável para a instância I, o que torna a instância inviável.

Problemas de minimização ou maximização estão, portanto, interessados em soluções viáveis de valor mínimo ou máximo, respectivamente. Como o objetivo do problema seria minimizar ou maximizar, é possível dizer que se trata de um problema de otimização em que se busca no espaço de soluções viáveis, uma solução na qual o valor seja ótimo. Por exemplo: um problema tem como objetivo selecionar o máximo de elementos possível de um conjunto finito. Para cada um destes elementos é associado um custo e, para cada subconjunto gerado, seu custo seria a soma dos custos de seus elementos, tendo como restrição uma capacidade máxima, na qual o custo do subconjunto selecionado como solução nunca poderá ultrapassar. Tem-se então um problema de otimização combinatória que tem como objetivo encontrar o subconjunto com o máximo de elementos possíveis com um custo que atenda a restrição de capacidade máxima. Este exemplo pode ser

facilmente resolvido com um algoritmo guloso, ordenando os elementos e escolhendo os que tem menor custo sempre.

2.2 Programação Linear Inteira

A programação linear inteira (PLI) é uma extensão da programação linear que tem como foco resolver problemas de otimização, cujas variáveis de decisão assumem apenas valores inteiros, enquanto na programação linear as variáveis de decisão podem assumir valores reais. Discretizar problemas para que as variáveis de decisão assumam apenas valores inteiros tornam os problemas muito mais complexos.

Segundo Klee (1980), muitos problemas práticos, especialmente aqueles de pesquisa operacional e ciência da computação, estão preocupados com a otimização de uma função de valor real f sobre um conjunto finito X de d-tuplas de números inteiros. Quando f é linear e X é definido por um número finito de restrições de desigualdade linear com coeficientes inteiros, o problema é de programação linear inteira.

A necessidade de formular e resolver problemas de programação linear inteira veio do fato de que para certas aplicações com programação linear, soluções fracionárias eram indesejáveis (PAPADIMITRIOU; STEIGLITZ, 1998), como o exemplo em que um valor x_j representa o número de aeronaves atribuídas à rota j, em um problema de otimização de rotas para voos, e, neste caso, não se pode obter uma solução fracionada.

Apesar de parecer uma alternativa válida, arredondar a solução obtida para o inteiro mais próximo pode levar a resultados subótimos ou até mesmo inválidos, pois pode resultar em uma solução que não satisfaz as restrições do problema. As restrições na PLI podem depender da natureza discreta das variáveis inteiras e, arredondar para um valor inteiro, pode fazer com que essas restrições sejam violadas, entre outros problemas.

Essa restrição adicional de integridade das variáveis de decisão torna a programação linear inteira mais complexa do que a programação linear. Enquanto a programação linear possui métodos e algoritmos eficientes que conseguem resolver em tempo polinomial problemas de otimização lineares contínuos, a

programação linear inteira requer técnicas específicas para lidar com a natureza discreta das variáveis, dado que estes problemas são considerados NP-Completo (GAREY; JOHNSON, 1990).

Existem algumas técnicas desenvolvidas para buscar a solução inteira em problemas de programação linear, como as técnicas de enumeração, que tem como objetivo enumerar, de forma inteligente, candidatos à solução ótima para o problema. O principal algoritmo que emprega esta técnica é chamado de *Branch-and-Bound*. (GOLDBARG; LUNA, 2005).

Outra forma de resolver problemas de PLI seria com abordagens não exatas. Estas, abrem mão da solução ótima, encontrando apenas uma solução considerada "boa" para tentar resolver os problemas em tempo de execução aceitável. A técnica que será apresentada neste trabalho que utiliza esta abordagem é a Relaxação Lagrangiana.

2.2.1 Relaxação Lagrangiana

A Relaxação Lagrangiana é uma técnica utilizada para resolver problemas de otimização que envolvem restrições. Ela consiste em transformar um problema com múltiplas restrições em um problema sem restrições, adicionando termos de penalidade à função objetivo (LEMARÉCHAL, 2001).

A ideia básica por trás da Relaxação Lagrangiana é introduzir multiplicadores de Lagrange para cada restrição do problema original. Esses multiplicadores são variáveis adicionais que representam o custo de violação de cada restrição, relaxando assim o problema (FISHER, 1985).

O problema original deve ser formulado com suas restrições e função objetivo. A partir daí, cria-se uma função Lagrangiana, adicionando os termos de penalidade à função objetivo original. Cada termo de penalidade é o produto do multiplicador de Lagrange correspondente da restrição associada.

A função Lagrangiana é resolvida sem as restrições. Isso pode ser feito usando técnicas de otimização, como *gradient descent* ou algoritmos de programação linear. O que resulta no que é chamado de solução primal, que são as soluções do problema sem as restrições, apenas com as penalizações na função objetivo (FISHER, 2004).

Após obter a solução da função Lagrangiana, os multiplicadores de Lagrange são atualizados para buscar uma solução viável. Isso pode ser feito usando regras de atualização, como o método de subgradiente ou *dual ascent* (FISHER, 2004).

O processo de atualização dos multiplicadores de Lagrange é repetido até que uma solução ótima seja encontrada ou até que um critério de parada seja alcançado. O critério de parada pode ser baseado no número de iterações, na melhoria da solução ou em outros critérios predefinidos. Um pseudocódigo de uma Relaxação Lagrangiana genérica será apresentado pelo Algoritmo 1.

Algoritmo 1: Relaxação Lagrangiana

Entrada: *Conjunto de restrições R, Função objetivo f(x)*
Saída: *Solução primal encontrada*
Início
  Inicialize os multiplicadores de Lagrange λ_i para cada restrição r_i com valores iniciais adequados;
  Defina um valor máximo de iterações maxIter e um critério de parada ε;
  Enquanto *não atingiu o número máximo de iterações (MaxIter)* ***faça***
    Atualize a solução primal;
    Para *cada variável de decisão x* ***faça***
      Encontre o valor que minimiza a função objetivo f(x) sob as restrições atualizadas com os multiplicadores de Lagrange;
    Fim
    Para *cada restrição r_i* ***faça***
      atualize o multiplicador de Lagrange λ_i usando um esquema de atualização adequado;
    Fim
    Se a diferença entre duas soluções primais consecutivas for menor que ε, pare a iteração;
  Fim
  Retorne a melhor solução primal encontrada;
Fim

Como mostrado no Algoritmo 1, é passado como parâmetro as restrições do problema e a função objetivo, os multiplicadores lagrangianos então são criados para cada uma das restrições com algum valor inicial. Então é feito um *loop* principal que itera até um máximo de iterações, definido no algoritmo por *maxIter*.

Dentro do *loop* principal, é criado um *loop* que busca uma solução primal para o problema, que nesse caso será uma solução que minimiza *f(x)*, aplicando multiplicadores de Lagrange que representam a penalidade de selecionar esta solução.

Ao fim do *loop* principal, os multiplicadores de Lagrange são atualizados para a próxima iteração e o critério de parada é verificado. Assim que o *loop* terminar de executar, a melhor solução primal encontrada é retornada.

Vale ressaltar que o problema pode encontrar uma solução que fira as restrições do problema, fazendo-se necessário, uma verificação ou um algoritmo complementar que complete a solução.

2.3 Abordagens Heurísticas

Problemas de otimização combinatória da classe NP (não polinomiais) exigem um custo computacional muito grande, o que significa que não há algoritmo que resolva estes problemas de forma exata em tempo polinomial quando a instância é muito grande (GAREY; JOHNSON, 1990). No entanto, nem todos os problemas necessitam serem resolvidos de forma ótima, de maneira que uma solução sendo considerada apenas boa, e que executa em um tempo aceitável, já seria o suficiente para resolver o problema. É aí que entram as abordagens heurísticas.

Abordagens ou algoritmos heurísticos são métodos de solução aproximada que buscam soluções satisfatórias em tempo viável. Elas são baseadas em julgamentos experientes e conhecimento específico do problema para guiar a busca por soluções em um espaço de solução complexo, permitindo uma exploração eficiente deste (GENDREAU; POTVIN, 2010). Um exemplo de algoritmo heurístico utilizado neste trabalho é a heurística gulosa.

2.3.1 Heurística Gulosa

Uma heurística gulosa é uma abordagem de solução para problemas de otimização que segue uma estratégia de fazer escolhas locais ótimas em cada etapa, com o objetivo de encontrar uma solução globalmente ótima. Essa abordagem é

chamada de "gulosa" porque, em cada etapa, faz a escolha que parece ser a melhor no momento, sem considerar as consequências futuras (GOLDBERG, 1989).

Em uma heurística gulosa, o algoritmo começa com uma solução vazia ou parcial e, a cada iteração, toma uma decisão localmente ótima para adicionar (ou remover) um elemento à solução, selecionar uma ação etc. A escolha é feita com base em critérios de otimização definidos para o problema em questão.

Além disso, as heurísticas gulosas são frequentemente utilizadas como uma etapa inicial para melhorar soluções por meio de outros métodos, como busca local ou algoritmos meta-heurísticos (GLOVER; KOCHENBERGER, 2006). O Algoritmo 2 mostra um pseudocódigo de uma heurística gulosa.

Algoritmo 2: Heurística Gulosa

```
Início
    Solucao ← ∅;
    Enquanto a solução não estiver completa faça
        melhorElemento ←∅;
        melhorValor ← -∞ (maximização) ou ∞ (minimização);
        Para cada elemento possível faça
            Se o elemento não viola nenhuma restrição do problema então
                Calcula o valor do elemento analisando informações
                heurísticas;
                Se o valor do elemento for melhor que o melhorValor então
                    melhorElemento ← elemento;
                    melhorValor ← valor;
                Fim
            Fim
        Fim
        Adicione o melhorElemento à solução;
    Fim
    Retorne a solução;
Fim
```

Como mostrado pelo Algoritmo 2, a heurística gulosa busca construir uma solução viável para o problema através de um *loop* principal, que executa até que uma solução seja construída. A cada iteração deste *loop*, é feito outro *loop* que varre o espaço de solução e seleciona o candidato que melhor resolve a etapa atual sem violar as restrições do problema. Ao fim do *loop* principal, o candidato

selecionado é adicionado como parte da solução, para que na próxima iteração, a próxima etapa do problema seja resolvida. Ao final, o algoritmo retorna a solução construída dentro deste *loop*.

Existem abordagens heurísticas que são mais flexíveis e genéricas para melhorar sua aplicabilidade, podendo ser facilmente adaptadas para atender às necessidades e restrições específicas de diversos problemas de otimização. Estas são chamadas meta-heurísticas (GENDREAU; POTVIN, 2010).

Meta-heurísticas são abordagens de alto nível, que se baseiam em princípios heurísticos gerais, estratégias de busca inteligente e mecanismos de aprendizado para explorar o espaço de solução de forma mais abrangente e eficiente, buscando soluções de alta qualidade (GLOVER; KOCHENBERGER, 2006).

Enquanto as heurísticas são abordagens mais específicas e direcionadas para problemas particulares, as meta-heurísticas são abordagens mais gerais e flexíveis que podem ser aplicadas a diferentes problemas de otimização. As meta-heurísticas tendem a fornecer melhores resultados, mas podem exigir mais tempo computacional devido à sua abordagem de busca mais abrangente. Problemas com múltiplos objetivos, restrições difíceis ou grandes dimensões podem se beneficiar do poder de busca das meta-heurísticas (SÖRENSEN, 2013). As meta-heurísticas utilizadas neste trabalho foram a Busca Tabu e *Ant Colony Optimization* (ACO).

2.3.2 Busca Tabu

O meta-heurística Busca Tabu é um algoritmo projetado para superar os desafios de busca local, fazendo com que as soluções geradas não fiquem presas à um ótimo local ou ciclos de soluções repetitivas (GENDREAU; POTVIN, 2010).

O algoritmo mantém uma lista chamada "lista tabu", que registra restrições ou soluções proibidas e repetidas que não podem ser utilizadas nas próximas iterações. Essas restrições podem ser baseadas em movimentos específicos, soluções fora do espaço de busca, atribuições de valores ou outras características do problema em questão. O objetivo da lista tabu é evitar que o algoritmo retorne a soluções já exploradas anteriormente, incentivando-o a explorar novas áreas do espaço de solução (GLOVER; KOCHENBERGER, 2006).

Durante a execução do algoritmo, são feitas iterações em que soluções vizinhas são geradas. A busca tem objetivo de encontrar a melhor solução possível dentro das restrições definidas para o problema. No entanto, mesmo se uma solução vizinha for pior do que a solução atual, ela ainda pode ser aceita se estiver fora da lista tabu. A aceitação dessas soluções é baseada em critérios de aspiração e permite que o algoritmo explore regiões diferentes do espaço de solução, evitando ficar preso em ótimos locais (GENDREAU; POTVIN, 2010).

O algoritmo continua iterando pela melhor solução encontrada e atualizando a lista tabu até atingir um critério de parada predefinido, como um número máximo de iterações sem melhora ou um limite iterações (GENDREAU; POTVIN, 2005). O Algoritmo 3 apresenta um pseudocódigo da busca tabu.

Algoritmo 3: Busca Tabu

```
Entrada: melhorSolucao, TabuList
Saída: Melhor solução encontrada (melhorSolucao)
Início
    melhorAvaliacao ← AvaliarSolucao(melhorSolucao);
    Enquanto não satisfaz a condição de parada faça
        Gere a lista de movimentos com a solução atual (melhorSolucao);
        Para cada movimento na listaMovimentos faça
            Selecione o melhorMovimento da lista de movimentos;
            Selecione a MelhorAvaliacaoMovimento do melhorMovimento;
        Fim
        Aplique o melhor movimento encontrado na solucaoAtual;
        Avalie a solução atual (avaliacaoAtual);
        Se avaliacaoAtual < melhorAvaliacao então
            melhorSolucao ← solucaoAtual;
            melhorAvaliacao ← avaliacaoAtual;
        Fim
        Adicione o melhorMovimento à tabuList;
       Verifique se o melhorMovimento é relevante para a solução;
       Adiciona o critério de aspiração ao melhorMovimento;
    Fim
  Retorna a melhor solução (melhorSolucao);
Fim
```

O Algoritmo 3 mostra uma busca tabu, que tem como entrada uma melhor solução parcial, que é avaliada, e em seguida o algoritmo executa através de um

loop principal, que itera até que uma solução razoável seja encontrada, ou que um número máximo de iterações seja atingido.

Dentro do *loop* principal é criada uma lista de movimentos, que representa alterações que são possíveis de fazer na solução parcial atual para tentar melhorá-la. Esta lista de movimentos é avaliada, e o melhor movimento é selecionado e aplicado à solução atual, o movimento aplicado a ela é colocado dentro da lista tabu, para que ele não possa ser feito novamente por alguma quantidade de iterações, para que o problema escape de ótimos locais. Caso a avaliação da solução atual seja melhor do que a melhor avaliação até agora, essa solução é tida como a nova melhor.

Caso o movimento que foi considerado o melhor, seja relevante para a solução, é feito um critério de aspiração, em que ele pode ser retirado da lista tabu e utilizado novamente. Ao final, o código retorna a melhor solução encontrada pelas iterações do *loop* principal.

2.3.3 Ant Colony Optimization

O Ant Colony Optimization (ACO), ou Otimização por Colônia de Formigas, é uma meta-heurística proposta por Colorni *et al.* em 1992, inspirada no comportamento das formigas quando procuram caminhos entre uma fonte de alimento e o ninho.

No ACO, um conjunto de formigas artificiais é usado para explorar o espaço de solução de um problema de otimização. Cada formiga constrói soluções parciais, consideradas como caminhos, e comunica informações sobre a qualidade dessas soluções para as outras formigas por meio de trilhas de feromônios (DORIGO et al., 2006).

No início, é definida a quantidade de formigas, com suas respectivas posições iniciais. Inicializa-se as trilhas de feromônios, geralmente com valores pequenos, em todas as posições do problema.

Cada formiga constrói uma solução parcial, fazendo escolhas baseadas em critérios como a qualidade das trilhas de feromônios e informações heurísticas específicas do problema. A probabilidade de uma formiga escolher um determinado caminho é influenciada tanto pelas trilhas de feromônios quanto pelas informações heurísticas (DORIGO et al., 2006).

Após todas as formigas terem construído suas soluções parciais, as trilhas de feromônios são atualizadas com base na qualidade das soluções encontradas. Geralmente, as trilhas de feromônios evaporam ao longo do tempo para evitar a estagnação em soluções subótimas (DORIGO; STÜTZLE, 2019).

O processo de construção de soluções e atualização das trilhas de feromônios é repetido por um número fixo de iterações ou até que um critério de parada seja atingido e a melhor solução encontrada pelas formigas ao longo do processo é considerada a solução final (DORIGO et al., 2006). O Algoritmo 4 apresenta um pseudocódigo do ACO:

Algoritmo 4: Ant Colony Optimization

```
Entrada: numFormigas, numIteracoes, taxaEvaporacao, alfa, beta
Saída: Melhor solução encontrada
Início
    Inicialize as trilhas de feromônios com valores iniciais em todas
    as posições do problema;
    Para cada iteração de 1 até numIteracoes faça
        Para cada iteração de 1 até numFormigas faça
            Construa a solução parcial para uma formiga específica levando
            em consideração trilhas de feromônios e informações heurísticas;
        Fim

        Realize a evaporação das trilhas de feromônios existentes
        Multiplicando-as pela taxa de evaporação;

        Atualize as trilhas de feromônios com base na qualidade das
        soluções construídas pelas formigas, depositando feromônios
        adicionais nas posições visitadas;

        Atualize a melhor solução global encontrada pelas formigas ao
        longo do processo caso uma nova solução melhor seja encontrada;
    Fim
    Retorne a melhor solução encontrada
Fim
```

Como mostrado pelo Algoritmo 4, é necessário definir alguns parâmetros iniciais, estes sendo, *numFormigas, numIteracoes, taxaEvaporacao, alfa* e *beta*, que representam a quantidade de formigas, o número total de iterações, a taxa de

evaporação do feromônio a cada iteração, a influência do feromônio na decisão da formiga de escolher algum caminho, e a informação de visibilidade da formiga para a decisão de escolher algum caminho.

Com os parâmetros definidos, é inicializada uma matriz de feromônios iniciais para cada posição do problema específico. É então criado um *loop* principal que itera até *numIteracoes*, e a cada iteração deste *loop*, será feito outro *loop* para cada formiga até *numFormigas*, dentro deste *loop* cada formiga irá criar uma solução parcial para o problema, tomando decisões baseadas nos parâmetros definidos e informações específicas do problema. Para cada solução parcial criada, é feita uma avaliação para definir qual a melhor solução da iteração. As trilhas de feromônios são atualizadas, utilizando a taxa de evaporação e com as soluções criadas pelas formigas. Caso a melhor solução da iteração seja melhor do que a melhor solução encontrada até agora, essa solução parcial passa a ser a melhor solução. Ao final, é retornado a melhor solução encontrada pelas formigas.

3 *COVERING SALESMAN PROBLEM*

O Covering Salesman Problem (CSP) é um problema de otimização combinatória proposto por John R. Current e David A. Schilling (1989), e pode ser definido como selecionar um ciclo de custo mínimo de um subconjunto de n cidades dadas, de modo que todas as cidades que não estão no ciclo estejam dentro de um padrão de distância de cobertura predeterminado de uma cidade que está no ciclo. O CSP pode ser visto como uma variação do Problema do Caixeiro Viajante, porém, incluindo os desafios do Problema da Cobertura de Conjuntos.

Uma definição formal para o CSP proposta por SALARI e NAJI-AZIMI (2012) apresenta o problema como a seguir: Suponha um grafo completo $G = (N, A)$ onde N é o conjunto de vértices que deve ser visitado ou coberto pelo ciclo e $A = \{(i,j) | \, i,j \in N\}$ é o conjunto de arestas do grafo e assumindo que para cada aresta $(i,j) \in A$, um custo c_{ij} é associado. Uma matriz binária de cobertura é dada por I_{nxn} em que I_{vu} será igual a 1 se, e somente se, o vértice v é coberto pelo vértice u. As variáveis de decisão do modelo são definidas como: $y_{ij} = 1$ se a aresta (i,j) fizer parte do ciclo, e 0 caso contrário; $w_i = 1$ se o vértice i fizer parte do ciclo, e 0 caso contrário. O modelo é formulado como a seguir:

$$\text{minimizar} \sum_{(i,j) \in A} c_{ij} y_{ij} \tag{1}$$

Sujeito a:

$$\sum_{(i,j) \in A} y_{ij} + \sum_{(i,j) \in A} y_{ij} = 2w_i \quad i \in N, \tag{2}$$

$$\sum_{j \in N} I_{ij} w_j \geq 1 \quad i \in N, \tag{3}$$

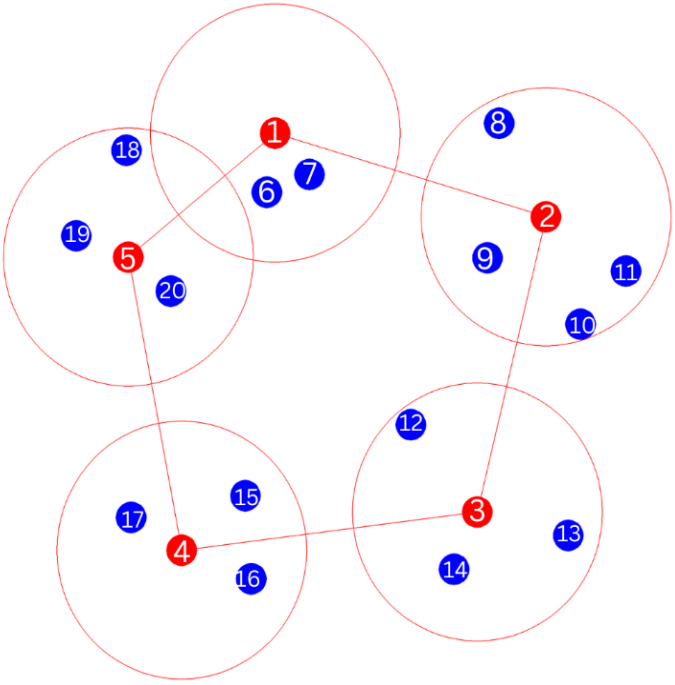
$$\sum_{l \in S} \sum_{k \in N \setminus S} y_{lk} + \sum_{l \in S} \sum_{k \in N \setminus S} y_{lk} \geq 2\left(w_i + w_j - 1\right) \quad S \subset N, 2 \leq |S| \leq n-2, i \in S, \; j \in N \setminus S, \tag{4}$$

$$y_{ij}, w_i \in \{0,1\} \qquad \forall (i,j) \in A, i \in N. \tag{5}$$

Neste modelo, o objetivo em (1) é minimizar o custo total do ciclo, o conjunto de restrições em (2) faz com que exista sempre uma aresta saindo e chegando aos vértices que estão no ciclo, ou seja, um vértice i com $w_i = 1$ tem duas arestas conectando a ele, com soma sendo igual a 2. As restrições em (3) são usadas para impor o requisito de cobertura para cada vértice $i \in N$. O último

conjunto de restrições é para eliminação de subciclos do grafo. A Figura 1 mostra um exemplo de solução para uma instância do CSP com 20 vértices.

Figura 2 – Exemplo de cobertura completa do CSP.



Fonte: adaptado de SALARI e NAJI-AZIMI (2012).

Na Figura 2 pode ser visto um ciclo que passa por todos os vértices de cobertura (representados pela cor vermelha), enquanto cada vértice coberto (representado pela cor azul) está dentro da área de cobertura de pelo menos um vértice de cobertura. Ou seja, a cobertura conseguiu ser concluída utilizando apenas 5 vértices como cobertura para os 15 restantes.

Como já dito anteriormente, o CSP pode ser representado como uma junção de dois problemas de otimização combinatória clássicos: o Problema do Caixeiro Viajante ou *Travelling Salesman Problem* (TSP), que é provado ser um problema NP-difícil por Karp, 1972; e o Problema de Cobertura de Conjuntos ou *Set Covering Problem* (SCP) citado ser, também, da classe NP-difícil por GROSSMAN e WOOS, 1997. Isso implica que o CSP também pertence à mesma classe de problemas NP-difíceis, o que significa que ele não pode ser resolvido de forma exata em tempo polinomial. Para entender a abordagem utilizada neste trabalho para a resolução do CSP, é necessário entender bem estes dois problemas.

3.1 O Problema do Caixeiro Viajante

O problema do Caixeiro Viajante (TSP) é um problema clássico de otimização combinatória que tem como objetivo encontrar um ciclo hamiltoniano de custo mínimo em um grafo. Adaptando a definição proposta por MILLER et al. (1960), o problema pode ser descrito da seguinte forma: Um vendedor deve visitar *n* cidades, indexadas de 1 a *n*. Ele sai de uma "cidade base" indexada por 0, visita cada uma das N outras cidades exatamente uma vez e retorna à cidade 0. Seja c_{ij} $(i \neq j \ = \ 0, 1, \ldots, n)$ a distância para viajar da cidade *i* para a cidade *j* e $x_{ij} \ = \ 1$ um caso o vendedor utilize o caminho de *i* para *j* como parte da solução e $x_{ij} \ = \ 0$ caso contrário. No problema deste trabalho, as cidades seriam os vértices de cobertura, que devem ser conectados por um ciclo hamiltoniano. A formulação matemática do problema, com base nessa definição, é:

$$minimizar \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{6}$$

sujeito a:

$$\sum_{j=1, i \neq j}^{n} x_{ij} = 1, \ \forall i \tag{7}$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \ \forall j \tag{8}$$

$$u_i - u_j + n x_{ij} \leq n - 1, \qquad 2 \leq i \neq j \leq n \tag{9}$$

$$u_i \leq n - 1, \qquad \forall i = 2, \ldots, n \tag{10}$$

$$x_{ij} \in \{0; 1\}, \qquad i \in N \tag{11}$$

Com este modelo, o objetivo em (6) é minimizar o custo total do ciclo, assim como no CSP (que busca minimizar o custo total do ciclo que conecta todos os vértices de cobertura). Os conjuntos de restrições em (7) e (8) garantem que uma cidade utilizou um caminho de saída e de chegada até ela apenas uma vez durante todo o ciclo. Os conjuntos de restrições (9) e (10) são para eliminação de subciclos do grafo, conhecida como a formulação Miller-Tucker-Zemlin de eliminação de subciclos.

3.2 O Problema da Cobertura de Conjuntos

O problema de Cobertura de Conjuntos (SCP) pode ser descrito como um problema de cobrir as linhas de uma matriz de *m*-linhas, *n*-colunas (a_{ij}) contendo apenas 0 ou 1 por um subconjunto das colunas a um custo mínimo (BEASLEY, 1987).

Dessa forma, é possível defini-lo formalmente como: Seja $x_j = 1$ se a coluna *j* (de custo c_j) esteja na solução e $x_j = 0$ caso contrário.

$$\text{minimizar} \sum_{j=1}^{n} c_j x_j \tag{12}$$

sujeito a:

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1\,, \qquad i = 1, \ldots, m \tag{13}$$

$$x_j \in (0,1), \qquad j = 1, \ldots, n. \tag{14}$$

O objetivo em (11) é minimizar o custo para cobrir todas as linhas da matriz utilizando os subconjuntos contidos nas colunas. Caso os subconjuntos não tenham custo (que é o caso do CSP), a variável c_j que tem o custo da coluna *j* recebe o valor 1, sendo assim, o objetivo se torna minimizar a quantidade de subconjuntos selecionados para cobrir as linhas da matriz. As restrições em (12) garantem que cada linha é coberta por pelo menos uma coluna selecionada. E em (13) estão as restrições de integralidade.

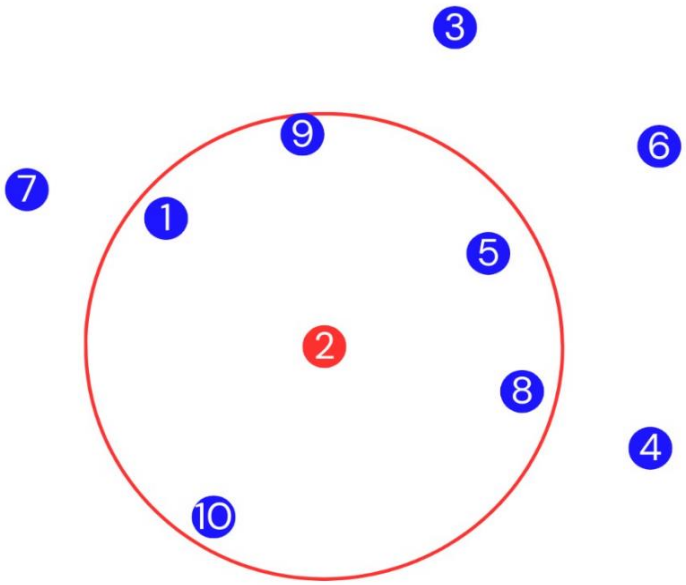
4 ABORDAGENS UTILIZADAS

A abordagem utilizada para resolver o CSP, foi separar o problema nos problemas clássicos de otimização combinatória SCP e TSP, com a intenção de aplicar métodos baseados em PLI, heurísticos e meta-heurísticos separadamente em cada um deles.

4.1 Relacionando o SCP ao CSP

No CSP, é possível imaginar um conjunto U contendo todos os vértices do grafo, podendo ser enumerados de 1 a N. Um vértice pode cobrir todos os outros vértices que estejam dentro de sua área de cobertura, que é representada por uma distância máxima predefinida para cada vértice ou um critério de cobertura diferente. Todos os vértices podem se tornar vértices de cobertura, fazendo assim, parte do ciclo. Portanto, cada vértice i do grafo pode ser representado como um subconjunto S_i, que contém os vértices por ele cobertos. A Figura 3 apresenta um exemplo de grafo com vértices enumerados de 1 a 10, mostrando a área de cobertura do vértice 2.

Figura 3 - exemplo de cobertura do vértice 2 em um grafo com 10 vértices

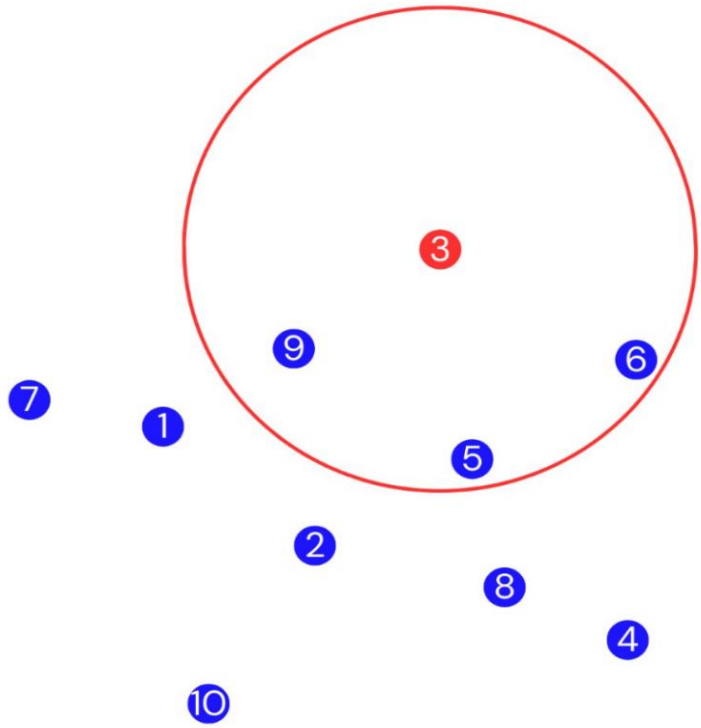


Fonte: Elaborada pelo autor

Na Figura 2, existe um grafo com 10 vértices, cada um com sua área de cobertura predefinida. O vértice 2, ao ser selecionado como vértice de cobertura, é representado como um subconjunto S_2 de elementos cobertos por ele: $S_2 = \{2, 1, 5, 8, 9, 10\}$, onde o primeiro elemento do subconjunto sempre será o próprio vértice de cobertura 2, pois ele cobre a si mesmo.

Dessa mesma forma, é possível criar um subconjunto S_3 utilizando o vértice 3 como vértice de cobertura, assim como qualquer outro. Como mostrado a seguir pela Figura 4, que apresenta o subconjunto $S_3 = \{3, 6, 5, 9\}$.

Figura 4 - exemplo de cobertura do vértice 3 em um grafo com 10 vértices



Fonte: elaborada pelo autor

Existirá, então, um conjunto total U contendo todos os 10 vértices, e 10 subconjuntos S_i $\{i = 1, 2, \ldots, 10\}$ (um para cada vértice), que conseguem cobrir o conjunto total. O problema então se torna um Problema de Cobertura de Conjuntos em que o objetivo é cobrir todos os vértices do grafo representados pelo conjunto U. No entanto, não existe um custo para selecionar um subconjunto S_i como no problema original do SCP, e, portanto, se deseja cobrir todos os vértices do grafo utilizando o mínimo de subconjuntos possível. Lembrando que ao selecionar um

subconjunto, seleciona-se, na verdade, o vértice que cobre todo aquele subconjunto.

A partir da cobertura de cada subconjunto, é possível então montar uma matriz binária de cobertura, como explicado anteriormente na seção 3.2 ao definir do problema CSP. Para o exemplo de 10 vértices, uma matriz dada por I_{10x10} em que I_{vu} será igual a 1 se, e somente se, o vértice v é coberto pelo vértice u. Como mostrado a seguir na Figura 5.

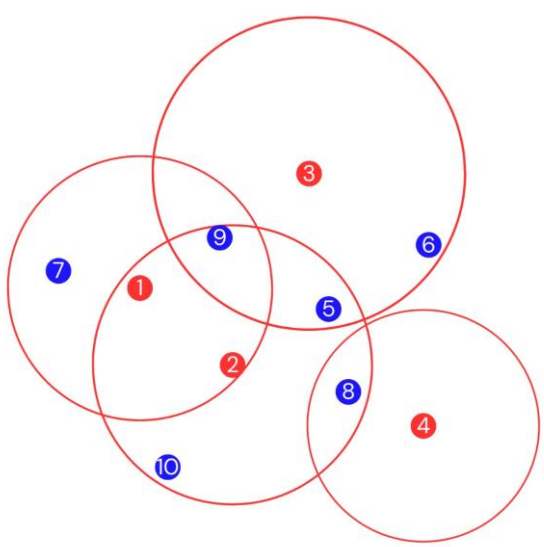
Figura 5 - Matriz de cobertura dos subconjuntos do exemplo com 10 vértices

	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	1	0	1	0
2	1	1	0	0	1	0	0	1	1	1
3	0	0	1	0	0	0	0	0	1	0
4	0	0	0	1	0	0	0	1	0	0
5	0	1	1	0	1	1	0	1	1	0
6	0	0	1	0	1	1	0	1	0	0
7	1	0	0	0	0	0	1	0	0	0
8	0	1	0	1	1	0	0	1	0	0
9	1	1	1	0	1	0	0	0	1	0
10	0	1	0	0	0	0	0	0	0	1
total coberto	4	6	4	2	5	2	2	5	5	2

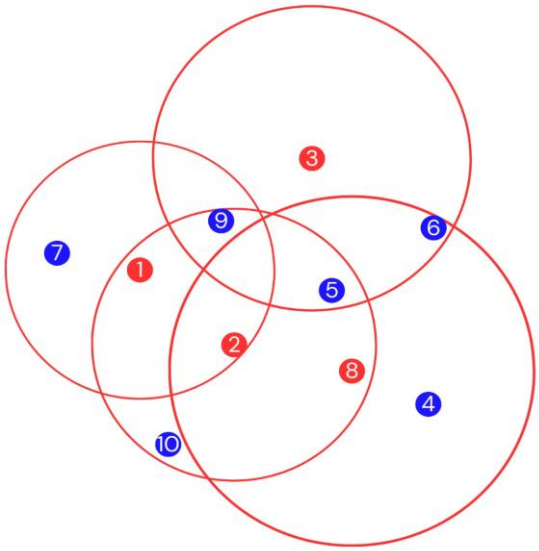
Fonte: Elaborada pelo autor

Para o exemplo representado pela matriz de cobertura da Figura 5, existem diversas soluções possíveis, sendo algumas delas, os vértices: 1, 2, 3, 4; 1, 2, 3, 8; ou 2, 3, 7, 8. Cada uma destas soluções gera uma cobertura completa do grafo com configurações diferentes, como mostrado na Figura 6.

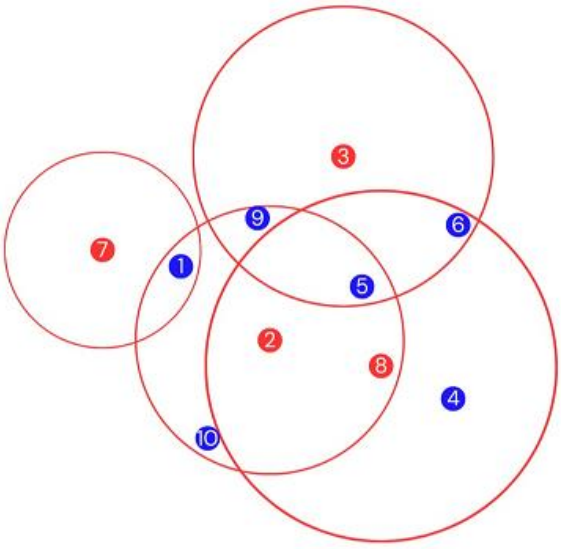
Figura 6 - Algumas soluções que geram cobertura máxima para o exemplo de 10 vértices. a) S_1, S_2, S_3, S_4, b) S_1, S_2, S_3, S_8 e c) S_2, S_3, S_7, S_8.



a)



b)



c)

Fonte: Elaborada pelo autor

A Figura 5 demonstra que, para cada instância do problema, podem existir diversas configurações de cobertura que cobrem todo o grafo, e o objetivo é encontrar uma configuração que utiliza o mínimo de vértices de cobertura possível que cubra todo o grafo. No entanto, quando a instância tem um número grande de vértices, existe um número imenso de soluções possíveis capazes de cobrir todo o grafo, tornando inviável a resolver o problema de forma ótima.

4.2 Relacionando o TSP ao CSP

Para resolver o CSP de forma exata, seria necessário gerar todas as soluções de SCP possíveis, ou seja, gerar todos os conjuntos de vértices, que, sendo escolhidos como vértices de cobertura, conseguem cobrir todo o grafo. Para então resolver um TSP apenas com os vértices de cobertura obtidos pelas soluções geradas para o SCP. Uma vez que uma dessas soluções gera um ciclo entre vértices de cobertura com o menor custo possível, sendo assim, a resposta final para o CSP. No entanto, pode ser necessário um subconjunto muito grande de vértices de cobertura para cobrir todo o grafo, e determinar a distância mínima entre esses vértices de cobertura se torna uma tarefa difícil para instâncias maiores, por conta da natureza não polinomial do TSP.

4.3 Solução Proposta

Como já dito no início da seção 3, o SCP e o TSP, bem como o CSP são NP-difíceis. O que torna inviável resolvê-los de forma exata em um tempo viável para instâncias grandes, como também reforçado pelas seções 4.1 e 4.2. Faz-se, então, necessário o uso de métodos heurísticos e aproximados para encontrar uma solução viável em um tempo de execução aceitável.

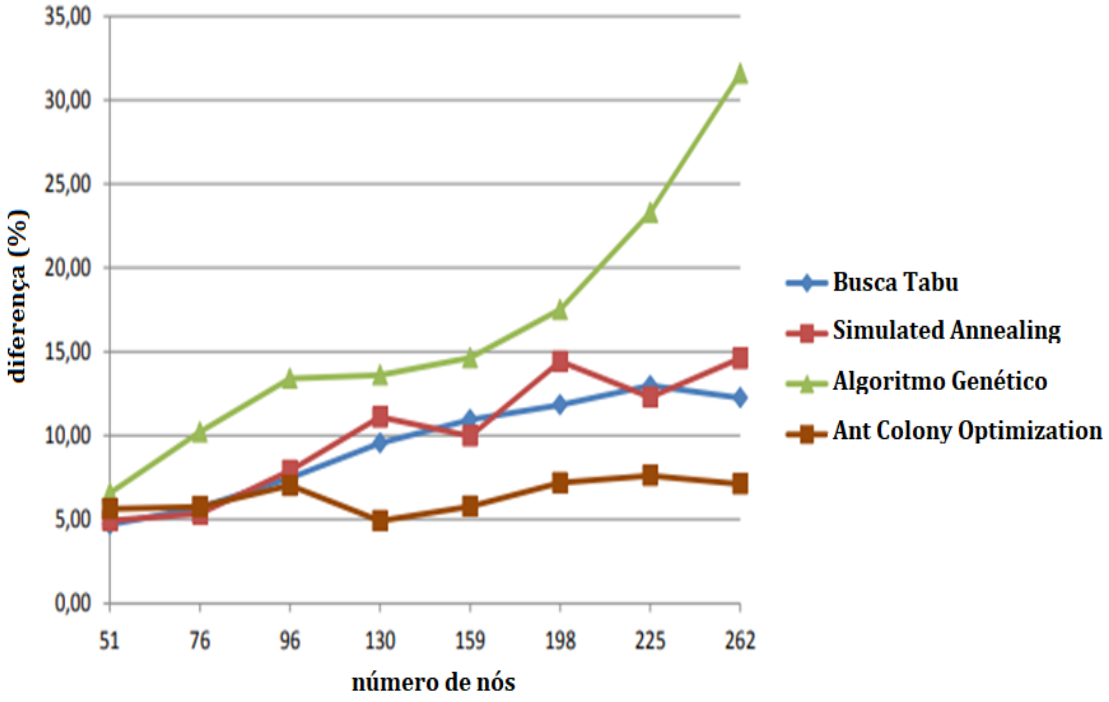
A estratégia proposta, é dividir o CSP em duas etapas, que podem ser representadas pelo SCP e pelo TSP. O SCP será resolvido de forma aproximada primeiramente, com o objetivo de utilizar o mínimo de vértices de cobertura possível para cobrir todo o grafo, imaginando que uma quantidade menor de vértices de cobertura poderia encontrar os vértices mais centralizados e com uma menor distância total entre eles. Com o grafo inteiro coberto pela configuração de subconjuntos de vértices de cobertura encontrada pela parte responsável pelo SCP, as restrições de cobertura do CSP serão satisfeitas. Em seguida, com os vértices de cobertura selecionados, será resolvido um TSP, também de forma aproximada, buscando o caminho mínimo entre os vértices de cobertura.

Para este trabalho, os algoritmos Relaxação Lagrangiana, Heurística Gulosa e Busca Tabu foram utilizados para resolver o SCP. E para o TSP, foi utilizada a meta-heurística ACO.

Os métodos para resolver a primeira etapa foram escolhidos por resolverem bem o SCP, e serem utilizados frequentemente em hibridizações e etapas iniciais para métodos mais poderosos, como em LU et al. (2021) que apresenta um algoritmo híbrido para o CSP que utiliza Busca Tabu em suas etapas iniciais, e em BEASLEY (1990), e em diversos artigos criados por ele, que utiliza a Relaxação Lagrangiana como base para resolver o SCP.

A meta-heurística ACO foi escolhida por ser a meta-heurística que resolve o TSP com a menor diferença entre a solução por ela encontrada e a solução ótima, como mostrado pelo estudo comparativo entre meta-heurísticas feito por ONDŘEJ (2015), e representado pela Figura 7, em que as soluções dadas pelo ACO se mantém com uma diferença de menos de 10% da melhor solução para a instância, mesmo com o aumento do número de vértices.

Figura 7 - diferença entre solução ótima e solução dada por heurísticas para instâncias do TSP.



Fonte: adaptado de ONDŘEJ (2015).

As instâncias do CSP, têm como entrada um conjunto U contendo todos os vértices, uma matriz de distância dentre todos eles e um conjunto S com todos os subconjuntos, em que S_i $(i \in U)$ representa o subconjunto de vértices cobertos pelo vértice i. Lembrando que o vértice i sempre estará contido e será o primeiro elemento do subconjunto S_i. Esses conjuntos são utilizados como entrada nas 3

implementações dos algoritmos para resolver o SCP, assim como a matriz de distância é adaptada e utilizada como entrada pelo ACO para resolver o TSP.

4.3.1 Algoritmo de Relaxação Lagrangiana para resolver o SCP

O algoritmo de Relaxação Lagrangiana foi adaptado para o SCP sem custos nos subconjuntos, com base no Algoritmo 1. Porém, como visto na seção 2.2.1, a Relaxação Lagrangiana irá "relaxar" as restrições do problema, gerando assim soluções que podem violar essas restrições. Para contornar isso, é feita uma verificação ao final do algoritmo que irá complementar a solução caso necessário, para que nenhuma restrição seja violada.

Na implementação para o SCP, as restrições são relaxadas através da introdução de multiplicadores lagrangianos, onde cada restrição é um multiplicador, representado por um valor real não negativo, eles serão ajustados durante o processo de otimização para buscar a solução ótima. O algoritmo tem como entrada além de *S* e *U*, um número máximo de iterações e um valor *alfa*, que é o parâmetro que controla o equilíbrio da função objetivo original e as penalidades que serão associadas às restrições.

O algoritmo começa inicializando os multiplicadores lagrangianos, que serão representados por um vetor *lambdas* (de tamanho de *U*), com cada posição tendo um valor inicial de 1.0 para cada subconjunto. A solução parcial é inicializada como um conjunto vazio, e é denominada *solucaoPrimal*. O vetor de conjuntos *subconjuntosSelecionados* representam os subconjuntos utilizados para cobrir todos os vértices. Um conjunto *verticesNaoCobertos* é criado inicialmente também contendo todo o conjunto *U*.

É inicializado então um *loop*, que irá iterar até que uma solução seja encontrada ou até que se alcance o número máximo de iterações, dada como parâmetro. Neste *loop*, é criado um vetor denominado *custosReduzidos* em que seus valores são uma medida que reflete a contribuição de cada subconjunto na cobertura do conjunto original. os custos reduzidos são calculados como a diferença entre o tamanho de $S_i \cap verticesNaoCobertos$ e o somatório dos multiplicadores lagrangianos correspondentes aos vértices desse subconjunto. Essa diferença reflete o "custo efetivo" de adicionar o subconjunto à solução parcial, considerando a quantidade de elementos ainda não cobertos por ela.

Em seguida, seleciona-se de *custosReduzidos* o índice do subconjunto com menor custo reduzido, e este será o subconjunto escolhido da iteração atual para fazer parte da solução. *verticesNaoCobertos* é atualizado removendo os vértices do subconjunto selecionado, e *solucaoPrimal* é atualizada adicionando os vértices cobertos pelo subconjunto selecionado, este subconjunto é adicionado à *subconjuntosSelecionados*. Caso a *solucaoPrimal* seja igual à *U* a execução irá parar. Em seguida são atualizados os multiplicadores de Lagrange para a próxima iteração, utilizando alfa e a diferença da solução parcial em relação ao conjunto original.

No entanto, com esta configuração o algoritmo não garante uma solução que cubra todos os vértices, por relaxar a restrição de cobertura do SCP, por isso, é necessário também uma verificação ao final do algoritmo, e caso a solução não consiga cobrir o conjunto total, um algoritmo guloso selecionaria os vértices que mais cobrem elementos não cobertos e os adicionaria à solução. O Algoritmo 5 mostra como a implementação do algoritmo de Relaxação Lagrangiana foi feita para o SCP.

Algoritmo 5: Relaxação Lagrangiana para o SCP

Entrada: *S, U, maxIter, alfa*

Saída: *solucaoPrimal*

Início

solucaoPrimal ← ∅;

verticesNaoCobertos ← *U;*

$lambdas_i$ ← *1.0* ***para cada*** *i* ∈ *U;*

Para *k = 0* ***até*** *MaxIter* ***faça***

custosReduzidos ← ∅;

Para cada S_i ***em*** *S* ***faça***

$custo \leftarrow S_i \cap verticesNaoCobertos - \sum_{j \in S_i} lambdas_j$;

Adiciona custo à custosReduzidos;

Fim

indexSelecionado ←*index do menor valor de custosReduzidos;*

selecionado ← $S_{indexSelecionado}$;

adicione selecionado à subconjuntosSelecionados;

Atualize a solucaoPrimal com o selecionado;

Atualize verticesNaoCobertos com o selecionado;

Se *solucaoPrimal = U* ***então***

Pare as iterações;

Fim

d ← $tamanho\ da\ solucaoPrimal$ *- tamanho de U;*

$lambdas \leftarrow l + alfa \times d$ ***para cada*** *l em lambdas;*

Fim

Enquanto *solucaoPrimal < U* ***faça***

Acidione à solucaoPrimal subconjunto S_i *que mais cobre verticesNaoCobertos;*

verticesNaoCobertos ← *verticesNaoCobertos - vértice selecionado;*

Fim

Retorna primeiras posições do vetor de conjuntos subconjuntosSelecionados;

Fim

4.3.2 Algoritmo heurístico guloso para resolver o SCP

A heurística gulosa foi implementada conforme o Algoritmo 2, da sessão 2.3.1. E como dito nesta mesma sessão, o algoritmo faz escolhas localmente ótimas com a esperança de encontrar a solução ótima global.

Para o algoritmo heurístico guloso então, cada etapa é representada pela iteração de um *loop*, que só terá a condição de parada satisfeita se, e somente se,

todo o conjunto de vértices *U* for coberto, ou seja, *verticesNaoCobertos* for vazio. Nesse mesmo *loop*, o objetivo é buscar o subconjunto S_i que cobre mais vértices ainda não cobertos, adicionando-o à variável *melhorSubconjunto*. Ao final do *loop*, o vértice de cobertura contido em *melhorSubconjunto* é adicionado ao conjunto *verticesDeCobertura*, e o conjunto *verticesNaoCobertos* é atualizado, removendo dele, os vértices cobertos pelo *melhorSubconjunto*. O Algoritmo 6 apresenta uma versão adaptada do Algoritmo 2, para resolver o SCP.

Algoritmo 6: Heurística Gulosa para o SCP

```
Entrada: S, U
Saída: verticesDeCobertura
Início
    verticesDeCobertura ← ∅;
    verticesNaoCobertos ← U;
    Enquanto verticesNaoCobertos ≠ ∅ faça
        melhorSubconjunto ← ∅;
        melhorCobertura ← -∞;
        coberturaAtual ← ∅;
        Para cada S_i em S faça
            coberturaAtual ← verticesNaoCobertos ∩ S_i;
            Se coberturaAtual > melhorCobertura então
                melhorSubconjunto ← S_i;
                melhorCobertura ← coberturaAtual;
            Fim
        Fim
        verticesDeCobertura ← verticesDeCobertura + primeiro vértice de
        melhorSubconjunto;
        verticesNaoCobertos ← verticesNaoCobertos - melhorCobertura;
    Fim
    Retorne verticesDeCobertura;
Fim
```

4.3.3 Algoritmo Busca Tabu para resolver o SCP

O algoritmo da busca tabu, mostrado no Algoritmo 3, também foi adaptado para resolver o SCP com foco na minimização da quantidade de subconjuntos em *S* necessários para cobrir todo o conjunto *U*.

Como entrada, além de S e U, existem alguns parâmetros que necessitam ser predefinidos, como o tamanho da Lista Tabu definido denominado *tamanhoLT*, e o número máximo de iterações, definido como *maxIter*. O algoritmo funciona com um *loop* que executa até o máximo de iterações. A cada iteração, é criado um vetor de candidatos, que será populado com os conjuntos S_j para todo vértice de cobertura j que tem $S_j \cap verticesNaoCobertos \neq \emptyset$ e que não esteja na *listaTabu* (estrutura que impedirá que qualquer elemento contido nela faça parte da solução). É selecionado de forma probabilística um dos subconjuntos do vetor *candidatos*, em que a probabilidade de um subconjunto ser selecionado é maior caso ele cubra mais vértices do grafo, e menor caso contrário. Ao selecionar aleatoriamente este candidato, ele agora fará parte da solução. Em seguida este subconjunto selecionado é adicionado à lista tabu, que caso esteja cheia, é removido seu elemento mais antigo. O conjunto de vértices não cobertos é atualizado, removendo os vértices agora cobertos pelo novo subconjunto.

Caso *verticesNaoCobertos* seja vazio, uma solução foi encontrada. Ela será tida como a melhor solução se a quantidade de subconjuntos utilizada para cobrir todos os vértices, for menor do que a melhor cobertura até então. Assim que uma solução é encontrada, ela é verificada para descobrir se é a melhor de todas, e a cobertura é reinicializada, porém mantendo os elementos da lista tabu, buscando fugir de um ótimo local escolhendo diferentes vértices de cobertura.

Existe também a função de aspiração, que é baseada em uma probabilidade de que um elemento que esteja na lista tabu e consegue cobrir muito bem o conjunto *verticesNaoCobertos* pode ser removido da mesma e usado como solução novamente. A resposta final do algoritmo será a melhor solução encontrada durante toda sua execução. O Algoritmo 7 mostra a implementação da busca tabu adaptada para o SCP.

Algoritmo 7: Busca Tabu para o SCP

Entrada: *S, U, maxIter, tamanhoLT*

Saída: *melhorResposta*

Início

coberturaAtual ← ∅;

tamanhoAtual ← 0;

verticesNaoCobertos ← U;

melhorResposta ← ∅;

melhorTamanho ← 0;

listaTabu ← ∅;

Para *i = 0* ***até*** *MaxIter* ***faça***

candidatos ← ∅;

Para cada S_j ***em*** *S* ***faça***

Se $verticesNaoCobertos \cap S_j \neq \emptyset$ ***e*** S_j *não pertence à listaTabu* ***então***

candidatos ← candidatos + S_j*;*

Fim

Fim

subconjuntoSelecionado ← algum subconjunto aleatório a partir das regras de probibalidade definidias;

coberturaAtual ← coberturaAtual + subconjuntoSelecionado;

tamanhoAtual ← tamanhoAtual + 1;

verticesNaoCobertos ← verticesNaoCobertos - subconjuntoSelecionado;

Adiciona subconjuntoSelecionado à listaTabu;

Se *listaTabu > tamanhoLT* ***então***

Remove elemento mais antigo da listaTabu;

Fim

Se $verticesNaoCobertos = \emptyset$ ***então***

Se *tamanhoAtual < melhorTamanho* ***então***

melhorResposta ← primeiras posições dos vetores de coberturaAtual;

melhorTamanho ← tamanhoAtual;

Fim

Reinicialize os valores decoberturaAtual, tamanhoAtual e verticesNaoCobertos;

Fim

Função de aspiração tem chance de remover algum subconjunto da lista tabu caso ele seja um subconjunto muito importante de cobertura;

Fim

Retorne melhorResposta;

Fim

4.3.4 Algoritmo da Colônia de Formigas para o TSP

Com os vértices de cobertura encontrados por algum dos algoritmos apresentados nas seções anteriores, agora, é necessário resolver um TSP para determinar o ciclo de custo mínimo entre eles, resolvendo assim o CSP.

Como as entradas dadas pelas instâncias do CSP têm uma matriz de distância entre todos os vértices, e para o TSP, só será resolvido com vértices de cobertura, é necessário criar outra matriz de distância apenas entre estes vértices, que serão obtidos pela parte responsável por resolver o SCP. Com essa matriz criada, considera-se o grafo apenas com os vértices de cobertura como um grafo completo.

Para inicializar o algoritmo, serão utilizados como parâmetros a matriz de distância entre os vértices de cobertura *matDist*, é necessário também definir um número de formigas *numFormigas*, assim como um número máximo de iterações *maxIter*.

Existem também alguns parâmetros que influenciam no comportamento e desempenho do algoritmo, o parâmetro α por exemplo, representa o coeficiente de trilha de feromônio, e controla o equilíbrio entre a influência do feromônio depositado e a informação de visibilidade na escolha dos próximos vértices. Um valor alto de α dá mais peso ao feromônio, fazendo com que as formigas sejam mais atraídas pelas trilhas de feromônio existentes, e um valor baixo de α dá mais peso à informação de visibilidade, fazendo com que as formigas considerem mais as distâncias entre os vértices.

O parâmetro β, também conhecido como coeficiente de informação de visibilidade, controla o equilíbrio entre a influência do feromônio depositado e a informação de visibilidade na escolha dos próximos vértices. Valores mais dão mais peso à informação de visibilidade, fazendo com que as formigas escolham vértices com distâncias menores, e valores mais baixos fazem com que as formigas sigam mais trilhas de feromônio.

O parâmetro ρ representa a taxa de evaporação de feromônio, e controla a taxa na qual o feromônio evapora ao longo do tempo. Um valor alto de ρ indica uma taxa de evaporação mais baixa, o que significa que as trilhas de feromônio permanecem por mais tempo, permitindo uma exploração mais intensa das

soluções encontradas, e um valor baixo de ρ indica uma taxa de evaporação mais alta, o que faz com que as trilhas de feromônio desapareçam rapidamente, permitindo uma exploração mais diversificada do espaço de busca.

E por fim o parâmetro *Q*, que determina a quantidade de feromônio depositado pelas formigas ao percorrerem uma aresta.

Com os parâmetros predefinidos, o algoritmo inicializa criando duas variáveis *numVertices* e *numArestas* que definem o número de vértices e o número de arestas (como é um grafo completo, o número de arestas é igual ao número de vértices menos um) respectivamente. Cria-se então uma matriz de feromônio $\tau_{numVertices\ x\ numVertices}$ com um valor igual para todas as arestas, sendo que τ_{ij} representa quanto feromônio está depositado na aresta entre o vértice *i* e o vértice *j*. É feito então, um *loop* principal que itera até *maxIter*. A cada iteração, é criada uma matriz *formigas* que armazena as trilhas de feromônios deixadas pelas formigas, em que cada linha representa uma formiga, e cada coluna representa uma aresta do TSP. Outro *loop* é criado, que irá iterar até *numFormigas*, com cada iteração representando uma formiga. Para cada formiga é realizada a construção de uma solução usando a regra de transição probabilística.

Cada solução é construída da seguinte forma: Uma formiga é posicionada em um vértice aleatório do grafo. Esse vértice é marcado como visitado, assim como todos os outros que a formiga passar. É calculada então, para cada vértice adjacente ao vértice atual que não tenha sido visitado, a probabilidade de que a formiga escolha a aresta que leva até ele, utilizando a regra de transição probabilística.

Na regra de transição probabilística, cada formiga avalia os vértices vizinhos disponíveis com base em duas informações cruciais, o feromônio depositado nas trilhas, e a informação de visibilidade. Para cada vértice atual *i,* será criado um conjunto *P*, em que P_j representa a probabilidade de uma formiga que está em *i* escolher a aresta que leva ao vértice *j*. P_j será calculado pela Equação 15:

$$P_j = ({\tau_{ij}}^{\alpha}) \times ((\frac{1}{MatDist_{ij}})^{\beta}) \quad (15)$$

A informação de visibilidade utilizada foi o inverso da distância entre o vértice *i* e *j*, sendo elevado a β, que é o parâmetro que controla a influência da informação de visibilidade. E τ_{ij} representa a quantidade de feromônio presente na

trilha entre o vértice *i* e *j*, sendo elevada a α que é o parâmetro que controla a influência do feromônio na escolha do próximo vértice. Ambos são multiplicados, combinando a influência da informação de visibilidade com a influência do feromônio para obter a probabilidade de transição para o vértice vizinho *j*.

Com todas as probabilidades calculadas para todos os vértices vizinhos não visitados, seus valores são normalizados com a Equação 16:

$$P_j = \frac{P_j}{\sum_{j \in numVertices} P_j} \tag{16}$$

A normalização é feita dividindo cada probabilidade pela soma de todas as probabilidades. Essa operação as distribui em uma escala relativa, garantindo que a soma resultante seja igual a 1.

Agora, com as probabilidades calculadas e normalizadas, a formiga escolhe aleatoriamente qual trilha seguir, utilizando *P* como pesos para a escolha aleatória, o que garante que a escolha seja feita de acordo com essas probabilidades.

Após a formiga escolher entre os vértices, a informação de que a formiga *f* que está em *i* escolheu o próximo vértice *j* é armazenada em $formigas_{fi}$, e o vértice atual, é atualizado de *i* para *j*, que é marcado também como um vértice visitado. Agora em *j*, a formiga irá repetir o procedimento, até que visite todos os vértices do grafo, criando assim, um ciclo hamiltoniano, ou seja, uma solução para o problema.

Assim que todas as formigas tiverem criado ciclos hamiltonianos para a iteração atual, o custo de cada ciclo é calculado a partir da matriz *formigas* e armazenado em *custos*, em que o custo do ciclo de uma formiga *f* é dado por $custos_f$. Os custos são verificados para analisar se algum deles tem custo melhor do que a melhor solução encontrada até agora. Caso sim, tanto o ciclo quanto seu custo são atualizados como a melhor solução.

Ao final do *loop*, é necessário atualizar as trilhas de feromônios, tanto com a evaporação, quanto com os feromônios deixados pelas formigas. Na evaporação, cada τ_{ij} que representa a quantidade de feromônio na aresta de *i* para *j*, é multiplicado pelo fator de evaporação (1.0 - ρ). Esse fator de evaporação (normalmente um valor entre 0 e 1) controla a taxa de evaporação do feromônio. A evaporação permite que o feromônio seja atualizado de forma dinâmica, com um

decaimento controlado, assim equilibrando a exploração e a intensificação do espaço de busca.

Por fim, para atualizar as trilhas deixadas pelas formigas, cada caminho contido na matriz *formigas* é percorrido, atualizando cada aresta de *i* para *j* da matriz τ. Sendo $\Delta\tau_{ij}$ a quantidade antiga de feromônio contida na aresta (*i, j*), Q a quantidade de feromônio deixado por cada formiga e a qualidade da solução da formiga, representada pela distância entre *i* e *j*. Cada posição da matriz τ é atualizada conforme a equação 17:

$$\tau_{ij} = \Delta\tau_{ij} + \left(\frac{Q}{matDist_{ij}}\right) \quad (17)$$

O *loop* principal irá repetir até *maxIter*, e ao final da execução, a melhor solução encontrada pelas formigas será a melhor solução para o TSP. O Algoritmo 8 mostra o funcionamento do código do ACO implementado para resolver o TSP.

Algoritmo 8: Ant Colony Optimization para o TSP

Entrada: *matDist, numFormigas, maxIter, TamanhoLT*

Saída: *melhorResposta, melhorCaminho*

Início

numVertices ← quantidade de linhas matDist;

numArestas ← numVertices - 1;

$\tau_{numVertices\ x\ numVertices}$ *← 1.0 para todas as posições;*

Para *i = 0* ***até*** *MaxIter* ***faça***

$formigas_{\ numFormigas\ x\ numArestas} \leftarrow 0\ para\ todas\ as\ posições;$

Para cada *f* ***em*** *formigas* ***faça***

verticeAtual ← vértice aleatório do grafo;

visitados ← 0 para todas as posições;

$visitados_{verticeAtual} \leftarrow 1;$

$formigas_{f,0} \leftarrow verticeAtual;$

Para cada *v* ***em*** *numVertices* ***faça***

Para cada *j* ***em*** *numVertices* ***faça***

Se *j não foi visitado* ***então***

P_j *← cálculo das probabilidades de a formiga ir de v para j;*

Fim

Fim

Normalize os valores de P;

proximoVertice ←vértice aleatório de P;

$formigas_{f,v} \leftarrow proximoVertice;$

$visitados_{proximoVertice} \leftarrow 1;$

verticeAtual ← proximoVertice

Fim

Fim

Atualize a melhorResposta com a melhor solução criada pelas formigas nesta Iteração caso o valor dela seja menor que o valor de melhorResposta;

τ ← τ(1.0 – ρ) para todas as posições de τ;*

Para cada *f* ***em*** *formigas* ***faça***

faça a atualização de feromônio do caminho feito pela formiga f na matriz τ

Fim

Fim

Retorne melhorResposta, melhorCaminho;

Fim

5 EXPERIMENTOS COMPUTACIONAIS

Neste capítulo são descritos os experimentos computacionais realizados a fim de comparar as abordagens elaboradas neste trabalho para resolver o CSP com as abordagens da literatura. Os algoritmos propostos foram implementados utilizando a linguagem Python em sua versão 3.11.1. Os experimentos foram executados em um computador com 32GB de memória RAM e com processador intel i5-9600k, 3.7GHz sobre o sistema operacional Windows (64 bits).

5.1 Instâncias para o CSP

As instâncias foram retiradas de TSPLIB (Reinelt, 1991) e utilizadas por GOLDEN et al. (2012), entre outros autores. Em cada uma delas, é dada a matriz de distância, assim como os subconjuntos de cobertura, no entanto, para estas instâncias, cada vértice de cobertura só pode cobrir seus NC vértices mais próximos. Portanto os conjuntos de cobertura terão seus tamanhos determinados por NC. No total são 19 instâncias, sendo 16 com a quantidade de vértices variando entre 51 e 200 e o valor de NC variando entre 7, 9 e 11, e 3 instâncias com o número de vértices entre 535 e 724, com seu NC variando entre 3, 5 e 7.

5.2 Algoritmo para fins de comparação

Para comparar os resultados obtidos deste trabalho, será utilizado um algoritmo baseado em busca local LS2, criado por GOLDEN et al. (2012), sua abordagem tenta melhorar uma solução deletando um vértice do ciclo e verificando se a solução ainda é viável, ou removendo um vértice e substituindo por uma sequência de vértices promissora. As respostas obtidas pelo LS2, foram retiradas de GOLDEN et al. (2012), para instâncias de 51 a 200 vértices, e retiradas de SALARI; NAJI-AZIMI, (2015), para instâncias de 535 a 724 vértices.

5.3 Inicialização dos parâmetros

Os parâmetros para os algoritmos utilizados neste trabalho foram definidos baseado na utilização destes métodos para resolver o TSP, com instâncias

proporcionais as utilizadas neste trabalho (GENDREAU; POTVIN, 2010), como apresentados nas seções 5.3.1 a 5.3.3.

5.3.1 Relaxação Lagrangiana

Para o algoritmo de Relaxação Lagrangiana (Algoritmo 5), foram utilizados um valor máximo de iterações igual a 200, o valor de *alfa* igual à 0.1 e os multiplicadores lagrangianos inicializados em 1.0.

5.3.2 Busca Tabu

Para a Busca Tabu (Algoritmo 7), foram utilizados uma lista tabu de tamanho 50, um critério de aspiração com probabilidade, igual a 50%, de que o elemento seja removido da lista tabu e um número máximo de iterações igual a 200.

5.3.3 Ant Colony Optimization

Para o algoritmo ACO (Algoritmo 8), os parâmetros foram definidos com o número de formigas igual a 100, número máximo de iterações igual a 200, uma taxa de evaporação (ρ) igual a 0.2, o peso do feromônio (α) igual a 0.5, peso da informação de visibilidade (β) igual a 1.0 e feromônio deixado pela formiga (Q) igual a 1.0.

5.4 Resultados

Para a apresentação dos resultados, foram criadas 3 tabelas, com objetivo de comparar o desempenho dos algoritmos elaborados deste trabalho, com o desempenho do algoritmo da literatura LS2 (GOLDEN et al., 2012), apresentado na seção 5.2. A primeira coluna de cada tabela contém o nome da instância, que inclui o número de vértices. A segunda coluna (NC) fornece o número de vértices mais próximos que podem ser cobertos por cada vértice. Para cada método, existem as colunas: Melhor Custo, NB e TT médio, que representam a melhor solução encontrada pelo algoritmo para aquela instância em 5 execuções, assim como o feito por GOLDEN et al. (2012) nos testes com o LS2, o número de vértices de

cobertura presentes na melhor solução, e o tempo total médio das 5 execuções, respectivamente.

No final de cada tabela, existem duas colunas. A primeira, contendo o *gap* que representa o aumento percentual do melhor custo encontrado pelo algoritmo elaborado para este trabalho, em relação ao LS2. A segunda coluna, contendo o *gap* que representa a redução percentual da quantidade de vértices de cobertura na melhor resposta do algoritmo elaborado para este trabalho, em relação a quantidade de vértices na melhor resposta do LS2. E na última linha de cada tabela, existe a média de ambos os *gaps*. Em cada linha, a melhor solução para a instância do CSP está destacada em negrito.

A Tabela 1 compara o desempenho do algoritmo Relaxação Lagrangiana+ACO (RL+ACO) descrito nas seções 4.3.1 e 4.3.4, em relação ao LS2. Ao analisar os resultados, percebe-se que o LS2 foi superior em todas as instâncias, obtendo custos menores que o RL+ACO. Por conta disso, os valores dos *gaps* de melhor custo foram sempre maiores que zero e, avaliando todas as instâncias, a média desses *gaps* foi de 5,4%, ou seja, a média dos melhores custos encontrados pelo RL+ACO foi 5,4% pior que a média dos melhores custos encontrados pelo LS2. Os maiores *gaps* de custos ocorreram nas instâncias rat575 e u724. Sendo que o maior *gap* (22,3%) ocorreu na instância u724, com NC igual à 7 e o menor *gap* (0,2%) ocorreu na instância KroA100, com NC igual à 7. Os *gaps* de NB, por outro lado, apresentaram uma redução média de 8,3%, o que significa que as soluções encontradas pelo RL+ACO utilizaram em média 8,3% menos vértices de cobertura em suas soluções. Com o maior *gap* sendo de 25,8%, o que representa uma redução de 25,8% na quantidade de vértices de cobertura, ocorrendo na instância KroB200 com NC igual à 9. E os menores *gaps* sendo 0%, o que significa que as soluções obtidas pelo RL+ACO e pelo LS2 utilizaram a mesma quantidade de vértices de cobertura em sua solução. Essa situação ocorreu para 13 instâncias.

Tabela 1 – comparação de resultados do LS2 e da Relaxação Lagrangiana+ACO para o CSP

		LS2			Relaxação Lagrangiana+ACO				
Instâncias	NC	Melhor Custo	NB	TT médio (s)	Melhor Custo	NB	TT médio (s)	*Gap* Melhor Custo (%)	*Gap* NB (%)
Eil51	7	**164**	10	1.05	185	9	2.04	12.8	10.0
	9	**159**	9	0.78	163	8	1.92	2.5	11.1
	11	**147**	7	0.87	148	6	1.76	0.7	14.3
Berlin52	7	**3887**	11	1.05	3976	10	1.91	2.3	9.1
	9	**3430**	7	0.95	3442	7	2.12	0.3	0.0
	11	**3262**	6	0.55	3314	6	2.01	1.6	0.0
St70	7	**288**	11	1.48	304	11	2.73	5.6	0.0
	9	**259**	10	1.65	283	10	2.40	9.3	0.0
	11	**247**	10	1.37	272	9	2.13	10.1	10.0
Eil76	7	**207**	15	1.35	210	15	2.81	1.4	0.0
	9	**185**	12	1.43	191	11	2.72	3.2	8.3
	11	**170**	11	1.50	178	9	2.29	4.7	18.2
Pr76	7	**50275**	14	1.89	50813	14	3.03	1.1	0.0
	9	**45348**	12	1.50	45771	11	2.94	0.9	8.3
	11	**43028**	10	1.39	45092	8	2.87	4.8	20.0
Rat99	7	**486**	18	2.45	495	17	3.65	1.9	5.6
	9	**455**	15	2.45	469	15	3.15	3.1	0.0
	11	**444**	13	2.26	446	12	3.21	0.5	7.7
KroA100	7	**9674**	19	2.78	9690	17	3.42	0.2	10.5
	9	**9159**	15	2.30	9288	15	3.87	1.4	0.0
	11	**8901**	13	2.05	9109	13	3.15	2.3	0.0
KroB100	7	**9537**	20	2.53	10395	19	3.81	9.0	5.0
	9	**9240**	15	2.62	9538	15	3.42	3.2	0.0
	11	**8842**	13	2.48	8978	10	3.43	1.5	23.1
KroC100	7	**9723**	17	2.55	10329	16	3.74	6.2	5.9
	9	**9171**	13	2.42	9925	13	3.78	8.2	0.0
	11	**8632**	13	2.44	9115	12	3.62	5.6	7.7
KroD100	7	**9626**	20	2.36	10543	16	3.91	9.5	20.0
	9	**8885**	13	2.73	9498	13	3.42	6.9	0.0
	11	**8725**	13	2.54	9296	12	3.63	6.5	7.7
KroE100	7	**10150**	19	2.16	10228	16	4.15	0.8	15.8
	9	**8991**	14	2.39	9429	13	3.53	4.9	7.1
	11	**8450**	13	2.64	8861	11	3.64	4.9	15.4
Rd100	7	**3461**	18	2.30	3718	18	3.84	7.4	0.0
	9	**3194**	16	2.31	3554	15	3.67	11.3	6.3
	11	**2922**	13	1.97	2999	12	3.28	2.6	7.7
KroA150	7	**11423**	28	4.10	11898	26	4.71	4.2	7.1
	9	**10056**	26	3.53	10282	21	4.68	2.2	19.2
	11	**9439**	21	3.46	9802	19	5.17	3.8	9.5
KroB150	7	**11457**	30	4.30	11567	26	5.21	1.0	13.3
	9	**10121**	24	3.50	11285	23	4.42	11.5	4.2
	11	**9611**	21	3.69	10088	17	5.04	5.0	19.0
KroA200	7	**13285**	35	5.64	13898	33	10.65	4.6	5.7
	9	**11708**	28	5.12	12272	26	9.75	4.8	7.1
	11	**10748**	29	4.61	11148	25	10.71	3.7	13.8
KroB200	7	**13100**	36	5.42	14356	33	10.57	9.6	8.3
	9	**11900**	31	5.11	11968	23	9.43	0.6	25.8
	11	**10676**	30	5.01	12180	24	9.18	14.1	20.0
ali535	3	**1387**	182	60.00	1442	179	201.42	4.0	1.6
	5	**1184**	122	60.00	1212	114	174.38	2.4	6.6
	7	**1094**	95	60.00	1125	92	188.74	2.8	3.2
rat575	3	**3755**	193	60.00	4151	192	235.13	10.5	0.5
	5	**3062**	133	60.00	3451	118	223.72	12.7	11.3
	7	**2667**	110	60.00	2779	91	230.75	4.2	17.3
u724	3	**25045**	230	60.00	29724	229	301.47	18.7	0.4
	5	**20563**	173	60.00	23295	152	274.51	13.3	12.1
	7	**18015**	130	60.00	22037	115	283.13	22.3	11.5
Média								5.4	8.3

Fonte: Elaborada pelo autor

A Tabela 2 apresenta a comparação dos algoritmos Busca Tabu+ACO (BT+ACO), apresentados nas seções 4.3.3 e 4.3.4, e do LS2. E para todas as instâncias testadas, o LS2 obteve resultados superiores ao BT+ACO, ou seja, os melhores custos encontrados pelo LS2 foram menores do que os encontrados pelo BT+ACO. Portanto, os *gaps* de melhor custo tiveram valores maiores que zero para todas as instâncias, com uma média de 6,5%, ou seja, a média dos melhores custos encontrados pelo BT+ACO foi 6,5% pior que a média dos melhores custos encontrados pelo LS2. Sendo o maior *gap* de 28,6%, encontrado na instância ali535 com NC igual à 5, o que significa que, para esta instância, a melhor solução encontrada pelo BT+ACO teve um custo 28,6% maior do que a solução encontrada pelo LS2. E o menor *gap* sendo 0,2%, encontrado na instância rat99 com NC igual à 11. Para o *gap* NB, o BT+ACO, obteve uma vantagem média de 7,8% em relação ao LS2, ou seja, para todas as instâncias, houve uma redução média de 7,8% da quantidade de vértices de cobertura utilizados nas melhores soluções de BT+ACO em relação ao LS2. Sendo a maior redução percentual 28,6%, obtida na instância Eil51 com NC igual à 11. E em 17 instâncias, o BT+ACO e o LS2 obtiveram respostas que utilizam a mesma quantidade de vértices, tendo um *gap* NB de 0%.

Tabela 2 – comparação de resultados do LS2 e da Busca Tabu+ACO para o CSP

		LS2			Busca Tabu+ACO				
Instâncias	NC	Melhor Custo	NB	TT médio (s)	Melhor Custo	NB	TT médio (s)	*Gap* Melhor Custo (%)	*Gap* NB (%)
Eil51	7	**164**	10	1.05	183	10	1.03	11.6	0.0
	9	**159**	9	0.78	172	8	1.12	8.2	11.1
	11	**147**	7	0.87	157	5	1.05	6.8	28.6
Berlin52	7	**3887**	11	1.05	3945	11	1.13	1.5	0.0
	9	**3430**	7	0.95	3494	7	1.05	1.9	0.0
	11	**3262**	6	0.55	3399	6	1.10	4.2	0.0
St70	7	**288**	11	1.48	310	11	1.28	7.6	0.0
	9	**259**	10	1.65	293	10	1.12	13.1	0.0
	11	**247**	10	1.37	278	10	1.21	12.6	0.0
Eil76	7	**207**	15	1.35	211	13	1.44	1.9	13.3
	9	**185**	12	1.43	191	12	1.35	3.2	0.0
	11	**170**	11	1.50	177	8	1.31	4.1	27.3
Pr76	7	**50275**	14	1.89	50491	13	1.58	0.4	7.1
	9	**45348**	12	1.50	45557	12	1.43	0.5	0.0
	11	**43028**	10	1.39	43610	8	1.52	1.4	20.0
Rat99	7	**486**	18	2.45	522	18	2.54	7.4	0.0
	9	**455**	15	2.45	466	13	2.43	2.4	13.3
	11	**444**	13	2.26	445	12	2.29	0.2	7.7
KroA100	7	**9674**	19	2.78	9778	16	2.47	1.1	15.8
	9	**9159**	15	2.30	9481	14	2.63	3.5	6.7
	11	**8901**	13	2.05	9034	10	1.93	1.5	23.1
KroB100	7	**9537**	20	2.53	9853	15	2.66	3.3	25.0
	9	**9240**	15	2.62	9669	15	2.41	4.6	0.0
	11	**8842**	13	2.48	9081	12	2.55	2.7	7.7
KroC100	7	**9723**	17	2.55	9893	16	2.72	1.7	5.9
	9	**9171**	13	2.42	9935	12	2.44	8.3	7.7
	11	**8632**	13	2.44	8754	13	2.50	1.4	0.0
KroD100	7	**9626**	20	2.36	9685	19	2.57	0.6	5.0
	9	**8885**	13	2.73	9610	13	2.43	8.2	0.0
	11	**8725**	13	2.54	9082	12	2.16	4.1	7.7
KroE100	7	**10150**	19	2.16	10516	19	2.28	3.6	0.0
	9	**8991**	14	2.39	9361	12	2.45	4.1	14.3
	11	**8450**	13	2.64	8865	13	2.21	4.9	0.0
Rd100	7	**3461**	18	2.30	3909	16	2.18	12.9	11.1
	9	**3194**	16	2.31	3261	16	2.22	2.1	0.0
	11	**2922**	13	1.97	3178	12	2.13	8.8	7.7
KroA150	7	**11423**	28	4.10	11583	24	2.97	1.4	14.3
	9	**10056**	26	3.53	11496	23	3.14	14.3	11.5
	11	**9439**	21	3.46	10133	20	2.86	7.4	4.8
KroB150	7	**11457**	30	4.30	11744	27	3.75	2.5	10.0
	9	**10121**	24	3.50	10860	20	3.59	7.3	16.7
	11	**9611**	21	3.69	9918	19	3.28	3.2	9.5
KroA200	7	**13285**	35	5.64	13387	34	4.41	0.8	2.9
	9	**11708**	28	5.12	12667	24	5.09	8.2	14.3
	11	**10748**	29	4.61	11940	22	4.37	11.1	24.1
KroB200	7	**13100**	36	5.42	14726	32	5.84	12.4	11.1
	9	**11900**	31	5.11	12989	29	5.96	9.2	6.5
	11	**10676**	30	5.01	12066	24	5.31	13.0	20.0
ali535	3	**1387**	182	60.00	1439	179	64.19	3.7	1.6
	5	**1184**	122	60.00	1523	118	72.44	28.6	3.3
	7	**1094**	95	60.00	1236	90	58.14	13.0	5.3
rat575	3	**3755**	193	60.00	4230	193	74.27	12.6	0.0
	5	**3062**	133	60.00	3160	124	81.71	3.2	6.8
	7	**2667**	110	60.00	2945	100	72.54	10.4	9.1
u724	3	**25045**	230	60.00	29260	226	94.87	16.8	1.7
	5	**20563**	173	60.00	23699	170	91.15	15.3	1.7
	7	**18015**	130	60.00	19434	123	104.38	7.9	5.4
Média								6.5	7.8

Fonte: Elaborada pelo autor

Por fim, na Tabela 3 é comparado o algoritmo de Heurística Gulosa+ACO (HG+ACO), apresentado nas seções 4.3.2 e 4.3.4, com o LS2. E novamente, o algoritmo LS2 obteve respostas melhores, com os melhores custos sendo menores do que os melhores custos encontrados pelo HG+ACO para todas as instâncias. O *gap* de melhor custos teve uma média de 11,1%, o que significa que o HG+ACO apresentou respostas 11,1% piores do que as encontradas pelo LS2, tendo o pior desempenho médio de todos os algoritmos elaborados neste trabalho. O maior *gap* de custos foi de 26,3%, obtido na instância KroB200 com NC igual à 11. E o menor *gap* de custos foi de 1,6%, obtido na instância Rat99 com NC igual à 11. Já para o *gap* NB, houve uma redução média de 17,2% da quantidade de vértices de cobertura utilizados pelo algoritmo HG+ACO em relação ao LS2. Sendo o maior *gap* NB de 33,3%, que acontece nas instâncias Eil51 com NC igual à 9 e KroB200 com NC igual à 11. Os *gaps* NP iguais a 0% acontecem em 3 instâncias apenas, mostrando que o algoritmo HG+ACO, foi o mais eficiente na redução da quantidade de vértices de cobertura das soluções.

Tabela 3 – comparação de resultados do LS2 e da Heurística Gulosa+ACO para o CSP

		LS2			Heurística Gulosa+ACO				
Instâncias	NC	Melhor Custo	NB	TT médio (s)	Melhor Custo	NB	TT médio (s)	*Gap* Melhor Custo (%)	*Gap* NB (%)
Eil51	7	**164**	10	1.05	194	7	1.15	18.3	30.0
	9	**159**	9	0.78	169	6	1.21	6.3	33.3
	11	**147**	7	0.87	167	5	0.92	13.6	28.6
Berlin52	7	**3887**	11	1.05	4050	9	1.04	4.2	18.2
	9	**3430**	7	0.95	3615	7	0.93	5.4	0.0
	11	**3262**	6	0.55	3429	6	0.91	5.1	0.0
St70	7	**288**	11	1.48	315	11	1.72	9.4	0.0
	9	**259**	10	1.65	293	9	1.58	13.1	10.0
	11	**247**	10	1.37	289	8	1.61	17.0	20.0
Eil76	7	**207**	15	1.35	245	12	1.96	18.4	20.0
	9	**185**	12	1.43	198	9	1.48	7.0	25.0
	11	**170**	11	1.50	183	8	1.42	7.6	27.3
Pr76	7	**50275**	14	1.89	51183	13	1.95	1.8	7.1
	9	**45348**	12	1.50	47239	11	1.93	4.2	8.3
	11	**43028**	10	1.39	46812	8	2.59	8.8	20.0
Rat99	7	**486**	18	2.45	564	15	2.60	16.0	16.7
	9	**455**	15	2.45	480	13	2.17	5.5	13.3
	11	**444**	13	2.26	451	10	2.10	1.6	23.1
KroA100	7	**9674**	19	2.78	10094	16	2.51	4.3	15.8
	9	**9159**	15	2.30	9715	12	2.23	6.1	20.0
	11	**8901**	13	2.05	9113	10	2.10	2.4	23.1
KroB100	7	**9537**	20	2.53	10541	15	2.78	10.5	25.0
	9	**9240**	15	2.62	9968	13	2.79	7.9	13.3
	11	**8842**	13	2.48	9165	10	2.41	3.7	23.1
KroC100	7	**9723**	17	2.55	10370	15	2.61	6.7	11.8
	9	**9171**	13	2.42	10103	12	2.45	10.2	7.7
	11	**8632**	13	2.44	8761	11	2.29	1.5	15.4
KroD100	7	**9626**	20	2.36	10765	15	2.84	11.8	25.0
	9	**8885**	13	2.73	10583	11	2.95	19.1	15.4
	11	**8725**	13	2.54	9345	10	2.15	7.1	23.1
KroE100	7	**10150**	19	2.16	10729	16	3.12	5.7	15.8
	9	**8991**	14	2.39	9472	12	3.02	5.3	14.3
	11	**8450**	13	2.64	9043	10	2.92	7.0	23.1
Rd100	7	**3461**	18	2.30	3945	15	2.28	14.0	16.7
	9	**3194**	16	2.31	3624	13	2.34	13.5	18.8
	11	**2922**	13	1.97	3192	11	2.40	9.2	15.4
KroA150	7	**11423**	28	4.10	11945	23	3.98	4.6	17.9
	9	**10056**	26	3.53	11670	19	3.81	16.1	26.9
	11	**9439**	21	3.46	10198	16	3.76	8.0	23.8
KroB150	7	**11457**	30	4.30	12095	24	4.01	5.6	20.0
	9	**10121**	24	3.50	11293	19	3.59	11.6	20.8
	11	**9611**	21	3.69	10285	16	3.63	7.0	23.8
KroA200	7	**13285**	35	5.64	13941	30	6.43	4.9	14.3
	9	**11708**	28	5.12	12728	23	7.15	8.7	17.9
	11	**10748**	29	4.61	12491	21	6.31	16.2	27.6
KroB200	7	**13100**	36	5.42	14826	30	7.13	13.2	16.7
	9	**11900**	31	5.11	13928	23	7.08	17.0	25.8
	11	**10676**	30	5.01	13484	20	6.98	26.3	33.3
ali535	3	**1387**	182	60.00	1662	179	64.50	19.8	1.6
	5	**1184**	122	60.00	1571	111	63.49	32.7	9.0
	7	**1094**	95	60.00	1295	81	65.91	18.4	14.7
rat575	3	**3755**	193	60.00	4238	192	88.14	12.9	0.5
	5	**3062**	133	60.00	3522	116	91.21	15.0	12.8
	7	**2667**	110	60.00	3047	88	85.81	14.2	20.0
u724	3	**25045**	230	60.00	29951	226	143.70	19.6	1.7
	5	**20563**	173	60.00	25948	147	141.03	26.2	15.0
	7	**18015**	130	60.00	22301	112	132.81	23.8	13.8
Média								11.1	17.2

Fonte: Elaborada pelo autor

Com os resultados apresentados nas 3 tabelas, é indiscutível poder afirmar que os algoritmos elaborados para este trabalho não resolvem o CSP tão bem quanto o algoritmo LS2, já que em todas as instâncias testadas, o LS2 obtém um ciclo de custo menor do que o gerado pelos outros algoritmos. O motivo disso, foi que, com a abordagem utilizada neste trabalho, apresentada na seção 4.3, o CSP foi dividido em dois problemas de otimização combinatória clássicos, em que primeiramente é resolvido um SCP, e com os vértices de cobertura selecionados por ele, é resolvido um TSP com o objetivo de determinar a menor distância entre eles.

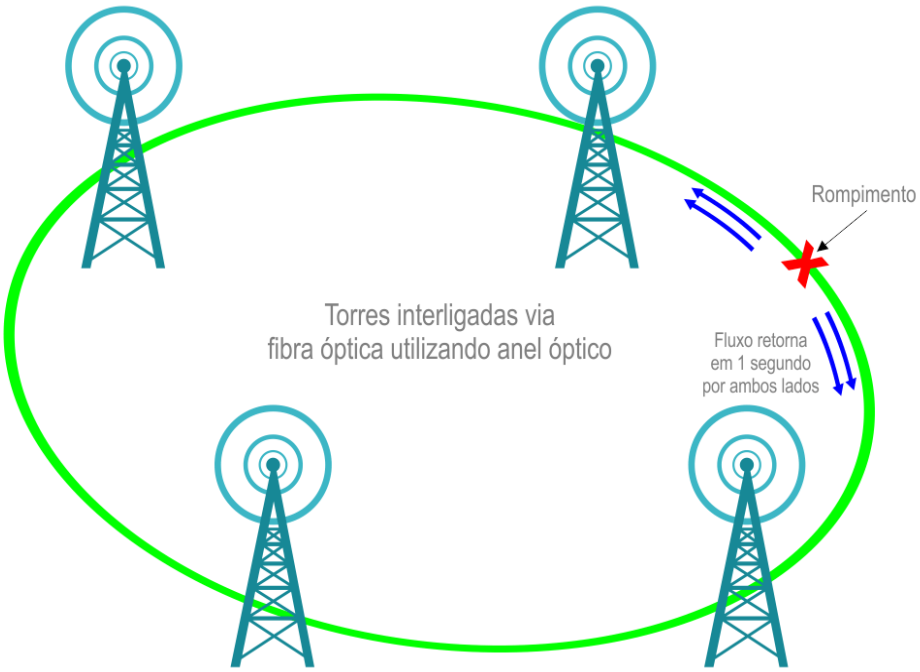
Nas abordagens propostas neste trabalho para resolver o SCP, a estratégia foi buscar uma configuração mínima de vértices de cobertura de forma aproximada, com a restrição de que essa configuração seja capaz de cobrir todo o grafo. Esta abordagem mostra-se ineficiente ao resolver o CSP, pois não é levada em consideração a distância entre os vértices de cobertura, mas sim a quantidade de vértices que um vértice de cobertura é capaz de cobrir, fazendo com que os algoritmos escolham os vértices de cobertura que cobrem mais vértices do grafo, sendo possível que eles sejam muito distantes uns dos outros, mesmo cobrindo mais vértices. Em uma configuração de vértices de cobertura escolhidos pelos algoritmos responsáveis por resolver o SCP, o ACO é executado logo em seguida, para resolver o TSP, iterando somente sobre esses vértices de cobertura selecionados, sem poder alterar essa configuração. Desta forma, o ACO consegue encontrar uma distância próxima da mínima entre os vértices de cobertura selecionados, mas que pode gerar uma solução muito distante da ótima para o CSP.

No entanto, ainda há algo interessante de se observar nos resultados das tabelas. Ao comparar principalmente o algoritmo Heurística Gulosa+ACO com o LS2, percebe-se que, em praticamente, para todas as instâncias, a heurística gulosa foi capaz de encontrar uma solução que utiliza menos vértices que o algoritmo LS2. O que para o CSP, não significa nada, já que o objetivo do problema é minimizar o custo total do ciclo entre os vértices de cobertura. Porém, é possível imaginar algum contexto em que a quantidade de vértices poderia afetar diretamente o custo total do ciclo.

5.5 Reelaboração do problema

Em um contexto de telecomunicações, pode-se supor um exemplo em que uma empresa deseja criar um sistema de telecomunicação para atender todos os clientes de uma região. Para isso, serão implantadas estações de atendimento, de forma que cada cliente deve ser atendido (coberto) por pelo menos uma delas. Os locais disponíveis para a implantação dessas estações são as localizações dos clientes. Uma vez que os locais para a implantação forem definidos, as estações serão construídas cobrindo todos os clientes. Essas estações são conectadas por cabos de fibra óptica a uma rede. Faz-se necessário criar um anel óptico entre as estações de atendimento, que proporciona segurança e redundância em caso de rompimento em algum trecho no cabo de fibra óptica, para que a rede continue operando normalmente. A Figura 8 mostra um exemplo do funcionamento de um anel óptico, em que caso haja um rompimento do cabo entre duas estações, o sinal que antes tinha o fluxo indo em apenas uma direção, agora retorna, indo em sentido contrário, sem deixar que nenhuma das estações perca conexão com a rede.

Figura 8 - Exemplo de anel óptico conectado por estações de atendimento.



Retirada de: site da provedora RQNet

Desta forma, é possível dizer que os vértices de cobertura seriam as estações de atendimento que serão construídas, as arestas seriam os cabos de fibra óptica que as conectam, e há a necessidade da criação de um ciclo entre as estações de atendimento para criar o anel óptico. Neste contexto, é visível que a construção de uma estação de atendimento afeta diretamente no custo de implantação do sistema. Portanto, uma empresa que deseja criar um sistema de atendimento a um conjunto de clientes, para minimizar gastos da implantação do sistema, teria como objetivo criar um ciclo utilizando o mínimo de estações de atendimento possível e com uma distância mínima entre elas para minimizar os gastos com cabos de fibra óptica. Então, é possível imaginar uma formulação para o CSP, em que também se tem um custo para adicionar um vértice de cobertura ao ciclo. O objetivo do problema será minimizar a quantidade de vértices que o ciclo terá, assim como minimizar a distância total do ciclo, da mesma forma que no CSP original, ou seja, o problema neste contexto, se torna multiobjetivo.

5.6 O *Covering Salesman Problem* multiobjetivo

O CSP Multiobjetivo pode ser definido da mesma forma que na seção 3, porém, agora adicionando um valor F_i para todo vértice i do grafo. Esse valor representa o custo para fazer do vértice i um vértice de cobertura. O CSP então, pode ser reformulado, para que busque o menor custo F_i dos vértices de cobertura presentes no ciclo, e, ao mesmo tempo, buscar um menor custo do ciclo que os conecta, como já era feito anteriormente, substituindo sua função objetivo, da seguinte forma:

$$\text{minimizar } Z = (Z_1, Z_2) \tag{18}$$

sujeito às mesmas restrições da formulação do CSP explicado na seção 3, e onde:

$$Z_1 = \sum_{i \in N} \sum_{j \in N} c_{ij} y_{ij} \tag{19}$$

$$Z_2 = \sum_{i \in N} \sum_{j \in N} F_i y_{ij} \tag{20}$$

A função objetivo (14) torna-se minimizar o valor de (15) e (16), em que Z_1 representa o custo das arestas que estão no ciclo, da mesma forma que a equação (1) na formulação do CSP original, e Z_2 o custo dos vértices de cobertura que estão

no ciclo. Esta variante do problema CSP e sua formulação matemática, foram propostas por Current e Schilling (1989).

5.7 Reformulação do problema

Na formulação apresentada na seção 5.6, o problema tem um valor F_i diferente para cada vértice *i*, ou seja, cada vértice de cobertura tem um custo diferente. No entanto, como simplificação do problema, pode-se supor que, no exemplo da seção 5.5, os vértices de cobertura, representados por estações de atendimento, terão o mesmo custo de implantação, ou seja, seu F_i possui o mesmo valor para todas as estações. Supondo um caso em que o custo de implantação de uma estação seja muito alto em relação ao custo do cabo da fibra óptica, fazendo com que a prioridade por selecionar menos vértices de cobertura seja maior do que minimizar a distância do ciclo.

Desta forma, o problema pode ser reescrito a partir do CSP multiobjetivo, modificando a equação (14) presente na formulação matemática, para minimizar a soma das duas funções objetivo apresentadas na seção 5.6, ou seja: minimizar $Z = (Z_1 + Z_2)$, o que faz com que o problema não seja mais multiobjetivo, pois pode ser representado por apenas uma equação.

Levando em consideração esta nova função objetivo, supondo que o custo de F_i seja o mesmo para todos os vértices de cobertura *i*, e que o valor deste custo seja muito alto em relação ao custo da soma das arestas, uma abordagem que poderia ser efetiva para este caso, seria minimizar primeiramente apenas a quantidade de vértices de cobertura presentes no ciclo, que foi o que os algoritmos de Relaxação Lagrangiana, Busca Tabu e Heurística Gulosa, elaborados neste trabalho, fizeram e em seguida, buscar minimizar apenas a distância entre os vértices de cobertura, criando um ciclo de custo mínimo entre eles, o que neste trabalho foi feito pelo ACO. Portanto a estratégia proposta neste trabalho, pode resolver o problema nestas condições de uma melhor forma que o LS2.

Para exemplificar, utilizando as mesmas instâncias e resultados obtidos nas Tabela 1, 2 e 3 e supondo que exista um custo $F_i = 2000$, o que significa que o custo de implantação de uma estação é 2000 vezes superior ao valor de implantação do cabo óptico por unidade de distância. Para cada vértice de cobertura *i* presente no ciclo que foi obtido pelos algoritmos (LS2, RL, BT e HG), é

possível multiplicar o custo F_i pela quantidade de vértices em cada solução e somá-la ao custo total do ciclo, ou seja, melhor custo = menor distância encontrada + (quantidade de vértices de vértices de cobertura × F_i).

A Tabela 4 mostra como os custos do ciclo se alteram nestas novas condições, onde o melhor custo agora é representado pelo valor da distância mínima encontrada para o ciclo somado com o custo da quantidade de vértices de cobertura utilizados para formá-lo, com um valor de F_i alto. Nessa tabela, o menor valor encontrado para cada instancia está destacado em negrito.

Tabela 4 - comparação de melhor custo dos algoritmos para o CSP com custo nos vértices F_i = 2000

		LS2	RL+ACO	BT+ACO	HG+ACO
Instâncias	NC	Melhor Custo	Melhor Custo	Melhor Custo	Melhor Custo
Eil51	7	20164	18185	20183	**14194**
	9	18159	16163	16172	**12169**
	11	14147	12148	**10157**	10167
Berlin52	7	25887	23976	25945	**22050**
	9	**17430**	17442	17494	17615
	11	**15262**	15314	15399	15429
St70	7	**22288**	22304	22310	22315
	9	20259	20283	20293	**18293**
	11	20247	18272	20278	**16289**
Eil76	7	30207	30210	26211	**24245**
	9	24185	22191	24191	**18198**
	11	22170	18178	**16177**	16183
Pr76	7	78275	78813	**76491**	77183
	9	69348	**67771**	69557	69239
	11	63028	61092	**59610**	62812
Rat99	7	36486	34495	36522	**30564**
	9	30455	30469	**26466**	26480
	11	26444	24446	24445	**20451**
KroA100	7	47674	43690	**41778**	42094
	9	39159	39288	37481	**33715**
	11	34901	35109	**29034**	29113
KroB100	7	49537	48395	**39853**	40541
	9	39240	39538	39669	**35968**
	11	34842	**28978**	33081	29165
KroC100	7	43723	42329	41893	**40370**
	9	35171	35925	**33935**	34103
	11	34632	33115	34754	**30761**
KroD100	7	49626	42543	47685	**40765**
	9	34885	35498	35610	**32583**
	11	34725	33296	33082	**29345**
KroE100	7	48150	**42228**	48516	42729
	9	36991	35429	**33361**	33472
	11	34450	30861	34865	**29043**
Rd100	7	39461	39718	35909	**33945**
	9	35194	33554	35261	**29624**
	11	28922	26999	27178	**25192**
KroA150	7	67423	63898	59583	**57945**
	9	62056	52282	57496	**49670**
	11	51439	47802	50133	**42198**
KroB150	7	71457	63567	65744	**60095**
	9	58121	57285	50860	**49293**
	11	51611	44088	47918	**42285**
KroA200	7	83285	79898	81387	**73941**
	9	67708	64272	60667	**58728**
	11	68748	61148	55940	**54491**
KroB200	7	85100	80356	78726	**74826**
	9	73900	**57968**	70989	59928
	11	70676	60180	60066	**53484**
ali535	3	365387	359442	**359439**	359662
	5	245184	229212	237523	**223571**
	7	191094	185125	181236	**163295**
rat575	3	389755	**388151**	390230	388238
	5	269062	239451	251160	**235522**
	7	222667	184779	202945	**179047**
u724	3	485045	487724	**481260**	481951
	5	366563	327295	363699	**319948**
	7	278015	252037	265434	**246301**

Fonte: Elaborada pelo autor

Como mostrado na Tabela 4, caso exista um custo alto relacionado a adicionar cada vértice de cobertura ao ciclo em relação às arestas, os algoritmos propostos neste trabalho, na maioria das instâncias, conseguem custos melhores do que o método LS2, com destaque para a abordagem Heurística Gulosa+ACO. Com F_i = 2000, o algoritmo HG+ACO obteve a melhor resposta entre todos os algoritmos testados em 37 das 57 instâncias. O algoritmo BT+ACO obteve a melhor resposta entre os algoritmos testados em 12 das 57 instâncias. O algoritmo RL+ACO obteve a melhor resposta em 5 das 57 instâncias, e o LS2 obteve uma resposta melhor em relação aos outros algoritmos em 3 das 57 instâncias. Portanto, HG+ACO foi o algoritmo que desempenhou melhor.

Foi criada também uma Tabela 5 para mostrar como as abordagens se comportariam caso o custo F_i tenha um valor baixo, com $F_i = 20$, ou seja, com o custo da implantação de cada estação sendo 20 vezes o valor do custo do cabo de fibra óptica por unidade de distância.

Tabela 5 - comparação de melhor custo dos algoritmos para o CSP com custo nos vértices F_i = 20

		LS2	RL+ACO	BT+ACO	HG+ACO
Instâncias	NC	Melhor Custo	Melhor Custo	Melhor Custo	Melhor Custo
Eil51	7	364	365	383	**334**
	9	339	323	332	**289**
	11	287	268	**257**	267
Berlin52	7	**4107**	4176	4165	4230
	9	**3570**	3582	3634	3755
	11	**3382**	3434	3519	3549
St70	7	**508**	524	530	535
	9	**459**	483	493	473
	11	**447**	452	478	449
Eil76	7	507	510	**471**	485
	9	425	411	431	**378**
	11	390	358	**337**	343
Pr76	7	**50555**	51093	50751	51443
	9	**45588**	45991	45797	47459
	11	**43228**	45252	43770	46972
Rat99	7	846	**835**	882	864
	9	755	769	**726**	740
	11	704	686	685	**651**
KroA100	7	10054	**10030**	10098	10414
	9	**9459**	9588	9761	9955
	11	**9161**	9369	9234	9313
KroB100	7	**9937**	10775	10153	10841
	9	**9540**	9838	9969	10228
	11	**9102**	9178	9321	9365
KroC100	7	**10063**	10649	10213	10670
	9	**9431**	10185	10175	10343
	11	**8892**	8761	9014	8981
KroD100	7	**10026**	10863	10065	11065
	9	**9145**	9758	9870	10803
	11	8985	**9536**	9322	9545
KroE100	7	**10530**	10548	10896	11049
	9	**9271**	9689	9601	9712
	11	**8710**	9081	9125	9243
Rd100	7	**3821**	4078	4229	4245
	9	**3514**	3854	3581	3884
	11	**3182**	3239	3418	3412
KroA150	7	**11983**	12418	12063	12405
	9	**10576**	10702	11956	12050
	11	**9859**	10182	10533	10518
KroB150	7	**12057**	12087	12284	12575
	9	**10601**	11745	11260	11673
	11	**10031**	10428	10298	10605
KroA200	7	**13985**	14558	14067	14541
	9	**12268**	12792	13147	13188
	11	**11328**	11648	12380	12911
KroB200	7	**13820**	15016	15366	15426
	9	12520	12428	13569	**14388**
	11	**11276**	12660	12546	13884
ali535	3	5027	5022	**5019**	5242
	5	3624	**3492**	3883	3791
	7	2994	2965	3036	**2915**
rat575	3	**7615**	7991	8090	8078
	5	5722	5811	**5640**	5842
	7	4867	**4599**	4945	4807
u724	3	**29645**	34304	33780	34471
	5	**24023**	26335	27099	28888
	7	**20615**	24337	21894	24541

Fonte: Elaborada pelo autor

A Tabela 5 mostra que, para um custo mais baixo associado a adicionar um vértice de cobertura ao ciclo, a estratégia de priorizar a minimização da quantidade de vértices, resolvendo-a primeiro, não se mostra tão efetiva quanto o LS2. Com F_i = 20, o algoritmo LS2 obteve respostas melhores em 41 das 57 instâncias. O algoritmo RL+ACO obteve a melhor resposta em 5 das 57 instâncias. O algoritmo BT+ACO obteve a melhor resposta em 6 das 57 instâncias, e o algoritmo HG+ACO obteve a melhor resposta em 6 das 57 instâncias.

Isso demonstra que os algoritmos que utilizaram a abordagem proposta de, primeiramente, buscar o mínimo de vértices para cobrir o grafo, e em seguida, buscar a distância mínima entre eles, são capazes de obter melhores respostas que o algoritmo LS2 em boa parte dos casos, para uma variação do CSP em que existe um custo alto para adicionar novos vértices de cobertura à solução, em relação ao custo da aresta por unidade de distância, perdendo sua efetividade, à medida que este custo se torna mais baixo.

Para essas instâncias, foi proposto que valor de F_i fosse o mesmo para todos os vértices, mas em uma situação em que cada vértice tem um custo diferente, seria necessário modificar a primeira parte da abordagem, buscando minimizar o custo dos vértices representados por F_i e não sua quantidade, desta forma, poderiam existir mais vértices no ciclo, mas com a soma de seus custos sendo mínima.

6 CONCLUSÃO

Neste trabalho de conclusão de curso foi abordado um problema de otimização combinatória, chamado *The Covering Salesman Problem* (CSP). O problema consiste em encontrar um ciclo de custo mínimo em um subconjunto de *n* vértices dados, tal que cada vértice que não pertence ao ciclo deve estar dentro de uma distância máxima predeterminada de algum vértice que está no ciclo.

Para resolver o CSP, foi proposta a estratégia de separar o problema em dois problemas clássicos, o *Set Covering Problem* (SCP) e o *Travelling Salesman Problem* (TSP), e resolvê-los com diferentes abordagens, utilizando algoritmos de aproximação, heurísticas e meta-heurísticas. O SCP foi resolvido primeiramente, com o objetivo de minimizar a quantidade de subconjuntos utilizados para cobrir o conjunto total, utilizando 3 abordagens diferentes, sendo estas, Relaxação Lagrangiana, Heurística Gulosa, e Busca Tabu. Utilizando as soluções geradas pelas abordagens responsáveis por resolver o SCP, o TSP foi resolvido com a meta-heurística *Ant Colony Optimization* (ACO). Para fim de comparação, foram utilizados os resultados obtidos pelo algoritmo baseado em busca local LS2, proposto por Golden et al. (2012).

Os resultados das abordagens elaboradas não se mostraram satisfatórios para o CSP, já que a estratégia de resolver primeiramente um SCP, busca minimizar a quantidade de vértices no ciclo, o que não garante gerar um ciclo de menor distância entre os vértices. Quando o TSP é resolvido pelo ACO, ele executa apenas com os vértices de cobertura definidos pelos algoritmos responsáveis por resolverem o SCP, fazendo com que ele não seja capaz de encontrar uma resposta melhor do que as já obtidas pelo LS2.

No entanto, a abordagem e os algoritmos apresentados neste trabalho podem ser úteis para resolver uma variação do CSP, em que se adiciona um custo fixo para inserir cada vértice de cobertura no ciclo, fazendo com que o custo do ciclo se torne a soma da distância mínima encontrada com a quantidade de vértices de cobertura multiplicado pelo custo de cada vértice utilizado no ciclo.

Os resultados obtidos com a alteração dos valores para estas novas condições demonstraram que os algoritmos elaborados neste trabalho obtiveram respostas melhores que o LS2 em 54 das 57 instâncias para um custo de inserir

cada vértice de cobertura ao ciclo igual à 2000 vezes o custo da aresta por unidade de distância. Já para um custo igual à 20 vezes o custo de cada aresta por unidade de distância, o LS2 obteve respostas melhores em 41 das 57 instâncias. Isso demonstra que as abordagens deste trabalho, por priorizarem a minimização da quantidade de vértices antes da minimização da distância, são mais efetivas para instâncias em que o custo de adicionar novos vértices de cobertura ao ciclo em relação ao custo de cada aresta por unidade de distância seja elevado, e perdendo sua efetividade à medida que esse custo se torna menor. Ou seja, quanto maior é este custo, mais interessante se torna a utilização da abordagem proposta neste trabalho.

Para trabalhos futuros, podem ser criadas diferentes estratégias para resolver a variação do CSP apresentada neste trabalho que podem priorizar a minimização da distância ao invés da quantidade de vértices, e comparar os resultados com as abordagens apresentadas. Também é possível criar abordagens para resolver o CSP multiobjetivo, que tem custos diferentes para cada vértice e visa minimizar a distância total do ciclo e o custo dos vértices ao mesmo tempo. Outro trabalho possível, seria analisar a relação do custo de adicionar cada vértice ao ciclo e o custo unitário da aresta, variando esta relação em um escopo razoável para determinar a partir de que ponto os algoritmos propostos neste trabalho são mais interessantes.

REFERÊNCIAS

BEASLEY, John E. **A lagrangian heuristic for set-covering problems**. Naval Research Logistics (NRL), v. 37, n. 1, p. 151-164, 1990.

BEASLEY, John E. **An algorithm for set covering problem**. European Journal of Operational Research, v. 31, n. 1, p. 85-93, 1987.

COLORNI, Alberto et al. **An Investigation of some Properties of an" Ant Algorithm"**. In: Ppsn. 1992.

CORMEN, Thomas H. et al. **Dynamic programming**. Cambridge, Massachusetts: The MIT Press, 2009.

CURRENT, John R.; SCHILLING, David A. **The covering salesman problem**. Transportation science, v. 23, n. 3, p. 208-213, 1989.

DORIGO, Marco; BIRATTARI, Mauro; STUTZLE, Thomas. **Ant colony optimization.** IEEE computational intelligence magazine, v. 1, n. 4, p. 28-39, 2006.

DORIGO, Marco; STÜTZLE, Thomas. **Ant colony optimization: overview and recent advances**. Springer International Publishing, 2019.

FISHER, Marshall L. **An applications oriented guide to Lagrangian relaxatio**n. Interfaces, v. 15, n. 2, p. 10-21, 1985.

FISHER, Marshall L. **The Lagrangian relaxation method for solving integer programming problems**. Management science, v. 50, n. 12_supplement, p. 1861-1871, 2004.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1990.

GENDREAU, M.; POTVIN, J.-Y. **Handbook of Metaheuristics.** [S.l.]: Springer, 2010.

GENDREAU, Michel; POTVIN, Jean-Yves. **Tabu search. Search methodologies: introductory tutorials in optimization and decision support techniques**, p. 165-186, 2005.

GLOVER, Fred W.; KOCHENBERGER, Gary A. (Ed.). **Handbook of metaheuristics**. Springer Science & Business Media, 2006.

GROSSMAN, Tal; WOOL, Avishai. **Computational experience with approximation algorithms for the set covering problem**. European journal of operational research, v. 101, n. 1, p. 81-92, 1997.

KARP, Richard M. **Reducibility among combinatorial problems. Springer Berlin Heidelberg, 2010.Reducibility among combinatorial problems, Complexity of computer computations** (RE Miller and JW Thatcher, editors). 1972.

LEMARÉCHAL, Claude. **Lagrangian relaxation**. Computational combinatorial optimization: Optimal or provably near-optimal solutions, p. 112-156, 2001.

LU, Yongliang; BENLIC, Una; WU, Qinghua. **A highly effective hybrid evolutionary algorithm for the covering salesman problem**. Information Sciences, v. 564, p. 144-162, 2021.

MATAI, Rajesh; SINGH, Surya Prakash; MITTAL, Murari Lal. **Traveling salesman problem: an overview of applications, formulations, and solution approaches**. Traveling salesman problem, theory and applications, v. 1, 2010.

MILLER, Clair E.; TUCKER, Albert W.; ZEMLIN, Richard A. **Integer programming formulation of traveling salesman problems.** Journal of the ACM (JACM), v. 7, n. 4, p. 326-329, 1960.

ONDŘEJ, Míča. **Comparison of metaheuristic methods by solving travelling salesman problem.** In: Proceedings of the 9th International Scientific Conference INPROFORUM: Common challenges-Different solutions-Mutual dialogue. Jihočeská univerzita v Českých Budějovicích, 2015.

RQNet, disponível em: <http://www.rqnet.com.br/site/estrutura/> Acesso em: 09 jun. 2023

SALARI, Majid; NAJI-AZIMI, Zahra. **An integer programming-based local search for the covering salesman problem**. Computers & Operations Research, v. 39, n. 11, p. 2594-2602, 2012.

SALARI, Majid; REIHANEH, Mohammad; SABBAGH, Mohammad S. **Combining ant colony optimization algorithm and dynamic programming technique for solving the covering salesman problem**. Computers & Industrial Engineering, v. 83, p. 244-251, 2015.

SÖRENSEN, Kenneth; GLOVER, Fred. **Metaheuristics**. Encyclopedia of operations research and management science, v. 62, p. 960-970, 2013.

TALBI, El-Ghazali. **Metaheuristics: from design to implementation**. John Wiley & Sons, 2009.

VENKATESH, Pandiri; SRIVASTAVA, Gaurav; SINGH, Alok. **A multi-start iterated local search algorithm with variable degree of perturbation for the covering salesman problem.** In: Harmony Search and Nature Inspired Optimization Algorithms: Theory and Applications, ICHSA 2018. Springer Singapore, p. 279-292, 2019.