UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL DIAS DE LIMA

UMA COMPARAÇÃO DE DESEMPENHO ENTRE DOIS ALGORITMOS DE BUSCA TABU APLICADOS AO PROBLEMA CAPACITADO DE LOCALIZAÇÃO DE FACILIDADES COM COBERTURA PARCIAL E FONTE ÚNICA

QUIXADÁ

2026

GABRIEL DIAS DE LIMA

UMA COMPARAÇÃO DE DESEMPENHO ENTRE DOIS ALGORITMOS DE BUSCA TABU APLICADOS AO PROBLEMA CAPACITADO DE LOCALIZAÇÃO DE FACILIDADES COM COBERTURA PARCIAL E FONTE ÚNICA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Fábio Carlos Sousa Dias

QUIXADÁ

2026

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

L698c Lima, Gabriel Dias de.
Uma comparação de desempenho entre dois algoritmos de busca tabu aplicados ao problema capacitado de localização de facilidades com cobertura parcial e fonte única / Gabriel Dias de Lima. – 2026.
77 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2026.
Orientação: Prof. Dr. Fábio Carlos Sousa Dias.

1. Busca tabu. 2. Localização de facilidades capacitadas. 3. Cobertura parcial. 4. Fonte única. 5. Comparação. I. Título.

CDD 004

GABRIEL DIAS DE LIMA

UMA COMPARAÇÃO DE DESEMPENHO ENTRE DOIS ALGORITMOS DE BUSCA TABU APLICADOS AO PROBLEMA CAPACITADO DE LOCALIZAÇÃO DE FACILIDADES COM COBERTURA PARCIAL E FONTE ÚNICA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: 19 de Janeiro de 2026

BANCA EXAMINADORA

Prof. Dr. Fábio Carlos Sousa Dias (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Henrique Macêdo de Araújo
Universidade Federal do Ceará(UFC)

Prof. Dr. Wladimir Araújo Tavares
Universidade Federal do Ceará(UFC)

Em memória de snoopy, o pior cachorro que já tive o prazer de conhecer.

AGRADECIMENTOS

Há muitas pessoas a quem eu gostaria de agradecer pela conclusão desta monografia. Agradeço imensamente a minha mãe, que me apoiou durante a minha graduação e, por ela, também agradeço a toda a minha família pelo suporte recebido. Não posso deixar de demonstrar a minha gratidão aos membros da banca avaliadora por toda a colaboração e aos dois *Fábios* que me instruíram ao longo deste trabalho, Fábio Carlos e Fábio Pires. O primeiro, por, atuando como meu orientador, ter me guiado durante a escrita desta monografia. Ao segundo, por ter tido tanta paciência ao ponto de ter respondido a todos os meus E-mails homéricos. Por último, agradeço a todos aqueles que fizeram parte da minha vida, mesmo que brevemente, durante estes quatro anos de graduação, pois todos conseguiram me ensinar algo de útil.

"Art. 1.° É declarada extincta, desde a data desta Lei, a escravidão no Brazil."

(Lei Áurea, 13 de maio de 1888)

RESUMO

Em problemas relacionados à cobertura parcial e à localização de facilidades, se objetiva selecionar um subconjunto de facilidades J' dentre um conjunto de facilidades candidatas J, de maneira que, ao menos, uma quantidade mínima das demandas de um conjunto de clientes I seja atendida. Este trabalho explora uma variação desse problema, que adiciona restrições quanto à capacidade das facilidades e à necessidade de que seus clientes sejam atendidos por, no máximo, uma facilidade — a restrição da fonte única. De acordo com a revisão da literatura realizada, o trabalho mais recente que explora esse mesmo problema foi elaborado por Mourão *et al.* (2024). Em seu trabalho, ele empregou a meta-heurística busca tabu aliada a um *solver* de programação inteira para resolver casos pontuais ligados à alocação de demandas. Seus resultados alcançaram grande precisão e tempos de execução mais rápidos do que os do resolvedor *CPLEX* na resolução do mesmo problema. No entanto, o auxílio de técnicas determinísticas para auxiliar a meta-heurística pode gerar dificuldades em compreender a sua eficácia. Desse modo, neste trabalho, a implementação da busca tabu conceituada por Mourão *et al.* (2024) foi desenvolvida na linguagem *Python 3.14*, e a análise comparativa entre o desempenho das duas versões, da busca tabu *pura* e da auxiliada pelo CPLEX, foi realizada por meio da comparação da qualidade das soluções encontradas e do tempo computacional gasto. Observou-se que, mesmo sem o auxílio de métodos determinísticos, a meta-heurística em si ainda consegue encontrar soluções de qualidade equivalentes às do CPLEX em $32,05\%$ das 78 instâncias testadas, enquanto seu tempo de execução apresentou uma queda em $56,41\%$ das instâncias.

Palavras-chave: busca tabu; localização de facilidades capacitadas; cobertura parcial; fonte única; comparação.

ABSTRACT

In problems related to partial covering and facility location, the objective is to select a subset of facilities J' from a set of candidate facilities J, so that at least a minimum amount of demand from a set of customers I is met. This work explores a variation of this problem that adds constraints regarding the capacity of the facilities and the requirement that customers be served by at most one facility — the single-source constraint. According to the literature review done, the most recent work exploring this same problem was developed by Mourão *et al.* (2024). In his study, he employed the tabu search metaheuristic combined with an integer programming *solver* to solve specific cases related to demand allocation. His results achieved high precision and faster execution times than those of the *CPLEX* solver when solving the same problem. However, the use of deterministic techniques to assist the metaheuristic may make it difficult to understand its standalone effectiveness. Therefore, in this work, the tabu tearch conceptualized by Mourão *et al.* (2024) was implemented in Python 3.14, and a comparative analysis between the performance of the two versions, the *pure* tabu search and the one aided by the CPLEX, was conducted by comparing the quality of the solutions and computational time spent. It was observed that, even without the aid of deterministic methods, the metaheuristic alone is able to find solutions of equivalent quality to CPLEX in 32.05% of the 78 tested instances, while its execution time decreased in 56.41% of the instances.

Keywords: tabu search; capacitated facility location; partial covering; single source; comparation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A área da Pesquisa Operacional (PO) surgiu no Reino Unido durante a Segunda Guerra Mundial, em um contexto em que o país passava por um período de escassez de recursos como medicamentos, munições e alimentos (Murthy, 2007). Nesse cenário, o exército britânico reuniu equipes de matemáticos, engenheiros, físicos e outros especialistas (Moreira, 2018) para realizar pesquisas relacionadas às operações militares em andamento de modo que, dentro das restrições impostas, fosse possível alocar quantidades de suprimentos em cada uma delas (Panneerselvam, 2006).

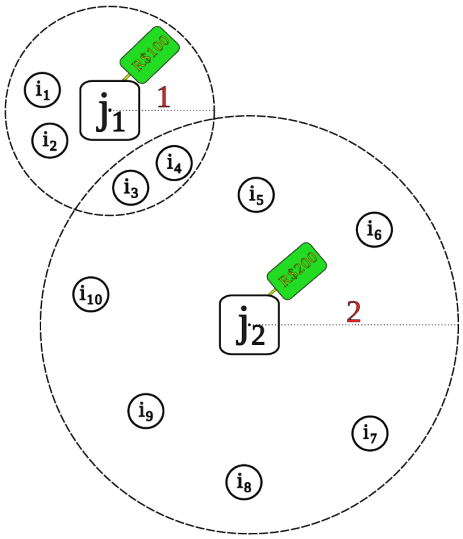
Mesmo após a guerra, a PO continuou a se desenvolver e passou a ser empregada na esfera civil (Taha, 2007), impulsionada pela necessidade de gerir os recursos escassos nos países europeus durante o pós-guerra (Murthy, 2007). Atualmente, trata-se de uma área de pesquisa de grande relevância econômica e com crescente investimento, visto que, diante da alta competitividade do mercado, a PO fornece subsídios fundamentais para a tomada de decisões empresariais, tornando os processos e serviços mais eficientes (Pollock; Maltz, 1994).

Um dos problemas mais difíceis da *PO* é o *Problema de Localização de Facilidades com Cobertura Parcial* (Goldbarg *et al.*, 2017), PLFCP em sua sigla. De acordo com Cordeau *et al.* (2019), seu objetivo é, dado um conjunto de clientes I de tamanho m, em que cada um deles possui uma demanda[1] d_i a ser atendida, e um conjunto de facilidades J de tamanho n — onde cada facilidade possui um raio de cobertura r e um custo de abertura f —, selecionar um subconjunto de facilidades $J' \subset J$ que atenda a uma demanda mínima D_m, frente a uma demanda total D_t, de forma que o somatório dos custos de abertura das facilidades em J' seja minimizado.

A Figura 1 ilustra uma instância desse problema. Nela, é possível observar que J é composto por duas facilidades, j_1 e j_2, com custos de instalação de R$ 100 e R$ 200, respectivamente. Além disso, o conjunto I possui 10 clientes e, para fins de simplificação, todas as suas demandas valem 1; portanto, a demanda total D_t é 10. Assim, supondo que D_m seja 8, a solução que melhor satisfaz o problema é $J' = \{j_2\}$, uma vez que j_2 é capaz de atender aos 8 clientes, enquanto j_1 é incapaz de suprir a demanda mínima e a escolha de $J' = \{j_1, j_2\}$, mesmo que factível, resultaria em uma solução mais custosa. Além disso, é interessante notar

[1] Os clientes, ou pontos de demanda — como também são referidos na literatura (Cordeau *et al.*, 2019; Mourão *et al.*, 2024) — podem ser interpretados como os indivíduos que pretendem usufruir dos serviços das facilidades, enquanto suas demandas podem ser entendidas de diferentes formas, como um peso associado a cada um deles ou a quantidade de certos itens que eles depositarão nas facilidades (Cordeau *et al.*, 2019), por exemplo. Sob essa interpretação, considera-se que um cliente é atendido quando suas demandas são alocadas a uma facilidade.

Figura 1 – Exemplo ilustrativo de uma instância do PLFCP.



Fonte: O próprio autor.

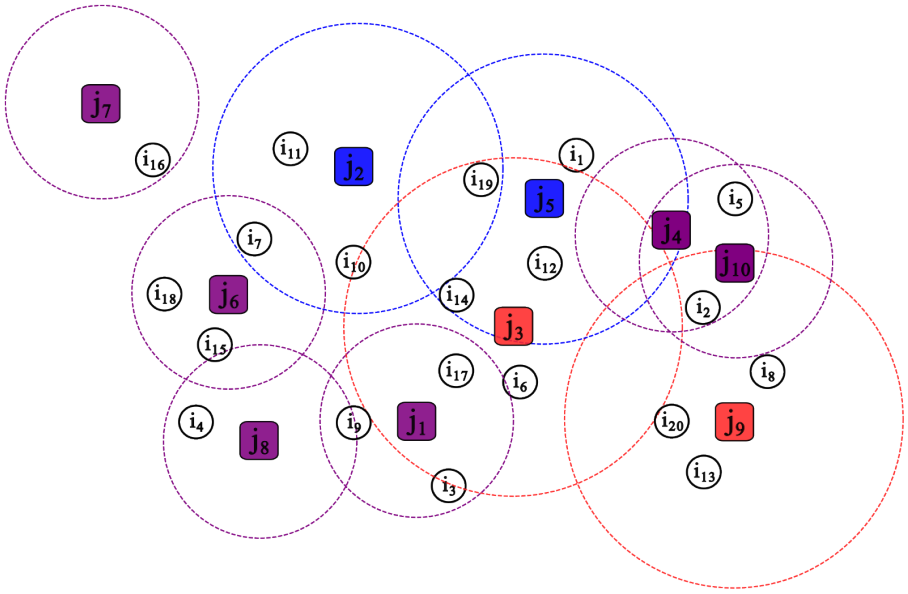
que, mesmo que j_2 seja mais cara que j_1, ao fim das contas, a sua instalação resulta na solução ótima, já que essa é factível e também é a mais barata possível.

Neste contexto, o presente trabalho trata de uma variação do *PLFCP* que adiciona uma restrição quanto à capacidade máxima das facilidades e uma segunda que limita o atendimento de cada cliente a uma única facilidade. Essas restrições fazem com que esta variação seja considerada *capacitada* e de *fonte única*, respectivamente. A essa variante deu-se o nome de *Single Source Capacitated Partial Set Covering Location Problem* (Mourão *et al.*, 2024) ou, em tradução livre, Problema de Localização de Facilidades com Cobertura Parcial, Capacitado e de Fonte Única (PCLFCPFU).

A Figura 2 apresenta uma instância do PCLFCPFU, na qual se considera que as facilidades roxas, azuis e vermelhas têm capacidades máximas de 2, 4 e 6, respectivamente. Além disso, existem 20 clientes com demandas unitárias ($d_i = 1$), 10 facilidades com custos de abertura idênticos e um D_m correspondente a 10. Ressalta-se, sobretudo, que o raio de cobertura não possui relação direta com a capacidade de atendimento da facilidade.

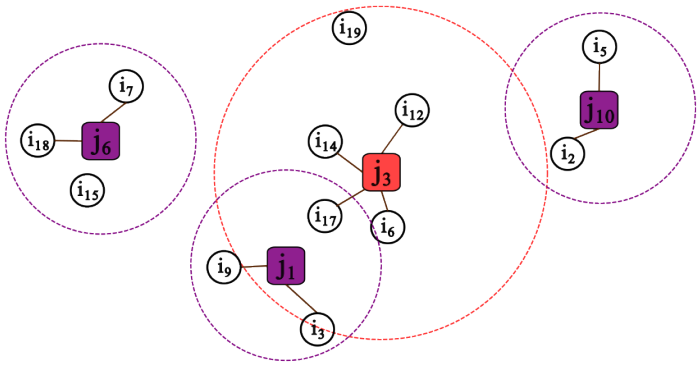
Desse modo, a Figura 3 representa uma solução viável para a instância apresentada. Nela, é possível observar que, devido ao esgotamento da capacidade das facilidades j_1, j_6 e j_{10}, existem demandas que não são atendidas, mesmo situadas dentro do raio de cobertura. No caso de j_3, embora essa facilidade ainda possa atender à demanda i_{19} sem prejuízo à qualidade da solução, o fato de não atendê-la não altera a factibilidade nem melhora o resultado da solução.

Figura 2 – Exemplo ilustrativo de uma instância do PCLFCPFU.



Fonte: Adaptado de Mourão *et al.* (2024).

Figura 3 – Solução factível para a instância apresentada na Figura 2. Os vértices que ligam os clientes às facilidades simbolizam que o i-ésimo cliente está, não está somente coberto pela j-ésima facilidade, mas também está sendo atendido por ela.



Fonte: Adaptado de Mourão *et al.* (2024).

Nesse sentido, um dos trabalhos mais recentes a explorar esse problema foi publicado em 2024, no Simpósio Brasileiro de Pesquisa Operacional (SBPO), por Mourão *et al.* (2024). Pelo uso do resolvedor CPLEX, o autor implementou tanto um modelo de programação inteira para obtenção das soluções ótimas, quanto um algoritmo baseado na meta-heurística *busca tabu*,

que resolvia alguns casos particulares relacionados às alocações das demandas por meio de um segundo modelo de programação inteira.

Ambos os algoritmos foram testados sobre as mesmas 78 instâncias e foi possível constatar que as soluções obtidas via busca tabu apresentaram boa qualidade, alcançando ótimos globais comprovados pelo resolvedor *CPLEX* em 73% dos testes, com um *gap* médio de $2,17\%$. Quanto ao desempenho computacional, a busca tabu obteve tempos de execução, em média, 108,94 vezes menores que os do *CPLEX*.

Entretanto, com a integração do resolvedor *CPLEX* à busca tabu, a avaliação do desempenho do *núcleo* da meta-heurística fica dificultada, já que um método de cunho determinístico é utilizado em certas circunstâncias para a alocação das demandas. Portanto, o presente trabalho busca, principalmente, avaliar o desempenho da busca tabu conceituada por Mourão *et al.* (2024) sem que esta seja auxiliada por um resolvedor de programação inteira. Desse modo, a qualidade das soluções encontradas e o tempo computacional foram avaliados, e uma análise comparativa entre os resultados foi feita.

1.1 Objetivos

Como já mencionado, a pesquisa de Mourão *et al.* (2024) carece de uma comparação entre a busca tabu *pura* e sua versão integrada ao resolvedor CPLEX, o que pode gerar dúvidas sobre a eficácia do funcionamento central da meta-heurística. Diante disso, as subseções seguintes descrevem o objetivo geral e os objetivos específicos desta pesquisa.

1.1.1 Objetivo geral

O objetivo geral consiste em suplementar o trabalho de Mourão *et al.* (2024) por meio da análise de desempenho da versão "pura" da busca tabu aplicada ao PCLFCPFU. Dessa forma, busca-se obter uma visão holística acerca da eficácia e eficiência dessa meta-heurística, fornecendo dados para que outras investigações possam ser realizadas. Visto que Cardoso *et al.* (2023) aponta que problemas de localização de facilidades ainda são pouco explorados na literatura, faz-se necessário contribuir com subsídios para que novas pesquisas sejam desenvolvidas na área.

1.1.2 Objetivos específicos

Para que a contribuição descrita seja concretizada, objetiva-se alcançar os seguintes objetivos específicos:

1. Implementação da meta-heurística busca tabu aplicada ao *PCLFCPFU*.
2. Explicação detalhada do código desenvolvido e das instâncias utilizadas.
3. Levantamento dos resultados com base nas métricas de qualidade das soluções e no tempo de execução, além da comparação desses resultados com os apresentados por Mourão *et al.* (2024).
4. Disponibilização dos artefatos de software implementados e das instâncias de teste usadas em um repositório público no *GitHub*, visando facilitar o desenvolvimento de novas pesquisas relacionadas a essa temática.

2 FUNDAMENTAÇÃO TEÓRICA

Para que todos os detalhes acerca desta pesquisa e de seus trabalhos relacionados sejam compreendidos em sua integridade, torna-se necessário que antes seja feita uma explicação dos conceitos fundamentais por trás deles. Portanto, esta seção é dedicada à apresentação desses fundamentos.

2.1 Conceitos essenciais

Como mencionado anteriormente, o principal objetivo da Pesquisa Operacional é estudar formas de alocar recursos de modo a satisfazer um conjunto de restrições e, ao mesmo tempo, garantir que esses recursos sejam utilizados da forma mais adequada possível (Panneerselvam, 2006). No problema analisado neste trabalho, tais recursos correspondem as facilidades, e é necessário selecionar aquelas que melhor atendem às demandas de atendimento da população.

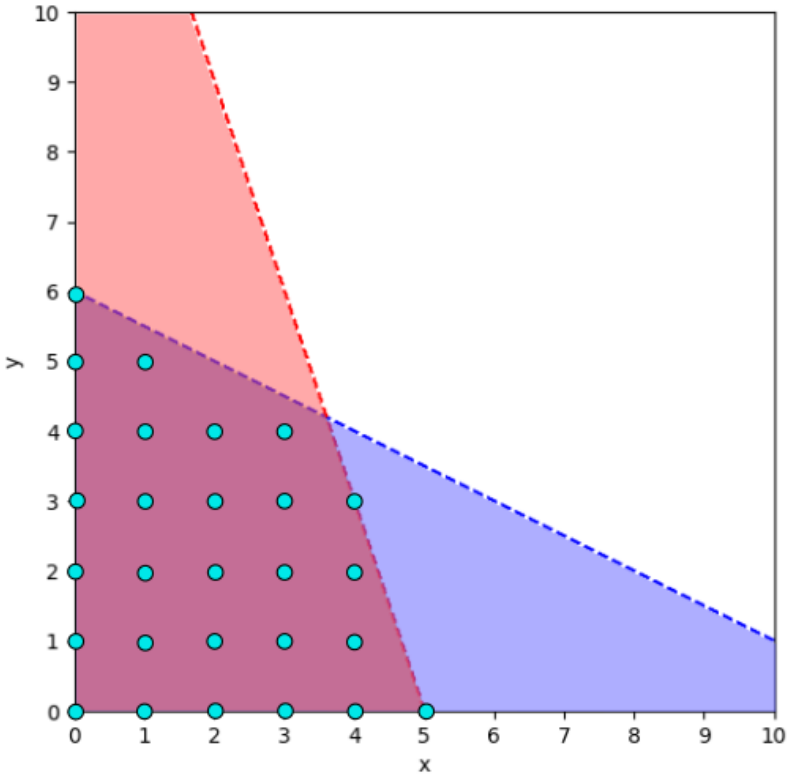
Apesar de a ideia ser simples, existem várias possíveis escolhas sobre quais facilidades, de fato, instalar e quais demandas alocar a cada uma delas; em outras palavras, existem várias soluções possíveis. Com isso em mente, chega-se a um conceito muito importante: o **espaço de busca**, que, como é bem demonstrado nas seções posteriores, é fundamental para compreender o funcionamento das heurísticas e das meta-heurísticas.

Desse modo, é possível definir o espaço de busca como o conjunto de todas as soluções possíveis de um problema (Russell; Norvig, 2016), as quais podem ser divididas em três categorias (Lachtermacher, 2016): **soluções factíveis** s^f, **ótimas** s^* e **infactíveis** s^i. As primeiras representam o conjunto de soluções que cumprem as restrições do problema, enquanto as ótimas correspondem às melhores soluções possíveis. Por fim, as soluções infactíveis são aquelas que não cumprem as restrições do problema

Em relação ao conceito de espaço de busca, a Figura 4 apresenta um exemplo prático. Nela, é possível observar uma área trapezoide em roxo claro, formada pela intersecção das retas vermelha e azul com o eixo das abscissas. Essas retas representam as restrições do problema (Hillier; Lieberman, 2013). Assim, observa-se que as regiões abaixo delas correspondem aos pares (x, y) que as satisfazem, de modo que a região roxa constitui o espaço em que as soluções factíveis estão localizadas, justamente por ser a única área situada abaixo de ambas as retas. Desse modo, as áreas vermelha e azul correspondem aos espaços em que os pares (x, y) satisfazem,

respectivamente, a restrição da reta vermelha e da reta azul.

Figura 4 – Exemplo de um espaço de busca.



Fonte: O próprio autor.

Note também que existem pontos em ciano marcados dentro da região roxa. Eles representam as soluções em que o par (x, y) é composto por números inteiros. A representação desses pontos é especialmente conveniente, pois, com eles, é possível notar mais um conceito fundamental: o de **vizinhança** de uma solução. Quando se fala em vizinhança, faz-se referência às soluções s' que são adjacentes a uma solução s (Papadimitriou; Steiglitz, 1998). Por exemplo, a vizinhança da solução $(2,2)$, cujos x, y são inteiros, corresponde a $\{(1,2),(3,2),(2,3),(2,1)\}$.

Esses conceitos serão especialmente importantes nas seções seguintes, em que a imagem da Figura 4 será explorada em mais detalhes, e os conceitos ligados ao espaço de busca serão amplamente utilizados para explicar o funcionamento das heurísticas e das meta-heurísticas

2.2 Programação matemática

De modo geral, grande parte dos problemas da PO são formalizados por meio de modelos matemáticos (Lachtermacher, 2016), os quais, por sua vez, são desenvolvidos por meio de técnicas de programação[1]. Desse modo, esses modelos compartilham três elementos principais (Taha, 2007): uma n-tupla de **variáveis de decisão** X, uma m-tupla de **restrições** R e uma **função objetivo** Z.

Uma forma didática de compreender esses conceitos é por meio de um exemplo. Um dos mais simples e conhecidos é o *problema da dieta*, proposto pelo governo dos Estados Unidos para possibilitar a criação de uma dieta que custasse o mínimo possível e que, ao mesmo tempo, atendesse a todas as quantidades recomendadas de nutrientes para os soldados que serviam na Segunda Guerra Mundial (Dooren, 2018).

Assim, no problema original, diversos alimentos e nutrientes foram considerados; entretanto, para fins de simplificação, apresenta-se aqui uma versão em que o conjunto de alimentos é reduzido a uma 6-tupla, isto é, $X = \{x_1, ..., x_6\}$, enquanto os nutrientes analisados se restringem a três: carboidratos, proteínas e gorduras. O Quadro 1 reúne os dados pertinentes a cada alimento x_i. Além disso, suponha que as quantidades recomendadas desses nutrientes para uma única refeição sejam, respectivamente, 125 g, 70 g e 20 g — ou seja essas são as restrições que compõe R.

Quadro 1 – Valor nutricional de cada um dos alimentos acompanhado de seus respectivos preços.

Alimentos	**carboidratos**	**proteínas**	**gorduras**	**custo/porção**
x_1	42.25g	4.00g	0.40g	R$2.00
x_2	13.60g	5.00g	0.50g	R$0.75
x_3	21.30g	4.00g	0.40g	R$1.00
x_4	0.00g	25.70g	7.85g	R$7.00
x_5	30.00g	0.60g	0.30g	R$0.80
x_6	0.00g	30.00g	7.70g	R$5.00

Fonte: O próprio autor.

Desse modo, a função objetivo Z é definida em função das quantidades de porções de cada alimento, de modo que ela calculará o custo da refeição de acordo com as porções selecionadas e, como o objetivo, é *minimizar* o custo da refeição, é dito que este problema é de **minimização** e, caso se deseje aumentar algum valor, como os lucros de uma empresa, ele seria

[1] Vale ressaltar que, nesse contexto, "programação" não se refere ao processo de desenvolvimento de algoritmos, mas sim ao planejamento da forma como os recursos disponíveis devem ser distribuídos de modo a atender às restrições do problema e cumprir um determinado objetivo da melhor forma possível.

um problema de **maximização** (Lachtermacher, 2016). Assim, a formulação seguinte apresenta a definição de Z e as restrições que ela está submetida:

$$\min\ Z = 2x_1 + 0,75x_2 + 1x_3 + 7x_4 + 0,80x_5 + 5x_6 \tag{1}$$

Sujeito a (s.a.):

$$42,25x_1 + 13,60x_2 + 21,30x_3 + 0x_4 + 30x_5 + 0x_6 \geq 125 \tag{2}$$

$$4,00x_1 + 5,00x_2 + 4,00x_3 + 25,70x_4 + 0,60x_5 + 30x_6 \geq 70 \tag{3}$$

$$0,40x_1 + 0,50x_2 + 0,40x_3 + 7,85x_4 + 0,30x_5 + 7,70x_6 \geq 20 \tag{4}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \tag{5}$$

No que diz respeito às restrições, é importante que alguns aspectos sejam destacados. Nesse contexto as restrições (2), (3) e (4) são responsáveis por, respectivamente, explicitar a quantidade mínima de carboidratos, proteínas e gorduras. Desse modo, os nutrientes presentes em cada porção de uma refeição são somados, de forma que a quantidade mínima recomendada de cada nutriente seja respeitada.

Também é relevante analisar a restrição (5), a qual determina que as variáveis de decisão devem assumir valores não negativos. Tal condição possibilita que um alimento seja ou não incluído na refeição e, caso venha a ser incluído, que a refeição contenha uma ou mais porções desse alimento.

Outrossim, a formulação deixa claro que não é necessário que a quantidade de nutrientes consumida seja exatamente igual à quantidade recomendada, já que, *de modo geral*, é bastante improvável que existam combinações de valores para as variáveis de decisão capazes de tornar as restrições (2), (3) e (4) exatamente iguais à quantidade mínima de nutrientes requerida.

Portanto, admite-se que o somatório dos nutrientes exceda a quantidade recomendada, pois tal solução ainda satisfazerá as exigências mínimas. Ademais, como uma refeição mais nutritiva implica porções maiores que, por sua vez, implicam maiores custos, é natural que a solução ótima, ao apresentar o menor custo possível, se aproxime ao máximo dos limites das restrições.

Desse modo, ao compreender plenamente os conceitos ligados à função objetivo Z e às tuplas X e R, torna-se pertinente a apresentação de um modelo genérico de programação matemática para que seja possível proporcionar uma visão ampla dos problemas modelados por meio desse recurso. Assim, o modelo logo abaixo constitui um arquétipo válido.

$$\min f = \sum_{j=1}^{n} c_j x_j = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

sujeito a.:

$$\sum_{j=1}^{n} a_{1j} x_j \geq r_1 \quad ou \quad a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \geq r_1$$

$$\sum_{j=1}^{n} a_{2j} x_j \geq r_2 \quad ou \quad a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n \geq r_2$$

$$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots$$

$$\sum_{j=1}^{n} a_{mj} x_j \geq r_m \quad ou \quad a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n \geq r_m$$

$$x_j \geq 0 \quad (j = 1, 2, \ldots, n)$$

Nessa formulação genérica, os coeficientes c da função objetivo representam os custos associados aos alimentos no problema da dieta, enquanto os coeficientes a de cada restrição correspondem aos nutrientes. Cabe destacar, sobretudo, que as restrições podem assumir diferentes formas, como desigualdades ou igualdades, conforme as exigências do problema.

Além disso, a própria estrutura do modelo pode variar de acordo com a natureza da função objetivo, de X e de R, o que permite classificá-lo em diferentes categorias. Por esse motivo, vale mencionar que os tipos mais comuns de modelos são: de **Programação Linear** (PL), **Programação Linear-Inteira** (PLI) e **Programação Inteira Pura** (PI). Na PL, as variáveis de decisão pertencem ao conjunto dos números reais e, tanto a função objetivo, quanto as restrições, são lineares. Já em modelos PLI, uma parcela das variáveis tem que ser inteira, enquanto, em modelos de PI todas as variáveis de decisão são inteiras(Hillier; Lieberman, 2013).

2.3 Métodos de resolução de problemas

Esta seção pretende apresentar de maneira ampla as formas possíveis de resolver problemas relacionados à programação matemática, de modo a pontuar seus pontos fortes e fracos.

2.3.1 Resolvedores (solvers)

Resolvedores são ferramentas de software desenvolvidas para encontrar uma solução ótima $s*$ de um problema de otimização, seja ele formulado como PL, PI, PLI ou outros modelos(Hillier; Lieberman, 2013). Eles permitem que o programador modele todos os aspectos dos modelos matemáticos e abstraia a complexidade associada ao desenvolvimento dos algoritmos responsáveis por procurar $s*$ entre as soluções viáveis. Nesse quesito, podem ser citados alguns resolvedores amplamente utilizados, como CPLEX, Gurobi e SCIP.

Contudo, uma das principais desvantagens associadas à utilização de resolvedores diz respeito ao elevado tempo computacional que eles podem levar para obterem soluções ótimas em instâncias de grande porte, especialmente para problemas com muitas variáveis de decisão(Taha, 2007). Isso ocorre porque a quantidade de restrições cresce exponencialmente de acordo com o número de variáveis de decisão (Hillier; Lieberman, 2013). Desse modo, mesmo que esse número seja relativamente pequeno, ele pode fazer com que o volume de restrições torne a execução do resolvedor substancialmente mais lenta.

2.3.2 Algoritmos heurísticos

De acordo com Coppin (2004), um algoritmo heurístico é aquele que faz uso de informações acerca do espaço de busca para guiar seu processo de exploração e intensificação, com o objetivo de encontrar uma solução ótima ou uma solução de boa qualidade.

Portanto, a principal vantagem associada ao uso de algoritmos heurísticos é que eles, ao utilizarem informações auxiliares acerca do espaço de busca, conseguem analisar menos soluções viáveis e analisar apenas as mais promissoras de acordo com a função objetivo, o que faz com que soluções desse gênero sejam bastante eficientes e, na maior parte das vezes, mais rápidas que a execução de resolvedores.

No entanto, a principal desvantagem associada a essa abordagem está no fato dela não garantir a solução ótima. Em alguns caso, uma heurística pode até encontrar a solução ótima, mas ela não tem condições teóricas de rotular a solução como ótima.

2.3.3 Algoritmos meta-heurísticos

Assim como as heurísticas, as meta-heurísticas empregam formas inteligentes de explorar o espaço de busca(Glover; Kochenberger, 2003); porém, ao contrário das primeiras, que

trabalham com informações factuais acerca do espaço de busca, as segundas utilizam técnicas que mimetizam processos genéticos, físicos, sociais e outros fenômenos para guiar a exploração do espaço de busca(Osman; Kelly, 1996).

Por exemplo, entre as meta-heurísticas mais conhecidas estão os algoritmos genéticos, inspirados na teoria da evolução darwiniana (Goldbarg *et al.*, 2017). De acordo com Russell e Norvig (2016), cada solução viável é codificada como uma sequência de caracteres, chamado *indivíduo*, em que cada caractere representa um *gene*. Por sua vez, os indivíduos participam de uma *população*, na qual aqueles que representam as soluções mais promissoras geram proles, e a sua descendência é usada para compor uma nova *população*. Após um número suficiente de gerações, espera-se que as soluções resultantes estejam suficientemente próximas à solução ótima ou que correspondam a ela.

Uma outra meta-heurística bastante conhecida é o *recozimento simulado*. Ela se baseia no processo termodinâmico de aquecimento e resfriamento lento de vidros e ligas metálicas, conhecido como "recozimento" pela indústria metalúrgica(Russell; Norvig, 2016). A ideia dessa técnica é permitir que os átomos do material se reorganizem de forma que a sua ductilidade e elasticidade aumentem, ou seja, que a sua capacidade de absorver energia sem fraturar melhore (Goldbarg *et al.*, 2017).

Deste modo, essa meta-heurística faz uso de uma variável T que representa uma temperatura e de uma estratégia de resfriamento α, de tal modo que T começa elevada e, ao longo das iterações, vai diminuindo conforme α(Coppin, 2004). Assim, quanto maior for T, maiores são as chances de que uma solução s' com qualidade inferior à do estado atual seja visitada(Coppin, 2004) e, de modo análogo, conforme o algoritmo se aproxima do seu fim, T estará tão baixa que, praticamente, apenas soluções s' melhores que a solução atual serão visitadas (Hillier; Lieberman, 2013).

No caso da meta-heurística estudada neste trabalho, a busca tabu, sua ideia central vem da noção de memória social associada a práticas consideradas proibidas, ou "tabu" (Glover, 1998). Mais detalhes sobre essa meta-heurística são apresentados na seção seguinte.

Portanto, em resumo, as meta-heurísticas providenciam um *modelo* para que soluções próximas ou correspondentes as soluções ótimas sejam encontradas. Portanto, de modo geral, elas conseguem ser mais rápidas que os resolvedores e mais flexíveis que as heurísticas (Hillier; Lieberman, 2013), o que faz com que elas sejam consideradas ferramentas muito importantes na resolução de problemas de PO(Hillier; Lieberman, 2013).

2.4 Busca tabu

Assim, de acordo com as conceitualizações feitas por Goldbarg *et al.* (2017), a variação da busca tabu aplicada neste trabalho é a *clássica*. Nessa abordagem, uma solução s' é considerada proibida caso contenha alguma configuração tida como tabu, ou seja, presente na *lista tabu*.

Por exemplo, suponha que se deseje encontrar um número inteiro x que maximize o resultado de uma função Z. Uma forma de obter uma solução suficientemente boa é usar a busca tabu, ao representar x em seu formato binário e alterar aleatoriamente o valor de um de seus.

Dessa forma, o i-ésimo bit que sofre uma mudança torna-se uma *configuração tabu*, e seu valor não pode ser alterado novamente durante um período de tempo t(Goldbarg *et al.*, 2017). Assim, uma *solução tabu* é qualquer solução que contenha uma configuração tabu ainda vigente, ou seja, cuja penalidade não expirou.

Na busca tabu clássica, uma solução tabu s^t só pode ser visitada se atender ao *critério de aspiração*, que consiste em verificar se ela é melhor do que a melhor solução encontrada até o momento. Se isso ocorrer, s^t é visitada; caso contrário, não(Goldbarg *et al.*, 2017). O Algoritmo 1 mostra o funcionamento da busca tabu aplicado ao exemplo, considerando para simplificação que x é um inteiro de 8 bits.

Ressalta-se, sobretudo, que, embora o algoritmo em questão se trate de uma solução específica para o exemplo dado, ele ainda é suficientemente geral para representar um arquétipo da busca tabu. Assim, todos os seus mecanismos de funcionamento podem ser encontrados em aplicações diversas.

Tendo isso em mente, a primeira observação é que o algoritmo recebe três parâmetros: uma solução inicial s_0, que serve como ponto de partida da busca; um número máximo $iter_{max}$ de iterações sem aprimoramentos; e uma duração máxima dur para que uma configuração seja tratada como tabu. Ao final da execução, espera-se que o algoritmo retorne um valor próximo do ótimo global. A seguir, são destacadas as partes mais importantes do programa:

- **linha 4:** A variável declarada nessa linha, $iter_{melhor}$, é responsável por registrar o valor de $iter$ no qual ocorreu a última melhora da solução. Essa informação é fundamental para a condição de parada do loop do programa.
- **linha 5:** T representa a lista tabu. Ela contém 8 elementos que, inicialmente, são iguais a -1, indicando que nenhum dos 8 *bits* de x é considerado uma configuração tabu. No que tange ao valor inicial, é necessário que ele seja um número negativo para que, durante

Algoritmo 1: Busca tabu

Entrada: Solução inicial s_0, número máximo de iterações sem melhora $iter_{max}$, duração tabu dur

Saída: Solução s^* suficientemente boa

```
1  s_atual ← s_0;
2  s* ← s;
3  iter ← 0;
4  iter_melhor ← 0;
5  T ← [−1,−1,−1,−1,−1,−1,−1,−1];
6  enquanto iter − iter_melhor ≤ iter_max faça
7      iter ← iter + 1;
8      bitFlipped, s' ← bitFlipper(s_atual);
9      se T[bitFlipped] ≥ iter então
10         se f(s') > f(s*) então
11             s_atual ← s';
12             s* ← s_atual;
13             iter_melhor ← iter;
14         continue;
15     s_atual ← s';
16     T[bitFlipped] ← iter + dur;
17     se f(s_atual) > f(s*) então
18         s* ← s_atual;
19         iter_melhor ← iter;
20 retorna s*;
```

a primeira iteração, em que $iter$ é 0, o critério de aspiração seja avaliado corretamente. Além disso, o i-ésimo elemento de T corresponde diretamente ao i-ésimo *bit* de x.

- **linha 6:** A ideia geral expressa nessa linha é que "enquanto soluções melhores que a melhor solução já encontrada estiverem sendo descobertas em tempo hábil, a busca deve continuar"
- **linha 8:** Nessa linha, a função bitFlipper é chamada e recebe s_{atual} como parâmetro. Ela seleciona aleatoriamente um dos bits de s_{atual} e retorna a solução vizinha gerada pela alteração desse bit, junto com o índice do bit modificado.
- **linha 9-14:** Esse intervalo corresponde ao procedimento de aspiração. Aqui, a condição da linha 9 verifica se o bit invertido está associado a uma configuração tabu. Se estiver, duas situações podem ocorrer: ou s' é melhor que $s*$, ou não. No primeiro caso, s' é aceito, $s*$ e $iter_{melhor}$ são atualizados e a execução segue para a próxima iteração. No segundo, como a solução é tabu e não atende ao critério de aspiração, ela é simplesmente descartada e o algoritmo prossegue para a próxima iteração.
- **linha 15-18:** Esse trecho trata do caso em que s' não é tabu. Nesse cenário, s_{atual} e T são

atualizados e, em seguida, verifica-se se s_{atual} supera a melhor solução já encontrada, s^*. Se isso ocorrer, tanto s^* quanto $iter_{melhor}$ são atualizados.

Ressalta-se, novamente, que o algoritmo apresentado é um arquétipo e, por isso, há espaço para melhorias. Dessa forma, ainda é possível acrescentar mecanismos adicionais ao seu funcionamento. Um exemplo seria incluir uma condição extra para interromper a execução quando a verificação da linha 10 falhar muitas vezes seguidas, mesmo que a condição do *loop* da linha 6 ainda continue verdadeira.

2.5 A métrica RPD

O *Relative Percentual Deviation*, ou *RPD*, foi uma das métricas utilizadas por Mourão *et al.* (2024) para avaliar a qualidade das soluções obtidas por meio da busca tabu em comparação com aquelas encontradas pelo *solver* CPLEX. Nesse contexto, ela permitiu mensurar o quão distante a qualidade das soluções encontradas pela busca tabu estavam em relação às retornadas pelo *solver* CPLEX. Dessa forma, quanto mais próximo de zero for o valor do RPD, menor é a diferença entre as qualidades das soluções comparadas. O cálculo dessa métrica segue a formulação apresentada a seguir.

$$RPD = \frac{Q_{exam} - Q_{ref}}{Q_{ref}}$$

Nesse contexto, Q_{exam} representa a qualidade da solução produzida pelo algoritmo sob exame, enquanto Q_{ref} denota a qualidade da solução obtida pelo algoritmo de referência. Considerando o uso dessa métrica por Mourão *et al.* (2024) na análise dos seus resultados, o presente trabalho também a adota para a comparação dos resultados, o que permite uma avaliação mais precisa.

3 TRABALHOS RELACIONADOS

Para que se conheça melhor a condição atual do estado da arte em relação ao *PCLFCPFU*, esta seção apresentará as obras que mais se relacionam com esta temática.

3.1 Meta-heurística *Busca Local Iterativa* aplicada ao *PLFCP*

Na edição de 2022 do *ENIAC*[1] Cardoso *et al.* (2022) desenvolveram uma meta-heurística *Busca Local Iterativa* (BLI) como uma possível forma de solucionar o *PLFCP*. Embora eles não tenham feito menção explícita da razão que os motivou a escolher essa meta-heurística, uma vez que a pesquisa teve cunho exploratório, é possível inferir que o intuito foi aumentar o arcabouço teórico relativo ao *PLFCP*.

Neste sentido, pode-se dizer que a pesquisa foi bem sucedida, pois para as 26 instâncias testadas, a BLI conseguiu encontrar as soluções ótimas 62.5% das vezes (Cardoso *et al.*, 2022), apesar de ter demorado mais que o resolvedor *CPLEX* em 57.69230% dos testes - em um dos casos tendo sido cerca de 20 vezes mais lento que ele.

O algoritmo foi testado sob um subconjunto das 78 instâncias desenvolvidas por Cordeau *et al.* (2019). Nesse caso, para os testes foram feitos paras todas as instâncias com parâmetro $m = \{10000, 50000\}$, além disso eles fixaram $n = 100$ e adotaram $p = \{50\%, 60\%, 70\%\}$ e, para cada combinação $m \times p$, diferentes conjuntos de valores para o raio de cobertura foram adotados.

Em quesito de equipamentos para os testes, um computador com um *Windows 10 Pro* rodando sobre uma máquina com processador Intel Core I5 9a geração 8250U CPU@1.60GHz e 8GB de RAM rodando foi usado. Por último, os autores pontuam que o custo computacional da *BLI* desenvolvida se deve à utilização da *Busca Local* para obtenção do vizinho a ser avaliado.

3.2 Meta-heurística *Recozimento Simulado* aplicada ao *PLFCP*

Em 2023, Cardoso *et al.* (2023) propuseram a meta-heurística *recozimento simulado* para solucionar o *PLFCP*. Por se tratarem dos mesmos autores do trabalho descrito na subseção anterior, o algoritmo foi testado sob o mesmo hardware descrito anteriormente. De modo semelhante, eles também utilizaram as mesmas instâncias de antes para realização dos testes computacionais. (Cardoso *et al.*, 2022; Cardoso *et al.*, 2023)

[1] Encontro Nacional de Inteligência Artificial e Computacional.

Em relação aos resultados, estes se mostraram mais promissores que os da pesquisa que empregou *BLI*, pois o tempo computacional caiu drasticamente para instâncias grandes do problema, e o *recozimento simulado* conseguiu encontrar as soluções ótimas em 90% dos testes.

3.3 Algoritmo Clustering Search para o *PLFCP*

Publicado em 2024 no Simpósio Brasileiro de Pesquisa Operacional (SBPO), o trabalho de Costa *et al.* (2024) utilizou a meta-heurística híbrida *CS*, combinada à meta-heurística "recozimento simulado" de Cardoso *et al.* (2023), para o refinamento das soluções. Nesse sentido, os testes foram realizados sobre o mesmo conjunto utilizado por Cardoso *et al.* (2023), em uma máquina com sistema Linux, processador AMD Ryzen 5 4600G com Radeon Graphics, operando a 4,2 GHz e com 16 GB de RAM.

Em termos de qualidade das soluções, os resultados foram promissores, pois cerca de 90% dos testes obtiveram soluções ótimas. No entanto, em todos os casos, a meta-heurística levou, no mínimo, 50% mais tempo que o algoritmo *simplex*.

3.4 Meta-heurística Busca Tabu para solucionar o *PCLFCPFU*

Como dito anteriormente, o trabalho mais recente que adiciona todas as restrições citadas deste foi escrito por Mourão *et al.* (2024) e publicado em 2024, no SBPO(Mourão *et al.*, 2024), no qual os autores formulam o problema pelo seguinte modelo matemático:

$$\min Z = \sum_{j \in J} f_j y_j \tag{1}$$

Sujeito a (s.a.):

$$a_{ij} y_i \geq z_{ij}, \quad \forall i \in I,\ j \in J \tag{2}$$

$$\sum_{j \in J} z_{ij} \leq 1, \quad \forall i \in I \tag{3}$$

$$\sum_{i \in I} d_i z_{ij} \leq U_{y_j}, \quad \forall j \in J \tag{4}$$

$$\sum_{i \in I} \sum_{j \in J} d_i z_{ij} \geq D_m \tag{5}$$

$$z_{ij} \in \{0,1\}, \quad \forall i \in I,\ j \in J \tag{6}$$

$$y_j \in \{0,1\}, \quad \forall j \in J \tag{7}$$

Nele o conjunto das demandas é expresso por $I = \{i_1, i_2, ..., i_m\}$, enquanto o das facilidades por $J = \{j_1, j_2, ..., j_m\}$. Além disso, ele faz uso de duas matrizes $A_{n\times m}$ e $Z_{n\times m}$, a primeira matém a relação de quais demandas estão dentro do raio de cobertura das facilidades, enquanto a segunda expressa quais demandas, de fato, serão atendidas por uma facilidade. Com isto, surge as variável binária $a_{i,j}$, que indicam se a i-ésima demanda está dentro do raio de cobertura da j-ésima facilidade, e a variável $z_{i,j}$ que indica se ele será atendido por ela ou não.

Em relação a função objetivo, ela visa minimizar o custo total de abertura das facilidades, assim, as variáveis f_j e y_j representam, respectivamente, o custo de abertura da j-ésima facilidade e se ela será aberta ou não. Tendo isso em mente, as restrições do problema são explicadas de forma intuitiva logo abaixo.

- (2): Caso a i-ésima demanda esteja no raio de cobertura da j-ésima facilidade, ela pode ou não ser atendida. Assim, caso ela nem sequer esteja dentro do seu raio de cobertura, com certeza ela não será atendida pela facilidade em questão.
- (3): As demandas dos clientes só podem ser alocadas a, no máximo, uma facilidade.
- (4): Expressa a ideia de que a quantidade de demandas atendidas pela j-ésima facilidade nunca será maior que a sua capacidade.
- (5): A quantidade de demandas atentidas por todo o conjunto de facilidades é de, no mínimo, D_m, exisitindo a possibilidade que mais demandas sejam atendidas.
- (6) e (7): Expressam os possíveis valores que as variáveis z_{ij} e y_j podem assumir. Desse modo, caso a primeira corresponda a 1, então a i-ésima demanda está no raio de cobertura da j-ésima facilidade, do contrário, isso é falso. Um racioncínio semelhante é aplicada a y_j. Se ela valer 1, logo ela foi instalada, do contrário, está fechada.

Em relação ao maquinário usado para a execução dos testes consistiu em um sistema Linux Mint, rodando sobre um processador Intel Xeon W-2223 de 3,60 GHz, com 64 GB de memória RAM. Isso possibilitou a realização de testes computacionais com instâncias de grande porte. A Tabela 1 ilustra os valores utilizados[2] para as n facilidades, os m clientes, a demanda mínima D_m, o raio r e a capacidade U_{y_j}.

Em relação aos resultados obtidos, o algoritmo demonstrou encontrar soluções ótimas em 75% dos casos e conseguiu finalizar a execução das instâncias em até 15 minutos —

[2] Ao analisar as instâncias utilizadas por Mourão *et al.* (2024), disponibilizadas em https://github.com/fabiopiresmourao/SSCPSCLPSBPO2024, constatou-se que a tabela apresentada pelo autor contém um equívoco nos valores do raio de cobertura para o caso $D = 70\%$. Enquanto a tabela do seu artigo indica os valores 3.35, 3.5, 3.75, 4.0 e 4.25, as instâncias disponíveis no repositório do *GitHub* não incluem o caso $r = 4.0$. Assim, na tabela deste trabalho, esse detalhe foi ajustado para refletir corretamente os dados disponibilizados.

Tabela 1 – Valores dos parâmetros utilizados nos experimentos.

n	m	D_m	r	U
100	{1.000; 10.000; 50.000}	50%	{5,5; 5,75; 6,0; 6,25}	{0,02D_m; 0,05D_m}
		60%	{4,0; 4,25; 4,5; 4,75; 5,0}	
		70%	{3,25; 3,5; 3,75; 4,25}	

Fonte: Mourão *et al.* (2024).

um resultado excelente, visto que o resolvedor de *PI*, desenvolvido por eles, demorou até 24 horas para concluir a execução.

4 METODOLOGIA

Nesta seção, detalham-se as métricas utilizadas para comparar a busca tabu desenvolvida neste trabalho com o modelo de programação inteira e a busca tabu apresentada por Mourão *et al.* (2024). Também são descritas as instâncias empregadas na comparação, bem como as configurações do sistema responsável pela execução dos testes.

4.1 Testes computacionais

Para a condução da análise comparativa, foram utilizadas as mesmas 78 instâncias empregadas por Mourão *et al.* (2024). Os valores dos parâmetros referentes ao número de pontos de demanda n, ao número de facilidades m, à demanda mínima D_m, ao raio de cobertura r e à capacidade U são apresentados na Tabela 2.

Tabela 2 – Valores dos parâmetros utilizados nos experimentos.

n	m	U	D_m	r
100	1 000	$0,02D_m$	$0,5D_t$	$\{5,5; 5,75; 6,0; 6,25\}$
	10 000		$0,6D_t$	$\{4,0; 4,25; 4,5; 4,75; 5,0\}$
	50 000	$0,05D_m$	$0,7D_t$	$\{3,25; 3,5; 3,75; 4,25\}$

Fonte: Mourão *et al.* (2024).

Desse modo, D_m corresponde a uma porcentagem da demanda total de cada instância. Nessa perspectiva, os valores da demanda total D_t das instâncias com m igual a 1.000, 10.000 e 50.000 são, respectivamente, 49.916, 502.399 e 2.520.167. De forma análoga, a capacidade U é definida como uma porcentagem de D_m. Nesse caso, consideram-se instâncias com capacidade equivalente a 2% e 5% do valor da demanda mínima.

No que se refere aos raios de cobertura, cada valor de D_m está associado a um conjunto específico de valores de raio. Essa relação é evidenciada na Tabela 2, na qual os valores apresentados na coluna D_m correspondem aos valores indicados na coluna r.

Dessa forma, a combinação de todos os parâmetros resulta nas 78 instâncias citadas. Além desses, o Mourão *et al.* (2024) também usou os valores de $itermax = 100$, $dur = 11$ e $\alpha = 1000$ para, respectivamente, a quantidade máxima de iterações sem melhora, duração de uma proibição tabu e coeficiente de penalidade associada a uma solução infactível, a qual é explicada em mais detalhes nas seções seguintes.

Nesse contexto, todas foram executadas em um computador com o sistema operacio-

nal *Ubuntu 20.04*, equipado com 8 GB de memória RAM e um processador *Intel Core i7-8700*, com frequência de 3.20 GHz e seis núcleos.

4.2 Implementação e comparação dos algoritmos

A busca tabu desenvolvida neste projeto foi implementada na linguagem *Python*, na versão 3.14, com compatibilidade a partir da versão 3.10. Essa linguagem foi escolhida por ser atual e por disponibilizar uma ampla gama de bibliotecas padrão, as quais facilitam o desenvolvimento de software. Outrossim, a versão 2.3 da biblioteca *NumPy* foi usada para que fosse possível definir variáveis de ponto flutuante de precisão simples[1].

Ademais, duas métricas foram utilizadas para a comparação dos algoritmos: a qualidade das soluções e o tempo computacional. Para avaliá-las em relação à primeira, o *gap* entre a qualidade das soluções obtidas pelo algoritmo deste trabalho e pelos outros dois desenvolvidos por Mourão *et al.* (2024) foi calculado. Quanto à segunda métrica, ela foi determinada por meio da cronometragem do tempo gasto para finalizar a execução da instância corrente.

[1] O idealizador da busca tabu cujo algoritmo está sob análise neste trabalho, Fábio Pires Mourão, foi contatado durante o andamento desta pesquisa para que algumas questões acerca da sua implementação fossem esclarecidas. Nesse contexto, foi aconselhada a utilização de variáveis de ponto flutuante de precisão simples em algumas partes desta implementação para que o estudo comparativo feito fosse mais preciso. Tais partes são evidenciadas no código por meio de comentários.

5 IMPLEMENTAÇÃO DA BUSCA TABU

O objetivo desta seção é apresentar, em detalhes, a implementação da busca tabu proposta e conceituada por Mourão *et al.* (2024). A seguir, descreve-se como esses conceitos foram traduzidos para código em *Python* 3.14. Ademais, como o desenvolvimento do projeto resultou em diversos arquivos e módulos auxiliares, são discutidas apenas as classes, métodos e funções essenciais para a compreensão da implementação. O código completo, acompanhado de uma explicação sobre a estrutura do projeto, encontra-se disponível no repositório do *GitHub* [1].

5.1 Criação das instâncias

Como mencionado anteriormente, as instâncias utilizadas neste trabalho são as mesmas empregadas por Mourão *et al.* (2024) e foram disponibilizadas publicamente no repositório do *GitHub* associado ao referido estudo[2]. Assim, considera-se pertinente descrever a natureza dessas instâncias, a fim de proporcionar uma compreensão mais detalhada do problema em análise.

5.1.1 Quanto as instâncias

No repositório, há três arquivos, dos quais são derivadas as 78 instâncias utilizadas. Esses arquivos são nomeados de acordo com os parâmetros empregados na geração das instâncias e, portanto, têm o seguinte formato:

$$GRID\_PSCLP\_n|x|\_m|y|\_d_i\_d_s\_f_i\_f_s\_s1.dat$$

O primeiro aspecto a ser considerado é que essas instâncias são originárias de problemas de cobertura de conjuntos, pois correspondem a instâncias do *Partial Set Covering Location Problem* (PSCLP), denominado *PLFCP* neste trabalho. Assim, a partir dos dados dessas instâncias, será possível inicializar a matriz de cobertura $A_{n\times m}$ e os vetores x e y, ao passo que a implementação da busca tabu será responsável por tratar as restrições adicionais do *PCLFCPFU*.

Outrossim, é possível identificar os parâmetros n, m, d e f presentes na nomenclatura dos arquivos. Eles correspondem, respectivamente, à quantidade de facilidades, à quantidade

[1] https://github.com/autumn-Days/PCLFCPFU_TCC/
[2] https://github.com/fabiopiresmourao/SSCPSCLPSBPO2024

de demandas, ao intervalo de valores das demandas e ao intervalo de custos das facilidades. Desse modo, d_i e d_s fazem referência aos limites inferior e superior dos possíveis valores das demandas; de forma análoga, f_i e f_s referem-se aos limites para os custos de abertura das facilidades. Por fim, $s1$ corresponde ao parâmetro utilizado como *seed* para a geração dos valores pseudoaleatórios das instâncias, que, em todos os casos, é igual a 1.

A compreensão desses aspectos facilita a interpretação do conteúdo dos arquivos. Estruturalmente, esses arquivos são compostos por $1+n+m$ linhas, sendo que a primeira contém apenas os inteiros n e m. Em seguida, encontram-se n linhas com os dados referentes a cada facilidade, as quais seguem o seguinte formato:

$$\begin{array}{ccccc} F & 0 & x_0 & y_0 & f_0 \\ F & 1 & x_1 & y_1 & f_1 \\ F & 2 & x_2 & y_2 & f_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ F & n-1 & x_{n-1} & y_{n-1} & f_{n-1} \end{array}$$

O caractere "F" presente no início de cada linha, indica que os dados em questão se referem às facilidades, enquanto os números da segunda coluna denotam o identificador de cada instalação. Além disso, x_j e y_j, nesse contexto, são números reais não negativos que representam as coordenadas (x,y) de cada facilidade, e f_j representa o custo associado a cada uma delas. Após as n linhas relativas às facilidades, têm início as m linhas correspondentes às demandas, que apresentam um formato semelhante.

$$\begin{array}{ccccc} C & 0 & x_0 & y_0 & d_0 \\ C & 1 & x_1 & y_1 & d_1 \\ C & 2 & y_2 & y_2 & d_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ C & m-1 & x_{m-1} & y_{m-1} & d_{m-1} \end{array}$$

Como se observa, na primeira coluna, em vez de "F", utiliza-se o caractere "C", correspondente ao *cliente*. Na última coluna, encontra-se o valor da demanda associada a esse ponto, enquanto o significado das demais colunas permanece inalterado.

Por fim, observa-se que, nos três conjuntos de instâncias mencionados, os valores de n, d_i, d_s, f_i e f_s permanecem fixos. O único parâmetro variável entre eles é o valor de m, conforme detalhado na Tabela 3.

Tabela 3 – Parâmetros usados para a geração das instâncias.

n	m	d_i	d_s	f_i	f_s
100	$\{1.000; 10.000; 50.000\}$	1	100	10	100

Fonte: Mourão *et al.* (2024).

Desse modo, em todas as instâncias, os valores das demandas situam-se no intervalo $[1, 100]$, enquanto o intervalo dos custos corresponde a $[10, 100]$. Além disso, o número de facilidades permanece fixo em 100, ao passo que a quantidade de clientes varia conforme a instância, assumindo os valores de 1.000, 10.000 e 50.000.

5.1.2 Quanto a inicialização da Matriz de cobertura

Com esses valores, é possível obter a matriz de cobertura de cada uma das 78 instâncias. Para isso, basta que, para cada facilidade y_j e para cada demanda d_i, seja calculada a distância euclidiana entre elas; se essa distância for menor ou igual ao raio de cobertura r da instância, considera-se que d_i está coberta por y_j e, portanto, a célula a_{ij} recebe o valor 1.

Visto que o código referente à inicialização da matriz $A_{n\times m}$ não é essencial para a compreensão da busca tabu, optou-se por não apresentá-lo nesta seção. Além disso, durante o processamento dessa matriz, realizou-se também a inicialização dos vetores x e y.

5.2 Estrutura de dados relevantes

Além da lista tabu e da matriz de cobertura $A_{n\times m}$, a busca tabu também utiliza duas estruturas de dados fundamentais: os vetores x e y. O primeiro representa a matriz de atendimento, que indica a associação entre clientes e facilidades, isto é, qual facilidade atende cada cliente. Já o vetor y registra o estado das facilidades, informando se elas estão abertas ou fechadas. Assim, quando $y_j = 1$, a j-ésima facilidade é considerada instalada; caso contrário, é considerada fechada.

Note que o vetor x desempenha o mesmo papel da matriz de atendimento $Z_{n\times m}$. Desse modo, um dos principais motivos para usar um vetor unidimensional em vez de uma matriz bidimensional é a otimização do uso de memória (Mourão *et al.*, 2024). Isso se deve ao fato de que, devido à restrição da fonte única, uma demanda arbitrária d_i é atendida, no máximo, por apenas uma facilidade y_j. Assim, se a matriz $Z_{n\times m}$ fosse utilizada para esse mapeamento, haveria um grande desperdício de memória, pois cada linha conteria, no máximo, um único

elemento 1. Portanto, caso a demanda mínima corresponda à totalidade dos clientes, seriam armazenadas $(n-1) \times m$ células nulas, caso todos os clientes fossem atendidos.

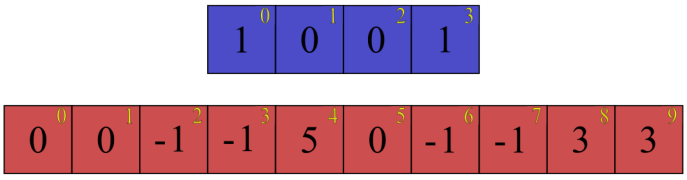
Desse modo, uma forma mais eficiente de manter o registro das alocações dos clientes é, justamente, por meio da utilização do vetor x, de modo que o valor vinculado a x_i corresponda à facilidade y_j que o atende. Assim, de acordo com Mourão *et al.* (2024) x_i pode assumir três valores:

1. j: Caso x_i esteja alocado para a j-ésima facilidade de y.
2. -1: No caso do cliente não estar no raio de cobertura de facilidades abertas.
3. $n+1$: Caso o cliente esteja dentro do raio de cobertura de, ao menos, uma facilidade com o *status* de aberta, mas não esteja sendo atendida por nenhuma.

Desse modo, é importante citar que, na implementação feita neste trabalho, -1 é atribuído aos clientes que não estão no raio de cobertura de facilidade alguma e $n+1$ é vinculado aos clientes que estejam no raio de cobertura de, ao menos, uma facilidade candidata. Esta medida foi tomada porque os *status* de abertura das facilidades não foram considerados fatores relevantes para a vinculação desses valores.

A Figura 5 ilustra possíveis valores para os vetores x e y. Nela, existem 4 facilidades candidatas e 10 clientes a serem atendidos. Também é possível observar que as demandas x_0, x_1 e x_5 estão alocadas em y_0, enquanto x_8 e x_9 estão alocadas em y_3.

Figura 5 – Exemplo de alocações. O vetor azul representa y, enquanto o vermelho, x.



Fonte: O próprio autor.

Já as demais demandas não estão alocadas a nenhuma facilidade. Dentro desse conjunto, apenas a demanda x_4 está no raio de cobertura de alguma facilidade candidata, mas ainda assim não recebe atendimento. As outras demandas dessa categoria permanecem sem atendimento por estarem fora do raio de cobertura de todas as facilidades.

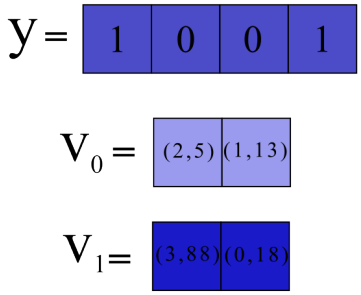
Além dos vetores x e y, também foram implementados, conforme descrito em Mourão *et al.* (2024), dois vetores auxiliares: v_0 e v_1. O vetor v_0 armazena os índices das facilidades de y

cujo *status* é fechado, enquanto v_1 registra os índices das facilidades abertas. Esses vetores são utilizados para otimizar determinadas buscas, como será explicado nas seções seguintes.

Além disso, é relevante explicar como eles foram implementados, pois a sua representação difere ligeiramente do que foi apresentado no artigo base. Nesta implementação, ambos os vetores auxiliares são listas de tuplas, em que o primeiro e o segundo elementos correspondem, respectivamente, ao índice da facilidade no vetor y e ao custo dessa facilidade. Mais uma vez, o motivo dessa escolha ficará mais claro nas seções seguintes.

Desse modo, a Figura 6 ilustra os valores dos vetores v_0 e v_1 para uma dada configuração de y, considerando custos arbitrários para as facilidades. Ademais, é importante notar que v_0 é ordenado em ordem crescente de acordo com o custo das facilidades, à medida que v_1 é ordenado decrescentemente de acordo com o mesmo critério.

Figura 6 – Exemplificação do funcionamento dos vetores auxiliares v_0 e v_1.



Fonte: O próprio autor.

5.3 Modelos

A implementação dos vetores x e y realizada neste trabalho seguiu os mesmos princípios idealizados por Mourão *et al.* (2024), com algumas adaptações. Nesse contexto, o vetor y consiste em uma lista de objetos *Facility*, enquanto o vetor x é formado por uma lista de listas de inteiros, nas quais cada uma contém exatamente três valores: a alocação, o valor da demanda d_i e o seu índice em x, respectivamente. Esse último elemento serve como identificador, permitindo diferenciar os *clientes* alocados para a mesma facilidade e com a mesma demanda. Para a representação dos *pontos de demanda*, um novo tipo chamado *Demand* foi definido, conforme o trecho de código abaixo.

Código-fonte 1 – Definição do tipo *Demand*

```
Demand = Annotated[List[int],"tamanho=3"]
```

Essa representação foi adotada para simplificar a visualização dos dados de x durante o processo de *debugging*, pois, caso uma classe fosse implementada, a visualização desses dados seria dificultada — uma característica secundária, advinda do uso do depurador escolhido. Outro detalhe a ser observado é a especificação do tamanho da lista que representa o tipo *Demand* por meio do módulo *Annotated*. Esse aspecto é equivalente a um comentário e não restringe, de fato, o tamanho da lista, nem altera sua utilização. Desse modo, por meio desta representação, caso x correspondesse a:

$$[[0,5,0],[-1,1,1],[-1,15,2],[-1,10,3],[0,10,4]]$$

Isso significaria que apenas o primeiro e o último *cliente* seriam atendidos por y_0, possuindo, respectivamente, cinco e dez demandas. Já x_1, x_2 e x_3 não seriam atendidos, pois não estão no raio de cobertura de nenhuma facilidade, possuindo, respectivamente, uma, quinze e dez demandas. Quanto à classe *Facility*, sua implementação é apresentada no Código-fonte 2.

Código-fonte 2 – Classe *Facility*.

```
class Facility:
    def __init__(self, capacity:int, cost:float, status:
       bool=False, coverage:int=0, served:int = 0) -> None:
        self.capacity = capacity
        self.cost = cost
        self.status = status
        self.coverage = coverage
        self.served = served

    def close(self) -> None:
        self.status = False
        self.served = 0

    def open(self) -> None:
        self.status = True
```

```
15
16      def hasSpace4Dem(self,dem:Demand) ->bool:
17          return (self.served+dem[1] <= self.capacity)
```

Como é possível constatar, a classe *Facility* tem 5 atributos, dos quais somente dois são obrigatórios para que um objeto seja instanciado: *capacity* e *cost*. Os demais atributos são inicializados com valores padrão e, posteriormente, serão atualizados conforme a progressão da busca tabu.

Dentre esses atributos, *status* é responsável por indicar se a facilidade está instalada ou não; *coverage* informa a quantidade de demandas que estão localizadas dentro do seu raio de cobertura; e *served* registra a quantidade de demandas que estão efetivamente atendidas pela facilidade. Perceba que esses dois últimos atributos dizem respeito à quantidade de *demandas*, e não à quantidade de *pontos de demanda*.

Além dos atributos, é possível observar a presença de três métodos. Assim, *close* é responsável por fechar a facilidade ao reinicializar os atributos *status* e *served*; *open* é uma forma alternativa de inicializar o atributo *status* com *verdadeiro*; e, por último, *hasSpace4Dem* retorna um valor booleano que indica se a facilidade tem capacidade suficiente para atender a uma certa demanda.

5.4 Representação da solução

No trabalho de Mourão *et al.* (2024), a solução foi representada como uma tupla dos vetores x e y. Neste trabalho, porém, uma classe *Sol* foi modelada para representar uma solução. Ela é responsável por armazenar os vetores x e y, bem como os vetores auxiliares v_0 e v_1, além de outros atributos e métodos. Para simplificação, apresentam-se nesta seção apenas os seus atributos e, conforme necessário, seus métodos serão apresentados nas seções posteriores.

Código-fonte 3 – Classe *Sol*

```
1  class Sol:
2      def __init__(self,facilities:List[Facility]=[],
           allocations:List[Demand]=[],minDemand:int=float("inf
           "),penalty:float=float("inf")):
3          self.facilities = facilities
```

```
4           self.allocations = allocations
5           self.objValue:float = float("inf")
6           self.totalCost = float("inf")
7           self.minDemand = minDemand
8           self.totalServed:int = 0
9           self.openFacil = []
10          self.closeFacil = []
11          self.penalty = penalty
12          self.waitingFlag = len(self.facilities)+1
```

Em relação aos atributos anteriormente introduzidos, é possível notar que os vetores x e y correspondem, respectivamente, a *allocations* e *facilities*, e os vetores auxiliares v_0 e v_1 estão representados na classe por meio de *closeFacil* e *openFacil*. Além disso, é conveniente introduzir os demais atributos; a lista a seguir os detalha.

- *objValue*: Representa o valor da solução quando avaliada pela função objetivo, que será detalhada nas seções seguintes.
- *totalCost*: O custo total associado à instalação das facilidades descritas em *openFacil*.
- *minDemand*: A demanda mínima D. Inicialmente, esse atributo é definido como infinito.
- *totalServed*: Quantidade total de demandas atendidas pelo conjunto de facilidades abertas.
- *penalty*: Como será apresentado adiante, uma penalidade proporcional à quantidade de demandas que faltam para que D seja alcançado é aplicada à *objValue*. Assim, o atributo *penalty* representa o valor da penalidade por cliente faltante. Por padrão, ele é inicializado com valor infinito.
- *waitingFlag*: Representa o valor $n+1$ atribuído aos pontos de demanda que estão dentro do raio de cobertura de alguma facilidade, mas que ainda não estão sendo atendidas.

5.5 Espaço de busca e vizinhança

O espaço de busca consiste em todos os valores possíveis para o vetor y e, consequentemente, também em todas as possíveis alocações e realocações das demandas expressas em x, decorrentes da configuração de y.

Nesse contexto, conforme Mourão *et al.* (2024), uma solução vizinha é gerada pela inversão de *apenas* um dos bits de y. Assim, caso y corresponda a $[1,0,0,1]$, uma solução vizinha

pode ser obtida ao inverter y_0, o que fará com que o componente y de s' corresponda a $[0,0,0,1]$ e exigirá a realocação das demandas de x que estavam previamente vinculadas a y_0, já que um *movimento de remoção* foi realizado.

Outra solução vizinha poderia ser obtida por meio de um *movimento de instalação* em y_1, o que resultaria em uma solução vizinha em que o componente y seria $[1,1,0,1]$ e exigiria a alocação dos clientes ainda *em espera* a y_1. A forma como a realocação e alocação ocorrem é descrita na seção seguinte.

5.6 Obtenção da solução inicial

A busca tabu desenvolvida neste trabalho, assim como a conceituada por Mourão *et al.* (2024), obtém uma solução inicial gerada por uma *heurística gulosa*. Nas duas subseções a seguir, o funcionamento dessa heurística auxiliar é explicado e, em seguida, a sua implementação é apresentada e detalhada.

5.6.1 Conceitualização

O propósito da heurística gulosa em questão é selecionar as facilidades mais promissoras para serem abertas com base em um critério específico. Nesse caso, o critério consiste em atribuir um quociente q a cada facilidade, que expressa a quantidade de demandas dentro de seu raio de cobertura — limitado pela sua capacidade — dividida pelo custo de instalação. Ao considerar os atributos da classe *Facility*, o valor de q é obtido conforme a expressão apresentada a seguir.

$$min(capacity, coverage) \div cost$$

Nesse sentido, as facilidades mais promissoras são aquelas que tiverem o maior quociente q e, portanto, serão as primeiras a atender aos clientes que estão em seu raio de cobertura. Desse modo, assim que não for mais possível alocar clientes à facilidade em questão, seja pelo esgotamento dos clientes não atendidos ou pela insuficiência de espaço, os clientes restantes são alocados às facilidades subsequentes. Esse processo continua até que a demanda mínima seja atendida.

Ademais, pode-se dizer que os clientes são alocados às facilidades baseando-se unicamente na preservação das restrições inerentes ao problema, ou seja, o respeito às capacidades, a limitação do atendimento dos clientes a, no máximo, uma facilidade e, é claro, os

clientes estarem sendo cobertos pela facilidade que os atendem. Portanto, o principal objetivo da heurística em questão é a configuração do componente *y* da solução, enquanto a alocação dos clientes é um aspecto secundário, realizado sem a preocupação com a otimização do espaço das facilidades.

5.6.2 Implementação

Considerando que a implementação da heurística gulosa possui uma extensão significativa, o que pode comprometer a fluidez da leitura, as subseções seguintes apresentam e descrevem, de forma estruturada, trechos da função `greedyHeuristic`, responsável pela implementação da funcionalidade em questão. Os trechos são exibidos na mesma ordem em que aparecem no código-fonte original.

5.6.2.1 Entrada e saída

Antes de apresentar a implementação propriamente dita da heurística gulosa, é fundamental compreender a sua assinatura. A função que a implementa, denominada *greedyHeuristic*, recebe como entrada a lista de facilidades (*facilities*), a lista de alocações (*allocations*), a matriz de cobertura (*coverageMatrix*) previamente inicializada e, por fim, a demanda mínima (*minDemand*). No que diz respeito à saída, a função retorna um inteiro que corresponde à quantidade de demandas atendidas pela solução inicial.

Código-fonte 4 – Assinatura de *greedyHeuristic*.

```
6 def greedyHeuristic(facilities:List[Facility],
7     allocations:List[Demand],
8     coverageMatrix:List[List[int]],
9     minDemand:int) -> int:
```

Por último, é importante notar que as listas *facilities* e *allocations* são modificadas por referência[3], o que faz com que os componentes *y* e *x* da solução inicial não precisem ser retornados.

[3] Em *Python*, objetos são passados por referência para funções e métodos, não por cópia. Logo, uma vez que listas são instâncias da classe *list*, *facilities* e *allocations* são passadas por referência por padrão.

5.6.2.2 *Inicialização de variáveis*

Quanto à execução da heurística gulosa propriamente dita, o algoritmo começa com a inicialização de algumas variáveis importantes, conforme mostrado no trecho de código a seguir.

Código-fonte 5 – Declaracação de variáveis.

```
10      totDemandServed:int =  0
11      waitingFlag:int = len(facilities)+1
12      quocients:List[float] = facUtils.calcQuocients(
           facilities)
```

Desse modo, é possível observar que três variáveis são declaradas. A variável *totDemandServed* mantém o registro do total de demandas atendidas pelo conjunto de facilidades; *waitingFlag* desempenha a mesma função do atributo de mesmo nome da classe *Sol*; e *quocientes* armazena a saída da função *calcQuocients*, que recebe uma lista de objetos *Facility* e calcula o quociente q de cada um, de modo que a j-ésima facilidade corresponda ao j-ésimo quociente. Em seguida, são inicializadas duas variáveis adicionais.

Código-fonte 6 – Declaração de *unservedDemands* e de *indexFacilities*.

```
13      unservedDemands:Set = demUtils.waitingDems(allocations,
           waitingFlag)
14      indexFacilities:List = [i for i in range(len(facilities
           ))]
```

Esse trecho inicializa duas estruturas de dados. Desse modo, *unservedDemands* consiste em um *Set* que tem por elementos tuplas de três valores, que correspondem aos pontos de demanda presentes em *allocations*, convertidos em tuplas pela função *waitingDems*. Além disso, somente os pontos de demanda que estão no raio de cobertura de, ao menos, uma facilidade pertencem a esse conjunto. Quanto à escolha da estrutura de dados, o *Set* foi adotado por permitir a remoção de elementos facilmente por meio da operação de *diferença entre conjuntos*.

Já na linha subsequente, uma lista de índices é criada. Ela é responsável por armazenar os índices dos elementos de *facilities*, de forma que o j-ésimo elemento de *indexFacilities*

corresponda ao índice da j-ésima facilidade. A princípio, essa medida pode parecer sem propósito, visto que já é possível acessar as facilidades por meio de seus próprios índices, sem o auxílio de uma estrutura de dados auxiliar. Contudo, a finalidade dessa ação torna-se mais clara à luz do funcionamento da heurística gulosa.

Vale lembrar que as demandas devem ser alocadas às facilidades associadas ao maior quociente q, o que naturalmente leva o programador a ordenar a lista *facilities* em ordem decrescente segundo esse critério. Entretanto, caso essa medida seja adotada, a j-ésima facilidade de *facilities* não corresponderia mais à j-ésima coluna de *coverageMatrix*, o que é problemático, uma vez que a manutenção dessa correspondência é indispensável para que as alocações ocorram ao longo do algoritmo.

Para contornar esse problema, a solução adotada foi mapear uma lista de índices à lista *quocients* e ordená-la de acordo com o valor dos elementos dessa lista. Desse modo, é possível preservar a relação entre os elementos de *facilities* e as colunas de *coverageMatrix*, enquanto também é possível obter as facilidades mais promissoras de acordo com o critério da heurística gulosa.

5.6.2.3 Ordenação da lista de índices

Para que essa ordenação fosse possível, foi necessário ordenar a lista de índices conforme a lista de quocientes, isto é, definir como critério de ordenação da lista *indexFacilities* a lista *quocients*. Neste sentido, o trecho a seguir descreve o processo adotado.

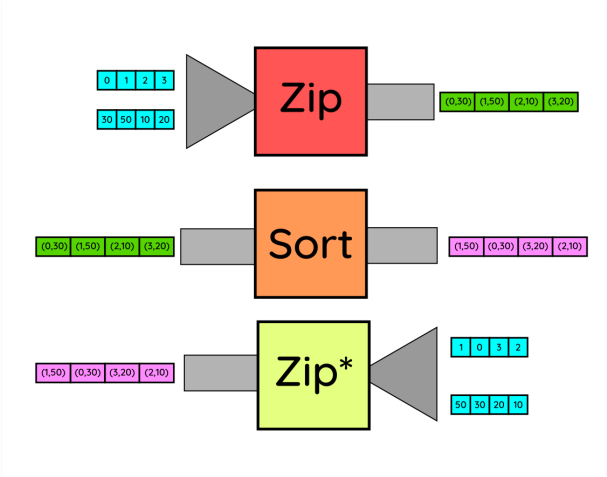
Código-fonte 7 – Ordenação de acordo os quocientes.

```
15    combined:List[Tuple[int,float]] = list(zip(
          indexFacilities,quocients))
16    combined.sort(key = lambda elem : elem[1], reverse=True
          )
17    indexFacilities,_ = zip(*combined)
```

Como é possível observar, a estratégia usada consistiu em combinar a lista de índices com a lista de quocientes, o que resultou na criação de uma lista de tuplas *combined*, composta por elementos $(\text{índice}_j, \text{quociente}_j)$. Em seguida, *combined* é ordenada de forma decrescente de acordo com o valor do quociente de cada tupla. Por fim, a lista *indexFacilities* é atualizada para

corresponder à sua versão ordenada de acordo com os quocientes associados a cada facilidade em *facilities*. A Figura 7 ilustra o funcionamento desse procedimento.

Figura 7 – Ilustração do funcionamento da ordenação da lista de índices. Na figura, duas listas, ambas representadas em azul, são passadas para a função *zip*. A de cima representa a lista de íncides, já a de baixo, a lista de quocientes. Ao final do processo, ambas são ordenadas e retornadas pela função *zip estrela*.



Fonte: O próprio autor.

5.6.2.4 Funcionamento central

Após a execução desses passos, chega-se à etapa central do algoritmo. Como este trecho de código é mais extenso do que os anteriores, sua explicação será apresentada por meio de uma lista de etapas, de modo a tornar a leitura mais clara e organizada.

Código-fonte 8 – Funcionamento da heurística gulosa.

```
18      j:int = 0
19      while ((j != len(indexFacilities)) and
20      (unservedDemands) and
21      (totDemandServed < minDemand)):
22          iFacility = indexFacilities[j]
23          facility = facilities[iFacility]
24          if (facility.coverage > 0):
25              capacity:int = facility.capacity
26              newServedDemands:Set = None
27              totDemandServed, newServedDemands = demUtils.
```

```
                allocate(coverageMatrix,facility,iFacility,
                allocations,totDemandServed,minDemand,
                waitingFlag,unservedDemands)
28              if (newServedDemands):
29                  facility.open()
30                  unservedDemands -= newServedDemands
31          j += 1
32      return totDemandServed
```

- 18: Inicialização da variável que será responsável por servir como índice de *facilities*.
- 19–21: Essas são as condições que determinam a continuidade da execução da heurística. Intuitivamente, podem ser entendidas da seguinte forma: "enquanto ainda existirem facilidades a serem percorridas, houverem clientes não atendidos e a demanda mínima não tiver sido atingida, a execução deve continuar".
- 22–23: Obtenção do j-ésimo elemento de *facilities*.
- 24–27: Caso a facilidade em questão possua algum ponto de demanda em seu raio de cobertura, a função *allocate* é chamada e recebe, entre outros parâmetros, *unservedDemands*, ou seja, o conjunto de clientes ainda não atendidos. O propósito dessa medida é fazer com que somente os clientes que ainda precisem ser alocados sejam analisados. Além disso, como mencionado anteriormente, a alocação dos clientes é feita sem preocupações quanto à otimização do espaço das facilidades, então *allocate* aloca os clientes assim que é constatado que o seu atendimento não viola as restrições do PCLFCPFU. Além disso, *allocate* também é responsável por atualizar *facility* por referência ao incrementar seu atributo *served*; por fim, a quantidade total de demandas atendidas *totDemandServed* é atualizada, e os clientes que acabaram de ser atendidos são adicionados a *newServedDemands*.
- 28–31: Dado que existe a possibilidade de *allocate* ser chamada e nenhum cliente seja efetivamente alocado — devido à insuficiência de capacidade, por exemplo —, torna-se necessária uma verificação prévia da nulidade de *newServedDemands* antes de atualizar o *status* da facilidade em questão. Assim, caso a condicional seja satisfeita, *unservedDemands* é atualizada, de modo que apenas os clientes que ainda necessitam de alocação sejam analisados nas próximas iterações. Concluída essa etapa, o loop prossegue com sua execução até que a condicional do *while* deixe de ser satisfeita.

- 32: A quantidade total de demandas atendidas é retornada.

5.7 Função objetivo

A função objetiva é dada pelo somatório dos custos das facilidades abertas, acrescido de uma penalidade proporcional à quantidade de demandas faltantes para que D_m sejam atingidas, denotada por $\bar{D}$. Dessa forma, é possível expressá-la da seguinte forma:

$$f(s) = \sum_{j \in J} y_j f_j + \alpha \bar{D}$$

Dessa forma, α representa o coeficiente de penalidade e, ao considerar os atributos da classe *Sol*, $\bar{D}$ pode ser expresso como $\max(0,\ minDemand - totalServed)$. Observa-se que a penalização ocorre somente quando a demanda mínima não é atendida. Portanto, caso a quantidade de demanda atendida exceda o mínimo estabelecido, nenhuma penalidade será aplicada, pois o valor zero será maior do que qualquer número negativo resultante da subtração entre *minDemand* e *totalServed*.

Além disso, ressalta-se que o *PCLFCPFU* é um problema de minimização e, portanto, ao adicionar um valor à soma dos custos das facilidades, a qualidade da solução é deteriorada. Por esse motivo, o valor referente à penalidade é somado, e não subtraído, ao custo total.

No código implementado, a função objetivo foi implementada como um método da classe *Sol*. Assim, é necessário percorrer o conjunto de facilidades abertas apenas uma vez para calcular o custo total. Esse cálculo é realizado logo após a obtenção da solução inicial por meio da heurística gulosa.

Desse modo, para calcular a qualidade das soluções ao longo das iterações da busca tabu, basta atualizar o atributo *totalCost* somando ou subtraindo o custo da facilidade cujo *status* foi alterado quando a busca passou para a solução vizinha.

Além disso, o atributo $totalServed$ é atualizado sempre que uma visita é realizada, de modo que $\bar{D}$ é obtido sem a necessidade de realizar uma busca linear sobre $openFacil$ sempre que a função objetivo for invocada. A implementação correspondente é apresentada a seguir.

Código-fonte 9 – Método de avaliação.

```
1 def evaluate(self, cost:float=float("inf"), opened=True,
    batch=False) -> None:
```

```
2     fine:float = max(0, self.minDemand - self.totalServed)
          * self.penalty
3     totalCost:float = 0
4     if (batch):
5         for i in range(len(self.openFacil)):
6             totalCost += self.openFacil[i][1]
7     else:
8         op = operator.add if (opened) else operator.sub
9         totalCost = op(self.totalCost,cost)
10    self.totalCost = totalCost
11    self.objValue = self.totalCost + fine
```

Primeiramente, é necessário analisar os parâmetros de entrada do método. O último parâmetro, *batch*, é uma variável booleana que define a forma como o somatório dos custos será calculado. Quando *batch* é verdadeiro, todas as facilidades abertas são percorridas em um loop para calcular o custo total. Caso contrário, aplica-se a técnica descrita nos parágrafos anteriores. O parâmetro *opened* indica se o movimento realizado foi de instalação ou remoção. Por fim, *cost* representa o custo da facilidade em questão e, por padrão, é inicializado com o valor infinito.

Na linha 2, a penalidade *fine*[4] é calculada conforme o procedimento descrito anteriormente. Caso a condição na linha 4 seja verdadeira, o somatório dos custos de abertura é calculado por meio de um loop. Caso contrário, executa-se a condição da linha 7, no qual, se o movimento realizado for de instalação, o custo da facilidade é adicionado a *totalCost*, ou, caso seja uma remoção, é subtraído. Por fim, nas linhas 10 e 11, são atualizados os atributos *totalCost* e *objValue*.

5.8 Lista tabu e critério de aspiração

Tanto o vetor tabu T, descrito por Mourão *et al.* (2024), quanto a lista tabu implementada neste trabalho consistem em estruturas do mesmo tamanho que y, de modo a existir uma correspondência direta entre y_j e T_j. Assim, T_j armazena o valor da última iteração na qual a modificação de y_j está proibida, indicando, portanto, o período durante o qual um movimento associado a y_j permanece *tabu*.

[4] *Fine* vem do inglês, *multa*.

Desse modo, inicialmente, todos os elementos de T são definidos como -1. Assim que y_j é modificado, T_j recebe o valor $dur + iter$, em que dur corresponde à quantidade de iterações durante as quais y_j não poderá ser alterado, e $iter$ é um contador que registra a iteração atual. Dessa forma, a soma desses dois valores define a última iteração em que um movimento associado a y_j ainda será considerado tabu.

Portanto, ao realizar qualquer movimentação para uma solução vizinha, verifica-se se T_j é maior ou igual a $iter$[5]. Caso essa condição seja satisfeita, y_j ainda constitui uma configuração tabu. Caso contrário, dependendo de outras condições que serão apresentadas nas seções posteriores, a solução é visitada e T_j recebe o valor $iter + dur$[6].

Nesse contexto, uma solução tabu s^t só pode ser visitada caso atenda ao critério de aspiração, ou seja, caso seja melhor do que a melhor solução s^* já encontrada. O funcionamento da lista tabu e do critério de aspiração é detalhado em maior profundidade na Seção 5.11, na qual o código correspondente à implementação da busca tabu é apresentado com mais detalhes.

5.9 O subproblema da alocação

Como comentado anteriormente, uma das características deste problema é que ele não se trata apenas da seleção de quais facilidades serão abertas, mas também da seleção de quais demandas serão alocadas para cada uma delas. Nesse sentido, surge o *subproblema da alocação*, cujo foco é justamente fazer essa seleção.

Nesta seção, é apresentado como a técnica de alocação e realocação de clientes descrita por Mourão *et al.* (2024) foi implementada. Outrossim, vale notar que, em alguns casos específicos, um resolvedor de *PLI* era invocado para resolver o problema. Desse modo, uma vez que o objetivo deste trabalho é analisar o desempenho da busca tabu sem o auxílio de métodos determinísticos, optou-se por não implementar o modelo de *PLI* proposto.

Em virtude da extensão do código, as subseções seguintes são dedicadas à explicação dos dois fluxos de execução principais: o caso em que a solução atual é factível e o caso em que ela é infactível.

---

[5] Aliás, o motivo pelo qual os valores iniciais dos elementos de T são -1 e não 0 é justamente que, na primeira iteração, em que $iter$ corresponde a 0, é necessário que esses valores iniciais sejam negativos para que a comparação *T[iter] >= iter*, que indica se uma configuração é tabu, seja falsa.

[6] Perceba que isso não corresponde a um incremento do valor anterior de T_j, mas sim à atribuição de um novo valor a T_j.

5.9.1 A solução atual é factível

Primeiramente, a sub-rotina responsável por resolver o *subproblema de alocação* foi implementada por meio de um método da classe *Sol*. Assim, além de realizar as alocações, ele também é responsável por atualizar o *status* da facilidade que acaba de ser instalada ou removida. Desse modo, ele configura tanto o componente *y* quanto o *x* da solução. Portanto, uma vez que está envolvido em todas as etapas da obtenção de uma solução vizinha, esse método foi chamado de *move*.

Desse modo, ela recebe três parâmetros: *pIter*, que, nesse contexto, representa o índice *p* de uma das listas auxiliares, seja de *openFacil* ou de *closeFacil*, dependendo da factibilidade da solução atual; a matriz de cobertura *coverageMatrix*; e *is2install*, que indica se o movimento a ser realizado se trata de uma *instalação* ou de uma *remoção*. Ao final, todos os atributos pertinentes são atualizados e nenhum valor é retornado.

Código-fonte 10 – método de movimentação: *s* é factível.

```
1  def move(self, pIter:int, coverageMatrix:List[List[int]],
2           is2install=False) -> None:
3      if (self.isFeasible()):
4          if (is2install):
5              facility:Tuple[int,int] = self.closeFacil.pop(
                   pIter)
6              indexFacility:int = facility[0]
7              self.__install(indexFacility, coverageMatrix)
8              self.__updateAuxList(facility,openFacil=True)
9          else:
10             facility:Tuple[int,int] = self.openFacil.pop(
                   pIter)
11             indexFacility = facility[0]
12             self.__close(indexFacility, coverageMatrix)
13             self.__updateAuxList(facility,closeFacil=True)
```

A execução do método propriamente dito começa com a avaliação da factibilidade da solução atual, na linha 2, por meio do método auxiliar *isFeasible*. Caso ela seja factível,

ainda existem duas possibilidades a serem avaliadas, de acordo com a conceitualização feita por Mourão *et al.* (2024): se um movimento de instalação ou de remoção está sendo feito.

Na primeira possibilidade, a facilidade de menor custo que ainda não foi aberta é instalada e tem demandas alocadas para ela. Esse processo é feito pela remoção do p-ésimo elemento de *closeFacil*, assim, uma tupla (y_j, c_j) é obtida. Em seguida, o método *__install* é invocado, sendo responsável por atualizar o status de y_j e alocar as demandas cabíveis[7] para ela, sem que haja a preocupação quanto à otimização do espaço da facilidade. Por último, a lista auxiliar *openFacil* é atualizada, uma vez que um elemento de *closeFacil* foi removido e precisa ser inserido novamente em uma lista auxiliar, nesse caso, em *openFacil*[8].

Já na segunda possibilidade, a p-ésima facilidade mais cara é removida e as demandas que antes estavam alocadas para ela são realocadas entre as facilidades que ainda estão abertas. Mais uma vez, a realocação é feita de modo a cumprir apenas as restrições do PCLFCPFU, sem que haja preocupações quanto à otimização do espaço. Em seguida, a tupla (y_j, c_j) é adicionada à *closeFacil*, de modo que sua ordenação seja preservada.

Após essa remoção, é possível que a solução deixe de ser factível e, por isso, logo após a linha 13 e ainda no escopo do *else*, na versão do algoritmo proposta por Mourão *et al.* (2024), verifica-se a factibilidade da solução gerada. Caso ela seja infactível, um *solver PLI* é chamado para maximizar a alocação das demandas para as facilidades abertas. Desse modo, o espaço das facilidades é otimizado, permitindo que mais demandas possam ser alocadas com o mesmo conjunto de facilidades abertas. No entanto, como este trabalho busca analisar o desempenho da busca tabu sem o auxílio de métodos determinísticos, essa parte do algoritmo não foi implementada.

5.9.2 A solução atual é infactível

Já no caso em que a solução é infactível, só existe uma opção de movimentação: *instalação*. Isso se deve à constatação de que, se não houver demandas atendidas o suficiente, a forma ordinária de resolução é por meio da instalação de uma nova facilidade e da alocação de demandas para ela, tal como já foi descrito na subseção anterior. Vale pontuar que, mesmo após a instalação e a alocação de demandas pelo método já descrito, ainda existe a possibilidade de a

[7] Ou seja, aquelas que ainda não foram atendidas pelas demais facilidades, que estão no seu raio de cobertura e que não excedam a capacidade da facilidade em questão.

[8] Vale destacar que, no método *__updateAuxList*, os elementos são inseridos nas listas auxiliares de modo a preservar sua ordem de acordo com o custo.

solução gerada ser infactível. Por isso, na versão proposta por Mourão *et al.* (2024), o mesmo *solver* citado na subseção anterior é chamado para otimizar a alocação das demandas quando esse caso se concretiza. Dessa forma, como já mencionado, esse *solver* não é chamado nesta implementação, pois isso foge ao escopo deste trabalho.

Código-fonte 11 – método de movimentação: *s* é infactível.

```
16      else:
17          facility:Tuple[int,int] = self.closeFacil.pop(pIter
                )
18          facilityIndex = facility[0]
19          self.__install(facilityIndex,coverageMatrix)
20          self.__updateAuxList(facility,openFacil=True)
```

5.10 Funções auxiliares usadas na busca tabu

Antes de se falar da implementação da *busca tabu* propriamente dita, é importante que as funções auxiliares usadas por ela sejam apresentadas. Nesse sentido, esta seção descreve o funcionamento de duas funções que são usadas com muita frequência ao longo do algoritmo: *install* e *remove*.

Diferente dos métodos *__install* e *__close* da classe *Sol*, que objetivavam atualizar os *status* de uma facilidade e alocar ou realocar suas demandas, essas duas funções auxiliares são responsáveis por abstrair comportamentos repetitivos que ocorrem durante a execução da busca tabu quando movimentos de *instalação* ou de *remoção* são feitos. Além disso, elas também atualizam as variáveis que forem pertinentes para as iterações futuras da metaheurística.

Nesse sentido, a primeira coisa a ser pontuada sobre essas duas funções é que elas servem apenas como uma interface para *doOperation*, que é uma terceira função responsável por, de fato, executar esses comportamentos e atualizar os dados pertinentes.

Essa escolha de *design* deve-se ao fato de o funcionamento de *install* e *remove* ser análogo. A única característica que as difere é que, enquanto a primeira acessa os dados de *closeFacil*, a segunda acessa os de *openFacil*. A implementação de ambas é apresentada abaixo.

Código-fonte 12 – install e remove

```
1 def install(s:Sol,sTabu:Sol,coverageMatrix:List[List[int]],
    tabu:List[int],i:int,pIter:int) -> Tuple[Sol,Sol,int]:
2     return doOperation(s,sTabu,coverageMatrix,tabu,i,pIter,
        is2install=True)
3
4 def close(s:Sol,sTabu:Sol,coverageMatrix:List[List[int]],
    tabu:List[int],i:int,pIter:int) -> Tuple[Sol,Sol,int]:
5     return doOperation(s,sTabu,coverageMatrix,tabu,i,pIter,
        is2install=False)
```

A similaridade entre as funções é evidente. Nesse aspecto, além os nomes serem diferentes, o parâmetro *is2install* passado para *doOperation* no retorno delas também é diferente. Enquanto *install* atribui *verdadeiro* e sinaliza que um movimento de instalação será feito, *remove* atribui o valor *falso* para esse parâmetro, o que faz com que um movimento de *remoção* seja realizado.

No que tange à entrada, dois objetos *Sol* são passados: a solução atual *s* e *sTabu*, que representa a melhor solução tabu encontrada durante uma iteração da metaheurística. Além desses, a matriz de cobertura, a lista tabu e os inteiros *i* e *pIter* também são passados. Em relação a esses últimos, *i* guarda a iteração atual da busca tabu, e *pIter* é um índice de uma das listas auxiliares — *closeFacil*, em caso de movimentos de instalação e *openFacil*, caso contrário.

Já no que se refere ao *output*, ambas as funções retornam a saída de *doOperation*, que, em todos os casos, se trata de uma tupla de três elementos. O primeiro diz respeito à solução vizinha gerada; o segundo, à atualização de *sTabu*; e o terceiro corresponde ao índice da facilidade que foi instalada ou removida durante a movimentação.

Ao compreender esses conceitos fundamentais acerca das funções *install* e *remove*, é pertinente que o funcionamento de *doOperation* seja descrito. Desse modo, seu código é apresentado logo abaixo.

Código-fonte 13 – doOperation

```
1 def doOperation(s:Sol,sTabu:Sol,coverageMatrix:List[List[
    int]],tabu:List[int],i:int,pIter:int,is2install:bool) ->
     Tuple[Sol,Sol,int]:
```

```
2       sNeighbour:Sol = cp.deepcopy(s)
3       sNeighbour.move(pIter,coverageMatrix,is2install=
            is2install)
4
5       config:int = None
6       if is2install:
7           config = s.closeFacil[pIter][0]
8           cost = s.facilities[config].cost
9           sNeighbour.evaluate(cost,opened=True)
10      else:
11          config = s.openFacil[pIter][0]
12          cost = s.facilities[config].cost
13          sNeighbour.evaluate(cost,opened=False)
14
15      if (tabu[config] >= i) and (sNeighbour < sTabu):
16          sTabu = cp.deepcopy(sNeighbour)
17
18      return (sNeighbour,sTabu,config)
```

A primeira coisa a ser notada é que uma cópia da solução atual *s* é feita e, com base nessa cópia, um movimento de instalação ou remoção é realizado, dependendo do valor de *is2install*. Assim, todos os atributos de *s* permanecem inalterados, enquanto *sNeighbour* corresponde à sua versão após a realização de um movimento.

Logo em seguida, o fluxo condicional das linhas 6 a 13 visa obter o índice da facilidade que protagonizou o movimento que deu origem a *sNeighbour*. Desse modo, o índice é salvo na variável *config* para que seja possível verificar se a movimentação em questão adveio de uma configuração tabu. Além disso, ambos os fluxos condicionais também são responsáveis por avaliar a qualidade de *sNeighbour*.

Desse modo, na linha 15, é verificado se $facilities_{\text{config}}$ se caracteriza como uma configuração tabu e se a solução gerada é melhor avaliada do que *sTabu*[9]. Em caso afirmativo,

[9] O operador de menor (<) da classe *Sol* foi sobrecarregado para que a comparação feita entre as suas instâncias tomem como critério de avaliação o atributo "objValue". Por causa disso, a comparação realizada na linha 49 verifica se *s_aux* é melhor avaliado que *sNeighbour*.

sTabu é atualizada e passa a armazenar uma cópia de *sNeighbour*. Por fim, uma tupla composta por *sNeighbour*, *sTabu* e *config* é retornada.

5.11 Algoritmo desenvolvido

Nesta seção, apresenta-se a implementação da busca tabu aplicada ao *PCLFCPFU*. Assim, descreve-se como os conceitos apresentados nas subseções anteriores se integram ao algoritmo completo. Ademais, por meio dessa explicação, também são introduzidos outros aspectos relevantes para o funcionamento da meta-heurística, como o critério de visitação.

Outrossim, como a implementação em Python corresponde ao pseudocódigo apresentado por Mourão *et al.* (2024), esta seção é dedicada exclusivamente à apresentação da implementação desenvolvida, uma vez que sua descrição também implica a explicação do referido pseudocódigo. Desse modo, as diferenças entre as implementações são pontuadas de forma clara, de modo a possibilitar a compreensão da lógica do algoritmo base e quaisquer diferenças também são pontuadas, porém também é possível consultar o pseudo-código idealizado por Mourão *et al.* (2024) no *Apêndice A*.

5.11.1 Entrada, saída e inicialização das variáveis

Além das estruturas de dados principais — *facilities*, *allocations* e a matriz de cobertura, *coverageMatrix* — a função *tabuSearch* recebe ainda outros quatro parâmetros. Nesse conjunto, *minDemand*, que já foi apresentado; *maxIter*, que estabelece o limite máximo de iterações sem melhoria; *tabuTimeOut*, que define a quantidade de iterações durante as quais uma configuração de *facilities* permanecerá como tabu; e, por fim, *penalty*, que representa o coeficiente de penalidade.

Código-fonte 14 – inicialização das principais variáveis e estruturas de dados da busca tabu.

```
8  def tabuSearch(facilities:List[Facility], allocations:List[
       Demand],coverageMatrix:List[List[int]], minDemand:int,
       maxIter:int,dur:int,penalty:float)-> Tuple[Sol,bool]:
9
10     config:int = -1
11     s:Sol = Sol(facilities, allocations, minDemand, penalty
          =penalty)
```

```
12      s.calcGreedySol(coverageMatrix)
13      sStar:Sol = cp.deepcopy(s)
14
15      i:int = 0
16      bestIter:int = 0
17      tabu:List[int] = [-1 for i in range(len(facilities))]
```

A partir da linha 10, têm início as inicializações das variáveis. Como o algoritmo envolve diversas variáveis e algumas etapas preparatórias antes de sua execução propriamente dita, a lista de pontuações a seguir resume o propósito de cada uma delas.

- *config*: Responsável por manter o registro da *configuração tabu* da lista *facilities*, isto é, o índice da facilidade que sofreu uma penalização durante a iteração em questão.
- *s*: diz respeito à solução atual. Na linha 11, ela é instanciada, e na linha 12 seus atributos são atualizados para corresponderem ao resultado da heurística gulosa. Nesse sentido, o método *calcGreedySol* invoca *greedyHeuristic* internamente e atualiza os atributos pertinentes, neste caso, *objValue*, *totalServed* e as listas auxiliares *openFacil* e *closeFacil*.
- *sStar*: representa a melhor solução encontrada até o momento. Inicialmente, ela é inicializada com uma cópia de *s*. Essa cópia é gerada pela função *deepcopy*[10].
- *i* e *bestIter*: a primeira corresponde à variável de iteração do loop principal da busca tabu, enquanto a segunda registra a iteração em que a melhor solução foi encontrada até então.
- *tabu*: corresponde à lista tabu. Inicialmente, todos os seus elementos começam com -1, o que indica que nenhuma das facilidades em *facilities* é considerada uma configuração tabu.

5.11.2 Primeiro fluxo de execução

No que tange à execução da metaheurística propriamente dita, o *loop* principal da busca-tabu é definido na linha 18, tal como foi pontuado durante a exposição do seu arquétipo. Já nas duas linhas seguintes, os objetos *sTabu* e *sNeighbour* são instanciados[11]. A primeira armazenará a melhor s^t avaliada pela função objetivo da iteração em questão e, ao fim dela, será avaliada pelo critério de aspiração; enquanto isso, *sNeighbour* armazenará a solução vizinha

[10] Essa medida é necessária porque o Python atribui objetos a variáveis por referência, o que faria com que toda alteração realizada em *s* também afetasse *sStar*. Portanto, tornou-se imprescindível criar uma cópia de *s*.

[11] Lembre-se de que, inicialmente, o atributo *objValue* é inicializado com infinito.

gerada pelos movimentos de instalação e remoção[12]. Em relação à variável *pIter*, declarada na linha 20, ela servirá como variável de iteração para percorrer as listas auxiliares *openFacil* e *closeFacil*.

Código-fonte 15 – Loop principal e primeiro fluxo de execução da busca tabu

```
18      while (i - bestIter < maxIter):
19          sTabu:Sol = Sol()
20          sNeighbour:Sol = Sol()
21          pIter:int = 0
22          if (s.isFeasible()):
23              if (s.openFacil):
24                  sNeighbour,sTabu,config = ta.close(s,sTabu,
                        coverageMatrix,tabu,i,pIter)
25                  pIter +=1
26              while (pIter < len(s.openFacil) and
27              (not sNeighbour.isFeasible() or
28              tabu[config] >= i)):
29                  sNeighbour,sTabu,config = ta.close(s,sTabu,
                        coverageMatrix,tabu,i,pIter)
30                  pIter +=1
31              if (not sNeighbour.isFeasible()) or (tabu[
                    config] >= i):
32                  pIter = 0
33                  if (s.closeFacil):
34                      sNeighbour,sTabu,config = ta.install(s,
                            sTabu,coverageMatrix,tabu,i,pIter)
35                      pIter += 1
36                  while (pIter < len(s.closeFacil) and (tabu[
                        config] >= i)):
37                      sNeighbour,sTabu,config = ta.install(s,
                            sTabu,coverageMatrix,tabu,i,pIter)
```

[12] Não é possível alterar a solução atual *s* diretamente, pois é necessário preservar a sua configuração para o caso de o vizinho gerado ser infactível ou tabu.

```
38                          pIter +=1
```

É possível dizer que a busca-tabu proposta por Mourão *et al.* (2024) possui dois fluxos de execução principais, os quais estão condicionados à factibilidade da solução atual. Desse modo, o segmento de código desta seção apresenta, principalmente, o fluxo em que a solução atual é factível, como fica evidenciado no escopo da condicional da linha 22. Já a seção seguinte apresentará o fluxo em que s é infactível.

Tendo isso em mente, na linha 22, a factibilidade da solução atual é verificada. Caso ela seja factível, *sNeighbour* é gerado pela remoção[13] da p-ésima facilidade de *s.closeFacil*[14]. Essa medida é tomada porque, uma vez que o objetivo é minimizar o custo total associado à instalação das facilidades selecionadas e já se sabe que a solução atual atende à demanda mínima estipulada, a facilidade mais cara é removida[15].

Após a remoção da p-ésima facilidade em *openFacil*, *pIter* é incrementada e, como existe a possibilidade de o vizinho gerado por meio desse movimento de remoção ser infactível ou tabu, o loop da linha 26 executa movimentos de remoção nas facilidades abertas subsequentes à $openFacil_{pIter}$. Esse processo se repete até que algum vizinho factível e não tabu seja gerado.

No entanto, mesmo assim, existe a possibilidade de que, ao fim desse loop, nenhum vizinho factível e não tabu tenha sido gerado. Para tratar dessa possibilidade, o escopo da condicional da linha 31 é responsável por gerar um vizinho por meio de um *movimento de instalação* e, é claro, tomando como base a configuração de s. Tal como nos casos descritos anteriormente, é possível que o vizinho gerado seja tabu. Nesse caso, o *loop* da linha 36 é responsável por executar movimentos de instalação, tendo como base as facilidades descritas em *closeFacil*, de modo semelhante ao ocorrido no loop da linha 26.

Desse modo, no pior dos casos, todos os loops que visavam à obtenção de uma solução factível e não tabu serão executados, de modo que o loop da linha 36 percorrerá todas as facilidades contidas em *closeFacil* sem que uma solução não tabu seja encontrada. Nesta possibilidade, caso *sTabu* não atenda ao critério de aspiração, a busca estagna e *sStar* é retornada

[13] A abreviatura *ta* foi usada para fazer referência a *tabuSearchAux*.

[14] Perceba que antes que o movimento de remoção ocorra, a nulidade da lista auxiliar de *openFacil* é verificada. Essa condicional não está presente no algoritmo base, porém a sua necessidade foi constatada pela possibilidade da lista em questão estar vazia, o que culminaria em um erro quando o movimento de remoção fosse performado, já que não haveriam mais facilidades abertas para serem fechadas. Verificações desse tipo acontecem também nas linhas 33 e 39, nenhuma das quais estão presentes no algoritmo base. Além disso, por se tratarem de condicionais muito simples, elas não são citadas diretamente durante a explicação do código.

[15] Lembre-se de que as facilidades de *openFacil* estão ordenadas de forma decrescente em ordem de custo de abertura. Portanto, ao remover a p-ésima facilidade para $p = 1$, a facilidade mais cara é removida.

antes que o condicional da linha 18 se torne falso.[16].

5.11.3 Segundo fluxo de execução

O segundo fluxo de execução diz respeito à possibilidade de a solução atual ser infactível. Nesse caso, apenas movimentos de instalação são feitos, como é possível observar no trecho de código a seguir.

Código-fonte 16 – Segundo fluxo condicional da busca tabu.

```
39          else:
40              if (s.closeFacil):
41                  sNeighbour,sTabu,config = ta.install(s,
                        sTabu,coverageMatrix,tabu,i,pIter)
42                  pIter +=1
43
44              while (pIter < len(s.closeFacil) and
45              (not sNeighbour.isFeasible() or
46              tabu[config] >= i)):
47                  s_aux = cp.deepcopy(s)
48                  s_aux.move(pIter,coverageMatrix,is2install=
                        True)
49                  iFac = s.closeFacil[pIter][0]
50                  cost = s.facilities[iFac].cost
51                  s_aux.evaluate(cost,opened=True)
52
53                  if (s_aux < sNeighbour):
54                      config = iFac
55                      sNeighbour = cp.deepcopy(s_aux)
56                      if tabu[config] >= i and sNeighbour <
                            sTabu:
57                          sTabu = cp.deepcopy(sNeighbour)
```

[16] É importante notar que o pseudocódigo apresentado por Mourão *et al.* (2024) não possui mecanismos para contornar essa possibilidade, então o mecanismo de parada descrito se trata de uma particularidade desta implementação.

```
58                  pIter += 1
```

Na linha 41, um movimento de instalação é feito e, como de costume, um loop o acompanha logo em seguida. Nesse caso, o laço da linha *44* tem uma funcionalidade dupla: garantir que uma solução factível e não tabu seja encontrada e, mais do que isso, fazer com que ela, necessariamente, seja melhor avaliada do que a solução encontrada na linha 41, o que incentiva a busca a progredir em direção a soluções mais promissoras.

Tendo isso mente, dentro do escopo desse loop, uma nova solução *s_aux* é gerada por meio da instalação das facilidades subsequentes a $closeFacil_p$. Das linhas 49 até 51, a solução é avaliada pela função objetivo e o condicional da linha 53 é responsável por verificar se *s_aux* é melhor avaliado que *sNeighbour*. Em caso positivo, o segundo é atualizado. Logo em seguida, *sTabu* é atualizada caso a condicional da linha 56 seja verdadeira, tal como já veio ocorrendo anteriormente.

5.11.4 Atualização de s e desfecho do algoritmo

Logo após a definição desses dois fluxos de execução, ainda dentro do escopo do condicional da linha 18, o critério de aspiração é verificado. O seu resultado determinará se a variável *s*, que guarda a configuração da solução atual, será atualizada para receber *sNeighbour* ou *sTabu*.

Código-fonte 17 – Critério de aspiração e desfecho da meta-heurística

```
59          if (sTabu < sStar):
60              s = cp.deepcopy(sTabu)
61              sStar = cp.deepcopy(sTabu)
62              bestIter = i
63          else:
64              if (tabu[config] >= i):
65                  return (sStar,False)
66              tabu[config] = i + dur
67              s = cp.deepcopy(sNeighbour)
68              if (s < sStar):
69                  sStar = cp.deepcopy(s)
```

```
70                      bestIter = i
71              i += 1
72          return (sStar,True)
```

Desse modo, como é possível observar de forma mais concreta no trecho de código acima, da linha 59 à 62 o critério de aspiração é definido. Assim, caso a melhor solução tabu encontrada durante a iteração atual seja melhor do que *sStar*, então *s* passa a armazenar uma cópia de *sTabu*, e o mesmo acontece para *sStar*. Além disso, *bestIter* é atualizado.

Note também que, na possibilidade em que o critério de aspiração é válido, a lista tabu não é atualizada. Isso se deve ao fato de a solução selecionada já ser uma solução com uma configuração tabu ativa e, nesses casos, não há uma dupla penalidade associada à sua visitação.

Já no caso do critério de aspiração não ser válido, é verificado se a solução selecionada é tabu, pois, se esse for o caso, ela não atende ao critério de aspiração, o que faria com que *sStar* fosse retornado imediatamente juntamente com o valor de *False*, para indicar a sua violação.

Caso isso não ocorra, as linhas subsequentes são responsáveis pela atualização da lista tabu de *s*, além de *sStar* e *bestIter*, na possibilidade de a solução encontrada superar *sStar*. Por último, *i* é atualizado e, assim que a condicional do *loop* da linha 18 for falsa, *sStar* é retornado, juntamente com o valor *True*, indicando que, em nenhum momento, o critério de aspiração foi violado.

6 RESULTADOS

Neste capítulo, os resultados dos testes computacionais realizados para as 78 instâncias executadas pelo algoritmo desenvolvido são comparados aos resultados obtidos por Mourão *et al.* (2024). Como já mencionado, a comparação se baseia nas métricas de qualidade das soluções e no tempo de execução. Além dessas métricas principais, a factibilidade da solução encontrada também foi verificada, assim como o valor da variável booleana associada à quebra do critério de aspiração durante a execução de uma instância.

Quanto a esses dois últimos dados, pontua-se de antemão que todas as soluções encontradas foram factíveis e que nenhuma das execuções foi interrompida devido a violações ao critério de aspiração. Desse modo, esses dois dados foram omitidos nas tabelas comparativas, pois se demonstram redundantes, já que todos os valores seriam iguais.

Ademais, em relação às tabelas apresentadas, devido à grande quantidade de dados, os resultados foram agregados, de modo que as 78 instâncias foram condensadas em 18. Assim, as variações das instâncias relacionadas aos raios de cobertura foram condensadas em uma única instância. Dessa forma, a média e o desvio padrão de cada agregação são apresentados.

O algoritmo pode ser encontrado em sua totalidade, incluindo os casos de teste utilizados, em https://github.com/autumn-Days/PCLFCPFU_TCC/.

6.1 Análise da qualidade das soluções

A Tabela 4 apresenta os *RPDs* da busca tabu desenvolvida neste trabalho, a $\mathrm{BT_D}$, em relação ao modelo de programação inteira e à busca tabu desenvolvida por Mourão *et al.* (2024), identificada por $\mathrm{BT_M}$. Desse modo, a média das porcentagens dos *RPDs* de cada instância agregada foi calculada, de forma que a última linha contém a média de todos os *RPDs* das 78 instâncias. Similarmente, o desvio padrão populacional das porcentagens dos *RPDs* também foi calculado para cada instância agregada. Assim, na última linha, o desvio padrão populacional dos *RPDs* das 78 instâncias é apresentado.

Pela Tabela 4, nas instâncias agregadas em que a D_m correspondeu a 50% de D_t, os dois algoritmos são similares e encontram o ótimo de todas as instâncias[1], com exceção do caso em que U corresponde a 5% de D_m e m é igual a 50.000. Observa-se a superioridade da $\mathrm{BT_M}$, já que as únicas instâncias em que a $\mathrm{BT_D}$ foi melhor correspondem àquelas em que $U = 0,05$,

[1] Existiram casos em que o resolvedor CPLEX não conseguiu provar a optimalidade da solução, pois sua execução foi interrompida por ter excedido o tempo máximo de execução.

Tabela 4 – Comparação da qualidade dos resultados obtidos em relação aos programas desenvolvidos por Mourão *et al.* (2024) por meio da métrica *RPD*.

Instâncias			BT_D x CPLEX		BT_M x CPLEX	
U	m	D_m	RPD (%)	Desv. Pad.	RPD (%)	Desv. Pad.
0,05	1.000	0,5	0	0	0	0
0,05	1.000	0,6	8,01	5,09	0,63	1,25
0,05	1.000	0,7	30,14	3,69	11,49	6,80
0,05	10.000	0,5	0	0	0	0
0,05	10.000	0,6	1,35	1,08	0	0
0,05	10.000	0,7	9,26	2,44	3,56	4,19
0,05	50.000	0,5	0,27	0,33	0,2	0,12
0,05	50.000	0,6	4,93	1,59	0,5	0,76
0,05	50.000	0,7	12,13	4,72	7,72	6,26
0,02	1.000	0,5	0	0	0	0
0,02	1.000	0,6	1,45	0,85	0,10	0,21
0,02	1.000	0,7	6,7	3,40	0,05	0,08
0,02	10.000	0,5	0	0	0	0
0,02	10.000	0,6	0	0	0	0
0,02	10.000	0,7	1,68	2,03	0	0
0,02	50.000	0,5	0	0	0	0
0,02	50.000	0,6	0,08	0,15	0,08	0,15
0,02	50.000	0,7	7,41	5,61	0	0
Média e Desv. Pad.			4,48	7,49	1,26	4,93

Fonte: O próprio autor.

$m = 50.000$ e $D_m = 0,5$, em que seu *RPD* foi cerca de 370% em comparação ao BT_M. Nas demais instâncias, o BT_M foi superior ou similar ao BT_D. Em média, a BT_M foi cerca de 28% melhor em comparação ao BT_D.

6.2 Análise dos tempos de execução

Ao observar a Tabela 5, o primeiro aspecto que chama a atenção é que, mesmo a BT_D tendo sido executada em um hardware com uma configuração inferior à da BT_M, em média, ela conseguiu finalizar as execuções mais rapidamente, além de apresentar maior constância quanto ao tempo gasto. Para 10 das 18 instâncias agregadas, seu desvio padrão esteve abaixo de 5%, o que não pode ser dito da BT_M, cujo desvio padrão esteve acima de 20% em 11 das 18 instâncias. Nesse sentido, as instâncias agregadas indicam que a BT_D foi mais rápida do que sua contraparte em 12 das 18 instâncias agregadas.

Porém, a agregação dos dados também pode dificultar uma análise mais detalhada. Por isso, ao examinar a tabela tabelado apêndice Apêndice, é possível constatar que, em 34 das 78 instâncias testadas, a BT_D foi mais lenta do que a BT_M. Dessas, $73,53\%$ estão concentradas

Tabela 5 – Comparação entre os tempos de execução da BT_D e da BT_M. O tempo está descrito em segundos.

Instâncias			Tempo BT_D		Tempo BT_M	
U	m	D_m	Média	Desv. Pad.	Média	Desv. Pad.
0,05	1.000	0,5	2,93	0,01	6,38	2,71
0,05	1.000	0,6	5,08	1,55	32,55	30,74
0,05	1.000	0,7	4,67	0,38	89,72	21,57
0,05	10.000	0,5	19,48	0,06	1,4	0,01
0,05	10.000	0,6	26,12	7,09	45,3	22,91
0,05	10.000	0,7	43,75	17,34	1.019,46	522,7
0,05	50.000	0,5	144,24	30,2	95,67	50,56
0,05	50.000	0,6	233,97	92,84	1.507,93	1.103,24
0,05	50.000	0,7	307,76	87,32	5.578,61	1.169,86
0,02	1.000	0,5	5,29	0,02	0,11	0,01
0,02	1.000	0,6	9,92	1,77	15,02	9,5
0,02	1.000	0,7	17,68	2,83	337,34	283,6
0,02	10.000	0,5	35,56	0,12	1,6	0,01
0,02	10.000	0,6	35,9	0,08	2,6	1,87
0,02	10.000	0,7	60,43	17,66	65,87	42,47
0,02	50.000	0,5	203,96	0,28	8,92	0,05
0,02	50.000	0,6	258,97	104,26	6.211,45	12.263
0,02	50.000	0,7	678,43	130,65	1.705,89	1.223,67
Média e Desv. Pad.			114,7	171,56	957,92	3659,27

Fonte: O próprio autor.

nas instâncias com $U = 0.02$, o que *talvez* indique que esta implementação em específico tenha dificuldades para prosseguir com o seu fluxo de execução quando a diminuição da capacidade das facilidades não é acompanhada de uma diminuição das demandas. Porém, em virtude da diferença entre os hardwares usados, não é possível afirmar categoricamente o motivo desse comportamento.

Outrossim, ao examinar os dados completos, é interessante notar que, durante a execução da BT_D, apenas 3 das instâncias executadas excederam 10 minutos de execução, sem chegar a exceder 15 minutos. Enquanto isso, sua contraparte teve 8 instâncias que alcançaram mais de 30 minutos de execução, com a mais longa chegando a cerca de 1 hora e 47 minutos.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como principal objetivo avaliar o desempenho da busca tabu proposta por Mourão *et al.* (2024), aplicada ao PCLFCPFU, sem o auxílio de métodos determinísticos, de modo a testar a eficácia do funcionamento central do algoritmo. Os resultados mostraram que, mesmo sem o uso do resolvedor CPLEX, a busca tabu foi capaz de alcançar bons resultados.

No entanto, para garantir que as soluções permaneçam próximas das soluções ótimas, com desvios relativos (*RPD*) de até 5% em comparação com o CPLEX, por exemplo, ainda é necessário que métodos auxiliares sejam utilizados, como resolvedores de programação inteira, uma vez que foi observado que a qualidade das soluções tende a se deteriorar conforme a demanda mínima aumenta ou a capacidade das facilidades aumenta.

Pela análise realizada, é possível deduzir que essa piora na qualidade decorre da adoção de uma estratégia de alocação que não busca otimizar o espaço das facilidades, limitando-se ao cumprimento das restrições do PCLFCPFU. Essa inferência fundamenta-se no fato de que, quanto mais otimizadas forem as alocações, menor é o número de facilidades que precisam ser instaladas para que a demanda mínima seja atendida.

Desse modo, uma vez que a implementação apresentada neste trabalho não é otimizada nesse aspecto, é correto inferir que, provavelmente, mais facilidades precisaram ser instaladas para que uma solução factível fosse encontrada. Portanto, à medida que as capacidades aumentam, assim como a demanda mínima, infere-se que a necessidade de otimizar o espaço das facilidades ocorra com maior frequência do que nos casos em que os valores desses parâmetros são menores.

Portanto, ao analisar os dados sob essa ótica, essa hipótese, caso seja demonstrada verdadeira, é capaz de explicar por que a versão da busca tabu auxiliada pelo *solver* CPLEX obteve soluções superiores. Isso ocorreria porque, mesmo com o aumento dos valores desses parâmetros, o resolvedor consegue otimizar o uso do espaço das facilidades instaladas.

Contudo, mais estudos são necessários para que uma relação de causalidade seja comprovada a partir dessa correlação, uma vez que o foco deste trabalho não foi identificar os fatores que levassem às disparidades na qualidade das soluções, mas sim comparar os resultados segundo as métricas adotadas. Quanto ao tempo de execução, foi constatado que a busca tabu apresentada neste trabalho, mesmo tendo sido executada em um hardware inferior, obteve resultados significativamente mais rápidos. No pior caso, a busca tabu desenvolvida levou cerca

de 14 minutos para completar a execução, enquanto a versão com o resolvedor CPLEX demorou até 1 hora e 47 minutos. Dessa forma, em virtude da semelhança entre as implementações, acredita-se que esse ganho de tempo é advindo, principalmente, da não utilização do *solver* para a resolução do subproblema da alocação.

Além disso, como dito na Seção 5.2, Mourão *et al.* (2024) vincula "$n+1$" e "-1" considerando os *status* de abertura das facilidades, enquanto na implementação feita neste trabalho, esta atribuição não considera este aspecto. Nesse sentido, é possível que as verificações quanto aos status das facilidades para a atribuição dos valores à lista de alocações tenham causado algum atraso computacional para a sua versão. Contudo, em virtude da complexidade da resolução do subproblema de alocação, ainda atribui-se a discrepância entre os tempos computacionais à utilização do resolvedor CPLEX. Entretanto, é importante ressaltar que as diferenças de hardware influenciam esses tempos e limitam a comparação direta.

No tocante aos outros objetivos, este trabalho também propôs a disponibilização dos artefatos de software relativos ao desenvolvimento do projeto, incluindo a implementação completa em um repositório público no GitHub, com as instâncias utilizadas, os resultados obtidos e a explicação detalhada da lógica da meta-heurística descrita nesta monografia.

Em síntese, este trabalho contribuiu para a avaliação e o aprimoramento da busca tabu aplicada ao PCLFCPFU, ao fornecer mais dados sobre o seu desempenho, bem como a análise desses dados. Futuras investigações podem explorar a implementação de outras estruturas de dados, além das descritas neste trabalho, com o objetivo de reduzir o tempo de execução em cenários complexos. Outrossim, para que a qualidade das soluções seja mantida à medida que a demanda mínima e as capacidades aumentam, recomenda-se a adoção de heurísticas gulosas para a alocação das demandas como alternativa ao uso de resolvedores de programação inteira.

REFERÊNCIAS

CARDOSO, L. C.; MOURÃO, F. P.; SÁ, E. M. d.; SOUZA, S. R. d. Metaheurística iterated local search aplicada ao problema de localização com cobertura parcial. In: ENCONTRO NACIONAL DE INTELIGÊNCIA ARTIFICIAL E COMPUTACIONAL, 19., Porto Alegre. **Anais [...]**. SBC, 2022. Disponível em: https://doi.org/10.5753/eniac.2022.227629. Acesso em: 25 jul. 2025.

CARDOSO, L. C.; SÁ, E. M. d.; SOUZA, S. R. d. Meta-heurística simulated annealing aplicada ao problema de localização com cobertura parcial. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 55., São José dos Campos. **Anais [...]**. SBPO, 2023. Disponível em: https://doi.org/10.59254/sbpo-2023-174885. Acesso em: 5 jul. 2025.

COPPIN, B. **Artificial Intelligence Illuminated**. [*S. l.*]: Jones and Bartlett Publishers, 2004. (Jones and Bartlett illuminated series). ISBN 9780763732301.

CORDEAU, J.-F.; FURINI, F.; LJUBIĆ, I. Benders decomposition for very large scale partial set covering and maximal covering location problems. **European Journal of Operational Research**, v. 275, n. 3, 2019. Disponível em: https://hal.science/hal-02152294. Acesso em: 25 jan. 2026.

COSTA, J. S. d.; AUGUSTO, G. S. d. J.; BAHIENSE, L.; GONZÁLEZ, P. H. Algoritmo clustering search aplicado ao problema de localização com cobertura parcial. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 56., Fortaleza. **Anais [...]**. SPBO, 2024. Disponível em: https://doi.org/10.59254/sbpo-2024-193368. Acesso em: 25 jan. 2026.

DOOREN, C. van. A review of the use of linear programming to optimize diets, nutritiously, economically and environmentally. **Frontiers in Nutrition**, v. 5, p. 48, jun. 2018. Disponível em: https://doi.org/10.3389/fnut.2018.00048. Acesso em: 25 jan. 2026.

GLOVER, F. Tabu search: wellsprings and challenges. **European Journal of Operational Research**, v. 106, n. 2, p. 221–225, 1998. ISSN 0377-2217. Disponível em: https://doi.org/10.1016/S0377-2217(97)00259-2. Acesso em: 25 jan. 2026.

GLOVER, F.; KOCHENBERGER, G. A. (Ed.). **Handbook of Metaheuristics**. 1. ed. New York, NY: Springer, 2003. v. 57. XII–557 p. (International Series in Operations Research & Management Science, v. 57). ISBN 978-0-306-48056-0. Disponível em: https://doi.org/10.1007/b101874. Acesso em: 25 jan. 2026.

GOLDBARG, E.; GOLDBARG, M.; LUNA, H. **Otimização Combinatória e Metaheurísticas**: Algoritmos e apliacações. [*S. l.*]: Elsevier Brasil, 2017. ISBN 9788535278132.

HILLIER, F.; LIEBERMAN, G. **Introdução à Pesquisa Operacional**. [*S. l.*]: AMGH, 2013. ISBN 9788580551198.

LACHTERMACHER, G. **Pesquisa operacional na Tomada de Decisões**: modelagem em excel. 5. ed. Rio de Janeiro, RJ: LTC, 2016. ISBN 9788521630487.

MOREIRA, D. A. **Pesquisa Operacional**: Curso introdutório. 1. ed. São Paulo: Cengage Learning Brasil, 2018. Ebook. ISBN 9788522128068.

MOURÃO, F. P.; SÁ, E. M. d.; SOUZA, S. R. d.; SOUZA, M. J. F. Um algoritmo busca tabu para o problema capacitado de localização de facilidades com cobertura parcial e fonte única. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 56., Fortaleza. **Anais [...]**. SBPO, 2024. Disponível em: https://www.doi.org/10.59254/sbpo-2024-193587. Acesso em: 25 jan. 2026.

MURTHY, P. **Operations Research**. [*S. l.*]: New Age International, 2007. ISBN 9788122420692.

OSMAN, I.; KELLY, J. **Meta-Heuristics**: Theory and applications. Springer US, 1996. ISBN 9780792397007. Disponível em: https://www.doi.org/10.1007/978-1-4613-1361-8. Acesso em: 25 jan. 2026.

PANNEERSELVAM, R. **Operations Research**. New Delhi: Prentice-Hall of India, 2006. 624 p. ISBN 978-81-203-2928-7.

PAPADIMITRIOU, C.; STEIGLITZ, K. **Combinatorial Optimization**: Algorithms and complexity. [*S. l.*]: Dover Publications, 1998. (Dover Books on Computer Science). ISBN 9780486402581.

POLLOCK, S. M.; MALTZ, M. D. Chapter 1 operations research in the public sector: an introduction and a brief history. In: **Operations Research and The Public Sector**. Elsevier, 1994, (Handbooks in Operations Research and Management Science, v. 6). p. 1–22. Disponível em: https://www.sciencedirect.com/science/article/pii/S0927050705800829. Acesso em: 25 jan. 2026.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence**: a modern approach. Third. England: Pearson Education, 2016. ISBN 9781292153964.

TAHA, H. A. **Operations Research**: An introduction. USA: Prentice-Hall, Inc., 2007. ISBN 0131889230.

APÊNDICE A – PSEUDOCÓDIGO DA BUSCA TABU APLICADA AO PCLFCPFU

Neste apêndice, apresenta-se uma versão adaptada do pseudocódigo desenvolvido por Mourão *et al.* (2024). A principal adaptação consiste na criação de duas funções auxiliares, denominadas *aplicar_remoção* e *aplicar_instalação*. Ambas são responsáveis por obter um vizinho s' a partir da solução atual s, atualizar as alocações em x conforme a nova configuração do vetor y e verificar se a solução gerada corresponde, ou não, a uma solução tabu s^t.

A única diferença entre as duas funções é que a primeira obtém s' pela remoção de um elemento de $v1$, enquanto a segunda o faz pela remoção de um elemento de $v0$. Isso implica que uma fecha uma facilidade, enquanto a outra abre uma. Devido a essa semelhança, apenas *aplicar_remoção* é apresentada, já que a implementação de *aplicar_instalação* se resume à substituição das ocorrências de $v1$ por $v0$. Desse modo, essa é a única diferença relevante entre o pseudocódigo original e o apresentado.

Outrossim, é necessário considerar que todas as variáveis declaradas ficam disponíveis no escopo global, inclusive aquelas declaradas nas funções auxiliares. Essa característica também pode ser observada no pseudocódigo original.

Algoritmo 2: Busca Tabu aplicada ao PCLFCPFU

Entrada: $x, y, f, A, c, d, D_{min}, M, dur, iter_{max}$
Saída: $s^*, f(s^*)$

$s \leftarrow (y,x) \leftarrow$ solução_gulosa(y,x,A,D_{min});
$s^* \leftarrow s$; $iter \leftarrow 0$;
$iter_{melhor} \leftarrow 0$; $T \leftarrow [\ ]$;
enquanto $(iter - iter_{melhor} \leq iter_{max})$ **faça**
  $iter \leftarrow iter + 1$;
  $f(s^t) \leftarrow \infty$;
  $v_1 \leftarrow$ vetor dos índices das facilidades abertas em ordem decrescente de custo;
  $v_0 \leftarrow$ vetor dos índices das facilidades fechadas em ordem crescente de custo;
  se *(s é factível)* **então**
    $p \leftarrow 1$;
    $s' \leftarrow$ aplicar_remoção();
    enquanto $((s'$ *infactível*$)$ ***OU*** $(s'$ *factível* ***E*** $T_{v1_p} \geq iter))$ ***E*** $(p \leq |v_1|)$ **faça**
      $p \leftarrow p + 1$;
      $s' \leftarrow$ aplicar_remoção();
    se $(s'$ *infactível*$)$ ***OU*** $(s'$ *factível* ***E*** $T_{v1_p} \geq iter)$ **então**
      $p \leftarrow 1$;
      $s' \leftarrow$ aplicar_instalação();
    enquanto $((s'$ *infactível*$)$ ***OU*** $(s'$ *factível* ***E*** $T_{v0_p} \geq iter))$ ***E*** $(p \leq |v_0|)$ **faça**
      $p \leftarrow p + 1$;
      $s' \leftarrow$ aplicar_instalação();
  senão
    $p \leftarrow 1$;
    $s' \leftarrow$ aplicar_instalação();
    enquanto $((s'$ *infactível*$)$ ***OU*** $(s'$ *factível* ***E*** $T_{v0_p} \geq iter))$ ***E*** $(p \leq |v_0|)$ **faça**
      $p \leftarrow p + 1$;
      $s_{aux} \leftarrow s \oplus M(v_{0_p})$;
      $x \leftarrow$ demanda atendida;
      se $f(s_{aux}) < f(s')$ **então**
        $s' \leftarrow s_{aux}$;
        se $(T_{v_{0_p}} \geq iter)$ ***E*** $(f(s') < f(s^t))$ **então**
          $s^t \leftarrow s'$;
  Atualize T;
  $s \leftarrow s'$ ou s^t de acordo com o critério de aspiração;
  se $(f(s) < f(s^*))$ **então**
    $s^* \leftarrow s$;
    $iter_{melhor} \leftarrow iter$;
retorna $(s^*, f(s^*))$;

Algoritmo 3: aplicar_remoção

Saída: Uma solução vizinha s'

1 $s' \leftarrow s \oplus M(v1_p)$;

2 $x \leftarrow$ Atualize a demanda;

3 **se** $(T_{v1p} \geq iter \; \boldsymbol{E} \; f(s') < f(s^t))$ **então**

4 $\quad s^t \leftarrow s'$;

5 **retorna** s';

APÊNDICE B – RESULTADOS INTEGRAIS

Neste apêndice, os principais resultados relacionados aos dados obtidos pela execução das 78 instâncias de teste são apresentados por meio da Tabela 6. Desse modo, além dos parâmetros já conhecidos, também são apresentadas a qualidade das soluções e os tempos computacionais decorrentes da execução da busca tabu desenvolvida neste projeto; a qualidade das soluções obtidas por Mourão *et al.* (2024) na execução de sua versão da busca tabu e do seu modelo de programação inteira executado pelo resolvedor CPLEX; e, por último, os status retornados pelo CPLEX. Em relação ao último, o valor 1 indica que o status *Optimal* foi retornado, o que significa que um ótimo global foi encontrado. Já o valor 0 indica o retorno do status *AbortTimeLim*, que sinaliza que o tempo limite de execução foi excedido e, portanto, não há garantia de que a solução encontrada seja ótima.

Tabela 6 – As colunas subsequentes à coluna *r* indicam, respectivamente, o tempo de execução em segundos da busca tabu desenvolvida neste trabalho, a qualidade da sua solução, a qualidade das soluções da busca tabu e do resolvedor CPLEX desenvolvidos por Mourão *et al.* (2024) e, por último, os status retornados pelo CPLEX.

U	m	D_m	r	T. BT_D	Q. BT_D	Q. BT_M	Q. CPLEX	S.
0,05	1000	0,5	5,5	2,945424	380	380	380	1
0,05	1000	0,5	5,75	2,938539	380	380	380	1
0,05	1000	0,5	6,0	2,920961	380	380	380	1
0,05	1000	0,5	6,25	2,925302	380	380	380	1
0,05	1000	0,6	4,0	4,184187	451	391	391	1
0,05	1000	0,6	4,25	7,905886	408	395	383	1
0,05	1000	0,6	4,5	3,438065	427	380	380	1
0,05	1000	0,6	4,75	4,468797	387	380	380	1
0,05	1000	0,6	5,0	5,396165	395	380	380	1
0,05	1000	0,7	3,25	5,21909	787	666	585	1
0,05	1000	0,7	3,5	4,628275	649	573	509	1
0,05	1000	0,7	3,75	4,707103	614	550	462	1
0,05	1000	0,7	4,25	4,144454	515	412	410	1
0,05	10000	0,5	5,5	19,384635	338	338	338	1
0,05	10000	0,5	5,75	19,499742	338	338	338	1

U	m	D_m	r	T. BT_D	Q. BT_D	Q. BT_M	Q. CPLEX	S.
0,05	10000	0,5	6,0	19,524405	338	338	338	0
0,05	10000	0,5	6,25	19,511299	338	338	338	1
0,05	10000	0,6	4,0	28,437281	354	343	343	1
0,05	10000	0,6	4,25	39,076724	343	338	338	1
0,05	10000	0,6	4,5	21,563897	343	338	338	1
0,05	10000	0,6	4,75	21,075879	340	338	338	1
0,05	10000	0,6	5,0	20,441415	338	338	338	1
0,05	10000	0,7	3,25	73,68938	601	590	533	1
0,05	10000	0,7	3,5	36,096932	479	451	442	1
0,05	10000	0,7	3,75	32,764751	434	401	395	1
0,05	10000	0,7	4,25	32,450715	369	348	348	1
0,05	50000	0,5	5,5	116,441355	379	377	376	1
0,05	50000	0,5	5,75	173,28528	377	377	376	1
0,05	50000	0,5	6,0	175,481416	376	377	376	0
0,05	50000	0,5	6,25	111,760716	376	376	376	0
0,05	50000	0,6	4,0	346,111211	434	410	403	1
0,05	50000	0,6	4,25	171,166811	413	402	398	0
0,05	50000	0,6	4,5	348,393864	404	386	386	0
0,05	50000	0,6	4,75	146,017196	407	386	386	0
0,05	50000	0,6	5,0	158,175152	399	386	387	0
0,05	50000	0,7	3,25	207,851328	733	712	611	1
0,05	50000	0,7	3,5	390,362755	566	560	508	0
0,05	50000	0,7	3,75	234,121753	507	482	463	0
0,05	50000	0,7	4,25	398,69513	452	420	420	0
0,02	1000	0,5	5,5	5,268619	1531	1531	1531	1
0,02	1000	0,5	5,75	5,277431	1531	1531	1531	1
0,02	1000	0,5	6,0	5,299307	1531	1531	1531	1
0,02	1000	0,5	6,25	5,31154	1531	1531	1531	1
0,02	1000	0,6	4,0	9,452887	1570	1543	1535	1
0,02	1000	0,6	4,25	11,262693	1564	1535	1535	1
0,02	1000	0,6	4,5	11,898237	1565	1531	1531	1

U	m	D_m	r	T. BT_D	Q. BT_D	Q. BT_M	Q. CPLEX	S.
0,02	1000	0,6	4,75	6,805619	1540	1531	1531	1
0,02	1000	0,6	5,0	10,191218	1535	1531	1531	1
0,02	1000	0,7	3,25	17,160636	1812	1614	1614	1
0,02	1000	0,7	3,5	21,173652	1695	1601	1598	1
0,02	1000	0,7	3,75	18,937954	1662	1577	1577	1
0,02	1000	0,7	4,25	13,432262	1607	1559	1559	1
0,02	10000	0,5	5,5	35,403132	1521	1521	1521	1
0,02	10000	0,5	5,75	35,632026	1521	1521	1521	1
0,02	10000	0,5	6,0	35,49652	1521	1521	1521	1
0,02	10000	0,5	6,25	35,721906	1521	1521	1521	1
0,02	10000	0,6	4,0	35,943683	1521	1521	1521	1
0,02	10000	0,6	4,25	35,847682	1521	1521	1521	1
0,02	10000	0,6	4,5	35,776037	1521	1521	1521	1
0,02	10000	0,6	4,75	36,015678	1521	1521	1521	1
0,02	10000	0,6	5,0	35,923393	1521	1521	1521	1
0,02	10000	0,7	3,25	84,757525	1599	1521	1521	1
0,02	10000	0,7	3,5	68,154367	1538	1521	1521	1
0,02	10000	0,7	3,75	50,849796	1528	1521	1521	1
0,02	10000	0,7	4,25	37,97092	1521	1521	1521	1
0,02	50000	0,5	5,5	204,404584	1588	1588	1588	1
0,02	50000	0,5	5,75	203,615581	1588	1588	1588	1
0,02	50000	0,5	6,0	203,925779	1588	1588	1588	0
0,02	50000	0,5	6,25	203,888848	1588	1588	1588	0
0,02	50000	0,6	4,0	467,471401	1594	1594	1588	0
0,02	50000	0,6	4,25	208,664911	1588	1588	1588	0
0,02	50000	0,6	4,5	206,268413	1588	1588	1588	0
0,02	50000	0,6	4,75	206,198632	1588	1588	1588	0
0,02	50000	0,6	5,0	206,234681	1588	1588	1588	0
0,02	50000	0,7	3,25	682,14517	1868	1615	1615	1
0,02	50000	0,7	3,5	851,181108	1750	1599	1599	1
0,02	50000	0,7	3,75	697,106241	1639	1594	1594	1

U	m	D_m	r	T. BT_D	Q. BT_D	Q. BT_M	Q. CPLEX	S.
0,02	50000	0,7	4,25	483,306242	1615	1588	1588	1

Fonte: O próprio autor.