UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

WALL BERG MIRANDA DOS SANTOS MORAIS

O USO DO PARALELISMO NO MÉTODO COLÔNIA DE FORMIGAS PARA RESOLUÇÃO DO PROBLEMA DA COBERTURA MÍNIMA DE VÉRTICES EM GRAFOS MASSIVOS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO - PR

07 de dezembro de 2018

WALL BERG MIRANDA DOS SANTOS MORAIS

O USO DO PARALELISMO NO MÉTODO COLÔNIA DE FORMIGAS PARA RESOLUÇÃO DO PROBLEMA DA COBERTURA MÍNIMA DE VÉRTICES EM GRAFOS MASSIVOS

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Engenharia de Computação do Departamento Acadêmico de Informática - DAINF - da Universidade Tecnológica Federal do Paraná - UTFPR, Câmpus Pato Branco, como requisito parcial para obtenção do título de "Engenheiro em Computação".

Orientador: Prof. Dr. Marco Antonio de Castro Barbosa

PATO BRANCO - PR

07 de dezembro de 2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 16 horas do dia 07 de dezembro de 2018, na sala V006, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Marco Antonio de Castro Barbosa (orientador), Bruno Cesar Ribas e Dalcimar Casanova para avaliar o trabalho de conclusão de curso com o título **O uso do paralelismo no método colônia de formigas para resolução do problema da cobertura mínima de vértices em grafos massivos**, do aluno **Wall Berg Miranda dos Santos Morais**, matrícula 01585550, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____
Prof. Marco Antonio de Castro Barbosa
Orientador (UTFPR)

_____
Prof. Bruno Cesar Ribas
(UTFPR)

_____
Prof. Dalcimar Casanova
(UTFPR)

_____
Profa. Beatriz Terezinha Borsoi
Coordenador de TCC

_____
Prof. Pablo Gauterio Cavalcanti
Coordenador do Curso de
Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

Todo sistema computável pode ser computado por uma Máquina de Turing.

Church-Turing

RESUMO

MORAIS, Wall Berg M. S. O Uso do Paralelismo no Método Colônia de Formigas para Resolução do Problema da Cobertura Mínima de Vértices em Grafos Massivos. 2018. 69f. Monografia (Trabalho de Conclusão de Curso 2) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

Este trabalho propõe solucionar o Problema da Cobertura Mínima de Vértices (PCMV) aplicadas em grafos massivos utilizando a meta-heurística Colônia de Formigas (ACO). Com a escassez da aplicação desta meta-heurística para resolução do PCMV na literatura, este trabalho propõe um método baseado no ACO para resolver este problema. Após a implementação do método proposto, os experimentos computacionais mostraram que as soluções obtidas pelo o método proposto se aproxima das soluções ótimas encontradas na literatura.

Palavras-chave: Formigas, Meta-heurística, Paralelismo, OpenMP, Instância.

ABSTRACT

MORAIS, Wall Berg M. S. The use of Parallelism in the Ant Colony method to solve The Minimum Vertex Cover Problem in Massive Graphs. 2018. 69f. Monografia (Trabalho de Conclusão de Curso 2) - Curso de Engenharia de Computação, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

This work proposes to solve the Minimum Vertex Cover Problem (MVCP) applied in massive graphs using the Ant Colony meta-heuristic (ACO). With the scarcity of the application of this meta-heuristic to solve the MVCP in the literature, this work proposes a method based on the ACO to solve this problem. After the implementation of propose method, the computacional experiments showed that the solutions obtained by the proposed method approaches the optimal solutions found in the literature.

Keywords: Ants, Meta-heuristic, Parallelism, OpenMP, Instance.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O computador é, atualmente, uma das principais ferramentas da atividade humana, pois, está presente na realização das mais diversas atividades, como, por exemplo, no comércio eletrônico, na indústria, no setor financeiro, na astronomia, entre outros, visando aumentar a produtividade destas atividades. O crescimento do poder computacional vem contribuindo para que aplicações de alto desempenho possam funcionar de forma eficiente. Aplicações que antes eram impossíveis de implementar, devido à falta de poder computacional, como as aplicações que exigem alta demanda de cálculos matemáticos, atualmente possuem soluções sendo desenvolvidas(NOBRE, 2011).

Com o crescimento do poder computacional veio o aumento da necessidade de armazenamento de dados. Os desafios computacionais decorrentes do crescimento da capacidade de armazenamento de dados estão centrados no armazenamento destes dados e nas respectivas busca e recuperação destes (LYNCH, 2008).

Muitos problemas reais, tais como o problema do limite de calado nos transportes marítimos (MORAIS *et al.*, 2017), logística (MIN *et al.*, 2006), roteamento de veículos (SOLOMON, 1987), alocação de recursos (JAIN *et al.*, 1984), entre outros, podem ser modelados matematicamente utilizando-se grafos. Desta forma, em algumas situações pode-se associar a solução destes problemas à solução de problemas abstratos clássicos da Teoria de Grafos e Teoria da Computação, tais como Árvore Geradora Mínima Capacitada (AGMC), Coloração de Grafos (CG), Circuito Hamiltoniano (CH), Caixeiro Viajante (PCV), Roteamento de Veículos (RV), Cobertura de Vértices (CV), Clique Máximo (CM), Conjunto Independente de Vértices (CIV), entre outros, nos quais (GAREY; JOHNSON, 1979) mostrou que são intratáveis. As alternativas de solução para estes problemas podem ser os métodos exatos, aproximados, heurísticos (meta-heurísticos) e paralelos (HOWARD, 1966; ALVIM, 1998; CAI *et al.*, 2010; UCHIDA *et al.*, 2012).

Soluções exatas garantem que se houver solução para a instância do problema, será encontrada a melhor solução. Entretanto, elas não garantem que a solução seja obtida em tempo razoável. São exemplos de métodos exatos: *programação dinâmica* (BELLMAN, 2003), *backtraking* (ZIVIANI, 2004; DASGUPTA *et al.*, 2008), *branch-*

and-bound (LAWLER; WOOD, 1966), para citar apenas alguns. Soluções aproximadas são aquelas que geram soluções viáveis para o problema em tempo razoável, com a garantia de ter um limite na razão de aproximação da solução obtida em relação a solução ótima (TALBI, 2009; ZIVIANI, 2004; GOODRICH; TAMASSIA, 2004).

Métodos heurísticos possuem como objetivo gerar uma solução para um problema sem ter garantia de aproximação com relação à solução ótima e a solução deve ser obtida em tempo razoável (TALBI, 2009). Uma especialização das heurísticas são as meta-heurísticas. Estes métodos têm como objetivo gerar uma solução para o problema em tempo razoável, e que esta resposta seja a melhor possível (ótima local). Dentre as meta-heurísticas existentes na literatura, pode-se citar o *Simulated Annealing* (KIRKPATRICK *et al.*, 1983; GENG *et al.*, 2007), Colônia de Formigas (DORIGO; GAMBARDELLA, 1997b), *Greedy Randomized Adaptive Search Procedure* (GRASP) (FEO; RESENDE, 1995; SALEHIPOUR *et al.*, 2011), entre outras (GENDREAU; POTVIN, 2010).

As soluções paralelas possuem como objetivo diminuir o tempo de execução para estes problemas, citados anteriormente, mantendo a mesma qualidade de solução que o método paralelizado oferece (TALBI, 2009). Tais soluções são adaptações de métodos exatos e heurísticos, citados anteriormente, porém, paralelizados, conforme descrito em (KARP; ZHANG, 1988; AMINI *et al.*, 1990; ALVIM, 1998).

Dentre os problemas descritos, o Problema da Cobertura de Vértices em um grafo não direcionado é um subconjunto de vértices que contém pelo menos um vértice no subconjunto para cada arestas presente no grafo (GAREY; JOHNSON, 1979). O Problema da Cobetura Mínima de Vértice (PCMV) tem como objetivo encontrar uma cobertura de vértices, em um grafo, que seja a menor possível. Segundo (GAREY; JOHNSON, 1979), o PCMV é classificado como um problema *NP-Difícil*. O PCMV é um proeminente problema de otimização combinacional com importantes aplicações práticas, incluindo segurança de redes, roteamento de circuitos elétricos, aplicações em redes de sensores e monitoramento de falhas em pontos de rede (*link failures*) (KAVALCI *et al.*, 2014). Entretanto, as soluções existentes para estes problemas possuem soluções eficientes para grafos relativamente grandes. Porém, para os grafos ditos massivos, grafos nos quais os conjuntos de vértices e arestas ultrapassam milhões, estas soluções não apresentam desempenho satisfatório (CAI, 2015). Em função disto, aplicar o paralelismo nas meta-heurísticas descritas neste texto pode ser uma possível solução para tratar de problemas com tal magnitude.

A Otimização por Colônia de Formigas, do inglês *Ant colony optimization* (ACO), é uma meta-heurística bioinspirada utilizada como solução de problemas de

otimização combinacional (DORIGO; BLUM, 2005). A ideia do ACO, segundo Dorigo e Gambardella (1997b), é baseado no comportamento das formigas reais que exploram um caminho entre seu ninho e uma fonte de comida, no qual elas não possuem visão, mas conseguem voltar para o ninho através de uma substância química chamada de feromônio. As formigas artificiais, definidas em (DORIGO *et al.*, 1996), são utilizadas para resolver inicialmente o Problema do Caixeiro Viajante (PCV). Dorigo e Blum (2005) afirma que o algoritmo do ACO é receptivo à eficiente paralelização, na qual pode melhorar o desempenho do algoritmo encontrando soluções melhores para problemas de grande magnitude.

Portanto, como objetivos deste trabalho, deseja-se implementar uma solução paralela para o Problema da Cobertura Mínima de Vértices, utilizando a metaheurística do Colônia de Formigas em sua forma paralela.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

Implementar uma solução paralela para o problema da Cobertura Mínima de Vértices com o algoritmo Colônia de Formigas em grafos massivos.

1.1.2 OBJETIVOS ESPECÍFICOS

Para obtenção de resultados teóricos e práticos que validem a aplicação do paralelismo em um problema NP-Difícil, neste caso, o PCMV, esse trabalho tem os seguintes objetivos específicos:

- Implementar o algoritmo colônia de formigas sequencial que resolva o problema proposto para instâncias convencionais e massivas.
- Elaborar uma estratégia de paralelização do problema para o algoritmo colônia de formigas.
- Implementar o algoritmo paralelo utilizando o *OpenMP*.
- Comparar as implementações propostas anteriormente com relação a seus tempos de execução.
- Comparar a qualidade da solução das implementações propostas com as soluções apresentadas no estado da arte para o PCMV.

Esse trabalho visa responder as seguintes perguntas com os resultados obtidos:

- A aplicação de paralelismo traz ganhos significativos em termos de tempo de solução e qualidade de resposta?
- Os esforços de aplicação de técnicas paralelas são compensadores para instâncias massivas?

1.2 JUSTIFICATIVA

Na literatura, muitos problemas NP-difíceis possuem algoritmos eficientes para instâncias relativamente grandes, mas para instâncias consideradas massivas, esses algoritmos não apresentam desempenhos satisfatórios, tanto em tempo de execução como em qualidade de solução (CAI, 2015). Em função disto, o uso do paralelismo pode ser uma alternativa viável para esses tipos de problemas. Entretanto, algumas questões precisam ser investigadas e esse trabalho propõe buscar técnicas de paralelização do Colônia de Formigas, que baseado no modelo para o PCV, será adaptada para o problema proposto.

1.3 ORGANIZAÇÃO DO DOCUMENTO

O trabalho está estruturado da seguinte forma. No capítulo 2 são apresentados os conceitos sobre complexidade de algoritmos. No capítulo 3 é apresentado a formulação do problema que este trabalho visa resolver. No capítulo 4 são apresentados opções de solução do problema. No capítulo 5 são apresentadas as metodologias adotadas e os materiais necessários. No capítulo 6 são apresentadas propostas de soluções para o problema. No capítulo 7 é apresentado os resultados obtidos a partir da solução proposta e no capítulo 8 é apresentado as conclusões deste trabalho.

2 COMPLEXIDADE DE ALGORITMOS

Para (TOSCANI; VELOSO, 2001), o mais importante questionamento sobre um problema é a sua complexidade, isto é, se ele pode ser resolvido mecanicamente (por um algoritmo). Para responder esses questionamentos, surgiram vários formalismos e máquinas abstratas, tais como a Máquina de Turing. A Máquina de Turing tornou-se a mais utilizada definição de algoritmo e assim deu precisão à definição de "Computável": um problema é computável se é resolvível por uma Máquina de Turing. Esse conceito ficou conhecido como Hipótese de Turing-Church (DEUTSCH, 1985).

2.1 TIPOS DE COMPLEXIDADE

Segundo TALBI, um algoritmo precisa de dois recursos para resolver um problema: tempo e espaço. A complexidade de tempo de um algoritmo é um número de passos necessários para resolver um problema de tamanho n. Para (GAREY; JOHNSON, 1979), o tempo requerido por um algoritmo (complexidade de tempo, em outras palavras) é expresso em termos do tamanho da instância de entrada do problema, que é definido de forma empírica. Para (CORMEN *et al.*, 2001), complexidade de tempo de um algoritmo possuindo uma determinada entrada é o número de operações primitivas ou "etapas" executadas para resolução de um determinado problema. É conveniente definir a noção de etapa de forma que ela seja tão independente da máquina quanto possível. Formalmente, (ROOSTA, 2000) representa complexidade de tempo por uma função $f(n)$ no qual n é o tamanho da entrada do problema.

Complexidade de espaço se referente à quantidade de memória que o algoritmo precisa para ser executado com completude (ROOSTA, 2000). Garey e Johnson (1979) definem que a complexidade de espaço não pode ser maior que a complexidade de tempo, pois em termos de uma Máquina de Turing, ele não pode possuir mais quadrados[1] do que os passos necessários para concluir uma computação.

[1] **Na abstração da Máquina de Turing, para simular memória, a máquina faz uso de uma fita no qual a computação é feita através dela. Portanto, um quadrado da fita é um pedaço de memória. (TURING, 1937)**

2.2 ANÁLISE DE MELHOR CASO, MÉDIO CASO E PIOR CASO EM ALGORITMOS

Análise de melhor caso, é aquela na qual, para se obter a resposta do algoritmo, é realizada a menor computação possível (CORMEN *et al.*, 2001). Por exemplo, no caso de um algoritmo de ordenação, a instância de entrada que exigirá o menor esforço computacional é um vetor totalmente ordenado. No caso de um algoritmo de pesquisa, o melhor caso é quando o valor a ser pesquisado é o primeiro elemento a ser comparado.

De acordo com Diverio *et al.* (2002), o caso médio de um algoritmo é dado por uma média ponderada do desempenho de cada entrada pela probabilidade de ela ocorrer, o que pode se tornar uma análise pouco prática por causa das dificuldades de determinação dessas probabilidades. Esse tipo de análise frequentemente exige matemática sofisticada e teoria das probabilidades (GOODRICH; TAMASSIA, 2004).

Segundo Cormen *et al.* (2001), o pior caso de algoritmo é quando ele realiza o tempo de execução mais longo para qualquer entrada de tamanho n. De acordo com o que Ziviani (2004) descreve, se f é uma função de complexidade baseada na análise de pior caso, então o custo de aplicar o algoritmo nunca é maior que $f(n)$. No caso de um algoritmo de ordem, o pior caso é quando a entrada do problema está ordenado de forma inversa à desejada, para o problema de pesquisa, o esforço computacional máximo será quando o elemento a ser pesquisado é o ultimo elemento comparado ou o elemento não esta presente no conjunto pesquisado.

GOODRICH e TAMASSIA (2004) afirmam que a análise de pior caso é mais fácil e pode ser definida de forma empírica. Além disso, usar uma abordagem de pior caso pode conduzir a algoritmos melhores, pois ter a certeza de que um algoritmo tem bom desempenho no pior caso, garante que ele tem bom desempenho em todos os casos.

2.3 TEMPOS DE COMPLEXIDADE DE ALGORITMOS

Diferentes algoritmos possuem uma variedade de diferentes funções de tempo de complexidade, e dizer quais desses algoritmos são eficientes ou não são, depende da situação do problema (GAREY; JOHNSON, 1979). A distinção entre estes algoritmos é saber se sua complexidade de tempo é dita polinomial ou exponencial.

Um algoritmo de tempo polinomial, segundo Garey e Johnson (1979), é definido como aquele cujo função de complexidade de tempo é $O(p(n))$ para alguma

função polinomial p, onde n é usado para para denotar o tamanho da instância de entrada do algoritmo. Qualquer algoritmo no qual a função de complexidade de tempo não pode ser limitada por um polinômio é chamado de algoritmo de tempo exponencial.

2.4 COMPLEXIDADE E TIPOS DE PROBLEMAS

De acordo com TALBI, a complexidade de um problema é equivalente à complexidade do melhor algoritmo conhecido que resolve este problema. Um problema é dito tratável (ou factível) se existir um algoritmo de tempo polinomial que possa resolvê-lo. Um problema é dito intratável (ou difícil) se não existir nenhum algoritmo de tempo polinomial que possa resolvê-lo (GAREY; JOHNSON, 1979).

Os problemas existentes na computação, podem ser divididos em três categorias de problemas: de decisão, de pesquisa e de otimização.

Garey e Johnson (1979) define um problema de decisão como um problema que tem como saída apenas duas opções: sim ou não. Formalmente, um problema de decisão Π consiste em um conjunto D_Π de instâncias e de um subconjunto $Y_\Pi \subseteq D_\Pi$ para instâncias que retornam “sim”.

Ainda de acordo com Garey e Johnson (1979), um problema de pesquisa (ou localização) Π consiste de um conjunto finito de objetos D_Π, chamados de instâncias e, para cada instância $I \in D_\Pi$, um conjunto de finito objetos $S_\Pi[I]$ chamado de solução de I. Então é dito que um algoritmo resolve um problema de pesquisa Π se, dado algumas instâncias de entrada $I \in D_\Pi$, ele retorna “não” como resposta se $S_\Pi[I]$ é vazia, caso contrário, o algoritmo retorna alguma solução s que pertence a S_Π.

Um problema é dito de otimização se, além de receber uma solução, caso houver, como retorna um problema de pesquisa, essa solução deve ser a menor solução, no caso de minimização (ou a maior no caso de maximização), para o problema (TALBI, 2009).

Cada tipo de problema pode ser associado a uma classe de problemas, como descrito a seguir.

2.5 CLASSE *P E NP*

De acordo com (GAREY; JOHNSON, 1979; GOODRICH; TAMASSIA, 2004), a classe de complexidade *P* é o conjunto de todos os problemas de decisão (ou linguagens) L

que podem ser aceitos em tempo polinomial no pior caso. Ou seja, existe um algoritmo A que, dado $x \in L$, responde "sim" em tempo $p(n)$, onde n é o tamanho de x e $p(n)$ é um polinômio.

Uma linguagem L é definida como da classe *P*, se e somente se, existir uma Máquina de Turing Determinística (MTD) que a reconheça em tempo polinomial (KARP, 1972).

Segundo GAREY; JOHNSON e GOODRICH; TAMASSIA, a classe *NP* é definida incluindo a classe de complexidade *P* permitindo, entretanto, a inclusão de linguagens que podem não estar em *P*. Especificamente com problemas *NP*, permite-se que algoritmos realizem uma operação adicional de escolha não-determinística chamada *choose*. Então, afirma-se que um algoritmo A aceita deterministicamente uma entrada x se existir um conjunto de saídas da operação *choose* que A poderia usar para responder "sim".

Portanto, a classe de complexidade *NP* é o conjunto de todos os problemas de decisão (ou linguagens) L que podem ser aceitos não-deterministicamente em tempo polinomial, ou seja, se existe um algoritmo não-determinístico A que, dado $x \in L$, então existe um conjunto de saídas da função *choose* em A tal que A responde "sim" em tempo $p(n)$, onde n é o tamanho de x e $p(n)$ é um polinômio.

Para (KARP, 1972), uma linguagem é definida como da classe *NP*, se e somente se, existir uma Máquina de Turing Não-Determinística (MTND) que a reconheça em um tempo polinomial.

2.6 CLASSE *NP-COMPLETO* E *NP-DIFÍCIL*

Formalmente, uma linguagem L é dita *NP-Completa* se $L \in$ *NP* e, para todas outras linguagens $L' \in$ *NP*, $L' \propto L$ (GAREY; JOHNSON, 1979). De outra maneira, um problema é dito ser *NP-Completo* se ele pertencer a classe *NP* e se ele poder ser redutível para outro problema que pertence a classe *NP*. Segundo o Teorema de Cook-Levin (COOK, 1971), existe pelo menos um problema *NP-Completo* que é o Problema da Satisfazibilidade booleana (SAT), logo, os demais problemas dessa classe podem ser redutíveis a ele.

A Figura 1, mostra alguns problemas *NP-Completos* e suas relações de redutividade, segundo (CORMEN, 2013).

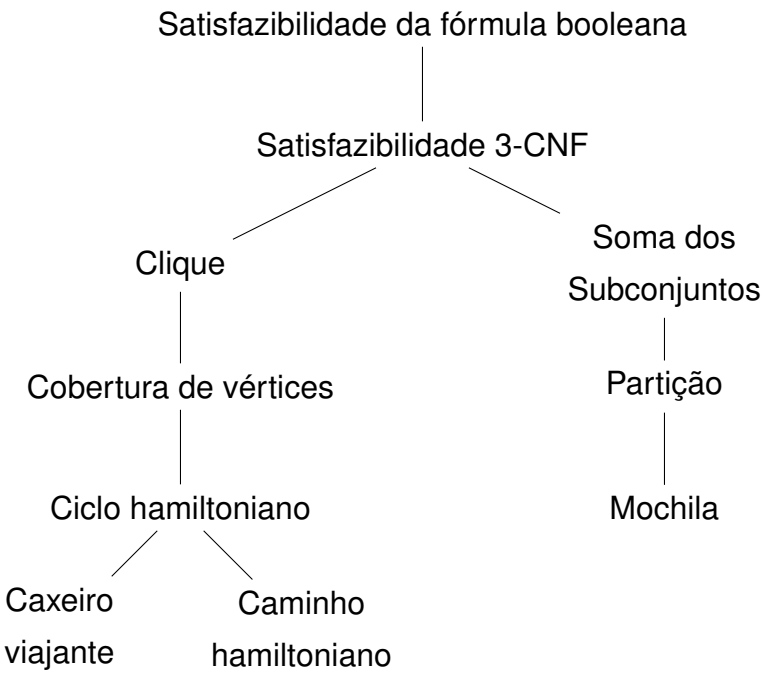

Figura 1: Problemas *NP-Completos* e sua arvore de redução
Fonte: (CORMEN *et al.*, 2001).

De acordo com (GAREY; JOHNSON, 1979), qualquer problema de decisão Π, sendo ele *NP* ou não, que pode ser transformado em um problema *NP-Completo* terá a propriedade de não ser resolvível em tempo polinomial a menos que $P = NP$. Então, pode-se afirmar que um problema Π é *NP-Difícil*, desde que é, em certo sentindo, no mínimo tão difícil quanto os *NP-Completos*. Geralmente, os problemas aos quais se aplica a definição de *NP-Difícil* são aqueles classificados como problemas de pesquisa e/ou de otimização.

3 PROBLEMA DA COBERTURA MÍNIMA DE VÉRTICES

Segundo o (CORMEN *et al.*, 2001), uma cobertura de vértices em um grafo não orientado $G = (V, E)$ é um subconjunto $V' \subseteq V$ tal que $(u, v) \in E$, então $u \in V'$ ou $v \in V'$ (ou ambos). Portanto, cada vértice deve cobrir suas arestas incidentes e uma cobertura de vértices para um grafo G é um subconjunto de vértices que cobrem todas as arestas E, conforme a Figura 2. O tamanho de uma cobertura de vértices é o número de vértices que ela contém.

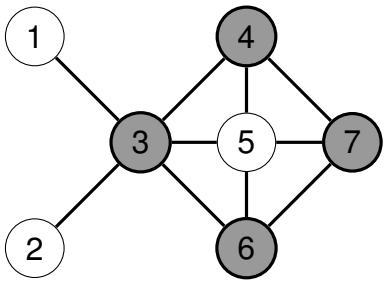

Figura 2: Exemplo de Cobertura de Vértices de tamanho 4
Fonte: (ZIVIANI, 2004).

O objetivo do PCMV é encontrar uma cobertura de vértices, em um grafo, que seja a menor cobertura possível, como ilustrado na Figura 3. Como o Problema da Cobertura de Vértices é um problema de *NP-Completo* demonstrado por Garey e Johnson (1979), o PCMV é classificado como um problema *NP-Difícil* pois segundo a definição do Garey e Johnson, um problema de otimização é no mínimo tão difícil quanto um problema *NP-Completo*. Além disto, o PCMV é semelhante a outros problemas de otimização como, por exemplo, o Conjunto Independente Máximo (*Maximum Independent Set*) e o Problema do Clique Máximo (*Maximum Clique - MC*) (GAREY; JOHNSON, 1979). Portanto, a solução deste problema é importante para auxiliar na solução de outros problemas correlatos tais como os de escalonamento e alocação de tarefas.

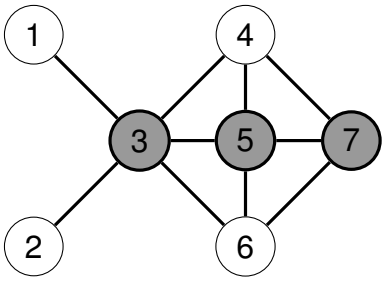

Figura 3: Exemplo de Cobertura Mínima de Vértices de tamanho 3 para o grafo anterior Fonte: (ZIVIANI, 2004).

Nas últimas décadas várias pesquisas foram realizadas com o propósito de buscar soluções para PCMV. Em (KARAKOSTAS, 2009) mostrou-se que algoritmos aproximativos para o PCMV podem alcançar um coeficiente de aproximação de $2-\Theta(\frac{1}{\sqrt{logn}})$. Em função disto, métodos suportados por heurísticas têm sido utilizados na tentativa de resolver o PCMV de maneira eficiente. Vários trabalhos têm mostrado a melhora de seus métodos aplicando heurísticas em suas abordagens e aplicações, como por exemplo, Algoritmos Evolutivos (EVANS, 1998), Colônia de Formigas (SHYU *et al.*, 2004), *Simulated Annealing* (XU; MA, 2006), para citar alguns.

4 MÉTODOS DE SOLUÇÃO PARA PROBLEMAS INTRATÁVEIS

De acordo com (TALBI, 2009), dependendo da complexidade de um problema, ele pode ser resolvido por métodos exatos ou por métodos aproximados. Métodos exatos (algoritmos exatos ou completos) sempre garantem soluções ótimas, porém, para problemas *NP-Completos*, a menos que $P = NP$, eles possuem algoritmos com tempo não polinomial. Métodos aproximados, ou heurísticos, conseguem gerar soluções com boas qualidades em tempo polinomial, porém, nada assegura encontrar a solução ótima.

4.1 MÉTODOS EXATOS

Algoritmos exatos sempre garantem a melhor solução de um dado problema, porém, para problemas *NP-Completos*, são limitados pelo o tamanho de entrada do problema por causa de sua complexidade (geralmente exponencial ou fatorial). São exemplos de métodos exatos a programação dinâmica (HOWARD, 1966),*backtraking* (ZIVIANI, 2004; DASGUPTA *et al.*, 2008) e *branch-and-bound* (LAWLER; WOOD, 1966).

De acordo com TOSCANI; VELOSO, a estratégia de programação dinâmica aborda um dado problema, dividindo-o em subproblemas mínimos, solucionando os subproblemas e guardando os resultados parciais. Os subproblemas menores e os subresultados são recompostos para resolver os problemas maiores que o geraram até obter a solução do problema. A programação dinâmica é aplicável aos problemas de otimização com a seguinte propriedade de otimalidade: “Uma solução não ótima de um subproblema não pode ser subsolução de solução ótima do problema". Isso significa que se a solução de um subproblema não é ótima, ela é descartada pelo algoritmo.

Segundo Dasgupta *et al.* (2008), *backtraking* é baseado na observação de que é frequentemente possível rejeitar uma solução examinando apenas uma parte dela. GOODRICH e TAMASSIA (2004) afirma que o algoritmo de *backtraking* percorre possíveis “caminhos de busca” para localizar soluções ou “becos sem saída”. Em outras palavras, o algoritmo de *backtraking* gera uma árvore de busca para um dado problema L , na qual, os ramos que não levam para uma possível solução são

eliminados. GOODRICH e TAMASSIA (2004) afirma este algoritmo sempre procura buscar a solução mais “fácil” para L.

Segundo Ziviani (2004), dado um problema de otimização L, o *Branch and bound*, implicitamente, enumera todas as possíveis soluções de L em uma árvore, na qual os ramos que não levam à soluções ótimas são descartados da solução, diminuindo, assim, o espaço de busca e apresentando uma solução ótima para L. GOODRICH e TAMASSIA (2004), Dasgupta *et al.* (2008) afirmam que o *branch-and-bound* é baseado no *backtraking* com as seguintes diferenças: enquanto o *backtraking* elimina os possíveis espaços de busca que não levam a uma solução válida, o *branch-and-bound* elimina as soluções que não satisfazem a cota dada como entrada do algoritmo. Como também, o *backtraking* busca apenas uma solução válida e o *branch-and-bound* procura a solução ótima.

4.2 MÉTODOS APROXIMADOS

De acordo com Ziviani (2004), para projetar algoritmos polinomiais para “resolver” problemas *NP-Completos*, deve-se “relaxar” no conceito de resolver o problema, pois, diferentemente dos métodos exatos, esses algoritmos possuem tempo polinomial cuja as soluções são sub-ótimas.

Segundo Ziviani (2004), um algoritmo aproximado é um algoritmo que gera soluções aproximadas dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado. O comportamento de algoritmos aproximados é monitorado do ponto de vista da qualidade de resultados.

Seja I uma instância de um problema Π e seja $S^*(I)$ o valor da solução ótima para I. Um algoritmo aproximado gera uma solução possível para I cujo valor $S(I)$ é maior (pior) do que o valor ótimo $S^*(I)$ (ZIVIANI, 2004).

A razão de aproximação é definida como:

$$R_a(I) = \frac{S(I)}{S^*(I)} \tag{1}$$

Na qual, para um problema Π, $S(I)$ é a solução do problema dada por um algoritmo aproximado, $S^*(I)$ é a solução ótima do problema e $R_a(I)$ a razão de aproximação. Dada a Equação 1, é dito que um problema é de minimização caso $R_a(I) > 1$ e de maximização caso $R_a(I) < 1$. Quanto mais próximo de 1 for a razão, melhor é o algoritmo aproximativo (GOODRICH; TAMASSIA, 2004; ZIVIANI, 2004).

GOODRICH e TAMASSIA (2004) apresentam um exemplo de algoritmo que possui uma razão de aproximação de 2 para o PCMV, no qual eles chamam de *2-aproximação* para *Vertex-Cover*. Este algoritmo é baseado no método guloso, que envolve escolher uma aresta do grafo, colocar seus dois extremos na cobertura e excluir do grafo essa aresta e seus vértices incidentes. O processo continua até que não hajam mais arestas.

Algoritmo 1: COBERTURA DE VÉRTICES APROXIMADO

Entrada: $G = (V, E)$
Saída: uma cobertura de vértices C para G

```
1 C ← ∅
2 enquanto G tiver arestas faça
3 |   escolha uma aresta e = (v, w) de G
4 |   coloque os vértices v e w em C
5 |   para cada aresta f incidente em v ou w faça
6 |   |   remova f de G
7 |   fim
8 fim
9 retorna C
```

Segundo GOODRICH e TAMASSIA (2004), o Algoritmo 1 possui o coeficiente de aproximação 2 porque, primeiramente, cada aresta $e = (v, w)$ escolhida pelo algoritmo e usada para adicionar v e w a C deve ser considerada em qualquer cobertura de vértices, ou seja, qualquer cobertura de vértices para G deve conter v ou w (possivelmente ambos). O algoritmo de aproximação adiciona v e w a C, e quando o algoritmo termina, não existem arestas não cobertas em G, pois a cada etapa, foram removidas as arestas cobertas pelos vértices v e w quando elas são adicionadas a C. Desta maneira, C forma uma cobertura de vértices para G. Além disso, o tamanho de C é no máximo duas vezes o tamanho de uma cobertura de vértices ótima para G, pois para cada dois vértices adicionados a C um deles deve pertencer à cobertura ótima.

4.3 MÉTODOS HEURÍSTICOS

Métodos heurísticos conseguem encontrar “boas” soluções para problemas que possuem instâncias relativamente grandes. Eles permitem obter um desempenho aceitável a custos aceitáveis em um vasto conjunto de problemas. Porém, métodos heurísticos não garantem uma certa aproximação entre a solução obtida em relação à ótima (TALBI, 2009). Como exemplo de heurística, cita-se o método guloso.

Segundo Dasgupta *et al.* (2008), algoritmos gulosos são aqueles que constroem soluções para problemas de otimização, nos quais a cada passo de escolha de

um elemento da solução é escolhido o elemento que possui melhor benefício naquele dado momento. De acordo com (TALBI, 2009), uma vez que determinado elemento é inserido na solução, ele não pode ser mudado, pois o algoritmo guloso não possui mecanismo de *backtraking*.

Algoritmo 2: Cobertura de Vértices Guloso

Entrada: $G = (V, E)$
Saída: uma cobertura de vértices C para G
1 $C \leftarrow \emptyset$
2 **enquanto** *G tiver arestas* **faça**
3 escolhe um vértice v de maior grau em G
4 coloca v em C
5 **para** *cada aresta* f *incidente em* v **faça**
6 remova f de G
7 **fim**
8 **fim**
9 **retorna** C

O Algoritmo 2 mostra uma versão gulosa para o PCMV. Tal algoritmo é dado como exemplo em (DASGUPTA *et al.*, 2008). O algoritmo tem como entrada um grafo não direcionado $G = (V, E)$, no qual V é o conjunto de vértices e E de arestas. E como saída, uma solução C na qual é uma cobertura de vértices. Como o algoritmo guloso é um método de construção, sua solução inicialmente é vazia (Linha 1) e a condição de parada é quando o grafo G não possuir mais arestas (Linha 2). Na Linha 3, é feita a escolha gulosa do vértice que será adicionado na solução e na Linha 5, é removido do grafo todas arestas incidentes ao vértice selecionado anteriormente.

Métodos heurísticos apresentam soluções nas quais não há garantia de otimalidade. Porém, tais soluções são obtidas em tempo polinomial. Métodos heurísticos geralmente são específicos para cada problema (TALBI, 2009).

4.4 META-HEURÍSTICAS

Meta-heurísticas são algoritmos de propósito geral que podem ser aplicados para resolver quase qualquer problema de otimização (TALBI, 2009). Elas podem ser vistas como metodologias de alto nível que podem ser usados em projetos de heurísticas que resolvem problemas de otimização. Diferentemente dos métodos exatos, meta-heurísticas permitem “resolver” problemas com instâncias relativamente grandes em tempo polinomial através de geração de soluções por métodos heurísticos e por processos de refinamento de solução (busca local) (GENDREAU; POTVIN, 2010).

Existem diferentes maneiras de classificar as meta-heurísticas, e dependendo da característica dessa classificação é possível dizer como elas se comportam (BLUM; ROLI, 2003; TALBI, 2009), tal classificação é dada por métodos inspirados na natureza (bio-inspirados), métodos que utilizam memória e aqueles que fazem uso de população ou partem de uma única solução.

Segundo Talbi (2009), algoritmos bio-inspirados são aqueles que fazem analogias a processos naturais, sejam eles biológicos, físicos ou evolutivos. Como exemplo, Algoritmo Genéticos (REEVES, 1995), Colônia de Formigas (DORIGO *et al.*, 1996), Colônia de Abelhas (KARABOGA; BASTURK, 2007), Enxame de Partículas (SHI; EBERHART, 1998) e *Simulated Annaling* (SA) (KIRKPATRICK *et al.*, 1983). Algoritmos que possuem memória são aqueles que guardam informações dinamicamente durante a busca de novas soluções, como exemplo, a Busca Tabu (GENDREAU *et al.*, 1994). Os métodos que não utilizam memória pode-se citar o GRASP e o SA (TALBI, 2009).

De acordo com Talbi (2009), métodos que partem de uma única solução manipulam e transformam a solução durante a busca por novas soluções. Tais métodos possuem mecanismos de busca local, como o SA. Métodos que são baseados em população, como o algoritmo genético e o colônia de formigas, geram diversas soluções nas quais buscam maior variedade no espaço de busca.

O texto que segue trata da meta-heurística Colônia de Formigas que será a meta-heurística a ser implementada neste trabalho.

4.4.1 OTIMIZAÇÃO POR COLÔNIA DE FORMIGAS

A Otimização por Colônia de Formigas, do inglês *Ant Colony Optimization* (ACO), foi proposta por Dorigo e Gambardella (1997a) como solução para o PCV. De acordo com Michalewicz (2010), o ACO é inspirado em colônias de formigas reais que depositam uma substância química, chamada feromônio, quando buscam alimento. Essa substância influencia no comportamento das formigas individuais. Quanto mais feromônio é acumulado em um caminho particular, maior é a probabilidade que uma formiga selecione esse caminho.

Segundo Dorigo e Gambardella (1997b), formigas artificiais se comportam de forma semelhante das formigas reais com as seguintes diferenças: formigas reais são cegas; uma trilha de feromônio é depositada pelas formigas quando elas estão buscando alimento; formigas reais não possuem memória. Entretanto, formigas artificiais possuem na memória todos os locais já visitados; elas podem escolher para onde ir;

as formigas artificiais depositam feromônio quando voltam da fonte de alimento para o ninho.

O algoritmo genérico para o ACO, descrito pelo Algoritmo 3, é constituído de três fases. São elas, a inicialização do algoritmo, a fase de construção de soluções e atualização do feromônio, como descrito nas Linhas 1, 4 e 5 respectivamente.

Algoritmo 3: COLÔNIA DE FORMIGAS

```
   Entrada: α, β, σ, G = (V, E)
   Saída: S /* Melhor Solução encontrada */
1  τ ← IniciaFeromônio
2  S_melhor ← ∞
3  enquanto condição de parada não satisfeita faça
4  |  S ← constroiSoluçãoFormigas
5  |  τ ← AtualizaFeromônio
6  |  se S é melhor que S_melhor então
7  |  |  S_melhor ← S
8  |  fim
9  fim
10 retorna S
```

O Algoritmo 3 possui como entrada, os parâmetros α, β, σ e a representação do problema em um grafo $G = (V, E)$. Os parâmetros α e β são utilizados na fase de construção do algoritmo e o parâmetro σ na atualização do feromônio, nas quais, cada parâmetro será abordado em suas respectivas fases de uso.

Na inicialização do algoritmo é preciso definir o valor inicial para o feromônio. Segundo Dorigo e Gambardella (1997a), o valor inicial do feromônio pode ser dado por:

$$\tau_0 \leftarrow (n \cdot L_{nn})^{-1} \tag{2}$$

No qual, L_{nn} é o valor da solução gulosa do problema, n é o tamanho da instância e τ_0 o valor inicial que será atribuído para todos os elementos da estrutura do feromônio. Para Michalewicz (2010), o valor inicial do feromônio pode ser dado por um valor baixo, mas não deixa explicito qual poderia ser esse valor.

O Algoritmo 4 descreve a fase de construção da solução de todas as formigas. No início, na Linha 1, cada formiga é colocada aleatoriamente em um ponto inicial da solução. Ao entrar no laço da Linha 2, cada formiga começa a construir sua própria solução independentemente umas das outras, com apenas a estrutura de feromônios como informação “comunitária” entre elas. Em seguida, a formiga presente no laço externo, calcula sua solução.

Inicialmente, cada formiga inicia com solução vazia e ao entrar no laço da Linha 4, a atual formiga calcula a probabilidade de escolha para cada elemento passível de ser escolhido, como descrito na Linha 5. Na Linha 6, um elemento presente na estrutura da probabilidade é escolhido para compor a solução utilizando o método da roleta, que é o método mais comumente utilizado (GAO *et al.*, 2005; LEE *et al.*, 2011; CECILIA *et al.*, 2013).

Algoritmo 4: CONSTROISOLUÇÃOFORMIGAS

```
   Entrada: α, β, τ, G = (V, A), n
   Saída: S /* Melhor Solução da iteração */
1  Colocar todas formigas em lugares aleatórios
2  para ∀k ∈ Formigas faça
3  |  S_k ← ∅
4  |  enquanto S não é solução faça
5  |  |  p_k ← CalculaProbabilidade de acordo com a Equação 3
6  |  |  s ← escolhe_s(p_k) /* Escolhe elemento através do método da
   |  |    roleta */
7  |  |  S ← S ⋃ {s}
8  |  fim
9  fim
10 retorna S
```

O laço interno é finalizado quando a formiga presente no laço externo encontra uma solução para o problema. O laço externo é finalizado quando todas as formigas terminam de construir suas respectivas soluções.

O cálculo de probabilidade é descrito na Equação 3 proposta por Dorigo e Gambardella (1997b) para resolver o PCV[1], é dada por:

$$p_k(i,j) \leftarrow \frac{[\tau_{(i,j)}]^{\alpha} \cdot [\eta_{(i,j)}]^{\beta}}{\sum_{u \in M_k} [\tau_{(i,u)}]^{\alpha} \cdot [\eta_{(i,u)}]^{\beta}} \tag{3}$$

Nas quais, $p_k(i,j)$ é a probabilidade da formiga k presente no vértice i ir para o vértice j, M_k é o conjunto de cidades não visitadas pela formiga k, $\tau_{(i,j)}$ é o feromônio para aresta i,j e $\eta_{(i,j)}$ representa a informação heurística. Para o PCV, $\eta_{(i,j)} = 1/d(i,j)$, onde $d(i,j)$ é a distância de i para j (MICHALEWICZ, 2010).

Os parâmetros α e β dizem qual informação possui maior importância ao escolher um novo elemento para a solução, se é a informação provinda da colônia (feromônio) ou da informação heurística (métrica do problema). Se $\alpha = 0$, a probabilidade de seleção será proporcional à informação heurística $\eta_{(i,u)}$, o que leva a seleção

[1] **Todas as equações presentes nessa seção foi proposta por Dorigo e Gambardella (1997b) para o caso especifico do PCV.**

comportar-se como o algoritmo guloso estocástico. Se $\beta = 0$, apenas o feromônio $\tau_{(i,u)}$ é levado em conta, logo, o algoritmo terá um comportamento aleatório (GENDREAU; POTVIN, 2010).

Diferente das formigas reais, as formigas artificiais descritas por Dorigo e Gambardella (1997b) depositam o feromônio no retorno para o ninho, na qual essa atualização é dada por:

$$\tau_{(i,j)} \leftarrow (1 - \sigma) \cdot \tau(i,j) + \Sigma_{s \in S_{atu}} \Delta\tau_s(i,j) \tag{4}$$

Onde $\tau_{(i,j)}$ é a quantidade de feromônio na aresta (i,j), $\sigma \in (0,1]$ é um parâmetro chamado de taxa de evaporação do feromônio, $\Delta\tau_k(i,j)$ é a quantidade de feromônio depositado e S_{atu} é o conjunto de soluções que será utilizado para atualizar o feromônio.

De acordo com Gendreau e Potvin (2010), a taxa de evaporação tem como objetivo evitar que o algoritmo tenha uma rápida convergência em direção à um ótimo local. Esse mecanismo é uma forma das formigas artificiais "esquecerem" um caminho, favorecendo, assim, a exploração de novas áreas do espaço de busca. Gendreau e Potvin (2010) afirma que os algoritmos do ACO são tipicamente diferentes pela maneira no qual a atualização do feromônio é implementada. Usualmente, S_{atu} é um subconjunto de S_{inter}, onde S_{inter} é o conjunto de todas as soluções na iteração atual.

O $\Delta\tau_s(i,j)$, descrito na Equação 4, é dado por:

$$\Delta\tau_s(i,j) = \begin{cases} \frac{1}{C_s}, & \text{se } i,j \in s \\ 0, & \text{caso contrário} \end{cases} \tag{5}$$

No qual C_s é o valor da função objetivo da solução s do problema. O ganho de feromônio só é dado para as arestas (i,j) que estão presentes na solução, as demais não são incrementadas.

Aqui, o ACO está sendo utilizado com o objetivo de paralelizar o problema e o método heurístico utilizado é inerente ao paralelismo. Segundo Dorigo e Gambardella (1997b), o algoritmo do Colônia de Formigas é receptivo ao paralelismo, podendo ter ganhos significativos de desempenho para encontrar boas soluções, especialmente em problemas grandes.

4.5 PARALELISMO

Nas seções anteriores, foram abordados métodos com o objetivo de resolver problemas intratáveis. Entretanto, apesar de possuírem um ganho significativo de tempo, para problemas maiores as estratégias utilizadas por esses métodos podem não apresentar desempenho satisfatório. Então, como possível solução, o paralelismo pode trazer ganhos maiores para os algoritmos existentes.

Segundo Talbi (2009), com o rápido crescimento da tecnologia em criação de processadores (e.g., processadores de múltiplos núcleos, arquiteturas dedicadas), redes (e.g., redes LAN e WAN)[2] e armazenamento de dados está fazendo com que o paralelismo se torne cada vez mais popular. Portanto, tais arquiteturas podem oferecer estratégias eficazes para o projeto e implementação de meta-heurísticas paralelas.

A computação paralela e distribuída[3] podem ser utilizadas para criar implementações de meta-heurísticas pelas seguintes razões, como descrito em (TALBI, 2009):

- Acelerar busca: um dos principais objetivos do paralelismo em meta-heurísticas é reduzir o tempo de busca. Isto ajuda na criação de métodos de otimização interativos e de tempo real.

- Melhorar a qualidade das soluções obtidas: alguns modelos de meta-heurísticas paralelas permitem melhorar a qualidade da solução. Troca de informações entre meta-heurísticas cooperativas pode alterar o comportamento em termos da busca associada ao problema. O principal foco das meta-heurísticas cooperativas é melhorar a qualidade de solução.

- Aumentar a robustez: uma meta-heurística paralela pode ser mais robusta em termos de resolver, de maneira eficiente, diferentes problemas de otimização e diferentes instâncias de um dado problema.

- Resolver problemas grandes: meta-heurísticas paralelas permitem resolver problemas com instâncias grandes[4] de problemas complexos de otimização. O desafio é resolver esses problemas com instâncias grandes que não podem ser resolvidos em uma máquina sequencial.

[2] **LAN é um acrônimo para *Local Area Network* e WAN para *Wide Area Network*.**

[3] **O foco deste trabalho é na computação paralela, e não na distribuída.**

[4] **Instâncias ou problemas de instâncias grandes podem ser relacionados aos ditos grafos massivos, definido pelo Cai (2015), como instâncias que ultrapassam milhares de vértices.**

Visto que o paralelismo pode ser utilizado para resolver problemas intratáveis, é preciso definir e analisar a relação entre custo e benefício, considerando o custo de investimento em equipamentos. Em função disto, é preciso aferir o desempenho de uma aplicação paralela em relação a sua sequencial (ou ainda de paralelo com paralelo), tal medida é feita pelo o cálculo do *Speedup*.

De acordo com Diverio *et al.* (2002), o *speedup* é definido como o quociente do tempo de execução do mais rápido algoritmo sequencial para o problema, no pior caso, pelo tempo do algoritmo paralelo, também no pior caso.

$$S(N) = \frac{T(1)}{T(N)} \tag{6}$$

O *speedup* é demonstrado pela Equação 6, onde $T(N)$ é o tempo de execução em N processadores e $T(1)$ é o tempo de execução em um único processador. O *speedup* é dado como um ganho de computação para N processadores.

Os algoritmos paralelos podem melhorar a solução de diversos problemas, porém, eles são dependentes da arquitetura do *hardware* na qual será implementado (ROOSTA, 2000). Tais arquiteturas são apresentadas em Flynn (1972).

A Taxonomia de FLYNN, segundo NAVAUX e ROSE (2008), se baseia no fato de um computador executar uma sequência de instruções sobre uma sequência de dados, no qual diferenciam-se o fluxo de instruções (*instruction stream*) e o fluxo de dados (*data stream*). Sendo estes fluxos múltiplos ou não, FLYNN propôs quatro classes apresentadas na Tabela 1.

	SD (*Single Data*)	MD (*Multiple Data*)
SI (*Single Instruction*)	SISD	SIMD
MI (*Multiple Instruction*)	MISD	MIMD

Tabela 1: Classificação de FLYNN segundo o fluxo de instruções e o fluxo de dados. Fonte: (NAVAUX; ROSE, 2008).

A classe SISD (*Single Instruction Single Data*), um único fluxo de instruções atua sobre um único fluxo de dados, conforme mostra a Figura 4. A unidade de processamento *UP* atua sobre um único fluxo de dados, na qual é lido, processado e reescrito na memória (NAVAUX; ROSE, 2008). Nessa classe é enquadrada as máquinas

de von Neumann tradicionais com apenas um processador.

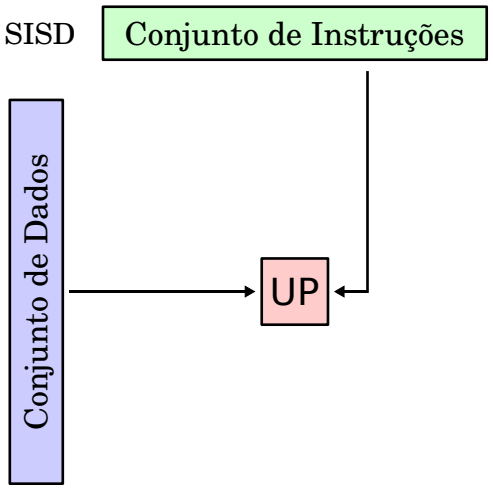

Figura 4: Diagrama da classe SISD
Fonte: https://en.wikipedia.org/wiki/File:SISD.svg.

A classe MISD (*Multiple Instruction Single Data*), é aquela que múltiplos fluxos de instruções atuariam sobre um único conjunto de dados, conforme a Figura 5. As unidades de processamento *UP* executam suas diferentes instruções em um único fluxo de dados. NAVAUX e ROSE (2008) concorda que esse tipo de arquitetura não faz qualquer sentido na qual é uma arquitetura impraticável.

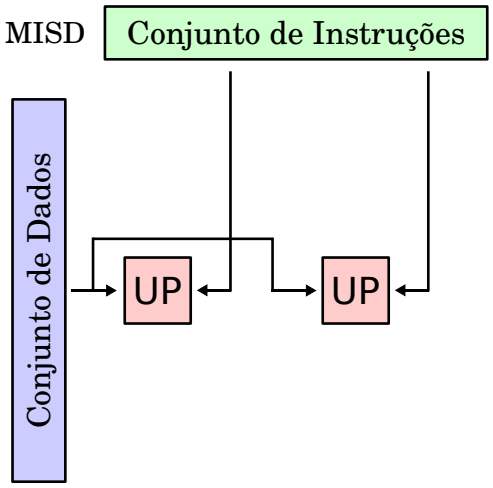

Figura 5: Diagrama da classe MISD
Fonte: https://en.wikipedia.org/wiki/File:MISD.svg.

A classe SIMD (*Single Instruction Multiple Data*), é aquela na qual uma única instrução é executada ao mesmo tempo sobre múltiplos dados, conforme a Figura 6. A mesma instrução é enviada para os diversos processadores *UP* envolvidos na execução. Todos os processadores executam suas instruções em paralelo de forma síncrona sobre diferentes fluxos de dados. Nessa classe são enquadrados os computadores vetoriais (NAVAUX; ROSE, 2008). Segundo Cheng *et al.* (2014), a maior van-

tagem da arquitetura SIMD, é que o programador pode pensar de forma sequencial e alcançar a aceleração paralela a partir de operações de dados paralelas, já que os compiladores garantem isso.

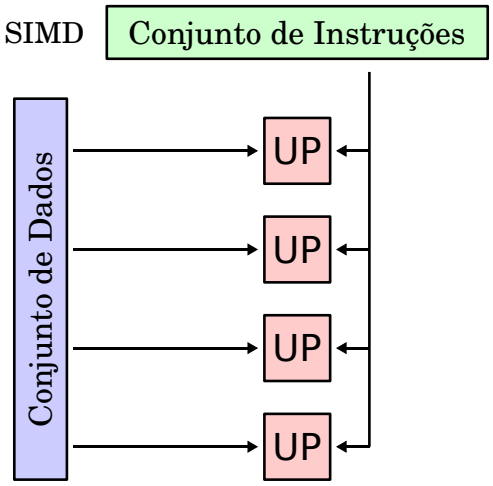

Figura 6: Diagrama da classe SIMD
Fonte: https://en.wikipedia.org/wiki/File:SIMD.svg.

E finalmente a classe MIMD (*Multiple Instruction Multiple Data*), cada processador executa o seu próprio programa sobre seus próprios dados de forma assíncrona, conforme mostra a Figura 7. Nessa classe, se enquadrada os computadores com múltiplos processadores.

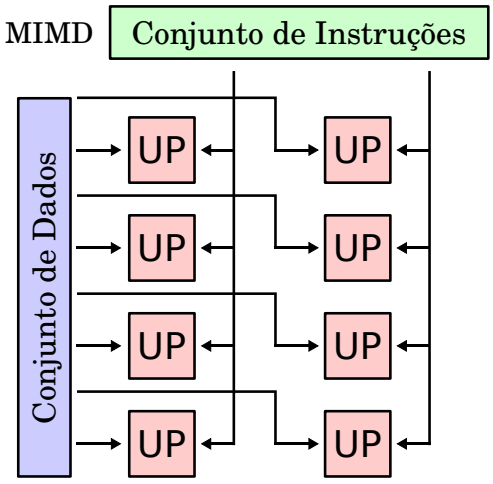

Figura 7: Diagrama da classe MIMD
Fonte: https://en.wikipedia.org/wiki/File:MIMD.svg.

Segundo (CHENG *et al.*, 2014), as arquiteturas de computadores podem ser subdivididas por sua organização de memória, no qual geralmente são classificados por dois tipos, são eles:

- Múltiplos nós com memória distribuída.

- Múltiplos processadores com memória compartilhada.

Um sistema de múltiplos nós, é constituído de muitos processadores conectados por uma rede. Cada processador possui sua própria memória local, na qual podem trocar conteúdo de sua memória através de trocas de mensagens na rede. Esse tipo de sistema é mais conhecido como *clusters* (CHENG *et al.*, 2014).

Arquiteturas de múltiplos processadores geralmente possuem tamanhos dentre pelos menos dois processadores até dezenas ou centenas de processadores. Esses processadores são fisicamente conectados em uma mesma memória, ou compartilham uma memória de baixa latência. Tais múltiplos processadores incluem em um único *chip*, sistemas com múltiplos núcleos, conhecidos como *multicore*. O termo *many-core* é usualmente utilizado para descrever arquiteturas *multicore* com a especialidade de possuir um número elevado de núcleos, como as GPUs (CHENG *et al.*, 2014).

Com o propósito de utilizar uma arquitetura MIMD para implementar o Colônia de Formigas, como visto na Seção 4.4.1, o texto que segue descreve o uma breve introdução do OpenMP, que é a ferramenta utilizada para paralelizar este algoritmo.

4.5.1 OPENMP

De acordo com CHAPMAN *et al.*, OpenMP é uma API para programação multiprocesso de memória compartilhada, no qual possui como objetivo padronizar e facilitar a programação paralela.

A programação paralela com o OpenMP é dada através de diretivas de compilação que são adicionadas no decorrer de um código sequencial (CHAPMAN *et al.*, 2007), como descrito no Listing 4.1, onde na Linha 5 foi declarado um bloco de código paralelo onde serão criados quatro *threads* que iram exibir na tela um: `Ola Mundo!`.

```
1 #include <omp.h>
2 #includie <stdio.h>
3
4 int main(){
5   #pragma omp parallel num_threads(4)
6   {
7     int id = omp_get_thread_num();
8     printf("Ola Mundo %d \n", id);
9   }
```

```
10    return 0;
11 }
```

Listing 4.1: Exemplo Programa OpenMP - O Olá Mundo

O OpenMP não se limita somente a diretiva utilizada acima, com ele é possível declarar regiões críticas, realizar laços de repetição em paralelo, entre outras funcionalidades, tudo por meio de diretivas.

Para este trabalho, esta ferramenta será utilizada para paralelizar o algoritmo Colônia de Formigas para o Problema da Cobertura Mínima de Vértices (PCMV).

5 MATERIAIS E MÉTODOS

5.1 MATERIAIS

A solução proposta neste trabalho foi implementada em um sistema baseado em GNU/Linux utilizando a biblioteca *OpenMP* do `gcc`. A máquina na qual foram implementadas as soluções e realizados os experimentos computacionais dispõe de um processador IntelR CoreTM i7-7700HQ(4), 2,8 GHz à 3,8 GHz de *clock*, com 8 GB de memória RAM.

As instâncias de testes dos algoritmos propostos estão presentes na biblioteca DIMACS, que conta com 76 instâncias de tamanhos variados, e no site do *NetworkRepository*[1].

5.2 MÉTODOS

Inicialmente, foi levantado o estado da arte para o Problema da Cobertura Mínima de Vértices a fim de adquirir uma base de comparação de resultados. A facilidade de encontrar soluções sequenciais e instâncias massivas na literatura foi determinante na escolha do problema proposto.

Em seguida, foi escolhida a meta-heurística implementada e as técnicas de paralelização, na qual, chegou-se a conclusão que o Colônia de Formigas é cabível para ser implementado sequencialmente e em paralelo. A escolha da meta-heurística foi influenciada pela facilidade de paralelização, fato este que o autor dessa meta-heurística confirma em (DORIGO; GAMBARDELLA, 1997b). O ACO foi inicialmente proposto para resolver o PCV. Entretanto, com base no trabalho proposto em (GILMOUR; DRAS, 2006), ele foi adaptado para o PCMV com as devidas alterações no algoritmo e nas equações.

Tendo definidos o problema, a meta-heurística e as técnicas paralelas, foram realizadas as elaborações e implementações dos algoritmos propostos.

O primeiro grupo de implementações deste trabalho constituiu de três implementações sequenciais que resolva o PCMV com o ACO. O foco desse primeiro grupo

[1] **Disponível em:** `www.networkrepository.com`

serviu como uma modelagem das estruturas de dados necessárias para o problema, assim como descobrir algumas características do próprio problema a fim de ter um implementação otimizada. Esse grupo servirá de comparação para os algoritmos paralelos.

O segundo grupo de implementações do problema, que foi a continuação do primeiro grupo, foi realizado em duas etapas: elaboração dos esboços dos algoritmos paralelos, possuindo como base os dois melhores algoritmos sequenciais e implementá-los. A técnica paralela aplicada é a mais simples para o ACO, na qual cada formiga executa em um processador separado concorrentemente. Tendo em vista que a CPU, na qual foi executada os algoritmos implementados, possui uma arquitetura *MIMD* e que no ACO a única sincronização entre cada formiga artificial é o feromônio, não foi necessário tomar precauções contra condições de corrida e/ou ao acesso à região crítica, pois cada formiga artificial pode ser executada de forma independente, não possuindo, assim, regiões críticas e nem condições de corrida. Para essa implementação serão utilizadas as bibliotecas do *OpenMP*.

As instâncias de teste para os algoritmos podem ser encontradas no site do *NetworkRepository*[2], o qual, conta com diversos tipos de grafos com tamanhos que se aplicam ao problema proposto. Cada implementação será submetida a testes exaustivos de desempenho para análise de resultados. Os resultados obtidos foram comparados entre si a fim de aferir os ganhos de desempenho de uma implementação em relação a outra. Em relação à qualidade de solução obtida, os resultados foram comparados com (CAI *et al.*, 2017), que possui os resultados mais recentes e eficientes na literatura. Tal comparação na qualidade de solução tem o objetivo de verificar se com o paralelismo, a qualidade de solução do algoritmo pode ser melhorada.

Com os resultados obtidos será possível responder as perguntas que este trabalho visa responder.

[2]**Disponível em:** `http://networkrepository.com/`

6 ALGORITMOS PROPOSTOS

Os algoritmos descritos a seguir foram adotados na implementação sequencial do problema. Em seguida, será abordado algumas alterações do algoritmo sequencial adotado aqui, e será explicado o porque destas modificações. Logo após, os algoritmos propostos serão adaptados para as implementações paralelas. Cada elemento das equações são vetores, com exceção dos parâmetros do ACO (*Ant Colony Optimization*, ou Otimização por Colônia de Formigas) e da informação heurística. Considera-se que as operações entre vetores são realizadas ponto a ponto. A notação matemática adotada é semelhante a que foi utilizada em (DORIGO; BLUM, 2005; GILMOUR; DRAS, 2006), na qual, foram descritas na Seção 4.4.1.

6.1 ORGANIZAÇÃO DOS DADOS

O algoritmo proposto possui como entrada um grafo $G = (V, A)$ e os demais parâmetros necessários para o ACO. São eles o α, β e σ. Os parâmetros do ACO possuem a mesma função e notação adotada por DORIGO; GAMBARDELLA, citado na Seção 4.4.1. O feromônio é representado por τ como uma informação vista por todas as formigas. A informação heurística é representada por η_k como uma informação individual para cada formiga. A heurística de escolha no caso do PCMV é o grau do vértice. A estrutura do feromônio é dada por um vetor de dimensão $|V|$ e a estrutura da informação heurística é dada por uma matriz de dimensão $Formigas \times |V|$. Uma solução S é representada por um vetor binário, no qual, os vértices não presentes na solução são representados por 1, caso contrário, com 0.

6.2 ACO PARA O PROBLEMA DA COBERTURA MÍNIMA DE VÉRTICES (PCMV)

O algoritmo proposto neste trabalho é representado pelo Algoritmo 5. Como detalhado anteriormente, ele possui como entrada um α, β e σ como entrada do ACO, $G = (V, A)$ como representação do problema por um grafo e os parâmetros $iteracoes$ e $converge$ como condição de parada. Esse algoritmo possui como retorno a melhor solução encontrada.

O algoritmo proposto é constituído de quatro etapas. Com relação ao ACO do

DORIGO; GAMBARDELLA é adicionada uma fase de otimização local, denominada LOT, proposta em (BARBOSA *et al.*, 2013). Essa fase será realizada para o conjuntos de soluções utilizadas para atualizar o feromônio, conforme a Linha 6.

Na Linha 1 é realizado a inicialização do feromônio. O valor inicial para ele será feito de acordo com a inicialização feita por DORIGO; GAMBARDELLA, citado na Seção 4.4.1.

Algoritmo 5: COLÔNIA DE FORMIGAS PROPOSTO

Entrada: $\alpha, \beta, \sigma, G = (V, A), iteracoes, converge$
Saída: S_{melhor}

```
1  τ ← IniciaFeromônio
2  S_melhor ← ∞
3  S_ant ← ∅
4  enquanto pass < iteracoes ∧ no_mud < converge faça
5      S_atu ← constroiSoluçãoFormigas()
6      S_atu ← buscaLOT()
7      τ ← AtualizaFeromônio()
8      S ← melhorSolucao(S_atu)
9      se S é melhor que S_melhor então
10         S_melhor ← S
11     fim
12     se S_ant = S_atu então
13         no_mud ← no_mud + 1
14     senão
15         S_ant ← S
16         no_mud ← 0
17     fim
18     pass ← pass + 1
19 fim
20 retorna S_melhor
```

Após as devidas inicializações, o ACO começa o seu processo de construção de soluções e atualização de feromônio, no laço descrito na Linha 4, até que todas as formigas convirjam para um único valor de solução ou até quando se esgotar a quantidade de iterações. Para deixar de forma genérica, adotou-se que o *constroiSoluçãoFormigas* retorna um conjunto de soluções S_{atu} as quais serão utilizadas para atualizar o feromônio.

Na Linha 6 será realizado um mecanismo de refinamento a fim de melhorar as soluções oriundas da fase de construção. Esse mecanismo faz varredura nas soluções procurando por vértices "duplamente cobertos"[1]. Em seguida, na Linha 7, o feromônio

[1] **Vértices "duplamente cobertos" são vértices que estão na solução, mas se removidos, as suas arestas não deixam de estarem cobertas.**

é atualizado e na Linha 8 é escolhido a melhor solução presente no conjunto. Se essa solução for melhor do que a melhor solução já encontrada, S_{melhor} é atualizado. Na Linha 12, é verificado se as soluções encontradas na iteração é igual as soluções encontradas na iteração anterior, caso afirmativo, o contador no_{mud} é incrementado, caso contrário, é atualizado o S_{ant}. No final do laço, o valor $pass$, que representa a iteração na qual se passa o algoritmo, é atualizado, e o algoritmo retorna a melhor solução encontrada.

6.2.1 FASE DE CONSTRUÇÃO

O Algoritmo 6 possui como função construir um conjunto de soluções para todas as formigas, onde tais soluções são utilizadas para atualizar o feromônio. Inicialmente, o vetor de solução é preenchido com 1 para todas as formigas (Linha 1) e a informação heurística é inicializado igualmente para todas as formigas com o grau de cada vértice, conforme a Linha 2. Em seguida, na Linha 5, começa o laço que irá gerar uma solução para todas as formigas. Nesse laço, cada formiga calcula previamente o somatório presente no denominador da Equação 7, conforme a Linha 6.

Algoritmo 6: CONSTROISOLUÇÃOFORMIGAS

Entrada: $\alpha, \beta, \tau, G = (V, A), n$
Saída: S_{atu}

1 $S^{\forall v \in V}_{\forall k \in Formigas} \leftarrow 1$
2 $\mu^{\forall v \in V}_{\forall k \in Formigas} \leftarrow grau(v)$
3 $S_{atu} \leftarrow \emptyset$
4 $v^{\forall k in Formigas}_{sol} \leftarrow 0$
5 **para** $\forall k \in$ *Formigas* **faça**
6 $\quad Den_{soma} \leftarrow \Sigma_{\forall v \in V} \tau_v^{\alpha} \cdot \mu_{k,v}^{\beta} \cdot S_k^v$
7 $\quad$ **enquanto** G *possuir arestas* **faça**
8 $\quad\quad v \leftarrow selVertProb(Den_{soma})$
9 $\quad\quad S_k \leftarrow S_k \bigcup \{v\}$
10 $\quad\quad v^k_{sol} \leftarrow v^k_{sol} + 1$
11 $\quad\quad \mu_k \leftarrow atualizaHeuristica(v)$
12 $\quad\quad Den_{soma} \leftarrow atualizaDen(Den_{soma})$
13 $\quad$ **fim**
14 $\quad S_{atu} \leftarrow S_{atu} \bigcup S_k$
15 **fim**
16 **retorna** S_{atu}

Na Linha 7, começa o laço que será executado com a condição de ainda não existir solução, que no caso do PCMV, é enquanto possuir arestas não cobertas. Antes de qualquer vértice entrar na solução, ele passa por um processo de seleção, descrito

na Linha 8, onde é escolhido um novo vértice para compor a solução. Logo após um vértice ser escolhido, ele é adicionado na solução, de acordo com a Linha 9.

A informação heurística para o PCMV, diferentemente do PCV, é alterada no decorrer das escolhas realizadas por cada formiga. Pois, quando um vértice é adicionado na solução, é preciso remover as arestas cobertas por aquele vértice. Nessa remoção, todos os vértices incidentes do vértice adicionado tem seu grau de vértice atualizado. Portanto, a informação heurística da formiga é atualizada toda vez que um vértice é adicionado, conforme a Linha 11.

Na Linha 14, é realizado a atualização do somatório descrito no denominador da Equação 7, onde nesta atualização, somente os vértices incidentes ao vértice adicionado na solução possui sua participação nesse somatório decrementada. No fim do laço de construção, a solução encontrada pela presente formiga é adicionada no conjunto de soluções, como descrito na Linha 14.

6.2.2 FASE DE SELEÇÃO

O Algoritmo 7 possui como função realizar o processo de seleção antes que um vértice componha a solução. Tal seleção é realizada de forma pseudo-aleatória. Cada vértice possui uma probabilidade de seleção, dada por:

$$p_v = \frac{[\tau_v]^\alpha \cdot [\eta_{k,v}]^\beta}{\Sigma[\tau]^\alpha \cdot [\eta_k]^\beta} \cdot S_k^v \tag{7}$$

onde τ_v é o feromônio presente no vértice v, $\eta_{k,v}$ é a informação heurística para a formiga k presente no vértice v e S_k^v é o valor de solução para a formiga k no vértice v.

O elemento S_k^v, descrito na Equação 7, possui como objetivo, garantir que nenhum vértice seja escolhido novamente. Pois se um vértice v é inserido na solução, $S_k^v = 0$, portanto, $p_v = 0$. Isso garante que o vértice v não seja escolhido novamente para solução, pois ele não possui chances de ser escolhido.

O cálculo de probabilidade é descrito pela Linha 4, possuindo como diferença da Equação 7, o denominador da equação, pois ele é previamente calculado. O somatório deste denominado é calculado em duas situações: a) antes de inicializar a construção de solução da formiga e b) depois que um vértice é adicionado na solução. O cálculo realizado antes de iniciar a construção da solução de uma formiga é realizado para todos os vértices presentes no grafo. O cálculo realizado após adicio-

Algoritmo 7: SELVERTPROB

Entrada: $\alpha, \beta, \tau, \mu_k, S_k, Den_{soma}, G = (V, A)$
Saída: v_{melhor}

```
1  Sel_melhor ← 0
2  v_melhor ← 0
3  para ∀v ∈ V faça
4  |  Prob_v ← (τ_v^α · μ_{k,v}^β) / Den_soma · S_k^v
5  |  Sel_v ← aleatorio(0, 1] · Prob_v
6  |  se Sel_v > Sel_melhor então
7  |  |  Sel_melhor ← Sel_v
8  |  |  v_melhor ← v
9  |  fim
10 fim
11 retorna v_melhor
```

nar um vértice na solução, é realizado apenas para os vértices incidentes ao vértice adicionado.

Após o cálculo de probabilidade, é necessário adotar um critério de escolha para selecionar o vértice que será adicionado na solução. Esse processo é realizado através do método *I-Roulette*, proposto em (CECILIA *et al.*, 2013). Esse método consiste em gerar um número aleatório entre 0 e 1 para cada vértice, e multiplica-lo pela a probabilidade de seleção que cada vértice possui, como descrito na Linha 5. O número calculado por essa multiplicação é atribuído à uma variável Sel_v. A seleção do vértice é dado para o vértice que possuir o valor máximo de Sel_v. A procura do melhor valor de Sel_v é descrita na Linha 6.

No final do Algoritmo, é retornado o vértice que possui o valor máximo de Sel_v, e este processo é realizado todas as vezes que um vértice é adicionado na solução.

6.2.3 ATUALIZAÇÃO DO SOMATÓRIO

O Algoritmo 8 possui como objetivo realizar o decremento do valor representado por Den_{soma}, que é o valor do denominado da Equação 7. Como citado na Seção 6.2.2, o cálculo deste somatório é realizado antes de uma formiga começar o processo de construção de solução e depois que um vértice é adicionado na solução. No primeiro caso, é necessário apenas de um algoritmo com um laço de repetição que possua uma variável que acumule o valor de soma para todos os vértices. No segundo caso, como descrito pelo Algoritmo 8, o valor final do somatório será apenas decrementado em relação ao valor de entrada, o Den_{soma}.

Esse algoritmo é dividido em duas etapas, são elas: a) decrementar o valor

de contribuição que os vértices adjacentes ao vértice que foi adicionado possui e b) decrementar o valor de contribuição que o vértice removido possuía na solução. A primeira etapa é iniciada com o laço da Linha 4, onde, para todos os vértices adjacentes, é calculado um Δ, que é dado por:

$$\Delta \leftarrow \tau_{v'}^{\alpha} \cdot (\mu_{v'}^{\beta} - (\mu_{v'} + 1)^{\beta}) \cdot S_{v'} \tag{8}$$

onde v' é um vértice adjacente do vértice v que foi adicionado na solução, $\tau_{v'}$ é o feromônio presente no vértice adjacente v', $\mu_{v'}^{\beta}$ é o valor da informação heurística atual, e $(\mu_{v'} + 1)^{\beta}$ é o valor da informação heurística antes do vértice ser adicionado. A diferença de 1 é pelo o fato que toda vez que um vértice é adicionado na solução, a informação heurística do vértices adjacentes ao vértice adicionado é decrementada apenas de 1 unidade. O elemento $S_{v'}$ serve para indicar que apenas os vértices que não estão presentes na solução possui sua parcela decrementada. No algoritmo, esse cálculo é realizado na Linha 5.

Algoritmo 8: atualizaDen

Entrada: $Den_{soma}, \mu_k, \tau, S_k, v$
Saída: a_{soma}

```
1  a_soma ← Den_soma
2  Δ ← 0
3  n_i ← 0
4  para ∀v' ∈ Adj_v faça
5  |  Δ ← τ_v'^α · (μ_v'^β − (μ_v' + 1)^β) · S_v'
6  |  a_soma ← a_soma + Δ
7  |  n_i ← n_i + S_v'
8  fim
9  Δ ← −τ_v^α · n_i^β
10 a_soma ← a_soma + Δ
11 retorna a_soma
```

Para realizar a segunda etapa, é necessário recuperar do vértice que foi adicionado na solução, a sua antiga informação heurística (já que ela foi zerada depois que o vértice foi adicionado na solução). Para isto, é preciso saber quantos dos vértices adjacentes não estão presentes na solução. Como $S_{v'}$ é 1 quando o vértice v' não está presente na solução, basta somar os valores de $S_{v'}$ para descobrir a antiga informação heurística, conforme é descrito na Linha 7. Após recuperar a informação heurística do vértice, é realizado o mesmo cálculo de Δ para o vértice que foi adicionado na solução e adicionado este valor em a_{soma}, de acordo com as Linhas 9 e 10. O novo valor de Den_{soma} é retornado no final do algoritmo pela variável a_{soma}.

6.2.4 ATUALIZAÇÃO DE FEROMÔNIO

Para o PCMV, atualizar o feromônio funciona de maneira igual do ACO para o PCV. A fim de deixar a equação de incremento para todos os vértices, sendo eles pertencentes a solução ou não, foi adicionado $!s$. Esse elemento adicionado é o vetor de solução "negados". Pois, como é feita a representação de um vértice que está na solução como 0, para o feromônio, deve ser dado um incremento para aquele vértice, então, aplicar a negação no vértice, faz como que seja aplicado um incremento de feromônio naquele vértice, conforme descrito pela Equação 10.

$$|s| = \Sigma_{\forall v \in s} s_v \tag{9}$$

$$\Delta\tau_s = \tfrac{1}{|s|} . !s \tag{10}$$

$$\tau = (1 - \sigma) \cdot \tau + \Sigma_{s \in S_{upd}} \Delta\tau_s \tag{11}$$

O novo feromônio é definido pela Equação 11, no qual, é feito a evaporação do mesmo no trecho $(1 - \sigma) \cdot \tau$ para que em seguida, ter o incremento do mesmo. O incremento é dado pelo somatório de todos os incrementos, pois, cada solução em S_{atu} irá incrementar o feromônio.

Os algoritmos descritos até aqui, fazem parte da primeira implementação do ACO sequencial. Com o objetivo de melhorar o tempo de execução destes algoritmos, foi necessário fazer algumas adaptações e alterações no algoritmo, como será descrito a seguir.

6.3 RETIRADA DO DENOMINADOR DA EQUAÇÃO DE PROBABILIDADES

A primeira modificação ocorrida no algoritmo adaptado do ACO para o PCMV, foi a retirada do denominador da Equação 7, formando uma nova equação, que agora é dada por:

$$p_v = [\tau_v]^\alpha \cdot [\eta_{k,v}]^\beta \cdot S_k^v \tag{12}$$

Com a nova equação, o ACO não precisa mais realizar os cálculos de atualização de denominador. A simplificação da Equação 7 foi realizada a fim de eliminar as atualizações do valor do somatório, e assim, reduzir tempo de execução do algoritmo.

Dado que o valor do denominador é uma constante e que o método de seleção

I-Roulette escolhe o vértice com o maior valor de Sel_v (variável descrita no Algoritmo 7). A retirada do denominador não influência na escolha do método *I-Roulette*.

Com a alteração ocorrida no cálculo de probabilidade de seleção, descrita pela Equação 12, o Algoritmos 6 foi atualizado, retirando dele os termos referentes ao denominador, conforme agora é descrito pelo Algoritmo 9.

Algoritmo 9: CONSTROISOLUÇÃOFORMIGASSEMDEN

Entrada: $\alpha, \beta, \tau, G = (V, A), n$
Saída: S_{atu}

$S^{\forall v \in V}_{\forall k \in Formigas} \leftarrow 1$
$\mu^{\forall v \in V}_{\forall k \in Formigas} \leftarrow grau(v)$
$S_{atu} \leftarrow \emptyset$
$v^{\forall k \in Formigas}_{sol} \leftarrow 0$
para $\forall k \in$ *Formigas* **faça**
  enquanto G *possuir arestas* **faça**
    $v \leftarrow selVertProbSemDen()$
    $S_k \leftarrow S_k \bigcup \{v\}$
    $\mu_k \leftarrow atualizaHeuristica(v)$
    $v^k_{sol} \leftarrow v^k_{sol} + 1$
  fim
  $S_{atu} \leftarrow S_{atu} \bigcup S_k$
fim
retorna S_{atu}

Na fase de seleção, também foi retirado as referências do termo do antigo denominador, resultando no Algoritmo 10.

Algoritmo 10: SELVERTPROBSEMDEN

Entrada: $\alpha, \beta, \tau, \mu_k, S_k, G = (V, A)$
Saída: v_{melhor}

$Sel_{melhor} \leftarrow 0$
$v_{melhor} \leftarrow 0$
para $\forall v \in V$ **faça**
  $Prob_v \leftarrow \tau^\alpha_v \cdot \mu^\beta_{k,v} \cdot S^v_k$
  $Sel_v \leftarrow aleatorio(0, 1] \cdot Prob_v$
  se $Sel_v > Sel_{melhor}$ **então**
    $Sel_{melhor} \leftarrow Sel_v$
    $v_{melhor} \leftarrow v$
  fim
fim
retorna v_{melhor}

Fazendo uma analise empírica de ambos os Algoritmos, com a simplificação da Equação 7, o tempo de execução da primeira modificação é menor se comparado

com a primeira implementação. As modificações ocorridas resultaram na segunda implementação do ACO para o PCMV.

O algoritmo de construção proposto pelo o Algoritmo 9, retorna um conjunto de solução de todas as formigas, e em seguida, essas soluções são utilizadas para atualizar o feromônio. Quando todas as formigas atualizam o feromônio, pode ocorrer que algumas formigas que possuem soluções ruins fazer parte da atualização do feromônio, o que não é desejado. Para que somente algumas formigas atualizem o feromônio, foi proposto o ACO Elistista, onde a diferença com o ACO clássico está no método de construção das soluções.

6.4 FASE DE CONSTRUÇÃO ELITISTA

O algoritmo Colônia de Formigas Elitista busca selecionar apenas as n melhores soluções encontradas pelas as formigas, onde n é a quantidade de formigas elitista. Diferentemente do PCV, onde todas as formigas encontram a solução para o problema ao mesmo tempo, no PCMV, cada formiga pode encontrar sua solução em tempos diferentes. Isso ocorre porque no PCMV, a medida de qualidade da solução é a quantidade de vértices na cobertura.

Sabendo que o PCMV é um problema de minimização, e que o processo de construção de solução adiciona um vértice na solução em cada rodada do processo. Então, a primeira formiga que encontrar uma solução, é a melhor solução daquela iteração, e a segunda formiga que encontrar outra solução, é a segunda melhor, e assim sucessivamente. Pensando nessa estrutura do problema e na ideia de que nem todas as formigas devem atualizar o feromônio, é proposto o ACO Elitista, onde somente o processo de construção de solução é alterado em relação ao algoritmo clássico.

O Algoritmo 11 mostra o processo de construção de solução para as formigas elitistas. Este algoritmo possui como entrada os parâmetros α, β, τ, $G = (V, A)$, que são iguais aos parâmetros do Algoritmo 9, mais o parâmetro $ELITE$. O parâmetro $ELITE$ representa a quantidade de formigas que irão atualizar o feromônio. O retorno do algoritmo é o conjunto S_{elite}, na qual, possui as melhores soluções que atualizarão o feromônio.

Antes do algoritmo começar o processo de solução, é necessário que ele faça as suas devidas inicializações. Portando, no inicio, o S_{elite} começa como um conjunto vazio (Linha 1) e as soluções de todas as formigas é inicializada com 1, conforme a

Linha 3.

O processo de construção do Algoritmo 9 é realizado para todas as formigas, porém, cada formiga constrói sua solução de forma sequencial. Isto é, para uma segunda formiga construir sua solução, a primeira formiga tem que ter terminado o seu processo de construção. Para o processo de construção elitista, todas as formigas procuram um vértice para adicionar na solução ao mesmo tempo. Assim, o laço que inicia na Linha 7, permite que as formigas fiquem procurando suas devidas soluções até que seja preenchido o conjunto S_{elite}, e o laço inicializado na Linha 8 permite que todas as formigas procurem um vértice de cada vez.

Algoritmo 11: CONSTROISOLUÇÃOELITE

Entrada: $\alpha, \beta, \tau, G = (V, A), ELITE$
Saída: S_{elite}

1 $S_{elite} \leftarrow \emptyset$
2 $Elite_{cont} \leftarrow 0$
3 $S^{\forall v \in V}_{\forall k \in Formigas} \leftarrow 1$
4 $A^{\forall k \in Formigas}_{rest} \leftarrow |A|$
5 $v^{\forall k \in Formigas}_{sol} \leftarrow 0$
6 $\mu^{\forall v \in V}_{\forall k \in Formigas} \leftarrow grau(v)$
7 **enquanto** $Elite_{cont} < ELITE$ **faça**
8 **para** $\forall k \in Formigas$ **faça**
9 **se** $S_k \notin S_{elite}$ **então**
10 $v \leftarrow selVertProbSemDen()$
11 $S_k \leftarrow S_k \bigcup \{v\}$
12 $\mu_k \leftarrow atualizaHeuristica(v)$
13 $v^k_{sol} \leftarrow v^k_{sol} + 1$
14 $A^k_{rest} \leftarrow calculaArestasRestantes(v)$
15 **se** $A^k_{rest} = 0$ **então**
16 $S_{elite} \leftarrow S_{elite} \bigcup S_k$
17 $S_{elite-sol} \leftarrow v^k_{sol}$
18 $Elite_{cont} \leftarrow Elite_{cont} + 1$
19 **fim**
20 **se** $Elite_{cont} = ELITE$ **então**
21 **retorna** S_{elite}
22 **fim**
23 **fim**
24 **fim**
25 **fim**
26 **retorna** S_{elite}

Como todas as formigas estão escolhendo os vértices para serem adicionados em suas soluções ao mesmo tempo, pode ocorrer que alguma formiga tenha encon-

trado uma solução antes que as outras. Para que não ocorra que a formiga que já achou a solução fique procurando algum vértice para adicionar, foi necessário adicionar a condição da formiga procurar um vértice somente se ela não estive em S_{elite}, conforme descrito na Linha 9.

Na Linha 10, um vértice é escolhido para compor a solução da formiga k. O processo de seleção de vértice é realizado pelo Algoritmo 10. Toda vez que um vértice é selecionado, ele é adicionado na solução da formiga k (conforme a Linha 11), assim como esta mesma formiga possui a sua informação heurística atualizada (Linha 12) e sua quantidade de arestas restantes para completar sua solução decrementada, de acordo com a Linha 14.

No momento que a formiga k não possuir mais arestas para serem cobertas (Linha 15), ela é adicionada em S_{elite}, conforme a Linha 16. Após a solução da formiga k ser adicionada em S_{elite}, o contador $Elite_{cont}$ é incrementado. Pode acontecer de mais de uma formiga achar uma solução na mesma iteração, e o conjunto S_{elite} está cheio. Neste caso, é necessário verificar se o contador $Elite_{cont}$ atingiu a quantidade máxima, que é $ELITE$ (Linha 20). Caso afirmativo, a execução da função é finalizada com o retorno de S_{elite}, conforme a Linha 21.

O método de construção das formigas elitistas resultou na terceira implementação do ACO para o PCMV. Possuindo em mãos três implementações do ACO (duas melhoras do algoritmo), foi realizado a paralelização em CPU dos dois últimos, que são: implementação sem o denominador da equação de probabilidades e método de construção elitista. A paralelização desses métodos ocorreram somente para a fase de construção.

6.5 ACO PARALELO

De acordo com DORIGO; GAMBARDELLA, o ACO é receptivo ao paralelismo, podendo ter ganhos significativos de desempenho para encontrar boas soluções, especialmente em problemas grandes. Sendo assim, foi realizado a paralelização dos Algoritmos 9 e 11.

A versão paralela do Algoritmo 9 é representado pelo Algoritmo 12. Para o ACO, todas as formigas procuram soluções independentemente umas das outras, possuindo apenas a informação do feromônio como uma informação compartilhada entre elas. Portanto, o paralelismo foi aplicado neste algoritmo para separar cada formiga em uma *thread* diferente. Assim, cada formiga constrói sua solução de forma

independente e paralela.

Algoritmo 12: CONSTROISOLUÇÃOFORMIGASSEMDENPARALELO

Entrada: $\alpha, \beta, \tau, G = (V, A), n$
Saída: S_{atu}

1 $S^{\forall v \in V}_{\forall k \in Formigas} \leftarrow 1$
2 $\mu^{\forall v \in V}_{\forall k \in Formigas} \leftarrow grau(v)$
3 $S_{atu} \leftarrow \emptyset$
4 **para** $\forall k \in$ *Formigas* **faça em paralelo**
5 **enquanto** G *possuir arestas* **faça**
6 $v \leftarrow selVertProbSemDen()$
7 $S_k \leftarrow S_k \bigcup \{v\}$
8 $\mu_k \leftarrow atualizaHeuristica(v)$
9 **fim**
10 $S_{atu} \leftarrow S_{atu} \bigcup S_k$
11 **fim**
12 **retorna** S_{atu}

O Algoritmo de construção paralela, em relação ao Algoritmo 9 continua o mesmo, a única diferença dos dois é que o laço que inicia na Linha 4 é executado em paralelo.

O Algoritmo 12 faz parte da primeira implementação paralela.

6.6 ACO ELITISTA PARALELO

A versão paralela do Algoritmo 11 é demonstrado pelo Algoritmo 13. Para a construção elitista, foi necessário criar uma região paralela na Linha 7, a fim de que o processo de construção de todas as formigas ocorressem de forma simultânea. Essa região paralela é divida em duas partes, são elas: a) região de processamento e b) região crítica.

A região de processamento é iniciada com o laço da Linha 9, no qual, executa para todas as formigas (em paralelo), uma iteração do processo de seleção de vértice e adição do mesmo na solução da formiga k.

O algoritmo de construção elitista, possui um recurso que todas as formigas "querem", que é entrar no conjunto S_{elite}. Entretanto, esse conjunto possui tamanho limitado, e não é permitido que mais de uma formiga "tente"entrar neste conjunto ao mesmo tempo, pois corre risco de ultrapassar o limite do conjunto ou até mesmo gerar inconsistências. Portanto, há um região crítica que deve ser sanada pelo o algoritmo, com relação a solução das formigas entrarem em S_{elite}.

Algoritmo 13: CONSTROISOLUÇÃOELITEPARALELO

Entrada: $\alpha, \beta, \tau, G = (V, A), ELITE$
Saída: S_{elite}

1 $S_{elite} \leftarrow 0$
2 $Elite_{cont} \leftarrow 0$
3 $S^{\forall v \in V}_{\forall k \in Formigas} \leftarrow 1$
4 $A^{\forall k \in Formigas}_{rest} \leftarrow |A|$
5 $v^{\forall k \in Formigas}_{sol} \leftarrow 0$
6 $\mu^{\forall v \in V}_{\forall k \in Formigas} \leftarrow grau(v)$
7 **região paralela**
8 **enquanto** $Elite_{cont} < ELITE$ **faça**
9 **para** $\forall k \in Formigas$ **faça em paralelo**
10 **se** $S_k \notin S_{elite}$ **então**
11 $v \leftarrow selVertProbSemDen()$
12 $S_k \leftarrow S_k \bigcup \{v\}$
13 $\mu_k \leftarrow atualizaHeuristica(v)$
14 $v^k_{sol} \leftarrow v^k_{sol} + 1$
15 $A^k_{rest} \leftarrow calculaArestasRestantes(v)$
16 **fim**
17 **fim**
18 **para** $\forall k \in Formigas$ **faça em paralelo**
19 **região crítica**
20 **se** $S_k \notin S_{elite} \wedge A^k_{rest} = 0 \wedge Elite_{cont} < ELITE$ **então**
21 $S_{elite} \leftarrow S_{elite} \bigcup S_k$
22 $S_{elite-sol} \leftarrow v^k_{sol}$
23 $Elite_{cont} \leftarrow Elite_{cont} + 1$
24 **fim**
25 **fim**
26 **fim**
27 **fim**
28 **fim**
29 **retorna** S_{elite}

Essa região crítica é inicializada na Linha 19, dentro de um laço que é executado para todas as formigas. Dentro da região crítica, é verificado se a formiga k possui todos os requisitos de entrar no conjunto S_{elite}. Tais requisitos são: a) a solução da formiga k não está em S_{elite}, b) a formiga k não possuir arestas para serem cobertas e c) o conjunto S_{elite} não está totalmente preenchido. Esses requisitos estão presentes na Linha 20. Como a região paralela é executada para todas as formigas, mesmo aquelas que já possuem uma solução definida, é necessário fazer a verificação se a solução da formiga k não está em S_{elite}. Como pode ocorrer de mais de uma formiga encontrar uma solução na mesma iteração, é necessário verificar se S_{elite} não

foi totalmente preenchido.

O Algoritmo 13 faz parte da segunda e última implementação paralela. Após a implementação dos algoritmos propostos neste capítulo, os mesmos foram submetidos a testes de exaustivos a fim de obter os resultados deste trabalho.

7 RESULTADOS E DISCUSSÕES

Os algoritmos propostos por este trabalho passaram por experimentos computacionais exaustivos a fim de aferir os seus desempenhos. Os experimentos foram realizados em duas etapas. A primeira etapa foi para determinar os valores dos parâmetros α, β, σ e quantidade de formigas, que apresentassem melhor desempenho em tempo de execução e melhor qualidade de solução. Depois de encontrados os parâmetros, a segunda etapa consistiu em submeter todos os algoritmos com os parâmetros encontrados anteriormente para as instâncias massivas e convencionais.

Os experimentos foram realizados para todas as instâncias clássicas encontradas na biblioteca DIMACS[1], e para algumas instâncias massivas presentes no site do *NetworkRepository*[2].

A biblioteca DIMACS[3] possui 76 instâncias de testes, que são dividas em grupos de tamanhos variados. Esses grupos são: `brock`, `C`, `cfat`, `gen`, `hamming`, `johnson`, `keller`, `MANN_a`, `p_hat`, `san` e `sanr`. Essas instâncias possuem quantidade de vértices variando entre 28 até 4 mil, e arestas variando entre 420 até 6 milhões.

O *NetworkRepository* possui diversas instâncias, todas separadas pelo tipo de rede, como por exemplo o grafo de redes sociais. Para este trabalho, foram selecionadas 29 instâncias do *NetworkRepository*, dividas em 7 grupos, são eles: `bio`, `ca`, `ia`, `inf`, `socfb`, `tech` e `web`. Cada uma representa um tipo de rede, que é melhor detalhada no site do *NetworkRepository*. As instâncias escolhidas possuem quantidade de vértices variando entre 1 mil até 33 mil, e quantidade de arestas variando aproximadamente de 1700 até 1,3 milhões.

Para a segunda etapa dos testes, cada algoritmo foi executado 100 vezes para as instâncias convencionais, com exceção do ACO Clássico sequencial que foi executada 20 vezes. Para as instâncias massivas, somente os algoritmos paralelos foram submetidos aos experimentos, em uma quantidade de 20 vezes. Para cada instância, foram aferidos os valores:

[1] **Disponíveis em:** `www.networkrepository.com/dimacs.php`

[2] **Disponíveis em:** `www.networkrepository.com`**, propostas em (ROSSI; AHMED, 2015)**

[3] **Disponível em:** `https://cse.unl.edu/~tnguyen/npbenchmarks/clique.html`**. Este *link* possui as soluções ótimas para o problema do Clique Máximo, porém, o complemento do grafo obtido no clique é a Cobertura Miníma de Vértices.**

- média do tempo de execução.
- média do valor de solução.
- desvio padrão do tempo de execução.
- desvio padrão do valor de solução.

Para as implementações paralelas, o *speed up* foi calculado com base nos tempos médios por grupo de instâncias para os seus respectivos algoritmos sequenciais.

É adotado nas figuras contendo os resultados em termos de qualidade de solução o termo solução objetivo. Esse termo é dado, pois para as instâncias da biblioteca DIMACS, é comprovado que as soluções objetivo são as soluções ótimas (XU; MA, 2006; GENG *et al.*, 2007). Porém, para as instâncias massivas, as soluções obtidas pelo CAI *et al.*, com o seu algoritmo *FastVC2*, não garantem que são soluções ótimas. Porém, os resultados obtidos pelo o CAI *et al.* é o que há de recente na literatura, para o problema em questão.

Os algoritmos foram implementados na linguagem C (gcc 7.3.0), com *flag* de compilação $-O2$. Os experimentos computacionais foram realizados em uma máquina que dispõe de um processador IntelR CoreTM i7-7700HQ(4), 2,80GHz - 3,80GHz, com 8 GB de memória RAM e o sistema operacional Ubuntu GNU/Linux 18.04.

7.1 OBTENDO MELHOR PARÂMETRO DE EXECUÇÃO DOS ALGORITMOS

A primeira etapa dos experimentos computacionais foi destinada para obter os parâmetros α, β, σ e a quantidade de formigas. Para tal, foi escolhida uma faixa de valores para cada um destes parâmetros, como descrito na Tabela 2. Os parâmetros α, β foram incrementados de 1, pois optou-se que esses parâmetros seriam variáveis inteiras. O σ foi incrementado de 0,1 e a quantidade de formigas, incrementado de 5.

Parâmetro	Início	Fim	Passo
α	1	3	1
β	1	3	1
σ	0,3	0,9	0,1
Formigas	20	40	5

Tabela 2: Parâmetros testados

No total, cada instância foi executada 315 vezes, e os melhores parâmetros de tempo e qualidade de solução da instância foram obtidos.

O algoritmo utilizado, para obter estes parâmetros, foi o ACO Elitista. A escolha deste algoritmo foi dada pelo o fato dele ser a versão do ACO mais rápida dentre os algoritmos propostos.

Os resultados obtidos nesta primeira etapa estão descritos na Tabela 3. Esta tabela é dividida em duas partes, que são referentes ao parâmetros obtidos para melhor tempo e para os parâmetros para melhor solução. Na primeira coluna esta presente o nome das instâncias e nas demais colunas os parâmetros α, β, σ e a quantidade de formigas. As instâncias de teste foram agrupadas de acordo com a identificação de seus grupos. Para os valores dos parâmetros, foram obtidos os valores médios por grupo.

Instância	**Qualidade de Tempo**				**Qualidade de Solução**			
	α	β	σ	**Forms.**	α	β	σ	**Forms**.
brock	1,25	1,08	0,63	20,00	1,08	1,67	0,58	25,00
C	1,29	1,14	0,54	20,00	1,00	2,29	0,53	27,86
c-fat	1,14	1,71	0,57	20,00	1,00	1,00	0,33	20,00
gen	1,20	1,40	0,66	20,00	2,00	1,40	0,44	26,00
hamming	1,33	1,83	0,62	20,00	1,00	1,17	0,45	21,67
johnson	1,25	1,50	0,55	20,00	1,00	1,00	0,33	20,00
keller	1,00	1,00	0,47	20,00	1,00	1,67	0,67	26,67
MANN	1,25	1,00	0,65	20,00	1,00	1,00	0,30	20,00
p_hat	1,13	1,20	0,70	20,00	1,07	1,33	0,45	25,67
san	1,00	1,00	0,71	20,00	1,00	1,09	0,35	22,73
sanr	1,25	1,25	0,48	20,00	1,00	1,75	0,40	27,50

Tabela 3: Média dos parâmetros para cada conjunto de instância

Após a obtenção dos parâmetros por grupo, foi obtida a média de todos os grupos para os parâmetros de qualidade de tempo e de qualidade de solução. Esses valores médios estão presentes na Tabela 4.

Descrição	α	β	σ	**Forms.**
Tempo	1,19	1,28	0,60	20
Solução	1,10	1,40	0,44	23,92
Média	**1,15**	**1,34**	**0,52**	**21,96**

Tabela 4: Média dos parâmetros para cada tipo de avaliação

Para obter o parâmetro geral, no qual todas as instâncias foram submetidas aos testes, foi aferido o valor médio entre as médias obtidas para os parâmetros de tempo e de qualidade de solução. Destes valores obtidos, o α e β são valores inteiros, portanto, seus valores médios foram truncados. Para a quantidade de formigas, o valor obtido foi arredondado. O σ, por ser um número decimal, foi mantido o valor encontrado.

7.2 EXPERIMENTOS DOS ALGORITMOS SEQUENCIAIS

Para cada algoritmo sequencial, foram aferidos os seus resultados em termos de qualidade de solução e tempo de execução. As soluções obtidas pelos os algoritmos foram confrontadas com as soluções ótimas para as instâncias do DIMACS.

As soluções obtidas do ACO Clássico Sequencial estão descritas na Figura 8.

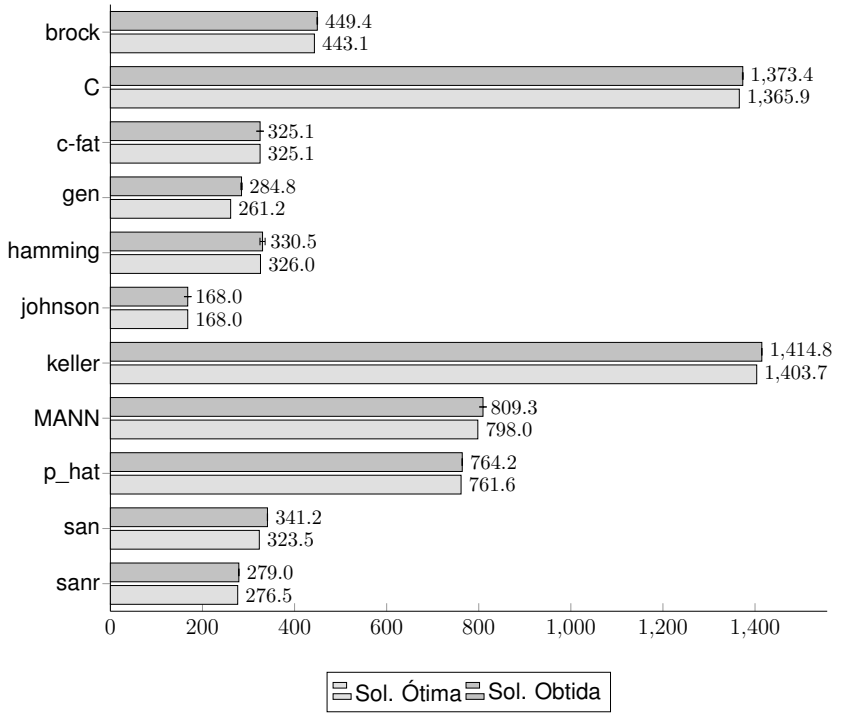

Figura 8: Gráfico comparativo das soluções obtidas (cinza escuro) em relação as soluções ótimas (cinza claro) para o ACO Clássico

As soluções obtidas do ACO Sem Denominador Sequencial estão descritas pela Figura 9.

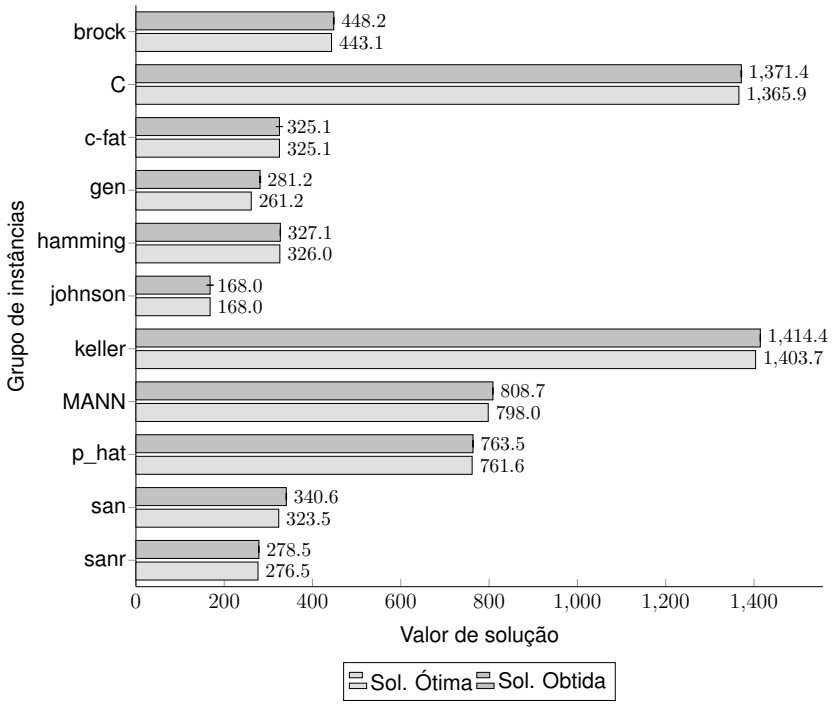

Figura 9: Gráfico comparativo das soluções obtidas (cinza escuro) em relação as soluções ótimas (cinza claro) para o ACO Sem Denominador Sequencial

As soluções obtidas pelo o ACO Elitista Sequencial estão descritos na Figura 10. Os testes do ACO Elite sequencial foi realizado para a quantidade de formigas elites igual a 20% de todas as formigas.

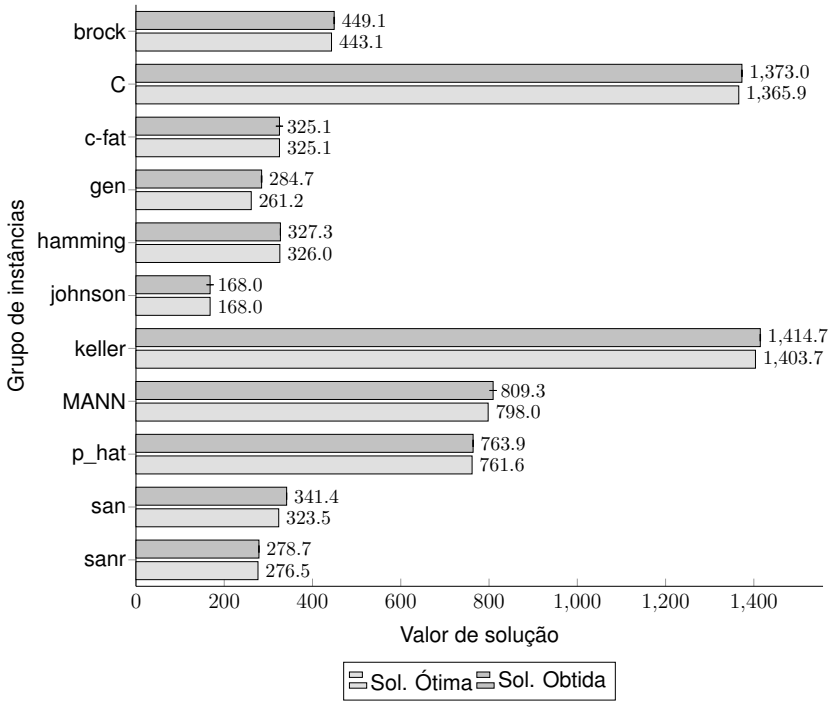

Figura 10: Gráfico comparativo das soluções obtidas (cinza escuro) em relação as soluções ótimas (cinza claro) para o ACO Elitista Sequencial

Observa-se que os três algoritmos propostos possuem valores médios de solução próximos das soluções ótimas e com desvios quase imperceptíveis, como descritos nas Figuras 8, 9 e 10. Por se tratar de métodos aproximados, é esperado que o Colônia de Formigas, quando não obtêm a solução ótima, possua valores de soluções próximos da solução ótima.

O baixo desvio padrão, presente nas soluções, é ocasionado devido a uma característica do algoritmo Colônia de Formigas, que é a estabilização de todas formigas em uma só solução. Essa estabilização ocorre devido ao maior acumulo de feromônio presente nos vértices que fazem parte da solução que todas as formigas encontraram.

A mudança da equação de probabilidades não ocasionou em mudanças perspectiveis na qualidade de solução, e o desvio padrão para as soluções obtidas pelo ACO Sem Denominador manteve o que foi obtido pelas demais implementações.

7.2.1 COMPARAÇÃO ENTRE AS IMPLEMENTAÇÕES SEQUENCIAIS

Os tempos obtidos pelos algoritmos sequenciais implementados estão descritos na Figura 11. A barra de cor cinza claro, é o tempo de execução médio do algoritmo ACO Clássico Sequencial. A barra de cor cinza intermediário (em relação aos três tons de cinza do gráfico), é o tempo de execução médio para o algoritmo ACO Sem Denominador Sequencial. E a barra de cor cinza escuro, é o tempo de execução média para o ACO Elistista Sequencial.

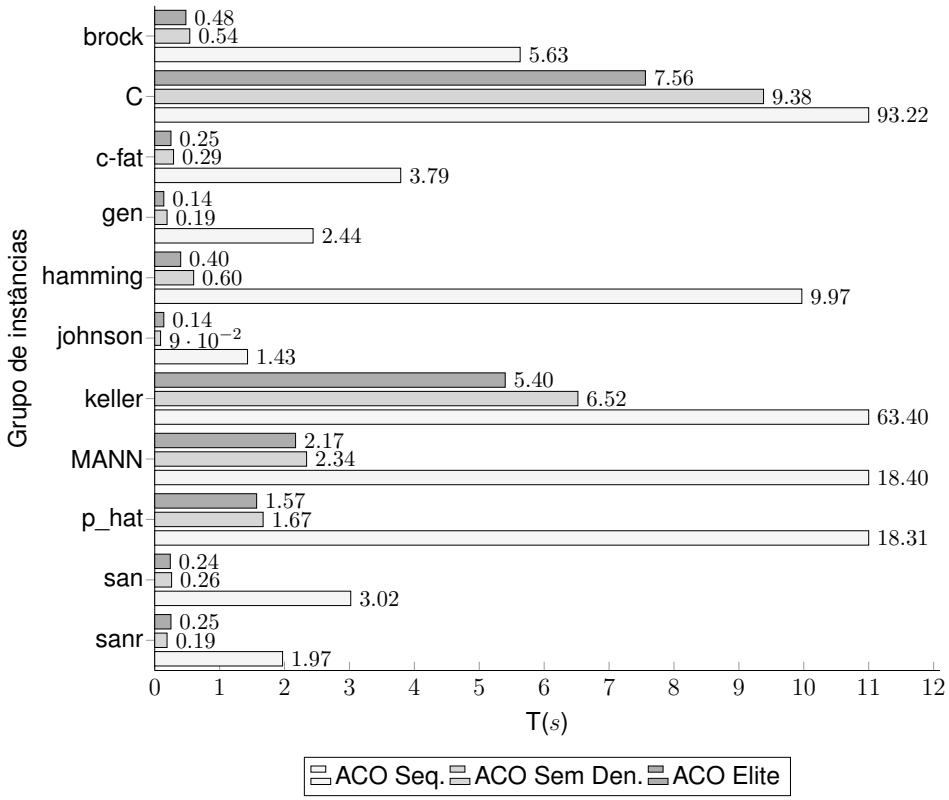

Figura 11: Gráfico comparativo entre os tempos de execução para as três implementações sequenciais.

Para as três implementações sequenciais do ACO, foram utilizadas as mesmas funções de cálculo de potência, e as mesmas funções para geração de número aleatório. Portanto, a diferença de tempo entre as implementações foi dado através da retirada do termo do denominador na equação de probabilidades, como descrito na Seção 6.3. A implementação do algoritmo ACO Sem Denominador é exatamente igual ao ACO Clássico, possuindo como diferença a retirada dos termos que faziam referência ao denominador da equação de probabilidade e a mudança da equação.

Os algoritmos ACO Sem Denominador e ACO Elitista possuem, em alguns casos como para a instância `C`, uma diferença de tempo maior, favorecendo o ACO Elitista. Porém, as vezes o ACO Sem Denominador pode possuir em média o tempo de execução menor, como no caso da instância `sanr`. É válido lembrar que o ganho do ACO Elitista é adquirido nas primeiras interações, pois é quando o feromônio está começando a ser adicionado nos vértices. Quando o feromônio está quase se estabilizando, as formigas possuem como tendência possuir o mesmo valor de solução, resultando que todas as formigas preenchem o grupo de soluções elite no mesmo ponto (somente as formigas de índices menores, nesse caso, conseguem entrar no conjunto de soluções elite).

7.3 EXPERIMENTOS PARA OS ALGORITMOS PARALELOS

Os experimentos computacionais para os algoritmos paralelos foram realizados para as instâncias massivas e para as instâncias convencionais. Ambas foram comparadas com as soluções de referencia (as soluções ótimas do DIMACS, e as soluções obtidas pelo CAI *et al.*).

Para ambos os algoritmos, os testes foram realizados executando-se 4 *threads*. A quantidade de *thread* foi dada pela quantidade de núcleos físicos que a máquina de teste possui. Os ganhos aferidos foram dados para o tempo execução total do algoritmo. Porém, cada execução do ACO pode possuir pontos de parada diferentes, ou seja, parar em diferentes interações.

Para os gráficos descritos a seguir, os grupos de instâncias (presentes no eixo y) entre os grupos `brock` e `sanr`, fazem partes das instâncias do DIMACS. As demais são grupo de instâncias do *NetworkRepository*.

As soluções obtidas pelo o ACO Sem Denominador Paralelo estão descritos na Figura 12. O algoritmo paralelo, como esperado, ele mantém sempre a mesma faixa de qualidade de solução, contendo baixos desvios e com as soluções se mantendo próxima da solução objetivo.

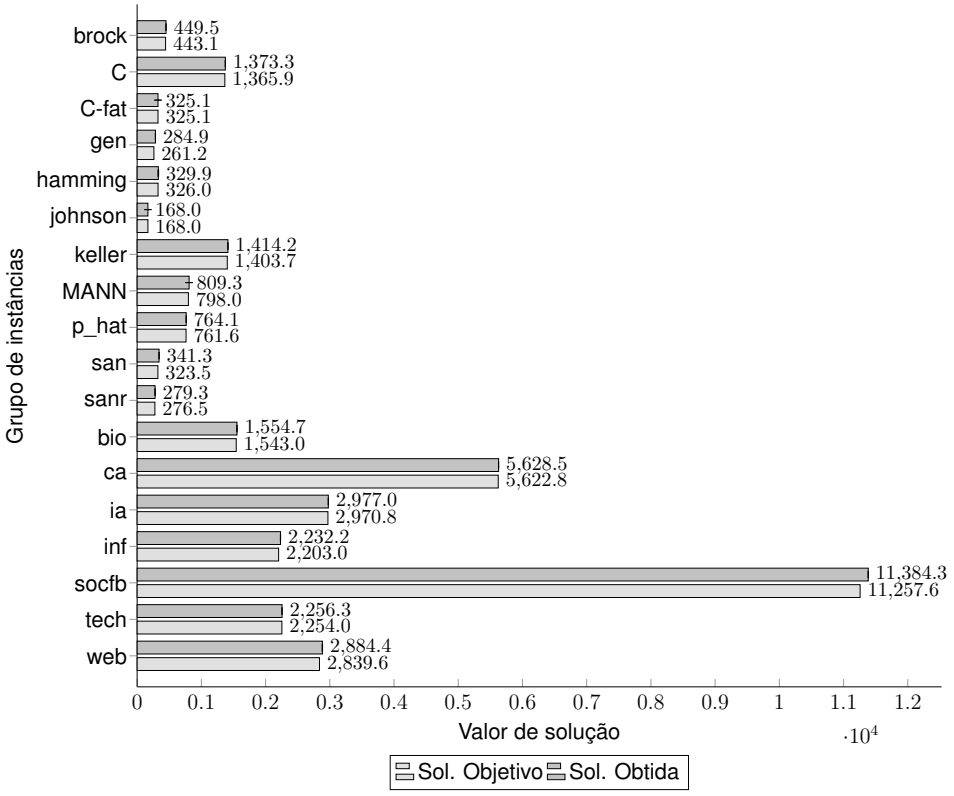

Figura 12: Gráfico comparativo das soluções obtidas (cinza escuro) em relação as soluções objetivos (cinza claro) para o ACO sem denominador paralelo

Na Figura 13 é mostrado um comparativo de tempo entre as implementações do ACO Sem Denominador sequencial (cinza claro) com o tempo do seu algoritmo paralelo. Como esperado, o algoritmo paralelo possui seu tempo de execução menor que o sequencial. Idealmente, o ganho de tempo total do algoritmo é 4. Entretanto, os comparativos de tempo foram feitos para os tempos totais de execução, não considerando o tempo de cada iteração do algoritmo. Em média, o ACO Sem Denominador Paralelo possui o *speed up* de 1,77. Para a comparação sequencial e paralelo, foi considerado apenas as instâncias do DIMACS, pois foram as instâncias testadas para os algoritmos sequenciais.

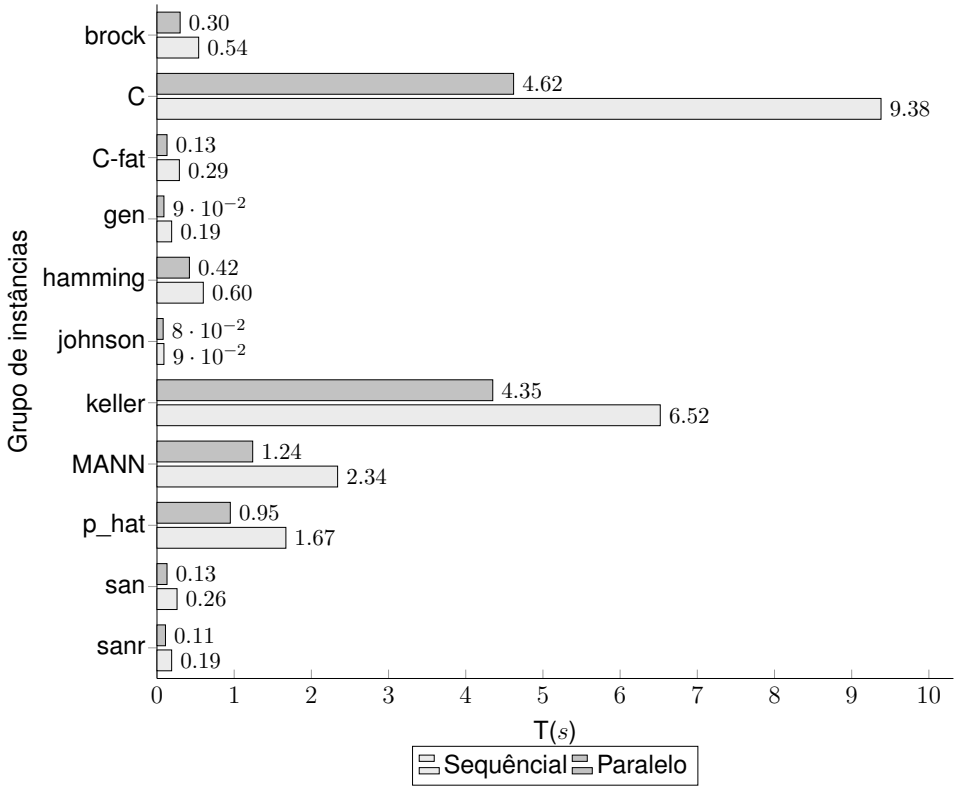

Figura 13: Gráfico comparativo dos tempos obtidos no ACO Sem Denominador sequencial (em cinza claro) com os tempos obtidos no ACO Sem Denominador paralelo (em cinza escuro)

As soluções obtidas pelo o ACO Elitista Paralelo estão descritos na Figura 14. A implementação paralela do ACO Elitista mantêm a qualidade de solução na mesma faixa que a sua versão sequencial, com baixos desvios das soluções obtidas. Os testes do ACO Elite sequencial foi realizado para a quantidade de formigas elites igual a 20% de todas as formigas.

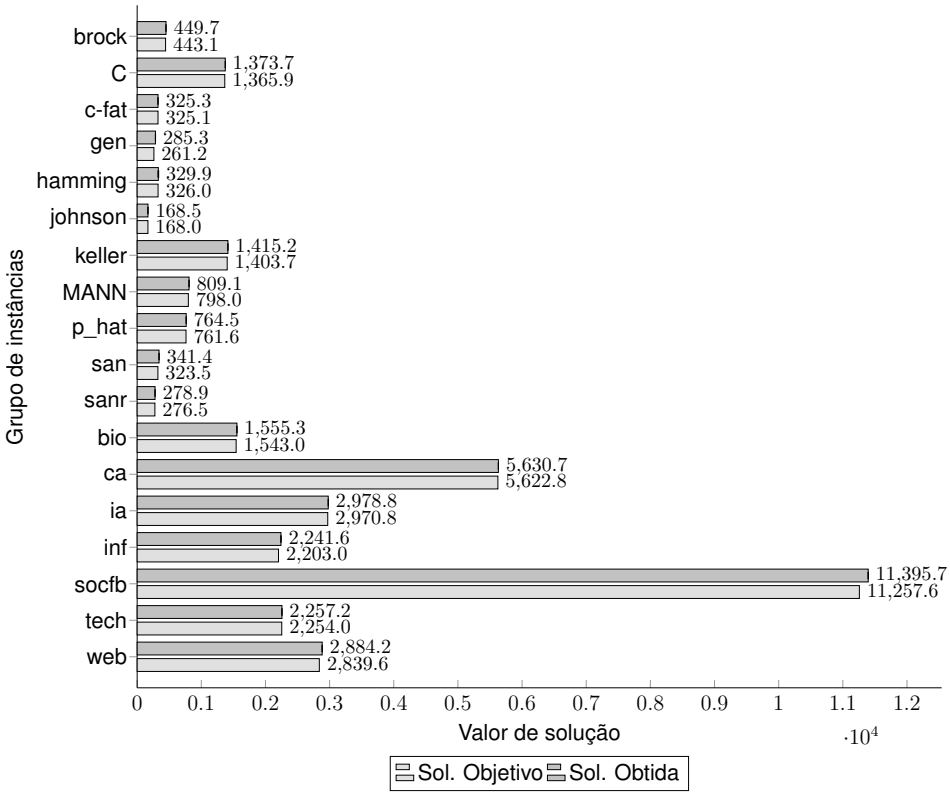

Figura 14: Gráfico comparativo das soluções obtidas (cinza escuro) em relação as soluções ótimas (cinza claro) para o ACO Elitista paralelo

O gráfico descrito pela Figura 15 mostra o comparativo entre as implementações sequenciais e paralelas do ACO Elitista. É de se esperar o ganho de tempo da implementação paralela. Os ganhos médios foram aferidos para os tempos totais de execução, não levando em conta os critérios de parada dos algoritmos. Em média, o ACO Elitista paralelo possui o *speed up* de 3,70. Para essas comparações, só foi considerado os testes para as instâncias do DIMACS.

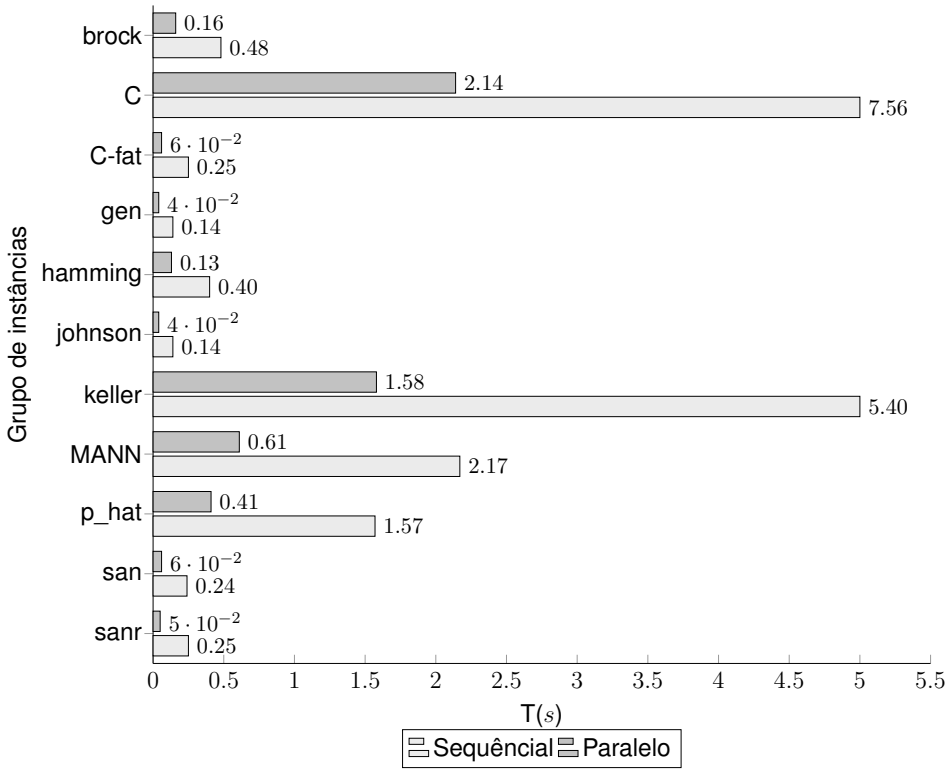

Figura 15: Gráfico comparativo dos tempos obtidos no ACO Elitista sequencial (em cinza claro) com os tempos obtidos no ACO Elitista paralelo (em cinza escuro)

7.3.1 COMPARAÇÃO ENTRE AS IMPLEMENTAÇÕES PARALELAS

O gráfico descrito pela Figura 16 faz comparação entre os tempos de execução dos dois algoritmos paralelos. Pelos resultados aferidos no gráfico, a execução paralela elitista apresenta melhor desempenho do que a implementação ACO Sem Denominador. Além disso, a qualidade de solução obtidas por ambos, se mantêm na mesma faixa de valores, e próximas das soluções ótimas.

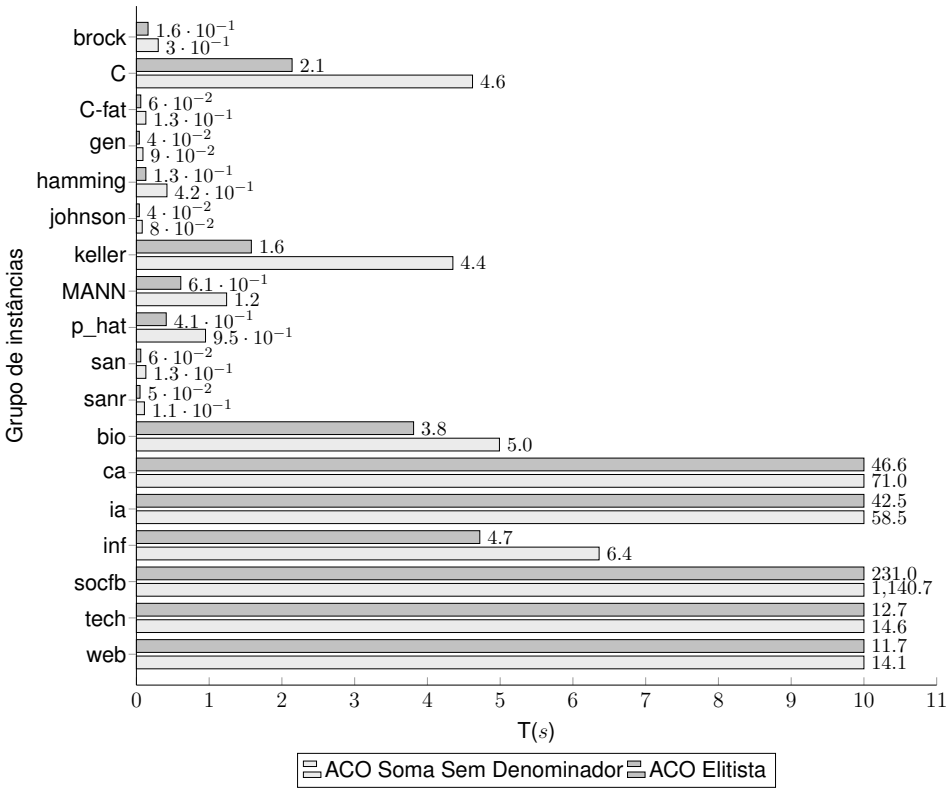

Figura 16: Gráfico comparativo dos tempos obtidos no ACO Sem Denominador paralelo (em cinza claro) com os tempos obtidos no ACO Elitista paralelo (em cinza escuro)

A diferença de tempo entre as duas implementações paralelas é dada, pois, para o ACO Sem Denominador, todas as formigas atualizam o feromônio, já o ACO Elistista, só um grupo de formigas faz o mesmo. O ganho do ACO Elitista é adquirido no início da execução do algoritmo, pois nesta etapa, o feromônio não está estabilizado, e portanto, existe uma diferença maior entre as soluções das formigas. Quando o feromônio está estabilizado, todas as formigas tendem a possuir o mesmo valor de solução, assim, todas as formigas que entram no conjunto das soluções elites, possuem o mesmo valor de solução.

8 CONCLUSÕES

O Problema da Cobertura Mínima de Vértices é um proeminente problema de otimização combinacional que possui importantes aplicações práticas, incluindo segurança de redes, roteamento de circuitos elétricos, aplicações em redes de sensores e monitoramento de falhas em pontos de rede. Portanto, a investigação de problema possui aplicações que podem ser utilizadas na prática.

Para resolver este problema, o presente trabalho optou por utilizar a metaheurística Colônia de Formigas, motivados pela a deficiência de implementações desta heurística na literatura e pela facilidade de paralelização da mesma.

A aplicação do paralelismo neste trabalho, trouxe ganhos significativos em termos de tempo de solução, sempre mantendo os mesmos valores de solução. Porém, a aplicação de técnicas paralelas para o Colônia de Formigas não é compensador para instâncias massivas, pois, mesmo o algoritmo ACO Elitista Paralelo, que possui o menor tempo de execução nas médias apresentadas, ele demora aproximadamente 15 minutos[1] para resolver uma instância com 29 mil vértices. Portanto, os algoritmos propostos ficam restritos à instâncias que possuem menos de 29 mil vértices.

[1] **Os dados mostrados são tempos médios para grupos de solução**

REFERÊNCIAS

ALVIM, Adriana C. F. **Estratégias de paralelização da meta-heurística GRASP**. Dissertação (Mestrado) — Pontifíca Universidade Católica do Rio de Janeiro, 4 1998.

AMINI, Amir A; WEYMOUTH, Terry E; JAIN, Ramesh C. Using dynamic programming for solving variational problems in vision. **IEEE Transactions on pattern analysis and machine intelligence**, IEEE, v. 12, n. 9, p. 855–867, 1990.

BARBOSA, Marco A; HOELSCHER, Igor; FAVARIM, Fábio; RIBEIRO, Richardson. Adaptando uma solução grasp ao problema da cobertura mínima de vértices. **Anais SULCOMP**, v. 6, 2013.

BELLMAN, Richard Ernest. **Dynamic Programming**. New York, NY, USA: Dover Publications, Inc., 2003. ISBN 0486428095.

BLUM, Christian; ROLI, Andrea. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 35, n. 3, p. 268–308, set. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm-.org/10.1145/937503.937505>.

CAI, Shaowei. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In: **Proceedings of the 24th International Conference on Artificial Intelligence**. [S.l.]: AAAI Press, 2015. (IJCAI'15), p. 747–753. ISBN 978-1-57735-738-4.

CAI, Shaowei; LIN, Jinkun; LUO, Chuan. Finding a small vertex cover in massive sparse graphs: Construct, local search, and preprocess. **J. Artif. Int. Res.**, AI Access Foundation, USA, v. 59, n. 1, p. 463–494, maio 2017. ISSN 1076-9757.

CAI, Shaowei; SU, Kaile; CHEN, Qingliang. Ewls: A new local search for minimum vertex cover. In: FOX, Maria; POOLE, David (Ed.). **AAAI**. [S.l.]: AAAI Press, 2010.

CECILIA, José M.; GARCíA, José M.; NISBET, Andy; AMOS, Martyn; UJALDóN, Manuel. Enhancing data parallelism for ant colony optimization on gpus. **Journal of Parallel and Distributed Computing**, v. 73, n. 1, p. 42 – 51, 2013. ISSN 0743-7315. Metaheuristics on GPUs. Disponível em: <http://www.sciencedirect.com/science/article-/pii/S0743731512000032>.

CHAPMAN, Barbara; JOST, Gabriele; PAS, Ruud van der. **Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)**. [S.l.]: The MIT Press, 2007. ISBN 0262533022, 9780262533027.

CHENG, John; GROSSMAN, Max; MCKERCHER, Ty. **Professional CUDA C Programming**. 1st. ed. Birmingham, UK, UK: Wrox Press Ltd., 2014. ISBN 1118739329, 9781118739327.

COOK, Stephen A. The complexity of theorem-proving procedures. In: **Proceedings of the Third Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1971. (STOC '71), p. 151–158. Disponível em: <http://doi.acm.org/10-.1145/800157.805047>.

CORMEN, Thomas H. **Algorithms Unlocked**. [S.l.]: The MIT Press, 2013. ISBN 0262518805, 9780262518802.

CORMEN, Thomas H.; STEIN, Clifford; RIVEST, Ronald L.; LEISERSON, Charles E. **Introduction to Algorithms**. 2nd. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0070131511.

DASGUPTA, Sanjoy; PAPADIMITRIOU, Christos H.; VAZIRANI, Umesh. **Algorithms**. 1. ed. New York, NY, USA: McGraw-Hill, Inc., 2008.

DEUTSCH, David. Quantum theory, the church–turing principle and the universal quantum computer. **Proc. R. Soc. Lond. A**, The Royal Society, v. 400, n. 1818, p. 97–117, 1985.

DIVERIO, Tiarajú Asmuz; TOSCANI, Laira Vieira; VELOSO, Paulo AS. Análise da complexidade de algoritmos paralelos. 2002.

DORIGO, Marco; BLUM, Christian. Ant colony optimization theory: A survey. **Theoretical Computer Science**, v. 344, n. 2, p. 243 – 278, 2005. ISSN 0304-3975.

DORIGO, Marco; GAMBARDELLA, Luca Maria. Ant colonies for the travelling salesman problem. **Biosystems**, v. 43, n. 2, p. 73 – 81, 1997. ISSN 0303-2647.

DORIGO, Marco; GAMBARDELLA, Luca Maria. Ant colony system: a cooperative learning approach to the traveling salesman problem. **IEEE Transactions on evolutionary computation**, IEEE, v. 1, n. 1, p. 53–66, 1997.

DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, v. 26, n. 1, p. 29–41, Feb 1996.

EVANS, Isaac K. Evolutionary algorithms for vertex cover. In: ____. **Evolutionary Programming VII: 7th International Conference, EP98 San Diego, California, USA, March 25–27, 1998 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 377–386.

FEO, Thomas A.; RESENDE, Mauricio G. C. Greedy randomized adaptive search procedures. **Journal of Global Optimization**, v. 6, n. 2, p. 109–133, Mar 1995. ISSN 1573-2916.

FLYNN, Michael J. Some computer organizations and their effectiveness. **IEEE transactions on computers**, IEEE, v. 100, n. 9, p. 948–960, 1972.

GAO, Hai-Hua; YANG, Hui-Hua; WANG, Xing-Yu. Ant colony optimization based network intrusion feature selection and detection. In: **2005 International Conference on Machine Learning and Cybernetics**. [S.l.: s.n.], 2005. v. 6, p. 3871–3875 Vol. 6. ISSN 2160-133X.

GAREY, Michael R.; JOHNSON, David S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447.

GENDREAU, Michel; HERTZ, Alain; LAPORTE, Gilbert. A tabu search heuristic for the vehicle routing problem. **Management science**, INFORMS, v. 40, n. 10, p. 1276–1290, 1994.

GENDREAU, Michel; POTVIN, Jean-Yves. **Handbook of Metaheuristics**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441916636, 9781441916631.

GENG, Xiutang; XU, Jin; XIAO, Jianhua; PAN, Linqiang. A simple simulated annealing algorithm for the maximum clique problem. **Information Sciences**, Elsevier, v. 177, n. 22, p. 5064–5071, 2007.

GILMOUR, Stephen; DRAS, Mark. Kernelization as heuristic structure for the vertex cover problem. In: DORIGO, Marco; GAMBARDELLA, Luca Maria; BIRATTARI, Mauro; MARTINOLI, Alcherio; POLI, Riccardo; STÜTZLE, Thomas (Ed.). **Ant Colony Optimization and Swarm Intelligence**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 452–459. ISBN 978-3-540-38483-0.

GOODRICH, M.T.; TAMASSIA, R. **Projeto de algoritmos: Fundamentos, análise e exemplos da internet**. [S.l.]: BOOKMAN COMPANHIA ED, 2004.

HOWARD, Ronald A. Dynamic programming. **Management Science**, v. 12, n. 5, p. 317–348, 1966. Disponível em: <https://doi.org/10.1287/mnsc.12.5.317>.

JAIN, Raj; CHIU, Dah-Ming; HAWE, William R. **A quantitative measure of fairness and discrimination for resource allocation in shared computer system**. [S.l.]: Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

KARABOGA, Dervis; BASTURK, Bahriye. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. **Journal of global optimization**, Springer, v. 39, n. 3, p. 459–471, 2007.

KARAKOSTAS, George. A better approximation ratio for the vertex cover problem. **ACM Trans. Algorithms**, ACM, New York, NY, USA, v. 5, n. 4, p. 41:1–41:8, nov. 2009. ISSN 1549-6325.

KARP, R. Reducibility among combinatorial problems. In: MILLER, R.; THATCHER, J. (Ed.). **Complexity of Computer Computations**. [S.l.]: Plenum Press, 1972. p. 85–103.

KARP, Richard; ZHANG, Yanjun. A randomized parallel branch-and-bound procedure. In: ACM. **Proceedings of the twentieth annual ACM symposium on Theory of computing**. [S.l.], 1988. p. 290–300.

KAVALCI, Vedat; URAL, Aybars; DAGDEVIREN, Orhan. Distributed vertex cover algorithms for wireless sensor networks. **International Journal of Computer Networks and Communications**, v. 6, 2014.

KIRKPATRICK, Scott; GELATT, C Daniel; VECCHI, Mario P. Optimization by simulated annealing. **science**, American Association for the Advancement of Science, v. 220, n. 4598, p. 671–680, 1983.

LAWLER, E. L.; WOOD, D. E. Branch-and-bound methods: A survey. **Operations Research**, v. 14, n. 4, p. 699–719, 1966.

LEE, J. W.; CHOI, B. S.; LEE, J. J. Energy-efficient coverage of wireless sensor networks using ant colony optimization with three types of pheromones. **IEEE Transactions on Industrial Informatics**, v. 7, n. 3, p. 419–427, Aug 2011. ISSN 1551-3203.

LYNCH, Clifford. Big data: How do your data grow? **Nature**, Nature Publishing Group, v. 455, n. 7209, p. 28, 2008.

MICHALEWICZ, Zbigniew. **How to Solve It: Modern Heuristics 2e**. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 3642061346, 9783642061349.

MIN, Hokey; KO, Hyun Jeung; KO, Chang Seong. A genetic algorithm approach to developing the multi-echelon reverse logistics network for product returns. **Omega**, Elsevier, v. 34, n. 1, p. 56–69, 2006.

MORAIS, Wall Berg; ROSA, Marcelo; TEIXEIRA, Marcelo; BARBOSA, Marco A. O problema do caixeiro viajante com limite de calado: uma abordagem usando simulated annealing. **XLIX - Simpósio Brasileiro de Pesquisa Operacional**, p. 2029–2040, Agosto 2017.

NAVAUX, P.O.A.; ROSE, C.A.F. **ARQUITETURAS PARALELAS**. BOOKMAN COMPANHIA ED, 2008. ISBN 9788577803095. Disponível em: <https://books.google.com.br-/books?id=eOrZPgAACAAJ>.

NOBRE, Ricardo Holanda. **Paralelismo como solução para redução de complexidade de problemas combinatoriais**. Dissertação (Mestrado) — Universidade Estadual do Ceará, 2011.

REEVES, Colin R. A genetic algorithm for flowshop sequencing. **Computers & operations research**, Elsevier, v. 22, n. 1, p. 5–13, 1995.

ROOSTA, Seyed H. Parallel processing and parallel algorithms - theory and computation. In: **SIGACT News**. [S.l.: s.n.], 2000.

ROSSI, Ryan A.; AHMED, Nesreen K. The network data repository with interactive graph analytics and visualization. In: **Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence**. [s.n.], 2015. Disponível em: <http:/-/networkrepository.com>.

SALEHIPOUR, Amir; SÖRENSEN, Kenneth; GOOS, Peter; BRÄYSY, Olli. Efficient grasp+vnd and grasp+vns metaheuristics for the traveling repairman problem. **4OR**, v. 9, n. 2, p. 189–209, Jun 2011. ISSN 1614-2411. Disponível em: <https://doi.org/10-.1007/s10288-011-0153-0>.

SHI, Yuhui; EBERHART, Russell. A modified particle swarm optimizer. In: IEEE. **Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on**. [S.l.], 1998. p. 69–73.

SHYU, Shyong Jian; YIN, Peng-Yeng; LIN, Bertrand M.T. An ant colony optimization algorithm for the minimum weight vertex cover problem. **Annals of Operations Research**, v. 131, n. 1, p. 283–304, Oct 2004. ISSN 1572-9338. Disponível em: <https://doi.org/10.1023/B:ANOR.0000039523.95673.33>.

SOLOMON, Marius M. Algorithms for the vehicle routing and scheduling problems with time window constraints. **Operations research**, Informs, v. 35, n. 2, p. 254–265, 1987.

TALBI, El-Ghazali. **Metaheuristics: From Design to Implementation**. [S.l.]: Wiley Publishing, 2009. ISBN 0470278587, 9780470278581.

TOSCANI, L.V.; VELOSO, P.A.S. **COMPLEXIDADE DE ALGORITMOS**. BOOKMAN COMPANHIA ED, 2001. ISBN 9788577803507. Disponível em: <https://books.google.com.br/books?id=SPVfPgAACAAJ>.

TURING, Alan Mathison. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London mathematical society**, Wiley Online Library, v. 2, n. 1, p. 230–265, 1937.

UCHIDA, A.; ITO, Y.; NAKANO, K. An efficient gpu implementation of ant colony optimization for the traveling salesman problem. In: **2012 Third International Conference on Networking and Computing**. [S.l.: s.n.], 2012. p. 94–102.

XU, Xinshun; MA, Jun. An efficient simulated annealing algorithm for the minimum vertex cover problem. **Neurocomputing**, v. 69, n. 7, p. 913 – 916, 2006. ISSN 0925-2312. New Issues in Neurocomputing: 13th European Symposium on Artificial Neural Networks. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0925231205003565>.

ZIVIANI, N. **Projeto de algoritmos: com implementações em Pascal e C**. Pioneira Thomson Learning, 2004. ISBN 9788522103904. Disponível em: <https://books.google.com.br/books?id=fQi_AAAACAAJ>.