FELIPE DE LIMA MESQUITA
JHOSER ALLAF DOS SANTOS MATHEUS

ALGORITMO FPT PARA ALIANÇAS DEFENSIVAS EM GRAFOS

(*versão pré-defesa, compilada em 6 de fevereiro de 2025*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: André Luiz Pires Guedes.

CURITIBA PR

2024

RESUMO

Este trabalho explora o problema de encontrar alianças defensivas em grafos, um tema relevante na teoria dos grafos com aplicações em diversas áreas, desde análise de mercado, redes sociais e biologia. Definimos uma aliança defensiva como um subconjunto de vértices onde cada vértice possui pelo menos tantos vizinhos dentro do conjunto quanto fora dele, de forma a ter sempre mais "aliados"do que "inimigos". O texto se concentra na formulação do problema, na análise da complexidade computacional e na implementação de um algoritmo eficiente para a identificação dessas alianças.

Apresentamos um algoritmo FPT semelhante a uma busca em profundidade, que explora sistematicamente os vértices do grafo para encontrar alianças defensivas de tamanho específico, juntamente com duas novas melhorias que apresentam melhora significativa no desempenho, e um visualizador web desenvolvido que permite a visualização passo a passo do processo de busca.

Palavras-chave: Algoritmo FPT. Alianças Defensivas. Grafos

SUMÁRIO

1 INTRODUÇÃO

O estudo das alianças em grafos é um campo fascinante que combina conceitos de teoria dos grafos e complexidade computacional. As alianças, especialmente as alianças defensivas, são subconjuntos de vértices que garantem uma forma de defesa mútua entre seus membros; conceito este que pode ser aplicado desde alianças para suporte mútuo entre nações em guerra (Kristiansen et al., 2004) até análise da estrutura secundária do RNA (Haynes et al., 2006).

Neste estudo, abordamos o problema de encontrar alianças defensivas em grafos, que pode ser formalizado como a identificação de subconjuntos de vértices que satisfazem condições específicas de conectividade e vizinhança. Um grafo $G = (V, E)$ é composto por um conjunto de vértices V e um conjunto de arestas E. Uma aliança defensiva é definida como um subconjunto $S \subseteq V$ tal que, para cada vértice $v \in S$, o número de vizinhos de v dentro de S é pelo menos igual ao número de vizinhos de v fora de S. Essa propriedade assegura que cada vértice na aliança possui mais aliados do que potenciais inimigos, promovendo assim a segurança do grupo.

A complexidade computacional associada à identificação de alianças defensivas é desanimadora, e fazemos dela o ponto central deste estudo. Através da perspectiva atenuante do algoritmo FPT proposto por (Enciso, 2009), buscamos entender a eficiência e a viabilidade de encontrar tais alianças em grafos de diferentes tamanhos e estruturas.

Além disso, apresentamos um visualizador web que ilustra o funcionamento do algoritmo, permitindo uma compreensão mais intuitiva dos passos envolvidos na busca por alianças defensivas, propomos duas melhorias para a eficiência do algoritmo e disponibilizamos em um repositório o algoritmo final e otimizado em *Python*. A seguir, detalharemos a fundamentação teórica necessária para a compreensão do tema, além de descrever a metodologia utilizada e os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

A fim de trabalharmos com o conceito de alianças defensivas, primeiro é importante lembrar alguns conceitos fundamentais de teoria dos grafos:

- **Grafo:** Representado como $G = (V, E)$, é composto por um conjunto de vértices V e arestas E que conectam pares de vértices.
- **Vizinhança:** Para um vértice v, a vizinhança $N(v)$ é o conjunto de vértices adjacentes a v. O grau de v é $|N(v)|$.
- **Subgrafo:** Um grafo F é subgrafo de G se $V(F) \subseteq V(G)$ e $E(F) \subseteq E(G)$.
- **Conectividade:** Um grafo é conexo se existe um caminho entre quaisquer dois vértices.

Agora, uma **aliança defensiva** é um subconjunto $S \subseteq V$ se, para cada vértice $v \in S$, o número de vizinhos de v dentro de S é pelo menos igual ao número de vizinhos de v fora de S, ou seja:

$$|N(v) \cap S| \geq |N(v) \setminus S| \tag{2.1}$$

Essa definição reflete a ideia de que os vértices de S formam uma estrutura em que cada um de seus vértices estará protegido por si mesmo e pelos seus vizinhos aliados em caso de "ataque"de seus vizinhos não-aliados.

Embora uma aliança defensiva não precise ser conexa, neste trabalho consideramos apenas alianças defensivas **conexas**. Mas vale destacar que, caso uma aliança defensiva seja desconexa, cada componente desconexa da aliança defensiva deverá, por definição, ser uma aliança defensiva.

2.1 CLASSE FPT(*FIXED-PARAMETER TRACTABLE*)

Como encontrar uma aliança defensiva em um grafo é um problema NP-completo, torna-se necessário o uso de diferentes técnicas para torná-lo tratável. Neste trabalho utilizamos o FPT como meio de tornar o problema controlado e conseguir encontrar soluções viáveis para o problema enfrentado.

FPT é, na verdade, uma **classe de complexidade** que aborda problemas parametrizados, em geral de decisão, onde o problema pode ser reformulado em função de um parâmetro específico denotado por k e um polinômio de n. O objetivo é separar a dependência de k da complexidade geral do problema, permitindo que o problema seja tratado eficientemente para valores pequenos de k, mesmo que o tamanho da entrada n seja grande.

Dizer que "um algoritmo está em FPT", portanto, é dizer que o problema que este algoritmo resolve está na classe FPT, e que ele trata esse problema isolando a variável k.

A complexidade de um algoritmo FPT é expressa na forma:

$$O(f(k) \cdot p(n)) \tag{2.2}$$

onde

- $f(k)$ é uma função que depende apenas do parâmetro k; ela pode crescer exponencialmente ou superpolinomialmente em relação a k, mas é independente do tamanho total da entrada n, e
- $p(n)$ é uma função polinomial no tamanho da entrada n.

Essa abordagem é útil em situações em que k pode ser fixado ou mantido pequeno, tornando o problema tratável em cenários em que abordagens tradicionais seriam inviáveis devido ao crescimento exponencial em n.

2.1.1 Aplicação ao problema

No caso do problema de alianças defensivas, k representa o tamanho da aliança que estamos buscando. Embora o problema geral de encontrar alianças defensivas seja NP-completo, parametrizá-lo por k permite projetar algoritmos relativamente eficientes para instâncias com valores pequenos de k, mesmo que o grafo G tenha um grande número de vértices.

O trecho a seguir da função $DefensiveAlliance$ aponta onde o parâmetro k é isolado:

```
1 DefensiveAlliance(G, S, k)
2 ...
3 Se v.c_w <= k - tamanho de S:
4 ...
```

Enquanto k limita a profundidade da busca, a expressão $v.c_w \leq k -$ tamanho de S limita a largura da busca, de acordo com o lema 14 (Enciso, 2009) que será enunciado mais à diante neste texto, se faz necessário visitar somente $\lfloor N[v]/2 + 1 \rfloor$ para garantir que um vértice pode ser defendido, ou não.

Dessa forma, o algoritmo FPT utilizado neste estudo foi proposto por (Enciso, 2009) e tem complexidade:

$$O(k^k \cdot n) \tag{2.3}$$

Essa é uma complexidade razoável em comparação com outros algoritmos publicados que resolvem o mesmo problema, mas ainda assim a função k^k cresce rapidamente com o aumento de k, como observado na tabela abaixo:

Como pode ser visto, mesmo valores moderados de k resultam em um crescimento muito rápido na complexidade. Assim, a eficiência do algoritmo depende diretamente de manter k pequeno.

k	k^k
2	4
3	27
4	256
5	3.125
6	46.656
7	823.543
9	387.420.489
10	10.000.000.000

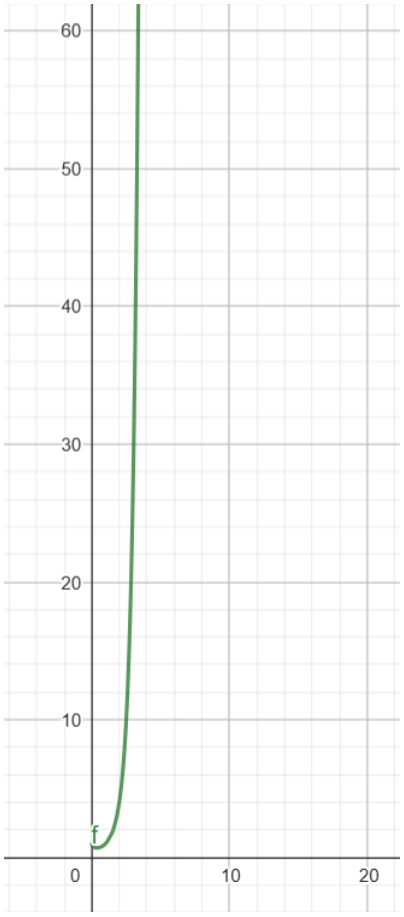

Figura 2.1: Gráfico da função k^k

2.2 O ALGORITMO

2.2.1 Propósito do algoritmo

O objetivo do algoritmo é garantir que, ao parametrizar o problema, a busca pela solução seja limitada a subconjuntos de tamanho k, reduzindo significativamente o espaço de busca comparado a uma abordagem que analisaria todos os subconjuntos possíveis. Além disso, ele explora propriedades estruturais dos grafos para otimizar a execução e evitar cálculos redundantes, tornando o problema mais acessível em cenários reais.

2.2.2 Explicação

O algoritmo é dividido em duas funções principais, a `main` e a `defensiveAlliance`. A `main` recebe como entrada um Grafo G e o tamanho da aliança desejada, um inteiro positivo k. De forma intuitiva, a abordagem do algoritmo é partir de um vértice do grafo por vez e olhar sua vizinhança numa tentativa de expandi-lo até formar uma aliança defensiva de tamanho k, ou todos os vértices terem servido de raiz da expansão.

```
Main(G,k)
      init_cw(G)
      Para cada vértice v de G:
            inicia uma aliança S <- {v}.
            aliança_encontrada <- DefensiveAlliance(S).
            Se aliança_encontrada:
                  retorne aliança_encontrada.
            retira v de S

      retorne "Sem aliança";
```

O papel da função `main` é garantir que todos os vértices foram usados como raiz da expansão. Para isso, primeiro é chamada a função auxiliar `init_cw` inicializa o estado de `c_w` de todos os vértices do Grafo `G`.

```
init_cw(G)
      Para cada vértice v de G:
            v.c_w <- teto(número de vizinhos de n / 2) - número de vizinhos de n fora
```

A variável `c_w` serve para verificar se o vértice está protegido dentro da aliança `S`, e isso faz parte da condição de sucesso da busca. Ou seja, quando todos os vértices de `S` estiverem protegidos (c_w <= 0), então uma aliança defensiva foi formada.

A seguir, a `main` chama a função `defensiveAlliance` para verificar se `S` é, ou pode ser expandida até, uma aliança defensiva de tamanho `k`.

```
DefensiveAlliance(G, S, k)
      inicia v <- vértice de maior c_w em S.
      Se v.c_w <= 0 e tamanho de S == k:
            devolve S.

      Se v.c_w <= k - tamanho de S:
            inicia t <- 1 + metade dos vizinhos de w.
            inicia o conjunto W <- t vizinhos de w que não pertencem a S.
            Para cada vértice w em W:
                  S <- S + w.
                  Para cada vizinho x de w em S:
                        subtrai 1 do c_w de x e de w.
                  aliança_encontrada <- DefensiveAlliance(G, S, k)
                  Se aliança_encontrada:
```

```
15                          Devolve aliança_encontrada.
16                    Para cada vizinho x de w em S:
17                          soma 1 ao c_w de x e de w.
18                    Retira w de S.
19
20       Retorne NULL;
```

No início de `defensiveAlliance` o algoritmo escolhe o vértice de maior `c_w` em `S`, que, em outras palavras, que seria o vértice mais vulnerável da aliança. Esse vértice serve para tanto verificar se `S` se tornou uma aliança quanto como ponto de expansão a fim de incluir novos vértices para defender conjunto `S` nesse nível da recursão.

A seguir, o algoritmo verifica se há espaço em `S` para os `c_w` vizinhos necessários serem adicionados, ou seja, para que `w` seja defendido dentro da restrição do tamanho máximo `k`. Essa verificação funciona de modo semelhante a uma heurística de busca, poupando tempo ao evitar vértices que não podem ser defendidos posteriormente. Por conta disso, pelo lema 14 de (Enciso, 2009) a seguir, vértices em G com $N[v] > 2k + 1$ nunca serão considerados na busca.

2.2.3 Lema 14

Assuma que $S \subseteq V$ é estendível para uma aliança defensiva S', onde $|S| < |S'| = k$. Então, para qualquer vértice desprotegido $w \in S$, $|S' \cap (N[w] - S|) \geq c_w$.
Em outras palavras se S é estendível e w é um vértice desprotegido de S então c_w é o número de vizinhos de w fora de S que é necessário para proteger w em S.

Esse lema é o que podemos considerar como o núcleo do algoritmo, pois ele nos garante também que, qualquer subconjunto $W \subseteq N[w] - S$ com $t = \lfloor \frac{d_w}{2} \rfloor + 1$ vértices contém ao menos um vértice w_i para o qual $S \cup w_i$ é estendível se e somente se S é estendível.

Esse lema é importante, pois a partir dele sabemos que precisamos verificar somente metade mais um dos vizinhos de w para encontrar um vizinho w_i que o protege em S se S é estendível para uma aliança defensiva ou para descartar w nessa árvore da recursão.

2.2.4 Evitando repetir conjuntos

Observando o comportamento do algoritmo no visualizador web, foi possível notar que um comportamento pouco eficiente: o critério de expansão de S (destacado a seguir) abre margem pra repetir várias vezes a mesma combinação de vértices, levando, principalmente em grafos de grande quantidade de vértices, a muito esforço improdutivo.

```
1  DefensiveAlliance(G, S, k)
2        inicia v <- vértice de maior c_w em S.
```

Pensando nisso, a equipe elaborou uma solução que armazena todas as combinações já analisadas anteriormente e impede de que novas iterações com elas sejam geradas, cortando toda

a sub-árvore subsequente. Isso é feito com a criação de um dicionário e a marcação única de cada combinação:

```
inicia combinacoes <- dicionário vazio
```

```
DefensiveAlliance(G, S, k)
[...]
      Para cada vértice w em W:
            S <- S + w.

            comb_id <- identificadores de S de forma ordenada.
            Se existe combinacoes[comb_id]:
                  Retira w de S.
                  pula para o próximo vértice.
            Caso contrário:
                  cria combinacoes[comb_id].

            Para cada vizinho x de w em S:
                  subtrai 1 do c_w de x e de w.
            aliança_encontrada <- DefensiveAlliance(G, S, k)
[...]
```

Caso não exista uma entrada da combinação no dicionário, cria-se uma e a instância corrente de `S` é analisada. Caso contrário, a instância é ignorada, podando todas as sub-árvores subsequentes. A complexidade de tempo é se resume a ordenação de, no máximo, $k-1$ elementos, e ao acesso e escrita no dicionário. Ambos são ofuscados pela complexidade geral.

Por outro lado, há um custo sério em termos de espaço. No pior caso, de não haver aliança e o algoritmo analisar todos os vértices e $k = n$, a combinação ocupa espaço $O(2^n)$, que corresponde a guardar todas as combinações de n vértices, variando de tamanho 1 até n. Isso pode ser mitigado ao limitar o tamanho das combinações armazenadas para a região crítica que vai ser repetida mais vezes. A análise desta região está na seção sobre Resultados e discussão.

Quanto ao desempenho, esta técnica permite ao algoritmo poupar muito tempo ao "amortizar"o custo k^k ao longo de várias iterações, pois, como nenhuma combinação é repetida, quanto mais exploradas são as combinações dos vértices, menos combinações existem para serem analisadas.

2.2.5 Priorização dos vértices expostos

```
DefensiveAlliance(G, S, k)
      inicia v <- vértice de maior c_w em S.
```

Ao iniciarmos $DefensiveAlliance(G, S, k)$ atribuindo v o vértice em W com maior c_w nós garantimos que a maior prioridade em cada chamada recursiva da função é proteger o vértice mais "exposto".

Assim obtemos também um critério de parada consistente, ou seja, quando o vértice com maior c_w ter $c_w \leq 0$ e $|S| = k$, teremos duas informações fundamentais sobre o contexto da execução:

1 - Se $c_w \leq 0$, então todos os vértices em S estão protegidos.

2 - Se $|S| = k$, encontramos a aliança defensiva procurada.

3 METODOLOGIA

O projeto é composto por duas partes principais: algoritmos de busca implementados em Python e um visualizador web criado para exibir os passos deste algoritmo, ambos disponíveis no repositório do Apêndice B. As partes funcionam de forma independente, sendo conectadas apenas pelo formato de entrada e saída dos programas.

3.1 ALGORITMO EM PYTHON

A implementação do algoritmo proposto por (Enciso, 2009) foi feita em Python e consta completa no Apêndice 1.

Nesta implementação foi utilizada a estrutura de dados da biblioteca *Networkx* para manipulação dos grafos, e, no mais, estruturada de forma semelhante ao algoritmo teórico, com a exceção do uso de uma estrutura de pilha para substituir a chamada recursiva.

Além do resultado final, o programa possibilita o retorno da aliança em formato JSON, com as características utilizadas no visualizador web. Essas características permitem a visualização passo a passo dos nós expandidos pelo algoritmo, e consistem do conjunto W a cada iteração da função `DefensiveAlliance`.

Há também *flags* para mostrar a quantidade de nós expandidos, para gerar um grafo aleatório segundo uma densidade especificada, e para alterar o comportamento da busca, como retornar a primeira aliança encontrada independente do tamanho.

3.2 VISUALIZADOR WEB

O projeto web foi desenvolvido com Typescript e React, e sua proposta é fornecer uma visualização passo-a-passo do algoritmo e do grafo de entrada. Assim como o algoritmo de busca, o visualizador pode ser encontrado no repositório que se encontra nas referências.

Para montar a visualização é necessário que seja fornecido como entrada um grafo disposto em formato JSON, contando com dois conjuntos extras de dados que são a aliança encontrada, caso exista, e um vetor de *steps*, que contém o conjunto W no dado *passo* da iteração.

Munido destas informações, o visualizador organiza os dados internamente para melhorar o desempenho e a decisão de cada caraterística visual do grafo e então personaliza uma *view* HTML, dada pela biblioteca 3d-force-graph (Hassan-Shafique, 2004), que cuida da renderização e simulação física do grafo.

Dentre as funcionalidades, vale destacar duas principais: a visualização do conjunto S a cada etapa do algoritmo e o mapa de calor dos nós explorados. O mapa de calor é a coloração dos vértices do grafo seguindo a regra de que, quanto mais visitado um vértice durante a execução do

algoritmo, mais quente é a cor usada; os nós mais quentes tem cores próximas do vermelho, e os mais frios, mais próximas do azul.

3.2.1 Visualização por passos

Ao carregar um grafo com o conjunto de passos, é possível navegar por cada um deles. Em um certo passo, os vértices e arestas da fronteira de S são desenhados com uma cor cinza escura, os além da fronteira tem cor cinza claro, e os vértices dentro de S ficam coloridos com a cor

- azul, se estiverem desprotegidos;
- ou verde, se estiverem devidamente protegidos.

Ao finalizar o algoritmo e desenhar a aliança, ela é colorida de verde escuro.

As figuras a seguir são de uma busca por uma aliança de tamanho $k = 15$ em um grafo de $v = 50$ vértices e $e = 40$ arestas. O algoritmo começa com S contendo o vértice colorido de azul.

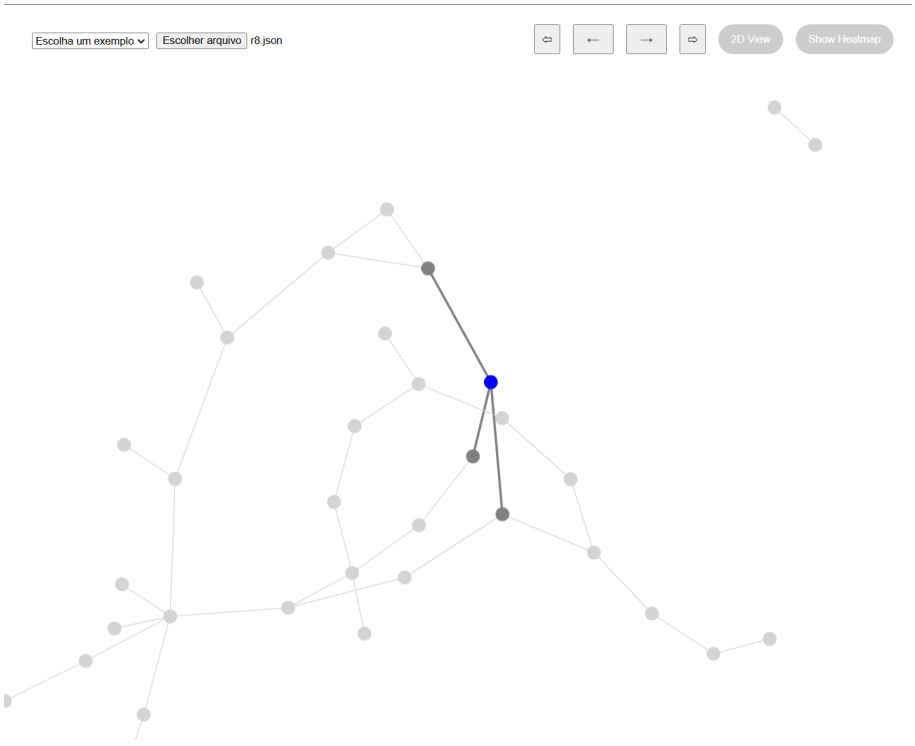

Figura 3.1: Passo 1 do algoritmo

3.2.2 Mapa de calor

É possível também ativar a visualização do mapa de calor, que colore os vértices de acordo com a quantidade de vezes que ele foi explorado pelo algoritmo.

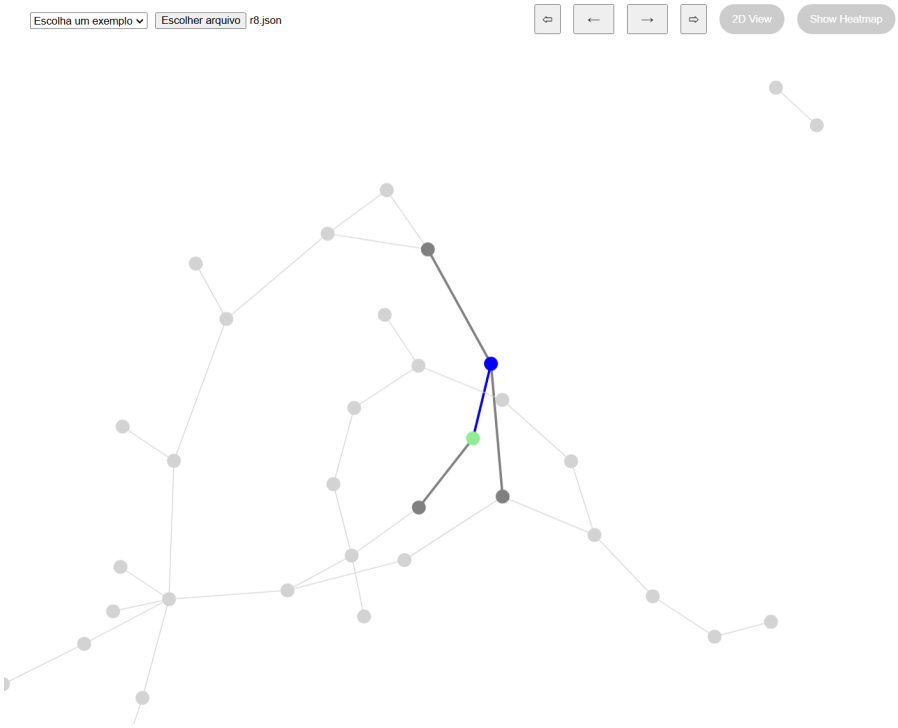

Figura 3.2: Passo 2 do algoritmo

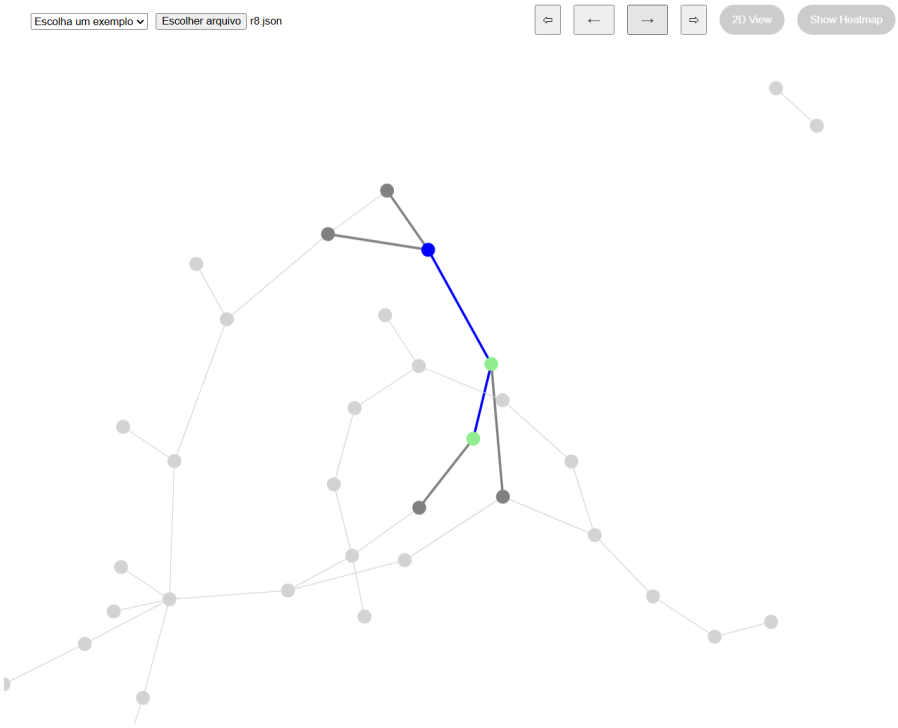

Figura 3.3: Passo 3 do algoritmo

Para ajudar na visualização, as cores mais frias também tem opacidade mais baixa.

Essa ferramenta em particular incentivou questionamentos interessantes a respeito da eficiência do algoritmo, como "como evitar a alta taxa de repetição de um grupo de vértices".

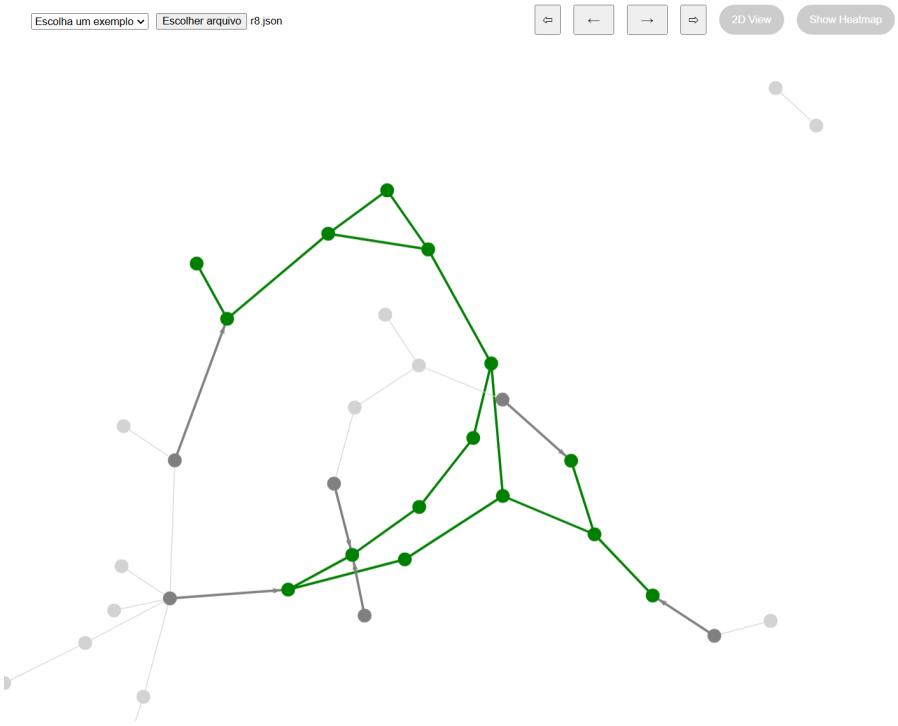

Figura 3.4: Passo final do algoritmo

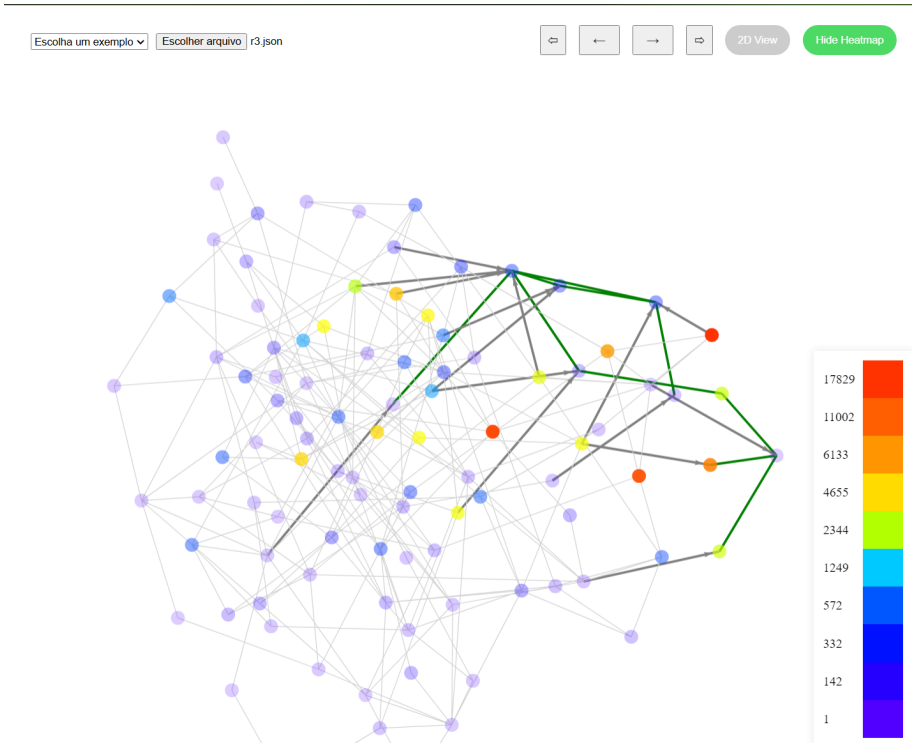

Figura 3.5: Mapa de calor do grafo

4 RESULTADOS E DISCUSSÃO

O algoritmo originalmente estudado e a versão com as melhorias propostas foram analisadas e submetidas a um conjunto de testes para melhor ilustrar o impacto e eficiência de cada uma. Visto que o problema continua sendo NP-completo, há pouco a ser feito para valores realmente grandes, mas foi possível sim observar uma ampliação dos valores considerados "razoáveis"pelo algoritmo FPT.

Para testar o algoritmo utilizamos a função da biblioteca `networkx` `nx.erdos_renyi_graph(v,e,seed)` onde `v` é o número de vértices `e` é a probabilidade de 2 vértices formarem uma aresta, e `seed` é uma semente para geração do Grafo.

Para os testes a seguir foram fixados os seguintes seguintes parâmetros `v=30` `e=0.333` e `seed=100` e executamos para `k` variando entre `1` e `29`.

Figura 4.1: Execução do algoritmo **evitando** a repetição de conjuntos

Na execução do algoritmo sem repetição de conjuntos é possível notar que o número máximo de nós explorados foi de aproximadamente 1 milhão de nós.

Por sua vez, na execução que permite a repetição de conjuntos o número de nós explorados nesse caso saltou de 1 milhão para 100 milhões.

Salvar os conjuntos resulta em uma melhora significativa na redução do número de nós a serem explorados, porém nós traz um novo problema pois o espaço necessário para armazenar todos esses conjuntos no pior caso é $\sum_{i=0}^{k} \binom{n}{i} < 2^n$, ou seja, acabamos trocando um tempo exponencial, por espaço exponencial.

Foi observado um padrão interessante na eficiência com relação ao grau médio dos vértices do grafo $d(G)$ e k; o número de nós explorados atinge um ápice para valores de k próximos de $d(G)$ criando uma "zona difícil", e suaviza a medida que a diferença aumenta.

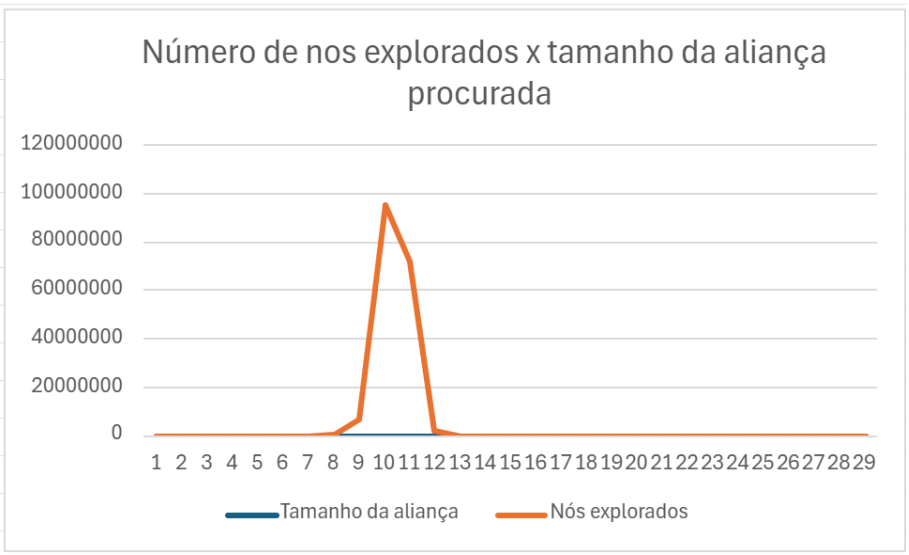

Figura 4.2: Execução do algoritmo **permitindo** a repetição de conjuntos

Para valores de $d(G)$ muito maiores que k isso acontece porque o algoritmo pode descartar muitas combinações através do critério `Se v.c_w <= k - tamanho de S:`. Essa linha garante que o próximo nó a ser expandido ao menos tem as condições de ser protegido dado o tamanho atual de S.

Por outro lado, valores de k muito menores do que $d(G)$, foi observado uma probabilidade maior de haver uma aliança defensiva. A modificação de ordenação dos vértices de `W` com base em quantos vizinhos ele possui em S e $\lfloor d(v)/2 \rfloor$, em especial, mostrou acelerar muito o processo de determinação da aliança, quando existente. Isso se deve as escolhas priorizarem a defesa dos vértices já em S, em prol de adições aleatórias.

Por fim, a modificação de "Evitar repetir conjuntos"mostrou-se acelerar o processo tanto no melhor caso quanto no pior, pois garante que somente novas combinações são testadas.

5 CONCLUSÃO

O estudo como um todo foi bastante produtivo dentro do tema, e possibilitou compreensão significativa do que são e como encontrar alianças defensivas. O visualizador web, como ferramenta didática, foi bastante aproveitado para a compreensão e elaboração das melhorias propostas ao algoritmo.

Também foi produtivo experimentar na prática a complexidade de um problema NP-completo e uma das ferramentas usadas para contornar esse degrau gigantesco na complexidade.

Dentre os diversos temas que podem ser abordados em discussões futuras, destacamos a implementação e análise do algoritmo proposto por (Enciso, 2009) para encontrar Conjuntos Seguros (*Secure Sets*), que segue uma abordagem FPT semelhante ao de alianças defensivas, e pode ser adaptado para o visualizador web para gerar resultados valiosos.

Outro ponto de possível expansão é o de pré-análise de grafos para a determinação de potencial de uma aliança de tamanho k, partindo da análise feita sobre o grau médio e a "zona difícil".

REFERÊNCIAS

Enciso, R. (2009). *Alliances In Graphs: Parameterized Algorithms And On Partitioning Series-parallel Graphs*. Tese de doutorado, University of Central Florida.

Hassan-Shafique, K. (2004). *Partitioning A Graph In Alliances And Its Application To Data Clustering*. Tese de doutorado, University of Central Florida.

Haynes, T., Knisley, D., Seier, E. e Zou, Y. (2006). A quantitative analysis of secondary rna structure using domination based parameters on trees. *BMC Bioinformatics*, 7(1):108.

Kristiansen, P., Hedetniemi, S. M. e Hedetniemi, S. T. (2004). Alliances in graphs. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 48:157–178.

APÊNDICE A – PSEUDO CÓDIGOS

A.1 ALGORITMO DEFENSIVEALLIANCE

```
Main(G,k)
  init_cw(G)
  Para cada vértice v de G:
    inicia uma aliança S <- {v}.
    aliança_encontrada <- DefensiveAlliance(S).
    Se aliança_encontrada:
      retorne aliança_encontrada.
    retira v de S

  retorne "Sem aliança";

init_cw(G)
  Para cada vértice v de G:
    v.c_w <- teto(número de vizinhos de n / 2) - número de vizinhos de n

DefensiveAlliance(G, S, k)
  inicia v <- vértice de maior c_w em S.
  Se v.c_w <= 0 e tamanho de S == k:
    devolve S.

  Se v.c_w <= k - tamanho de S:
    inicia t <- 1 + metade dos vizinhos de w.
    inicia o conjunto W <- t vizinhos de w que não pertencem a S.
    Para cada vértice w em W:
      S <- S + w.
      Para cada vizinho x de w em S:
        subtrai 1 do c_w de x e de w.
      aliança_encontrada <- DefensiveAlliance(G, S, k)
      Se aliança_encontrada:
        Devolve aliança_encontrada.
      Para cada vizinho x de w em S:
        soma 1 ao c_w de x e de w.
      Retira w de S.
```

```
34
35  Retorne NULL;
```

A.2 MODIFICAÇÃO PARA EVITAR REPETIR CONJUNTOS

```
1 inicia combinacoes <- dicionário vazio
```

```
1  DefensiveAlliance(G, S, k)
2  [...]
3    Para cada vértice w em W:
4      S <- S + w.
5
6      comb_id <- identificadores de S de forma ordenada.
7      Se existe combinacoes[comb_id]:
8        Retira w de S.
9        pula para o próximo vértice.
10     Caso contrário:
11       cria combinacoes[comb_id].
12
13     Para cada vizinho x de w em S:
14       subtrai 1 do c_w de x e de w.
15     aliança_encontrada <- DefensiveAlliance(G, S, k)
16 [...]
```

A.3 MODIFICAÇÃO PARA PRIORIZAÇÃO DOS VÉRTICES EXPOSTOS

```
1 DefensiveAlliance(G, S, k)
2   inicia v <- vértice de maior c_w em S.
```

APÊNDICE B – REPOSITÓRIO

O código-fonte e demais arquivos utilizados neste trabalho estão disponíveis no seguinte repositório:

`https://github.com/lipemesq/tcc2024-felipe-jhoser`