Análise de Grafos de Coocorrência de Termos em Textos Científicos

Lincoln Gondin; Gustavo Henrique

Junho 2025

Sumário

1 Descrição da Extração de Termos

Para a base de textos científicos foi utilizado os resumos dos textos encontrados no site *arxiv.org*. O arXiv é um serviço de distribuição gratuita e aberta para mais de 2,4 milhões de artigos científicos, em uma série de campos de estudo.

Para a extração foi feito um script em Python para o *scraping* dos resumos de vários textos científicos em todas as categorias disponíveis no site, sendo elas: Ciência da Computação, Economia, Engenharia Elétrica e Ciência de Sistemas, Matemática, Física, Biologia Quantitativa, Finança Quantitativa e Estatística.

Foram extraídos 1466 resumos dessas diferentes categorias e salvos em arquivos de texto separados para posterior processamento.

2 Construção do Grafo de Coocorrência

O grafo foi construido seguindo uma implementação direta do pseucodigo que podemos visualizar abaixo. Cada termo da base de texto se traduziu em um vertice e se os termos coocorrem na mesma janela ao menos uma vez uma aresta unica é criada entre eles. Foi utilizado grafos não direcionados para as análises.

Algorithm 1 Geração do Grafo de Coocorrência

function GerarGrafoCoocorrencia(representacao, textos, textosOriginais, n, w)
    $Voc \leftarrow$ ColetarNTermosMaisFrequentes(n, textos)
    $mapa\_indices \leftarrow$ mapa de cada termo em Voc para seu índice
    if representacao é "lista" **then**
        $grafo \leftarrow$ GrafoLista(Voc)
    else
        $grafo \leftarrow$ GrafoMatriz(Voc)
    end if
    for cada texto T em $textosOriginais$ **do**
        for $i \leftarrow 0$ até $|T| - w$ **do**
            $janela \leftarrow T[i : i + w]$
            $termos \leftarrow$ termos da janela que estão em $mapa\_indices$
            $pares \leftarrow$ todas as combinações únicas de dois termos em $termos$
            for cada par (t_a, t_b) em $pares$ **do**
                $u \leftarrow mapa\_indices[t_a]$
                $v \leftarrow mapa\_indices[t_b]$
                if não grafo.arestaExiste(u, v) **then**
                    grafo.adicionarAresta(u, v)
                end if
            end for
        end for
    end for
    return $grafo$
end function

3 Representações de Grafo

Foi utilizada uma abordagem OOP para a criação das duas representações dos grafos, dessa maneira ambas possuiam a mesma interface de interação.

Por conta dessa escolha tecnica não foi necessario, por exemplo criar duas funções diferentes dos algoritmos de detecção de componentes conexas. É possivel visualizar na figura abaixo a classe base que a representação em lista e em matriz herdam. Cada representação terá sua própria implementação para cada função.

```python
#classe basse grafo que sera usada como modelo pelos grafos em lista e matriz de adjacencia
class Grafo:
    def __init__(self):
        pass
    def adicionarAresta(self, i: int, j: int):
        pass
    def getArestas(self, vertice: int) -> list[int]:
        return []
    def arestaExiste(self, vertice: int, verticeQueConecta: int) -> bool:
        return False
    def getTermo(self, vertice: int) -> str:
        return ""
    def getQuantidadeDeVertices(self) -> int:
        return 0
```

Figura 1: Classe Grafo base.

3.1 Lista de Adjacência

A representação em lista foi feita como se é esperado; cada vertice armazena uma lista de todos os seus vertices vizinhos, representado-os por inteiros, para a consulta do termo ao qual o vertice se refere foi utilizado uma lista externa onde o i-esimo elemento representa o i-esimo termo extraido da base de texto, o mesmo foi feito na representação em matriz.

```python
class GrafoLista(Grafo):
    def __init__(self, termos: list[str]):
        # Inicializa o grafo como uma lista de adjacência
        (parameter) self: SelfGrafoLista mo, e a lista em cada posição representa seus vizinhos
        self.vertices: list[list[int]] = [[] for _ in range(len(termos))]
        self.termos = termos.copy()  # guarda uma cópia dos termos associados aos vértices
        self.n = len(termos)  # número total de vértices
    def adicionarAresta(self, i: int, j: int):
        # Adiciona uma aresta não-direcionada entre os vértices i e j
        # Garante que a conexão só seja feita uma vez (sem duplicatas)
        if j not in self.vertices[i]:
            self.vertices[i].append(j)
        if i not in self.vertices[j]:
            self.vertices[j].append(i)
    def getArestas(self, vertice: int) -> list[int]:
        # Retorna a lista de vizinhos (arestas) do vértice informado
        return self.vertices[vertice]
    def arestaExiste(self, vertice: int, verticeQueConecta: int) -> bool:
        # Verifica se há uma aresta entre os dois vértices
        return verticeQueConecta in self.vertices[vertice]
    def getTermo(self, vertice: int) -> str:
        # Retorna o termo (palavra) associado ao vértice
        return self.termos[vertice]
    def getQuantidadeDeVertices(self) -> int:
        # Retorna o número total de vértices no grafo
        return self.n
```

Figura 2: Implementação do grafo usando lista de adjacência.

3.2 Matriz de Adjacência

Na representação em matriz foi utilizado uma lista bidimensional n por n, onde o elemento Aij é 1 caso haja uma aresta de i para j.

```python
class GrafoMatriz(Grafo):
    def __init__(self, termos: list[str]):
        # Inicializa o grafo como uma matriz de adjacência (n x n)
        # Cada posição (i, j) da matriz indica se existe uma aresta entre os vértices i e j (1 existe 0 não existe)
        self.matriz = [[0 for _ in range(len(termos))] for _ in range(len(termos))]
        self.termos = termos.copy()  # armazena os termos associados aos vértices
        self.n = len(termos)  # número total de vértices
    def adicionarAresta(self, i: int, j: int):
        # Adiciona uma aresta não-direcionada entre os vértices i e j
        # Atualiza as duas posições da matriz para representar a conexão
        self.matriz[i][j] = 1
        self.matriz[j][i] = 1
    def getArestas(self, vertice: int) -> list[int]:
        # Retorna a lista de vértices conectados ao vértice informado
        # Faz isso verificando quais posições na linha correspondente da matriz possuem valor 1
        return [idx for idx, valor in enumerate(self.matriz[vertice]) if valor == 1]
    def arestaExiste(self, vertice: int, verticeQueConecta: int) -> bool:
        # Verifica se há uma aresta entre os dois vértices na matriz
        return self.matriz[vertice][verticeQueConecta] == 1
    def getTermo(self, vertice: int) -> str:
        # Retorna o termo (palavra) associado ao vértice
        return self.termos[vertice]
    def getQuantidadeDeVertices(self) -> int:
        # Retorna o número total de vértices no grafo
        return self.n
```

Figura 3: Implementação do grafo usando matriz de adjacência.

4 Algoritmos para Detecção de Componentes Conectados

4.1 Força Bruta (Grafo Não Direcionado)

O algoritmo de força bruta para detecção de componentes conexas em grafos não direcionados utiliza uma busca em profundidade (DFS) iniciada a partir de cada vértice não visitado. A cada nova DFS, é identificada uma nova componente conexa, agrupando todos os vértices acessíveis a partir daquele ponto inicial. O processo se repete até que todos os vértices tenham sido visitados. É uma abordagem simples e eficaz para grafos não direcionados e densos.

Algorithm 2 Algoritmo de Força Bruta para Componentes Conexas

```
1: function ForcaBrutaComponentes(grafo)
2:     visitado ← vetor de tamanho n inicializado com FALSO
3:     componentes ← lista vazia
4:     for v ← 0 até n − 1 do
5:         if não visitado[v] then
6:             componente ← lista vazia
7:             DFS(v, componente)
8:             Adicionar componente à lista componentes
9:         end if
10:    end for
11:    return componentes
12: end function
13: function DFS(v, componente)
14:    visitado[v] ← VERDADEIRO
15:    Adicionar v à lista componente
16:    for cada vizinho em grafo.getArestas(v) do
17:        if não visitado[vizinho] then
18:            DFS(vizinho, componente)
19:        end if
20:    end for
21: end function
```

4.2 Algoritmo Adaptado de Tarjan (Grafo Não Direcionado)

A versão original do algoritmo de Tarjan foi desenvolvida para encontrar componentes fortemente conexas em grafos direcionados. No entanto, ao lidar com grafos não direcionados, o uso direto da versão clássica de Tarjan pode causar problemas, como a identificação errada de conexões ou loops desnecessários. Por isso, uma adaptação foi realizada: ao invés da abordagem recursiva baseada em índices e low-link, utiliza-se uma DFS iterativa com pilha, que mantém o rastreamento de vértices visitados para formar as componentes conexas de forma eficiente e compatível com grafos não direcionados.

Algorithm 3 Versão Iterativa do Algoritmo de Tarjan para Componentes Conexas

```
1: function TarjanAdaptado(grafo)
2:     visitado ← vetor de tamanho n inicializado com FALSO
3:     componentes ← lista vazia
4:     for v ← 0 até n − 1 do
5:         if não visitado[v] then
6:             componente ← DFS(v)
7:             Adicionar componente à lista componentes
8:         end if
9:     end for
10:    return componentes
11: end function
12: function DFS(v_inicial)
13:    pilha ← lista contendo v_inicial
14:    componente ← lista vazia
15:    visitado[v_inicial] ← VERDADEIRO
16:    while pilha não está vazia do
17:        v ← pilha.pop()
18:        Adicionar v à lista componente
19:        for cada vizinho em grafo.getArestas(v) do
20:            if não visitado[vizinho] then
21:                visitado[vizinho] ← VERDADEIRO
22:                pilha.push(vizinho)
23:            end if
24:        end for
25:    end while
26:    return componente
27: end function
```

5 Procedimento de Medição de Tempo

Para avaliar o desempenho dos algoritmos de detecção de componentes conexas implementados neste trabalho, foi desenvolvido um sistema de medição de tempo de execução. O objetivo é comparar a eficiência das abordagens (Força Bruta e Tarjan adaptado), bem como o impacto da representação do grafo (lista ou matriz de adjacência) sobre o tempo de execução.

5.1 Função de Medição de Tempo Individual

A função `medir_tempo_algoritmo` executa um determinado algoritmo sobre um grafo, repetidas vezes, e retorna o tempo médio de execução em milissegundos. Ela recebe como parâmetros:

- **nome_algoritmo**: nome do algoritmo a ser testado (apenas para fins de exibição);
- **representacao**: tipo de estrutura utilizada (lista ou matriz);

- **funcao**: referência à função do algoritmo que será executado;
- **grafo**: o grafo sobre o qual o algoritmo será executado;
- **quantidade_execucoes**: número de vezes que a função será chamada para cálculo da média.

O tempo total é medido com a função `time.perf_counter()`, que fornece alta resolução para medições de desempenho.

5.2 Função de Medição Global

A função `medirDesempenho` organiza os testes para três tamanhos diferentes de grafo, com 500, 1000 e 2000 termos mais frequentes extraídos dos textos. Para cada tamanho, são criadas representações dos grafos em lista e matriz de adjacência. Em seguida, são realizados testes para os dois algoritmos (Força Bruta e Tarjan adaptado) sobre ambas as representações.

- Para cada combinação algoritmo/representação, o tempo médio de execução é calculado;
- Os resultados são armazenados em um dicionário para posterior geração de gráficos comparativos;
- A quantidade de execuções padrão utilizada foi de 1.000.000 repetições por teste.

Este processo garante que as variações causadas por pequenas flutuações de desempenho da máquina não interfiram significativamente nos resultados.

5.3 Pseudocódigo das Funções de Medição

Algorithm 4 Função medir_tempo_algoritmo

function MEDIR_TEMPO_ALGORITMO(nome_algoritmo, representacao, funcao, grafo, quantidade)
    $inicio \leftarrow$ tempo atual
    for $i \leftarrow 1$ até quantidade **do**
        Executar `funcao(grafo)`
    end for
    $fim \leftarrow$ tempo atual
    $tempo\_medio \leftarrow \dfrac{(fim - inicio)}{quantidade} \times 1000$
    Imprimir: "A combinação nome/representação tomou tempo_medio ms"
    return *tempo_medio*
end function

Algorithm 5 Função medirDesempenho

```
1: function MEDIRDESEMPENHO(quantidadeExecucoes)
2:     dadosParaGrafico ← {500 : [], 1000 : [], 2000 : []}
3:     nTermos ← [500, 1000, 2000]
4:     for cada n em nTermos do
5:         (grafoLista, grafoMatriz) ← INICIALIZARGRAFOS(n)
6:         Imprimir: "Para um conjunto de n termos:"
7:         MEDIR_TEMPO_ALGORITMO("Força Bruta", "Lista", algoritmoForcaBruta, grafoLista, quantidadeExecucoes)
8:         MEDIR_TEMPO_ALGORITMO("Força Bruta", "Matriz", algoritmoForcaBruta, grafoMatriz, quantidadeExecucoes)
9:         MEDIR_TEMPO_ALGORITMO("Tarjan", "Lista", algoritmoTarjan, grafoLista, quantidadeExecucoes)
10:        MEDIR_TEMPO_ALGORITMO("Tarjan", "Matriz", algoritmoTarjan, grafoMatriz, quantidadeExecucoes)
11:    end for
12: end function
```

5.4 Saída Exemplar do Programa

```
Para um conjunto de 500 termos:
    A combinação Força Bruta/Lista tomou um tempo médio de: 0.1.957590 milissegundos
    A combinação Força Bruta/Matriz tomou um tempo médio de: 16.077170 milissegundos
    A combinação Tarjan/Lista tomou um tempo médio de: 1.889870 milissegundos
    A combinação Tarjan/Matriz tomou um tempo médio de: 15.304440 milissegundos

    Para um conjunto de 1000 termos:
    A combinação Força Bruta/Lista tomou um tempo médio de: 5.075190 milissegundos
    A combinação Força Bruta/Matriz tomou um tempo médio de: 62.492520 milissegundos
    A combinação Tarjan/Lista tomou um tempo médio de: 3.932200 milissegundos
    A combinação Tarjan/Matriz tomou um tempo médio de: 61.619660 milissegundos

    Para um conjunto de 2000 termos:
    A combinação Força Bruta/Lista tomou um tempo médio de: 15.895280 milissegundos
    A combinação Força Bruta/Matriz tomou um tempo médio de: 248.319690 milissegundos
    A combinação Tarjan/Lista tomou um tempo médio de:  6.8856901 milissegundos
    A combinação Tarjan/Matriz tomou um tempo médio de: 243.304470 milissegundos
```

Esses resultados são utilizados na próxima seção para construir gráficos de comparação entre as abordagens.

6 Resultados Experimentais

Uma vez que os tempos médios não variam muito, e o tempo necessario para executar cada combinação 1000000 de vezes seria extremamente alto e desnecessário, podendo se extender

durante varias horas e dias, fizemos para cada combinação, 10; 100; 1000 e 10000 execuções dos algoritmos de componentes conexas,e depois toma-mos os tempos médios obtidos. Os valores podem ser visualizados na tabela e grafico abaixo.

6.1 Tabelas

Tabela 1: Tempos médios para 10, 100, 1000 e 10000 execuções.

Tamanho do Vocabulário (n)	**Força Bruta Lista (ms)**	**Força Bruta Matriz (ms)**	**Tarjan Lista (ms)**	**Tarjan Matriz (ms)**
500	1.950880	16.078558	1.838019	15.446336
1000	4.359374	63.382609	3.931208	61.406054
2000	8.985045	249.268669	6.822162	243.171462

6.2 Gráfico Comparativo

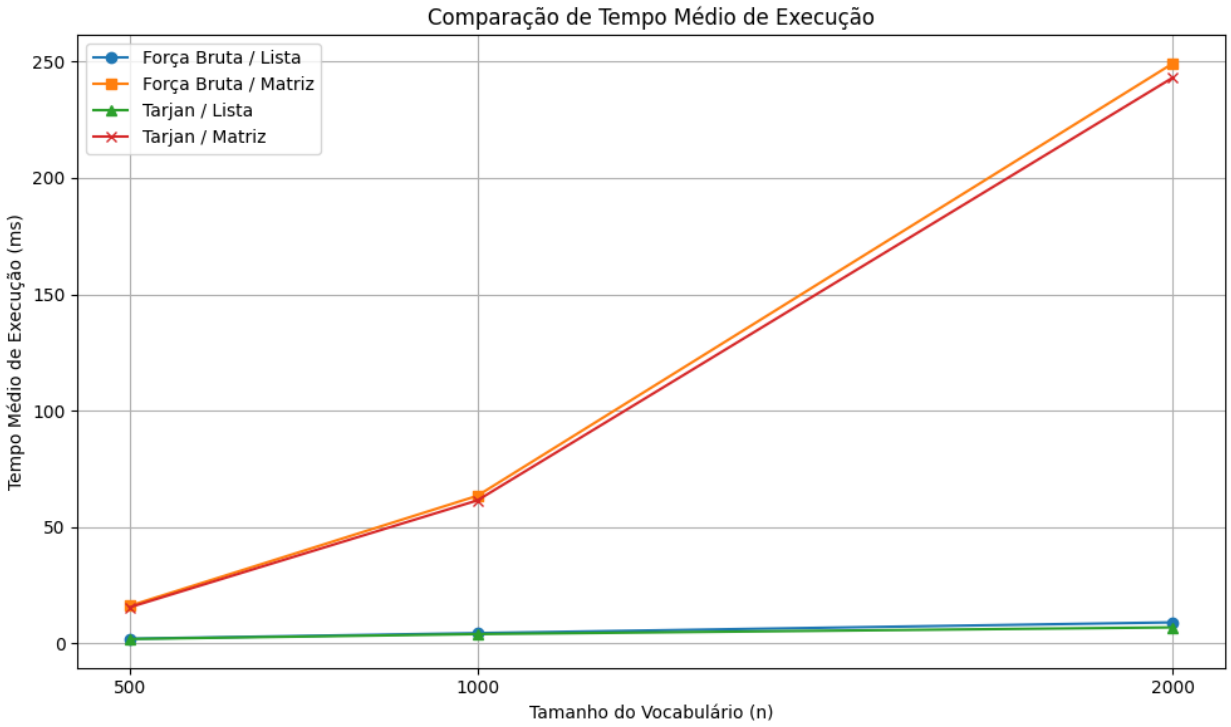

Figura 4: Comparação de desempenho entre algoritmos e representações.

7 Discussão dos Resultados

7.1 Desempenho dos Algoritmos de Detecção de Componentes Conexas

Observações

Compararam-se os tempos de execução médios de dois algoritmos (Força Bruta e Tarjan), utilizando duas representações distintas de grafo: **lista de adjacência** e **matriz de adjacência**.

Conclusões

- A principal variável de impacto no desempenho foi a **representação do grafo**, e não o algoritmo em si.
- A **lista de adjacência** foi significativamente mais eficiente que a matriz, principalmente em grafos grandes e esparsos — como é típico nos grafos de coocorrência.
- A diferença entre Força Bruta e Tarjan foi pequena dentro da mesma estrutura, com Tarjan tendo leve vantagem.

Implicação prática

Para textos grandes ou aplicações que exigem performance, **usar lista de adjacência é fundamental**, independentemente do algoritmo escolhido.

7.2 Comportamento dos Grafos em Diferentes Tipos de Texto

Em textos pequenos e controlados

- Os grafos gerados tendiam a se dividir em **várias componentes conexas distintas**.
- Cada componente geralmente refletia um **tema ou tópico central** do texto (exemplo: palavras relacionadas a "educação" formando uma subestrutura separada de palavras relacionadas a "tecnologia").

Em bases textuais amplas

- O grafo passou a apresentar uma **única componente conexa gigante**, onde todas as palavras estão, de algum modo, conectadas.

Interpretação

Esse fenômeno representa uma **mudança estrutural significativa** no grafo, decorrente da maior diversidade e sobreposição semântica dos textos.

7.3 Palavras Hub e sua Influência Estrutural

Definição

Palavras **hub** são termos que ocorrem com muita frequência e aparecem em diferentes contextos. Elas funcionam como **pontes linguísticas**, conectando palavras que, de outro modo, pertenceriam a temas distintos.

Exemplos comuns

"problema", "forma", "uso", "importante", "sistema", "realizar", "trabalho".

Efeitos no grafo

- Essas palavras possuem **grau elevado** e criam **conexões transversais** no grafo, unindo diferentes clusters semânticos.
- São as responsáveis diretas por **colapsar componentes separadas em uma só**, criando a componente conexa gigante.

7.4 Consequências para Análise e Interpretação dos Grafos

Problemas

- Dificulta a **identificação automática de tópicos** ou **comunidades temáticas**, já que tudo parece conectado.
- **Dilui o significado local** das coocorrências, tornando mais difícil interpretar relações semânticas específicas.

Estratégias para mitigar

- **Remoção de palavras hub**: além de *stopwords* tradicionais, remover palavras com grau extremamente alto.
- **Aplicação de limiar de frequência**: manter apenas arestas com número de coocorrências acima de um certo valor.
- **Análise de comunidades internas**: mesmo com uma única componente, algoritmos como *Louvain* podem revelar subgrupos semânticos.
- **Janelas menores de coocorrência**: diminuem o alcance das pontes semânticas, favorecendo agrupamentos locais.

8 Reflexão Final

O projeto demonstrou, na prática, como o **comportamento estrutural de um grafo de coocorrência** varia com a escala, a representação e a granularidade semântica dos dados.

Os testes controlados foram fundamentais para observar padrões e validar a teoria. Já os testes com dados reais mostraram desafios que só emergem com **complexidade e escala**.

Em última análise, o sucesso na análise de grafos linguísticos não depende apenas do algoritmo, mas da **combinação entre representação, filtragem, escala de análise e objetivos semânticos**.

9 Figura do Grafo com Componentes Destacados (Opcional)

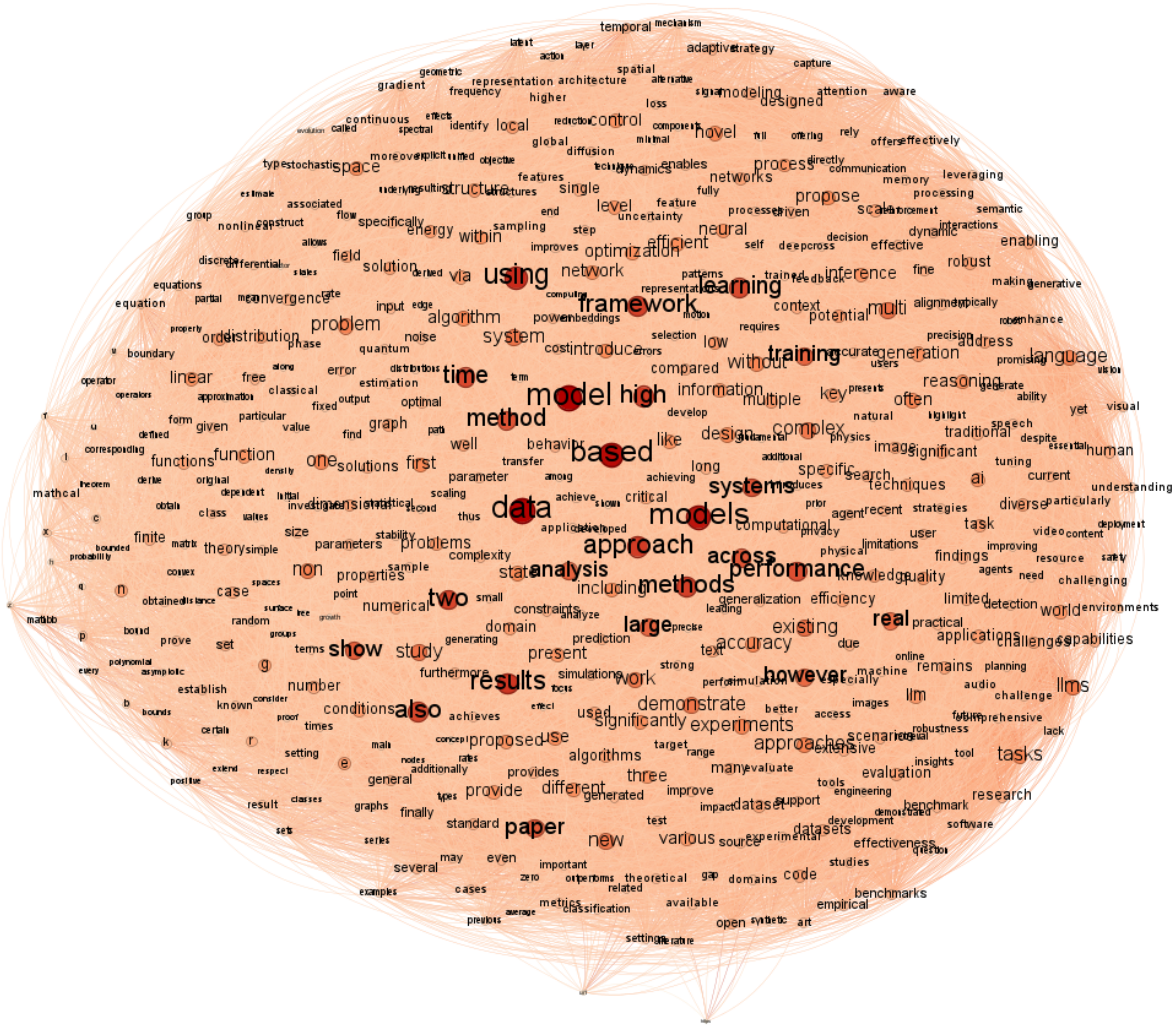

Figura 5: Grafo de Coocorrência para $n = 500$ termos, gerado a partir das amostras reais mencionadas anteriormente no relatório. Cada nó representa um termo frequente, e as arestas indicam coocorrência entre termos dentro de uma janela deslizante.

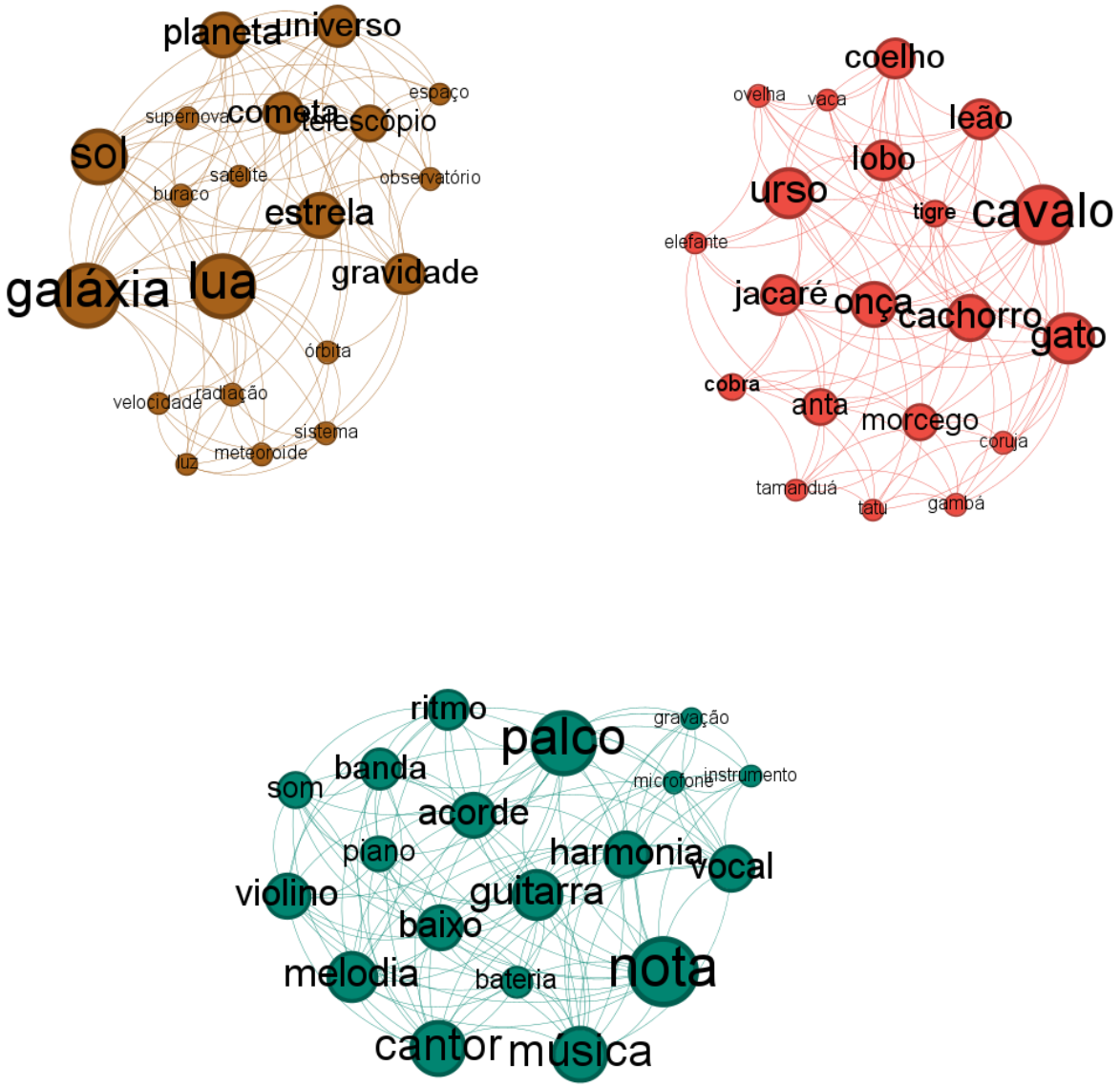

Figura 6: Grafo de Coocorrência gerado a partir de uma base de textos sintéticos criada especificamente para testes controlados. Essa representação foi utilizada para validar o comportamento dos algoritmos de detecção de componentes conexas em diferentes cenários. Cada componete conexa está representada com uma cor diferente (como pedido no trabalho)