



UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Instituto de Geociências e Ciências Exatas
DEMAC
Ciências da Computação



André Luis Dias Nogueira
Felipe Melchior de Britto
Rafael Daiki Kaneko
Ryan Hideki Tadeo Guimarães
Vitor Marchini Rolisola

Relatório sobre Grafos Hamiltonianos

Relatório Acadêmico

Rio Claro, 2024

André Luis Dias Nogueira
Felipe Melchior de Britto
Rafael Daiki Kaneko
Ryan Hideki Tadeo Guimarães
Vitor Marchini Rolisola

Relatório sobre Grafos Hamiltonianos

Este relatório apresenta a implementação de testes para verificar se grafos aleatórios satisfazem os teoremas hamiltonianos de Dirac, Ore e Bondy-Chvátal, utilizando modelos de grafos com ciclo inicial e arestas adicionadas com probabilidade p . A análise foi realizada para diferentes valores de N e p , com resultados apresentados em gráficos e tabelas.

UNIVERSIDADE ESTADUAL PAULISTA

Orientador: Prof. Emílio Bergamim Júnior

Rio Claro

2024

RESUMO

Este trabalho tem como objetivo de realizar a análise de grafos hamiltonianos por meio da implementação e avaliação dos teoremas de Dirac, Ore e Bondy-Chvátal. Foi aplicado esses critérios em grafos aleatórios gerados a partir de dois modelos principais: o modelo Cíclico-Aleatório, que inicia com um ciclo hamiltoniano e incrementa conexões com uma probabilidade p , e o modelo de Erdos-Renyi, onde cada par de vértices recebe uma aresta com probabilidade fixa. Foram gerados grafos aleatórios para diferentes combinações de N (número de vértices) e p , e aplicados testes para determinar a conformidade com os teoremas citados. Também foi desenvolvido um programa para realizar a automação desses testes e gerar os resultados apresentados neste trabalho. Os resultados, apresentados em gráficos e tabelas, demonstram a eficácia de cada teorema e modelo na identificação de grafos hamiltonianos, oferecendo uma análise empírica sobre a aplicabilidade desses critérios em redes complexas.

Palavras-chaves: grafos hamiltonianos, teorema de Dirac, teorema de Ore, teorema de Bondy-Chvátal, modelo de Erdos-Renyi, teoria dos grafos.

SUMÁRIO

1	INTRODUÇÃO	4
1.1	Justificativas e Relevância	4
1.2	Metodologia	4
1.2.1	Teorema de Dirac	5
1.2.2	Teorema de Ore	5
1.2.3	Teorema de Bondy-Chvátal	6
1.2.4	Modelo Cíclico-Aleatório	6
1.2.5	Modelo de Erdos-Renyi	6
1.3	Objetivos	6
2	IMPLEMENTAÇÃO	8
2.1	Código Principal	8
2.1.1	PRIMEIRA REGIÃO (VETORES)	8
2.1.2	SEGUNDA REGIÃO (FILAS)	8
2.1.3	TERCEIRA REGIÃO (MATRIZES)	11
2.1.4	QUARTA REGIÃO (GRAFOS)	13
2.1.5	REGIÃO FINAL (MAIN)	17
2.2	Automação para testes	20

1 INTRODUÇÃO

A Teoria dos Grafos é uma área essencial da matemática discreta, com aplicações em diversos campos, incluindo ciência da computação, logística, redes de comunicação, biologia computacional e pesquisa operacional. Um conceito particularmente relevante nessa teoria é o de ciclos hamiltonianos, onde se busca um percurso cíclico que passa por todos os vértices de um grafo exatamente uma vez. Grafos que contêm tais ciclos são denominados **grafos hamiltonianos** e têm implicações práticas em problemas de otimização, como o problema do caixeiro viajante e o roteamento de redes, onde se procura uma rota eficiente que minimize o custo de deslocamento.

1.1 JUSTIFICATIVAS E RELEVÂNCIA

O estudo de grafos hamiltonianos ganha importância na medida em que muitos problemas complexos podem ser simplificados pela verificação de hamiltonianidade em suas representações gráficas. Contudo, a determinação exata da presença de ciclos hamiltonianos é um problema computacionalmente difícil (NP-completo). Para contornar essa dificuldade, a teoria propõe critérios suficientes de hamiltonianidade, que, embora não garantam uma solução exata para todos os grafos, oferecem maneiras eficientes de inferir a presença de ciclos hamiltonianos em grafos que satisfaçam certas condições. Os teoremas de **Dirac**, **Ore** e **Bondy-Chvátal** são três desses critérios, cada um propondo condições suficientes que, quando satisfeitas, garantem a hamiltonianidade do grafo. A relevância desses teoremas está no potencial de reduzir significativamente a complexidade do problema da hamiltonianidade, o que tem implicações diretas em áreas como o planejamento urbano e a configuração de redes, onde rotas e conexões precisam ser eficientes e bem estruturadas.

Explorar e comparar os modelos de grafos que satisfaçam esses teoremas em condições variáveis de conexão oferece uma base empírica valiosa para avaliar a aplicabilidade e a robustez de cada critério. Esse estudo também contribui para uma melhor compreensão dos modelos aleatórios de grafos, que frequentemente são usados para simular redes reais, onde a distribuição de conexões segue padrões probabilísticos.

1.2 METODOLOGIA

A metodologia proposta para este estudo envolve a implementação de um conjunto de testes para verificar se um grafo dado satisfaz os critérios de hamiltonianidade estabelecidos pelos teoremas de Dirac, Ore e Bondy-Chvátal. Para isso, serão aplicados algoritmos específicos para cada teorema:

1. **Teste de Dirac:** Será verificado se todos os vértices de um grafo possuem grau $\delta \geq \frac{n}{2}$, sendo n o número de vértices do grafo.
2. **Teste de Ore:** Para cada par de vértices não adjacentes u e v , será avaliado se a soma dos graus $d(u) + d(v) \geq n$.
3. **Teste de Bondy-Chvátal:** Utilizando o método de fechamento do grafo, serão inseridas arestas entre vértices não adjacentes sempre que a soma de seus graus seja pelo menos n , e em seguida, será avaliado se o grafo resultante é hamiltoniano.

Os testes serão aplicados a grafos gerados aleatoriamente de acordo com dois modelos:

- **Modelo Cíclico-Aleatório:** Um grafo inicialmente configurado como um ciclo simples de N vértices (o que garante que ele seja hamiltoniano) e, em seguida, arestas adicionais são inseridas entre pares de vértices com uma probabilidade p .
- **Modelo de Erdos-Renyi:** Cada par de vértices recebe uma aresta com uma probabilidade fixa p , sem uma configuração inicial de ciclo, resultando em grafos com conectividade aleatória.

Para cada combinação de N (número de vértices) e p (probabilidade de conexão), serão gerados dez grafos aleatórios. Cada grafo será submetido aos três testes, e os resultados serão organizados em tabelas e gráficos, comparando a frequência com que cada teorema é satisfeito em cada modelo.

1.2.1 TEOREMA DE DIRAC

O Teorema de Dirac, um dos primeiros critérios suficientes para a hamiltonianidade, postula que um grafo simples com $n \geq 3$ vértices é hamiltoniano se todos os seus vértices possuem grau $d(v) \geq n/2$. Esse teorema baseia-se na premissa de que, quando cada vértice possui um número mínimo de conexões, o grafo torna-se suficientemente "denso" para conter um ciclo hamiltoniano. O Teorema de Dirac é relevante por sua simplicidade e pela garantia que fornece em grafos densos, mas aplica-se apenas a grafos onde o grau de cada vértice atinge um limite mínimo específico.

1.2.2 TEOREMA DE ORE

O Teorema de Ore amplia a condição de Dirac ao considerar pares de vértices não adjacentes. Segundo esse teorema, se em um grafo simples com $n \geq 3$ vértices a soma dos graus de cada par de vértices não adjacentes u e v satisfaz $d(u) + d(v) \geq n$, então o grafo é hamiltoniano. Ao incluir pares de vértices não conectados diretamente, o Teorema de Ore apresenta uma condição menos restritiva, aplicando-se a uma gama mais ampla de grafos e oferecendo uma abordagem mais geral para verificar a hamiltonianidade.

1.2.3 TEOREMA DE BONDY-CHVÁTAL

O Teorema de Bondy-Chvátal propõe uma abordagem iterativa para verificar a hamiltonianidade, conhecida como operação de fechamento do grafo. Esse teorema afirma que um grafo G com n vértices é hamiltoniano se e somente se seu fechamento G^* for hamiltoniano, onde G^* é obtido ao adicionar arestas entre pares de vértices não adjacentes u e v sempre que $d(u) + d(v) \geq n$. Essa condição permite construir um grafo equivalente em termos de hamiltonianidade ao adicionar conexões entre pares de vértices conforme necessário, simplificando o problema ao permitir uma verificação gradativa.

1.2.4 MODELO CÍCLICO-ALEATÓRIO

No modelo cíclico-aleatório, inicia-se com um ciclo simples de N vértices, o que garante que o grafo possui um ciclo hamiltoniano desde o início. Em seguida, são adicionadas arestas aleatórias entre pares de vértices não adjacentes com uma probabilidade p . Esse modelo permite que o grafo mantenha um ciclo básico enquanto aumenta gradualmente a conectividade, possibilitando a análise da transição de grafos com hamiltonianidade garantida para grafos mais complexos e densamente conectados.

1.2.5 MODELO DE ERDOS-RENYI

O modelo de Erdos-Renyi, proposto por Paul Erdős e Alfréd Rényi, é um dos modelos mais tradicionais para a geração de grafos aleatórios. Nesse modelo, cada par de vértices em um grafo recebe uma aresta com uma probabilidade fixa p , resultando em grafos com distribuição de conexões aleatória e sem uma estrutura cíclica inicial. Esse modelo é amplamente utilizado para estudar propriedades estatísticas de grafos e para modelar redes complexas onde as conexões entre vértices ocorrem de maneira independente e sem padrões definidos.

1.3 OBJETIVOS

Este estudo possui os seguintes objetivos principais:

1. Explorar a Aplicabilidade dos Teoremas de Dirac, Ore e Bondy-Chvátal: Aprofundar a compreensão dos critérios de hamiltonianidade em grafos aleatórios, identificando em que circunstâncias cada teorema é aplicável.
2. Desenvolver Testes Computacionais para Verificação da Hamiltonianidade: Implementar algoritmos que verifiquem a conformidade de grafos com os três teoremas, de modo a avaliar a eficiência de cada critério como indicador de hamiltonianidade.

3. Comparar Modelos de Grafos Aleatórios: Examinar a eficácia dos modelos cíclico-aleatório e Erdos-Renyi na produção de grafos que satisfaçam os critérios de hamiltonianidade, e comparar as taxas de grafos hamiltonianos produzidos por cada modelo para diferentes valores de N e p .

Este estudo pretende fornecer uma visão prática e teórica sobre os critérios hamiltonianos, contribuindo para o entendimento de sua aplicabilidade e oferecendo uma base empírica para o uso desses critérios na análise e simulação de redes complexas.

2 IMPLEMENTAÇÃO

A implementação deste projeto consiste na criação de um algoritmo para verificar se um grafo satisfaz os três teoremas hamiltonianos: Dirac, Ore e Bondy-Chvátal. Além disso, foi desenvolvido um script em Python para automatizar a geração e análise dos resultados.

2.1 CÓDIGO PRINCIPAL

O código principal, escrito em linguagem C, está organizado em várias seções, cada uma dedicada a funcionalidades específicas relacionadas à geração, manipulação e análise de grafos, com relação aos ciclos hamiltonianos.

2.1.1 PRIMEIRA REGIÃO (VETORES)

```
1  int *criar_vetor(int n) {
2      int *vetor = (int *)calloc(n, sizeof(int));
3      return vetor;
4  }
5
6  void troca_lugares(int *vetor, int num1, int num2) {
7      int aux = vetor[num1];
8      vetor[num1] = vetor[num2];
9      vetor[num2] = aux;
10 }
11
12 void liberar_vetor(int *vetor) {
13     free(vetor);
14 }
```

A primeira região, **vetores**, contém funções para gerenciamento de array dinâmico (*vetor*). Inclui funções para criar um vetor (*criar_vetor*), trocar elementos dentro de um vetor (*troca_lugares*) e liberar a memória alocada para um vetor (*liberar_vetor*).

2.1.2 SEGUNDA REGIÃO (FILAS)

```
1  typedef struct dado {
2      int dado;
```

```
3     struct dado *proximo;
4 } DADO;
5
6 typedef struct {
7     DADO *entrada;
8     DADO *saida;
9 } Fila;
10
11 typedef Fila *p_fila;
12
13 p_fila criar_fila() {
14     p_fila f = malloc(sizeof(Fila));
15     f->entrada = NULL;
16     f->saida = NULL;
17     return f;
18 }
19
20 int fila_vazia(p_fila f) {
21     return (f->saida == NULL);
22 }
23
24 void esvaziar_fila(p_fila f) {
25     DADO *aux;
26     while (!fila_vazia(f)) {
27         aux = f->saida;
28         f->saida = f->saida->proximo;
29         free(aux);
30     }
31     f->entrada = NULL;
32 }
33
34 void liberar_fila(p_fila f) {
35     DADO *aux;
36     while (!fila_vazia(f)) {
37         aux = f->saida;
38         f->saida = f->saida->proximo;
39         free(aux);
40     }
```

```

41     free(f);
42 }
43
44 void enqueue(p_filha f, int k) {
45     DADO *aux = malloc(sizeof(DADO));
46     aux->dado = k;
47     aux->proximo = NULL;
48     if (!fila_vazia(f)) {
49         f->entrada->proximo = aux;
50         f->entrada = aux;
51     } else {
52         f->entrada = aux;
53         f->saida = aux;
54     }
55 }
56
57 int dequeue(p_filha f) {
58     DADO *aux;
59     int i;
60     if (!fila_vazia(f)) {
61         aux = f->saida;
62         if (f->entrada != f->saida) {
63             f->saida = f->saida->proximo;
64         } else {
65             f->entrada = NULL;
66             f->saida = NULL;
67         }
68         i = aux->dado;
69         free(aux);
70         return i;
71     }
72     return INT_MIN;
73 }
74
75 bool remove_item(p_filha f, int k) {
76     DADO *atual = f->saida;
77     DADO *anterior = NULL;
78     while (atual != NULL) {

```

```

79     if (atual->dado == k) {
80         if (anterior != NULL) {
81             anterior->proximo = atual->proximo;
82             free(atual);
83         } else {
84             desenfileirar(f);
85         }
86         return true;
87     }
88     anterior = atual;
89     atual = atual->proximo;
90 }
91 return false;
92 }

```

A segunda região, **filas**, define e gerencia uma estrutura de dados de fila (*FIFO*). Inclui a definição da estrutura (*DADO*), que representa um elemento na fila, e a estrutura *Fila*, que representa a própria fila. As funções nesta região incluem criar uma fila (*criar_fila*), verificar se uma fila está vazia (*fila_vazia*), esvaziar uma fila (*esvaziar_fila*), liberar a fila (*liberar_fila*), enfileirar (*enfileirar*), desenfileirar (*desenfileirar*) e remover um item específico da fila (*remover_item*).

2.1.3 TERCEIRA REGIÃO (MATRIZES)

```

1  typedef struct matricial {
2      int n; // linhas
3      int **matriz; // ponteiro para matriz
4      int *grau; // ponteiro para vetor com o grau dos vértices
5      p_fila *lista_n_adjascencia; // ponteiro para vetor da lista de
        ↳ não adjascência
6  } Matriz;
7
8  Matriz *inicializar_matriz(int qtd_vertices) {
9      Matriz *matricial = (Matriz *)malloc(sizeof(Matriz));
10     int **matriz_adjascencia = (int **)malloc(qtd_vertices *
        ↳ sizeof(int *));
11     for (int i = 0; i < qtd_vertices; i++) {
12         matriz_adjascencia[i] = (int *)malloc(qtd_vertices *
        ↳ sizeof(int));

```

```

13     }
14     matricial->n = qtd_vertices;
15     matricial->matriz = matriz_adjascencia;
16     matricial->lista_n_adjascencia = malloc(sizeof(p_fila) *
17         ↪ qtd_vertices);
18     for (int i = 0; i < qtd_vertices; i++) {
19         matricial->lista_n_adjascencia[i] = criar_fila();
20     }
21     matricial->grau = criar_vetor(qtd_vertices);
22     return matricial;
23 }
24
25 Matriz *copiar_matriz(Matriz *matricial) {
26     Matriz *copia = inicializar_matriz(matricial->n);
27     DADO *aux;
28     for (int i = 0; i < copia->n; i++) {
29         copia->grau[i] = matricial->grau[i];
30         aux = matricial->lista_n_adjascencia[i]->saida;
31         while (aux != NULL) {
32             enfileirar(copia->lista_n_adjascencia[i], aux->dado);
33             aux = aux->proximo;
34         }
35         for (int j = 0; j < copia->n; j++) {
36             copia->matriz[i][j] = matricial->matriz[i][j];
37         }
38     }
39     return copia;
40 }
41
42 void liberar_matriz(Matriz *matricial) {
43     for (int i = 0; i < matricial->n; i++) {
44         free(matricial->matriz[i]);
45         liberar_fila(matricial->lista_n_adjascencia[i]);
46     }
47     liberar_vetor(matricial->grau);
48     free(matricial->lista_n_adjascencia);
49     free(matricial->matriz);
50     free(matricial);

```

A terceira região, **matrizes**, lida com operações de matriz, especificamente para representar grafos. Ela define a estrutura `Matriz`, que inclui a matriz de adjacência, o grau de vértices e uma lista de vértices não adjacentes. As funções nesta região incluem inicializar uma matriz (*inicializar_matriz*), copiar uma matriz (*copiar_matriz*) e liberar a memória alocada para uma matriz (*liberar_matriz*).

2.1.4 QUARTA REGIÃO (GRAFOS)

```

1  typedef Matriz *Grafo;
2
3  void gerar_grafo(Grafo grafo, bool orientado, float probabilidade) {
4      int porcentagem = (int)(100 * probabilidade);
5      if (!orientado) { // garante espelhamento
6          for (int i = 0; i < grafo->n; i++) {
7              for (int j = i; j < grafo->n; j++) {
8                  if (i != j) { // evitar ligacoes proprias
9                      grafo->matriz[i][j] = (rand() % 100 <
10                         ↪ porcentagem) ? 1 : 0; // pesos entre 1 e 0
11                         ↪ (tem ou nao tem)
12                      grafo->matriz[j][i] = grafo->matriz[i][j];
13                      if (grafo->matriz[i][j]) {
14                          grafo->grau[i]++;
15                          grafo->grau[j]++;
16                      } else {
17                          enfileirar(grafo->lista_n_adjascencia[i], j);
18                          enfileirar(grafo->lista_n_adjascencia[j], i);
19                      }
20                  } else {
21                      grafo->matriz[i][j] = 0; // falso para quando for
22                      ↪ a diagonal principal
23                  }
24              }
25          }
26      } else {
27          for (int i = 0; i < grafo->n; i++) {
28              for (int j = 0; j < grafo->n; j++) {
29                  if (i != j) { // evitar ligacoes proprias

```

```

27         grafo->matriz[i][j] = (rand() % 100 <
    ↪   porcentagem) ? 1 : 0; // pesos entre 1 e 0
    ↪   (tem ou nao tem)
28         if (grafo->matriz[i][j]) {
29             grafo->grau[i]++;
30         } else {
31             enfileirar(grafo->lista_n_adjascencia[i], j);
32         }
33     } else {
34         grafo->matriz[i][j] = 0; // falso para quando for
    ↪   a diagonal principal
35     }
36 }
37 }
38 }
39 }
40
41 void gerar_grafo_hamiltoniano(Grafo grafo, bool orientado, float
    ↪   probabilidade) {
42     gerar_grafo(grafo, orientado, probabilidade);
43     int *ciclo = criar_vetor(grafo->n);
44     for (int i = 0; i < grafo->n; i++) {
45         ciclo[i] = i;
46     }
47     if (!orientado) {
48         troca_lugares(ciclo, 0, rand() % (grafo->n));
49         for (int i = 1; i < grafo->n; i++) {
50             troca_lugares(ciclo, i, rand() % (grafo->n - i) + i);
51             if (!(grafo->matriz[ciclo[i - 1]][ciclo[i]])) {
52                 grafo->matriz[ciclo[i - 1]][ciclo[i]] = 1;
53                 grafo->matriz[ciclo[i]][ciclo[i - 1]] =
    ↪   grafo->matriz[ciclo[i - 1]][ciclo[i]];
54                 grafo->grau[ciclo[i - 1]]++;
55                 grafo->grau[ciclo[i]]++;
56                 remover_item(grafo->lista_n_adjascencia[ciclo[i - 1]],
    ↪   ciclo[i]);
57                 remover_item(grafo->lista_n_adjascencia[ciclo[i]],
    ↪   ciclo[i - 1]);

```

```

58     }
59 }
60 if (!(grafo->matriz[ciclo[grafo->n - 1]][ciclo[0]])) {
61     grafo->matriz[ciclo[grafo->n - 1]][ciclo[0]] = 1;
62     grafo->matriz[ciclo[0]][ciclo[grafo->n - 1]] =
        ↪ grafo->matriz[ciclo[grafo->n - 1]][ciclo[0]];
63     grafo->grau[ciclo[grafo->n - 1]]++;
64     grafo->grau[ciclo[0]]++;
65     remover_item(grafo->lista_n_adjascencia[ciclo[grafo->n -
        ↪ 1]], ciclo[0]);
66     remover_item(grafo->lista_n_adjascencia[ciclo[0]],
        ↪ ciclo[grafo->n - 1]);
67 }
68 }
69 liberar_vetor(ciclo);
70 }
71
72 bool dirac(Grafo grafo) {
73
74     for (int i = 0; i < grafo->n; i++) {
75         if (grafo->grau[i] < grafo->n / 2)
76             return false;
77     }
78     return true;
79 }
80
81 bool ore(Grafo grafo) {
82     DADO *aux;
83     for (int i = 0; i < grafo->n; i++) {
84         aux = grafo->lista_n_adjascencia[i]->saida;
85         while (aux != NULL) {
86             if (grafo->grau[i] + grafo->grau[aux->dado] < grafo->n)
87                 return false;
88             aux = aux->proximo;
89         }
90     }
91     return true;
92 }

```



```

93
94 Grafo fecho_hamiltoniano(Grafo grafo) {
95     Grafo fecho_hamiltoniano = copiar_matriz(grafo);
96     int aux;
97     for (int i = 0; i < fecho_hamiltoniano->n; i++) {
98         while (fecho_hamiltoniano->lista_n_adjascencia[i]->saida !=
99             ↪ NULL) {
100             if (ore(fecho_hamiltoniano))
101                 return fecho_hamiltoniano;
102             aux = desenfileirar(fecho_hamiltoniano->
103                 ↪ lista_n_adjascencia[i]);
104             fecho_hamiltoniano->matriz[i][aux] = 1;
105             fecho_hamiltoniano->matriz[aux][i] =
106                 ↪ fecho_hamiltoniano->matriz[i][aux];
107             fecho_hamiltoniano->grau[i]++;
108             fecho_hamiltoniano->grau[aux]++;
109         }
110     }
111     liberar_matriz(fecho_hamiltoniano);
112     return NULL;
113 }
114
115 bool bondy_chvatal(Grafo fecho) {
116     if (fecho == NULL)
117         return false;
118     for (int i = 0; i < fecho->n; i++) {
119         for (int j = i + 1; j < fecho->n; j++) {
120             if (!(fecho->matriz[i][j])) {
121                 return false;
122             }
123         }
124     }
125     return true;
126 }

```

A quarta região, **grafo**, foca em operações específicas de grafos. Inclui funções para gerar um gráfico aleatório (*gerar_grafo*), gerar um gráfico hamiltoniano (*gerar_grafo_hamiltoniano*), verificar se um gráfico satisfaz o teorema de Dirac

(*dirac*), verificar se um gráfico satisfaz o teorema de Ore (*ore*), gerar um fechamento hamiltoniano de um gráfico (*fecho_hamiltoniano*) e verificar se um gráfico satisfaz o teorema de Bondy-Chvátal (*bondy_chvatal*). Além disso, inclui funções para imprimir o gráfico em um arquivo (*imprimir_grafo_arquivo*) e visualizar o gráfico e suas informações (*visualizar_grafo_e_informacoes*).

2.1.5 REGIÃO FINAL (MAIN)

```
1  int main() {
2      int opcao, n, salvar;
3      Grafo grafo=NULL, grafo=NULL;
4      float probabilidade;
5      static int orientado = false;
6      while (true) {
7          srand(time(NULL)); // garantir boa aleatorizacao
8          printf("+-----Menu-----+\n");
9          printf("Escolha uma opcao:\n");
10         printf("1. Gerar Grafo\n");
11         printf("2. Gerar Grafo Hamiltoniano\n");
12         printf("3. Verificar Teorema de Dirac\n");
13         printf("4. Verificar Teorema de Ore\n");
14         printf("5. Verificar Bondy-Chvatal\n");
15         printf("6. Gerar Fecho Hamiltoniano\n");
16         printf("7. Visualizar Grafo\n");
17         printf("8. Visualizar o Fecho Hamiltoniano\n");
18         printf("0. Sair\n");
19         printf("+-----+\n");
20         scanf("%d", &opcao);
21         switch (opcao) {
22             case 1:
23                 if (grafo) {
24                     liberar_matriz(grafo);
25                     printf("Grafo anterior existente foi excluido!\n");
26                 }
27                 printf("Digite o numero de vertices: ");
28                 scanf("%d", &n);
29                 printf("Digite a probabilidade de aresta (0 a 1): ");
30                 scanf("%f", &probabilidade);
31                 grafo = inicializar_matriz(n);
```

```

32     gerar_grafo(grafo, orientado, probabilidade);
33     printf("Grafo gerado com sucesso!\n");
34     break;
35 case 2:
36     ...
37     printf("Digite o numero de vertices: ");
38     scanf("%d", &n);
39     printf("Digite a probabilidade de aresta (0 a 1): ");
40     scanf("%f", &probabilidade);
41     grafo = inicializar_matriz(n);
42     gerar_grafo_hamiltoniano(grafo, orientado, probabilidade);
43     printf("Grafo Hamiltoniano gerado com sucesso!\n");
44     break;
45 case 3:
46     ...
47     if (dirac(grafo)) {
48         printf("O grafo satisfaz o Teorema de Dirac.\n");
49     } else {
50         printf("O grafo NAO satisfaz o Teorema de Dirac.\n");
51     }
52     break;
53 case 4:
54     ...
55     if (ore(grafo)) {
56         printf("O grafo satisfaz o Teorema de Ore.\n");
57     } else {
58         printf("O grafo NAO satisfaz o Teorema de Ore.\n");
59     }
60     break;
61 case 5:
62     ...
63     if (ore(grafo)) {
64         fecho = grafo;
65     }
66     if (bondy_chvatal(fecho)) {
67         printf("O fecho hamiltoniano satisfaz o Teorema de
68              $\hookrightarrow$  Bondy-Chvatal.\n");
69     } else {

```

```

69         printf("O fecho hamiltoniano NAO satisfaz o Teorema de
           ↪ Bondy-Chvatal.\n");
70     }
71     break;
72 case 6:
73     ...
74     if (ore(grafo)) {
75         printf("O grafo já possui um fecho hamiltoniano.\n");
76     } else {
77         fecho = fecho_hamiltoniano(grafo);
78         if (fecho != NULL) {
79             printf("Um fecho hamiltoniano para o grafo foi
               ↪ gerado!\n");
80         } else {
81             printf("Não foi possível modificar o grafo.\n");
82         }
83     }
84     break;
85 case 7:
86     ...
87 case 8:
88     if (ore(grafo)) {
89         printf("O próprio grafo já é um fecho
               ↪ hamiltoniano.\n");
90         break;
91     }
92     ...
93     ...
94 }
95 }
96 }

```

A região final, **main**, contém a função (*main*) que fornece uma interface orientada a menu para o usuário interagir com o programa. O usuário pode gerar grafos, verificar vários teoremas, gerar fechamentos hamiltonianos e visualizar os grafos. A função principal manipula a entrada do usuário, chama as funções apropriadas com base na escolha do usuário e garante o gerenciamento de memória adequado ao liberar recursos alocados antes de sair.

2.2 AUTOMAÇÃO PARA TESTES