Documentação do trabalho prático

Fractais

Leonardo Borges de Oliveira — Turma TW — 2021014368

1. Introdução

O presente documento consiste na documentação do Trabalho Prático da disciplina de Matemática Discreta do semestre 2023/1. Esse se trata da implementação e análise de dois fractais a serem definidos com base no número de matrícula do autor e um novo fractal criado, também, pelo autor.

As regras para definição dos fractais a ser implementado são:

$$\sum algs\ n.\ de\ matrícula\ mod\ 4\ =\ 27\ mod\ 4\ =\ 3$$
$$Número\ de\ matrícula\ par$$

Assim, a primeira regra define a implementação da *Ilha de Koch* e a segunda define a implementação do *Preenchimento de espaço de Hilbert.*

2. Estratégia de Implementação

Sobre as estratégias para a implementação dos fractais, foram consideradas duas principais: uma iterativa com arquivos intermediários e uma recursiva. A estratégia utilizada foi a mesma para os três fractais desenvolvidos nesse trabalho.

A primeira, criaria um arquivo que armazenaria a sequência de caracteres para cada um dos estágios. E, a partir do estágio anterior leria a sequência de caracteres e criaria o arquivo para o estágio atual.

A principal vantagem desse método é que se teria um controle de todos os estágios passados e seria possível manipulá-lo a qualquer momento. O ponto negativo desse método é a utilização de memória secundária, que diminui o desempenho do algoritmo, além de estar mais suscetível a erros por conta de falhas no acesso à memória secundária.

Já a segunda, que foi a alternativa adotada, usaria o princípio recursivo. Para isso, foi criada uma função *get_fractal(int)* que recebe como parâmetro o estágio atual do fractal, para os fractais com duas regras essa função foi substituída por *get_x(int) e get_y(int)*. Caso o estágio seja igual a 0 a *get_fractal* retorna o caractere ‘F’ e as demais retornam nulo, a fim de eliminar o símbolo intermediário. Caso não seja, é chamada a própria função passando o estágio atual menos um, para o caso da *get_x* também é chamada a *get_y* com o mesmo parâmetro, para a *get_y,* o oposto. E, por fim, retorna a concatenação da regra de recorrência equivalente.

Nessa segunda, a principal vantagem é que é o algoritmo se mantém completamente em memória principal, fazendo, assim, com que o seu tempo de execução seja menor para estágios maiores. Além disso, evita, também, erros na abertura e fechamento de arquivos.

2.1. Implementação

Comentando um pouco mais especificamente sobre a implementação utilizada, temos que todos os códigos se iniciam numa função *main*, onde é feita a interação direta com o usuário, recebendo parâmetros de entrada, a abertura de arquivo e gravação de informações preliminares nele. Após isso, o método chama, por um laço de 4 iterações, a função *write_fractal*.

Essa função, é responsável por lidar com o axioma. Assim, ela chamará os métodos que retornam as regras presentes no axioma e imprimirá no arquivo a sequência de caracteres obtidas após as n interações.

```
void write_fractal(int size, FILE* f ){
    String* s = get_fractal(size - 1);

    fprintf(f, "-------------------------------------------------
    fprintf(f, "\nEstágio %d: %s+%s+%s+%s\n",
        size,
        s->_string,
        s->_string,
        s->_string,
        s->_string);

}
```

Figura 1: Função *write_fractal* para o fractal Koch Island

É possível perceber na Figura 1 que o método que retorna a regra necessária no axioma é chamado apenas uma vez, mesmo sendo necessária quatro vezes no axioma. Essa foi a principal motivação para a criação dessa função. Nos demais fractais ela também está presente, chamando as funções *get_x* e *get_y* necessárias para a implementação do axioma.

```
void write_fractal(int size, FILE* f ){
    String* x = get_X(size);
    String* y = get_Y(size);

    fprintf(f, "-------------------------------------------------------------
    fprintf(f, "\nEstágio %d: %s+%s\n", size, x->_string, y->_string);

}
```

Figura 2: Função *write_fractal* para o fractal Preenchimento de Espaço de Leonardo

Seguindo, têm-se as funções *get_fractal, get_x e get_y* que são o cerne da estratégia da aplicação sendo detalhadas acima.

```
String* get_fractal(int stage){
    String* f = new_string("F");

    if(stage == 0)
        return f;

    String* plus = new_string("+");
    String* minus = new_string("-");

    String* fractal = get_fractal(stage - 1);

    String *recurrency_rule [15] = {···

    return concatenate(recurrency_rule, 15);
}
```

Figura 3: Função *get_fractal* para o fractal Ilha de Koch

```
String* get_X(int stage){
    if(stage == 0)
        return NULL;

    String* y = get_Y(stage - 1);
    String* x = get_X(stage - 1);
    String* f = new_string("F");
    String* plus = new_string("+");
    String* minus = new_string("-");

    if(y == NULL || x == NULL){
        String *recurrency_rule [8] = { minus, f, f, plus, f, f, minus, minus};
        return concatenate(recurrency_rule, 8);
    } else {
        String *recurrency_rule [13] = {minus, y, f, f, plus, x, y, f, x, f, y, minus, minus};
        return concatenate(recurrency_rule, 13);
    }
}
```

Figura 4: Função *get_x* para o fractal Preenchimento de Espaço de Leonardo

Percebe-se, que nas funções presentes nas Figuras 3 e 4 também foi adotada a estratégia de chamar apenas uma vez o método que retorna a regra do caractere recursivamente, mesmo sendo utilizada mais de uma vez na regra de recorrência. Essa decisão também foi tomada visando melhorar o desempenho do código.

Além dessas funções, também foi criada a função *concatenate* com o intuito de evitar repetição de código. A principal função dela é lidar com a alocação de memória para a sequência de caracteres e juntar em uma *String* todos os caracteres passados como parâmetro.

```
String* concatenate(String** s_array, int array_size){
    String* destination = (String *) malloc(1 * sizeof(String));

    for(int j = 0; j < array_size; j++){
        int initial_position = destination->size;
        destination->size += s_array[j]->size;
        destination->_string = (char *) realloc(destination->_string,  destination->size * sizeof(char*));

        for(int i = initial_position; i < destination->size; i++)
            destination->_string[i] = s_array[j]->_string[i-initial_position];
    }

    return destination;
}
```

Figura 5: Função *concatenate* para o fractal Preenchimento de Espaço de Hilbert

Por fim, também foi criado um tipo *String*, necessário para armazenar uma sequência de caracteres e o seu tamanho, e um construtor para esse tipo para evitar repetição de código e centralizar a criação de novas *Strings*.

```
typedef struct string {
    int size;
    char* _string;
} String;

String * new_string(char* c){
    String* s = (String *) malloc(1 * sizeof(String));
    s->size = 1;
    s->_string = c;

    return s;
}
```

Figura 6: Tipo *String* e seu construtor

3. Ilha de Koch

O primeiro fractal a ser tratado é o da Ilha de Koch que pode ser definido pelo seguinte L-sistema:

$$Axioma\colon F + F + F + F$$
$$\theta = 90º$$
$$Regra\colon F \rightarrow F + F - F - FFF + F + F - F$$

A tabela a seguir lista a quantidade de Segmentos (F) e a quantidades de Símbolos gerados para os quatro primeiros estágios da Ilha de Koch.

Tabela 1: Número de Símbolos e Segmentos para Ilha de Koch

n	#F	#Símbolos
1	36	63
2	324	567
3	2916	5103
4	26244	45927

3.1. Equação de Recorrência

A partir da Tabela 1 e do L-sistema da Ilha de Koch é possível escrever a equação de recorrência que define a quantidade de Segmentos e Símbolos que o fractal vai apresentar para um estágio n, como será demonstrado a seguir.

Para o número de segmentos é possível fazer:

$$T(1) = 4 * 9$$
$$T(2) = 4 * 9 * 9$$
$$T(3) = 4 * 9 * 9 * 9$$
$$\dots$$
$$T(n) = 4 * 9^n$$

Já para o número de total de símbolos é possível escrevê-la como:

$$T(n) = h(n) + g(n)$$

Em que $h(n)$ é o número total de Fs presentes no estágio e $g(n)$ é o total de símbolos diferentes de F presentes no estágio.

A função $h(n)$, como apresentado anteriormente, é dada por $4 * 9^n$. Já para a função $g(n)$ é possível pensar:

$$g(1) = 3 + 4 * 6$$
$$g(2) = 3 + 4 * 6 + 4 * 9 * 6$$
$$g(3) = 3 + 4 * 6 + 4 * 9 * 6 + 4 * 9 * 9 * 6$$
$$\dots$$
$$g(n) = 3 + \sum_{i=0}^{n-1} 24 * 9^i$$

Assim, tem-se:

$$T(n) = 4 * 9^n + \sum_{i=0}^{n-1} 24 * 9^i + 3$$

3.2. Análise de Complexidade

Analisando as equações de recorrência geradas para a contagem do número de símbolos e de Fs presentes na Ilha de Koch, é possível perceber que o algoritmo cresce exponencialmente. Isso é visível tanto na primeira equação com o 9^n quanto na segunda com o 9^i. Dessa maneira, a ordem de complexidade do algoritmo é $\theta(c^n)$, onde c é uma constante e n o número de estágios.

4. Preenchimento de Espaço de Hilbert

O segundo fractal a ser tratado é o Preenchimento de Espaço de Hilbert que pode ser definido pelo seguinte L-sistema:

$$Axioma: X$$
$$\theta = 90º$$
$$Regra: X \rightarrow -YF + XFX + FY -$$
$$Y \rightarrow +XF - YFY - FX +$$

A tabela a seguir lista a quantidade de Segmentos (F) e a quantidades de Símbolos gerados para os quatro primeiros estágios da Preenchimento de Espaço de Hilbert.

Tabela 2: Número de Símbolos e Segmentos para Preenchimento de Espaço de Hilbert

n	#F	#Símbolos
1	3	7
2	15	35
3	63	147
4	255	595

4.1. Equação de Recorrência

A partir da Tabela 2 e do L-sistema da Preenchimento de Espaço de Hilbert é possível escrever a equação de recorrência que define a quantidade de Segmentos e Símbolos que o fractal vai apresentar para um estágio n, como será demonstrado a seguir.

Para o número de segmentos é possível fazer:

$$T(1) = 3$$
$$T(2) = 4 * 3 + 3$$
$$T(3) = 4 * 4 * 3 + 4 * 3 + 3$$
...

$$T(n) = \sum_{i=0}^{n-1} 3 * 4^i$$

Já para o número de símbolos é possível fazer parecido, tendo:

$$T(1) = 7$$
$$T(2) = 4 * 7 + 7$$
$$T(3) = 4 * 4 * 7 + 4 * 7 + 7$$
$$\dots$$
$$T(n) = \sum_{i=0}^{n-1} 7 * 4^i$$

4.2. Análise de Complexidade

Analisando as equações de recorrência geradas para a contagem do número de símbolos e de Fs presentes na Preenchimento de Espaço de Hilbert, é possível perceber que o algoritmo cresce exponencialmente. Isso é visível em ambas equações com o 4^i. Dessa maneira, a ordem de complexidade do algoritmo é $\theta(c^n)$, onde c é uma constante e n o número de estágios.

5. Preenchimento de Espaço de Leonardo

O último fractal a ser tratado é o Preenchimento de Espaço de Leonardo que pode ser definido pelo seguinte L-sistema:

$$Axioma: X + Y + X + Y + X + Y + X + Y$$
$$\theta = 60º$$
$$Regra: X \rightarrow -YFF + XYFXFY --$$
$$Y \rightarrow +XFF - YXFYFX ++$$

A tabela a seguir lista a quantidade de Segmentos (F) e a quantidades de Símbolos gerados para os quatro primeiros estágios da Preenchimento de Espaço de Leonardo.

Tabela 3: Número de Símbolos e Segmentos para Preenchimento de Espaço de Leonardo

n	#F	#Símbolos
1	24	71
2	144	391
3	744	1991
4	3744	9991

5.1. Equação de Recorrência

A partir da Tabela 3 e do L-sistema da Preenchimento de Espaço de Leonardo é possível escrever a equação de recorrência que define a quantidade de Segmentos e Símbolos que o fractal vai apresentar para um estágio n, como será demonstrado a seguir.

Para o número de segmentos é possível fazer:

$$T(1) = 6 * 4$$
$$T(2) = 5 * 6 * 4 + 6 * 4$$
$$T(3) = 5 * 5 * 6 * 4 + 5 * 6 * 4 + 6 * 4$$
$$\dots$$
$$T(n) = \sum_{i=0}^{n-1} 6 * 4 * 5^{i} = \sum_{i=0}^{n-1} 24 * 5^{i}$$

Já para o número de símbolos é possível fazer parecido, tendo:

$$T(1) = 6 * 8 + 7$$
$$T(2) = 5 * 6 * 8 + 6 * 8 + 7$$
$$T(3) = 5 * 5 * 6 * 8 + 5 * 6 * 8 + 6 * 8 + 7$$
$$\dots$$
$$T(n) = \sum_{i=0}^{n-1} (6 * 8 * 5^{i}) + 7 = \sum_{i=0}^{n-1} (48 * 5^{i}) + 7$$

5.2. Análise de Complexidade

Analisando as equações de recorrência geradas para a contagem do número de símbolos e de Fs presentes na Preenchimento de Espaço de Leonardo, é possível perceber que o algoritmo cresce exponencialmente. Isso é visível em ambas equações com o 5^{i}. Dessa maneira, a ordem de complexidade do algoritmo é $\theta(c^{n})$, onde c é uma constante e n o número de estágios.

5.3. Desenho dos Estágios

Para a última parte desse trabalho, foi proposto que a turma interagisse levantando possíveis soluções simples para o desenho dos fractais. Após a conversa com alguns colegas de classe, decidimos a utilização da biblioteca SDL2 por ser na linguagem C e pela simplicidade de implementação. A partir disso, foi criado um pequeno script que calculava a partir do seno e do cosseno do ângulo passado as posições de início e fim do segmento representado pela letra F na sequência de caracteres passado para o script.

Assim, os fractais gerados estão representados nas imagens a seguir:

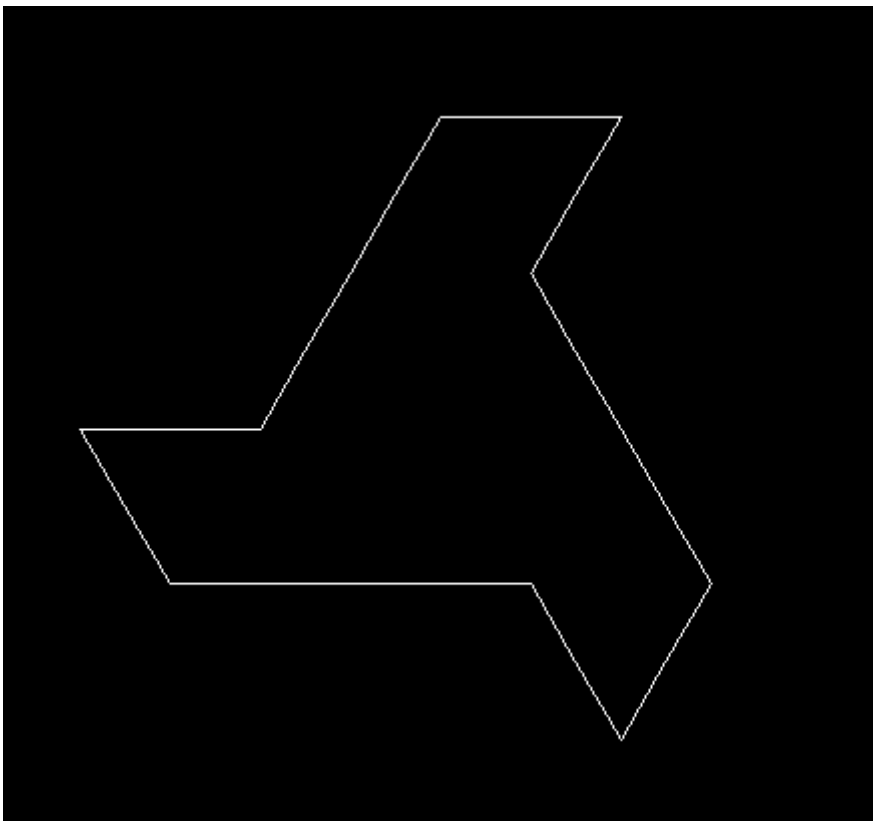

Figura 7: Estágio 1 do Preenchimento de Espaço de Leonardo

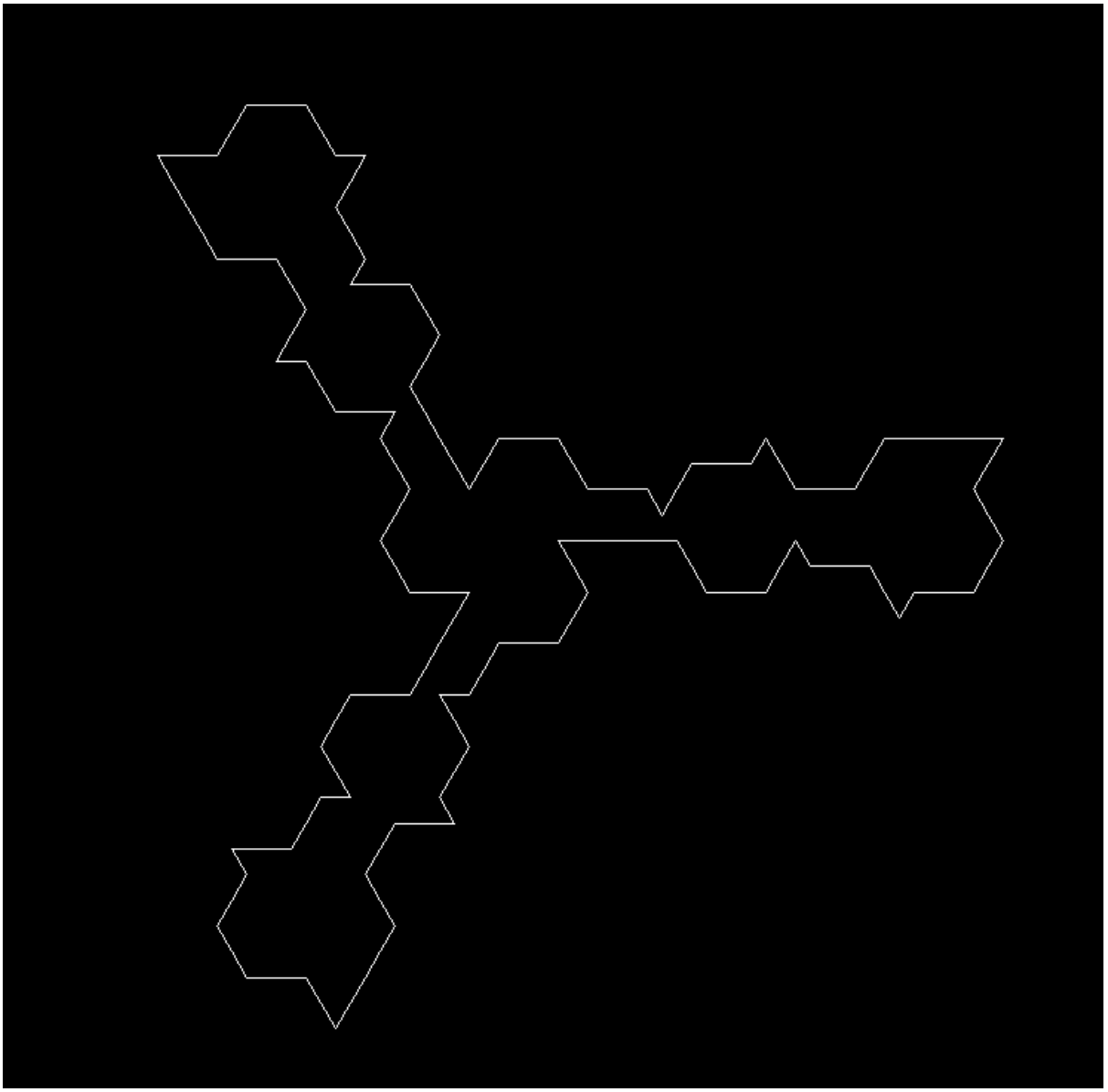

Figura 8: Estágio 2 do Preenchimento de Espaço de Leonardo

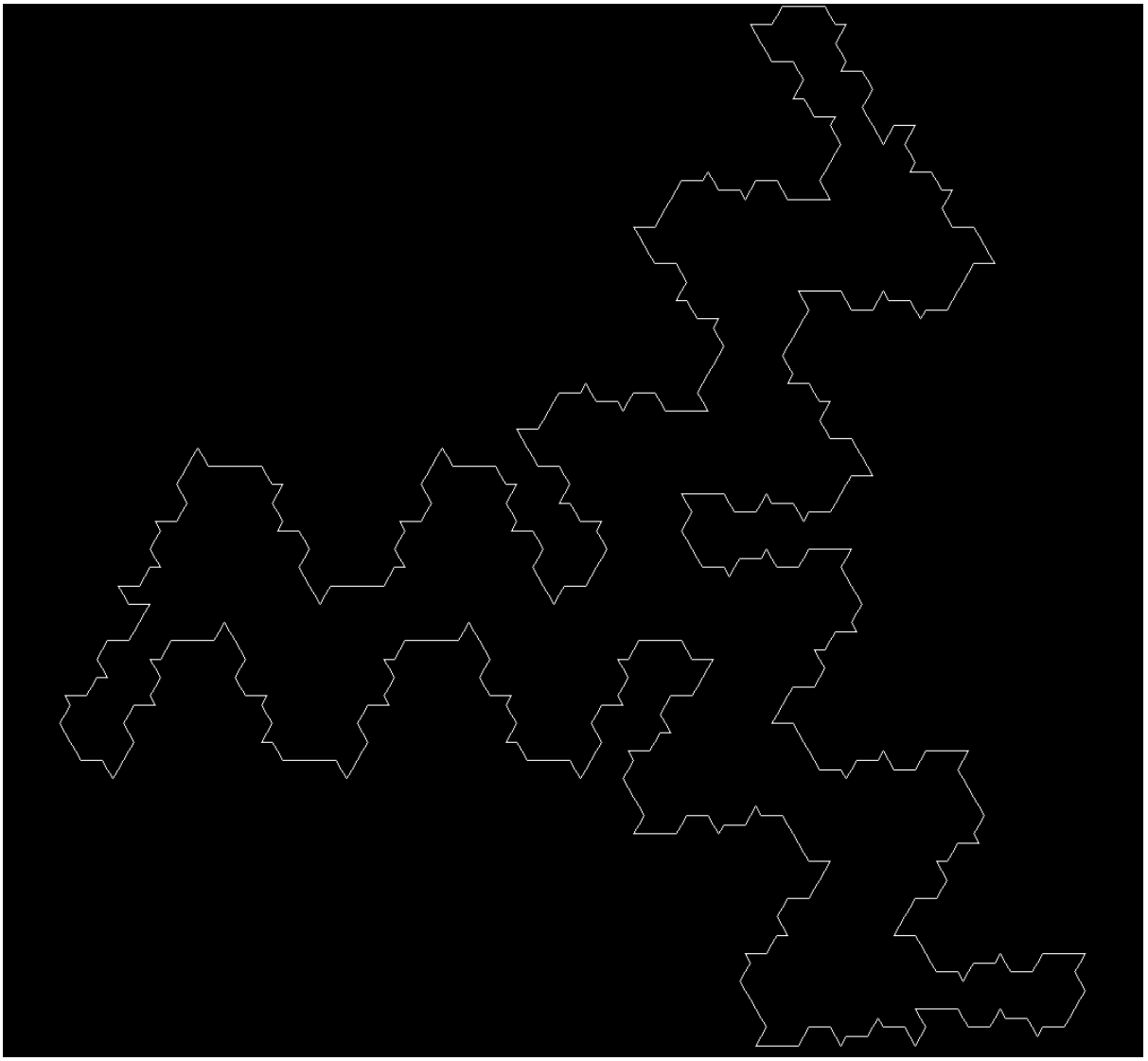

Figura 8: Estágio 3 do Preenchimento de Espaço de Leonardo

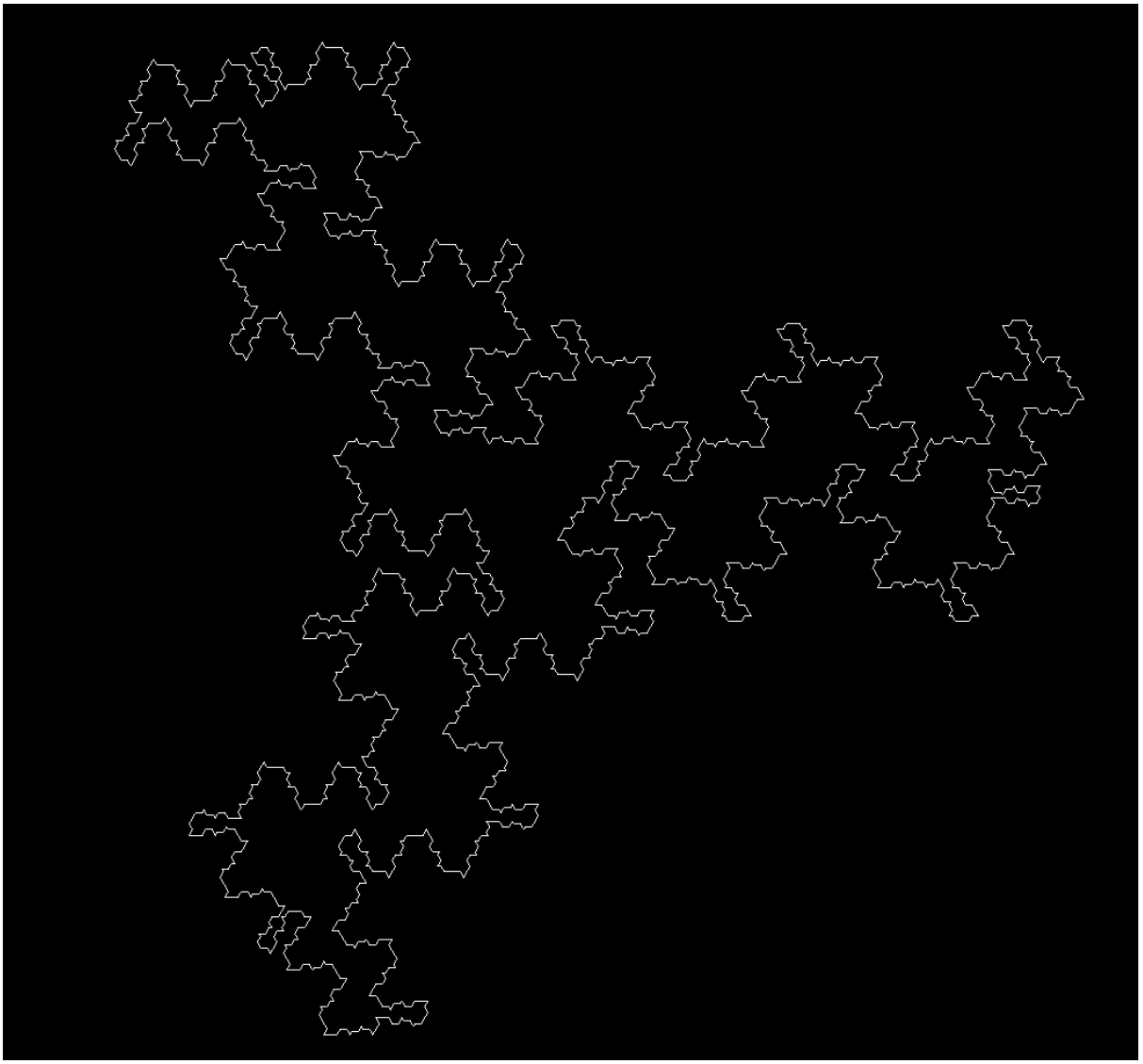

Figura 9: Estágio 4 do Preenchimento de Espaço de Leonardo