Ciclos Hamiltonianos

João Vitor Gonçalves
Paulo David

Sumário

Primeira versão do código

```c
#include <math.h>
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>

#include <gsl/gsl_blas.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_matrix.h>

#define MAX_TEXT_SIZE 255

bool is_string_valid (char *);
bool is_consonant (char );
void print_matrix (gsl_matrix *, char *);
void encryption (char *);
void decryption ();

// declaring global variables
int n = 0;
int s = 0;
gsl_matrix * A;
gsl_matrix * B;
gsl_matrix * N;
gsl_matrix * K;
gsl_matrix * Q;
gsl_matrix * C1;
gsl_matrix * C2;
```

```c
int main()
{
    char plain_text[MAX_TEXT_SIZE];

    // Gets the plain text
    printf("Type the message to be ciphered\n");
    fgets(plain_text, MAX_TEXT_SIZE, stdin);

    // Check if the plain text is valid
    if (!is_string_valid(plain_text)){
        printf("The plain text only allows letters A through Z, space and dot\n");
        return 1;
    }

    encryption(plain_text);

    gsl_matrix_set_all(C1, 0);
    gsl_matrix_set_all(N, 0);
    gsl_matrix_set_all(B, 0);

    decryption();

    gsl_matrix_free(A);
    gsl_matrix_free(B);
    gsl_matrix_free(N);
    gsl_matrix_free(K);
    gsl_matrix_free(C1);
    gsl_matrix_free(Q);
    gsl_matrix_free(C2);
    return 0;
}
```

```c
bool is_string_valid (char * plain_text){

    for (char *sub_string = plain_text; sub_string[0] != '\n'; sub_string++){
        if(!isalpha(sub_string[0]) && sub_string[0] != ' ' && sub_string[0] != '.'){
            return false;
        }

        // Turn to upperCase every letter
        sub_string[0] = toupper(sub_string[0]);
    }
    return true;
}


bool is_consonant (char letter){

    for (char* vowels= "AEIOU"; vowels[0] != '\0'; vowels++)
        if(vowels[0] == letter)
            return false;

    return true;
}

void print_matrix (gsl_matrix * matrix, char * title){

    printf("\n\n %s\n", title);

    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            printf("%g ", gsl_matrix_get(matrix, i, j));
        }
        printf("\n");
    }
}
```

	0	1	2	3	4	5	6
7	A	B	C	D	E	F	G
8	H	I	J	K	L	M	N
9	O	P	Q	R	S	T	U
10	V	W	X	Y	Z	space	dot

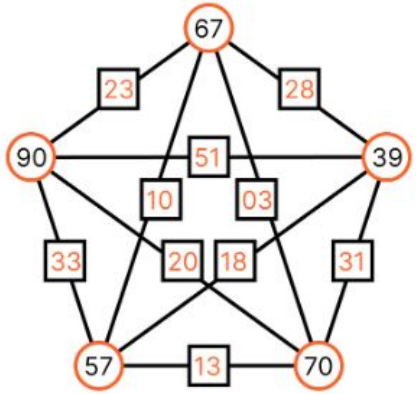
```c
void encryption (char plain_text[MAX_TEXT_SIZE]){

    // Translate the plain_text to an string of numbers
    int coded_plain_text[MAX_TEXT_SIZE];
    for ( ;n < MAX_TEXT_SIZE && plain_text[n] != '\n'; n++){

        char letter = plain_text[n];

        // Handle corner cases when the character is space or an dot
        if(letter == ' '){
            coded_plain_text[n] = 105;
            continue;
        } else if(letter == '.'){
            coded_plain_text[n] = 106;
            continue;
        }

        // Get the position of the letter in the alphabet
        int letter_num = letter- 64;

        // Calculate the row and column of the letter according to the especified table
        int row = ceil((double)letter_num /7) +6;
        int column = letter_num -(row -7)*7 -1;

        // Set the final value of the translation [A-Z] -> number
        if (is_consonant(letter)){
            coded_plain_text[n] = column*10 + row;
        } else{
            coded_plain_text[n] = row*10 + column;
        }
    }
```

	0	1	2	3	4	5	6
7	A	B	C	D	E	F	G
8	H	I	J	K	L	M	N
9	O	P	Q	R	S	T	U
10	V	W	X	Y	Z	space	dot

Grafo => G = 67, R = 39, A = 70, F = 57, O = 90

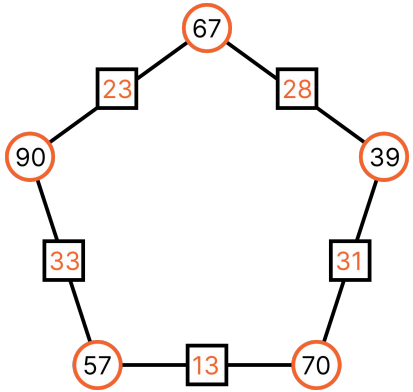
```c
// Check if the message has 3 characters minimum
if (n < 3){
    printf("The plain text should have at least 3 characters\n");
    exit(2);
}

// Generate matrix A
A = gsl_matrix_alloc(n, n);

for(int i = 0; i < n; i++){
    for(int j = 0; j < i; j++){
        int value = abs(coded_plain_text[i]-coded_plain_text[j]);
        gsl_matrix_set(A, i, j, value);
        gsl_matrix_set(A, j, i, value);
    }
}

print_matrix(A, "matriz A >>>");

// Calculate the matrix B
B = gsl_matrix_alloc(n, n);

for(int i = 0; i < n; i++){
    int y = i-1;
    int x = i+1;

    if (y == -1){
        y = n -1;
    } else if ( x == n ){
        x = 0;
    }

    gsl_matrix_set(B, i, y, gsl_matrix_get(A, i, y));
    gsl_matrix_set(B, i, i, plain_text[i]);
    gsl_matrix_set(B, i, x, gsl_matrix_get(A, i, x));
}

print_matrix(B, "matriz B >>>");
```

Grafo => G = 67, R = 39, A = 70, F = 57, O = 90



A =

	G	R	A	F	O
G	0	28	3	10	23
R	28	0	31	18	51
A	3	31	0	13	20
F	10	18	13	0	33
O	23	51	20	33	0



B* =

	G	R	A	F	O
G	7	28	0	0	23
R	28	18	31	0	0
A	0	31	1	13	0
F	0	0	13	6	33
O	23	0	0	33	15

```
// Calculate the matrix N
N = gsl_matrix_alloc(n, n);
gsl_blas_dsymm(CblasLeft, CblasUpper, 1.0, A, B, 0.0, N);

print_matrix(N, "N =>>>");

// Calculate the key matrix K
K = gsl_matrix_alloc(n, n);

for (int i = 0; i < n; i++){
    for (int j = 0, max = n -i; j < max; j++){
        gsl_matrix_set(K, i, i+j, j+1);
    }
}

print_matrix(K, "matriz K >>>");

// Calculate the matrix C1
C1 = gsl_matrix_alloc(n, n);
gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, N, K, 0.0, C1);


print_matrix(C1, "matriz C1 >>>");
```

1313	597	1001	858	675
1369	1745	265	2194	2003
1349	642	1130	738	798
1333	1007	571	1258	725
1589	2182	2030	458	1618

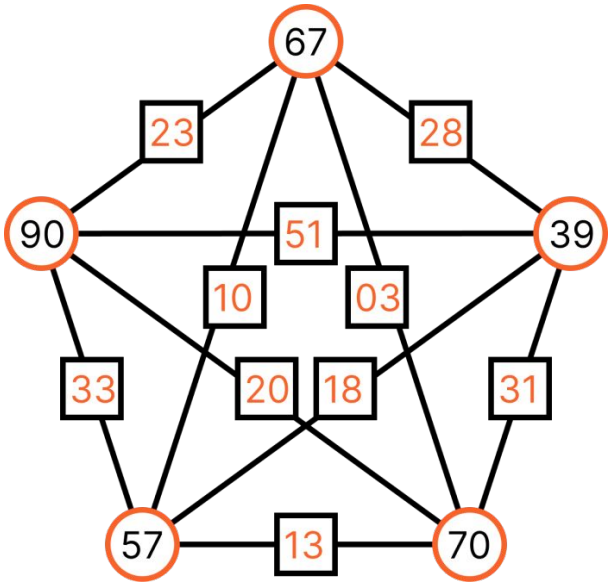
K =

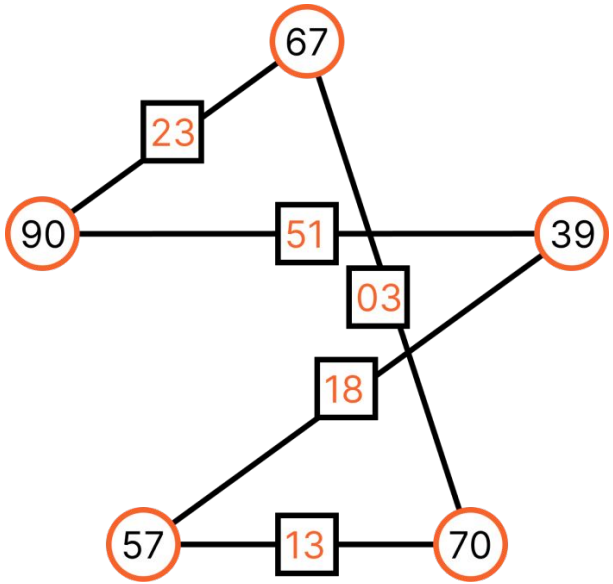
1	2	3	4	5
0	1	2	3	4
0	0	1	2	3
0	0	0	1	2
0	0	0	0	1

1313	3223	6134	9903	14347
1369	4483	7862	13435	21011
1349	3340	6461	10320	14977
1333	3673	6584	10753	15647
1589	5360	11161	17420	25297

```
// Calculate S
int hamilton_cycle[n], index = 0;
hamilton_cycle[0] = 0;

while (index < n-1){

    int vertexes_weight[n];
    int idx_vertex_lowest_weight = -1;

    // Calculate the weight of the direct transitive closure from the last vertex in hamilton_cycle
    for (int i = 0; i < n; i++){

        // The matrix diagonal is discarted
        if (hamilton_cycle[index] == i){
            vertexes_weight[i] = -1;
            continue;
        }

        vertexes_weight[i] = gsl_matrix_get(A, hamilton_cycle[index],i);
        if (idx_vertex_lowest_weight == -1 || vertexes_weight[idx_vertex_lowest_weight] > vertexes_weight[i]){
            idx_vertex_lowest_weight = i;
        }
    }
```

```
    // Ensure the vertex is not repeated in the hamilton_cycle
    while(true){
        bool is_repeated = false;

        for (int i = 0; i <= index; i++){
            if (hamilton_cycle[i] == idx_vertex_lowest_weight ){
                is_repeated = true;
                break;
            }
        }

        if (!is_repeated)
            break;

        // Discards idx_vertex_lowest_weight in vertexes_weight because it is repeated in hamilton_cycle
        vertexes_weight[idx_vertex_lowest_weight] = -1;
        idx_vertex_lowest_weight = -1;

        // Finds the next vertex with the lowest weight
        for (int i = 0; i < n; i++)
            if ((idx_vertex_lowest_weight == -1 || vertexes_weight[idx_vertex_lowest_weight] > vertexes_weight[i]) && vertexes_weight[i] != -1)
                idx_vertex_lowest_weight = i;
    }

    s += vertexes_weight[idx_vertex_lowest_weight];         // Sum the vertex add in the hamilton cycle
    hamilton_cycle[++index] = idx_vertex_lowest_weight;    // add the Vertex to the hamilton cycle
}

s += gsl_matrix_get(A, 0, hamilton_cycle[index]);       // sum the vertex of the first and last vertex to calculate the final value of s
printf("\nvalor de s: %d\n",s);

```





S = 03+ 13+ 18+ 51+ 23 = 108

```c
    // Calculate C2 and Q
    C2 = gsl_matrix_alloc(n, n);
    Q  = gsl_matrix_alloc(n, n);

    printf("\nCifra final em forma linear: ");


    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            int C1_value = gsl_matrix_get(C1, i, j);
            gsl_matrix_set(C2, i, j, C1_value % s);

            printf("%g", gsl_matrix_get(C2, i, j));         // Final cypher printed in linear form

            gsl_matrix_set( Q, i, j, floor(C1_value/s));
        }
    }

    print_matrix(C2, "matriz C2 >>>");
    print_matrix(Q, "matriz Q >>>");
}

```

$C2 = C1 \bmod S =$

1313	3223	6134	9903	14347
1369	4483	7862	13435	21011
1349	3340	6461	10320	14977
1333	3673	6584	10753	15647
1589	5360	11161	17420	25297

$\bmod 108 =$

17	91	86	75	91
73	55	86	43	59
53	100	89	60	73
37	1	104	61	95
77	68	37	32	25

1791867591.7355864359.53100896073.3711046195.7768373225

$Q = \lfloor C1 / S \rfloor =$

$\left\lfloor$

1313	3223	6134	9903	14347
1369	4483	7862	13435	21011
1349	3340	6461	10320	14977
1333	3673	6584	10753	15647
1589	5360	11161	17420	25297

$\right\rfloor / 108 =$

12	29	56	91	132
12	41	72	124	194
12	30	59	95	138
12	34	60	99	144
14	49	103	161	234

Usado para a descriptografia

```c
void decryption (){

    // Obtein matrix C1
    C1 = gsl_matrix_alloc(n, n);

    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            int remainder = gsl_matrix_get(C2, i, j);
            int quocient = gsl_matrix_get(Q, i, j);
            gsl_matrix_set(C1, i, j, quocient *s + remainder);
        }
    }

    print_matrix(C1, "matrix C1 reconstruída >>>");

    // Calculate the inverse matrix of K
    gsl_matrix * inverse_K = gsl_matrix_alloc(n, n);
    for(int i = 0; i < n; i++){
        gsl_matrix_set(inverse_K, i, i, 1);
        if (i+1 < n){
            gsl_matrix_set(inverse_K, i, i+1, -2);
        }
        if (i+2 < n){
            gsl_matrix_set(inverse_K, i, i+2, 1);
        }
    }


    print_matrix(inverse_K, "inverso da matriz K >>>");
```

$$[Q]ij \times S + [C2]ij = C1$$

$K =$

1	2	3	4	5
0	1	2	3	4
0	0	1	2	3
0	0	0	1	2
0	0	0	0	1

$K^{-1} =$

1	-2	1	0	0
0	1	-2	1	0
0	0	1	-2	1
0	0	0	1	-2
0	0	0	0	1

```
1
2  // Calculate the matrix N
3  gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, C1, inverse_K, 0.0, N);
4
5  print_matrix(N, "matriz N reconstruída");
6
7  // Calculate the inverse matrix of A
8  gsl_matrix * inverse_A = gsl_matrix_alloc(n, n);
9
10 int signum;
11 gsl_permutation * p = gsl_permutation_alloc(n);
12
13 gsl_linalg_LU_decomp (A, p, &signum);
14 gsl_linalg_LU_invert (A, p, inverse_A);
15
16 gsl_permutation_free(p);
17
18 print_matrix(inverse_A, "inverso da matriz A >>>");
```

Calcular a matriz N multiplicando C1 e K^{-1}.

1313	3223	6134	9903	14347
1369	4483	7862	13435	21011
1349	3340	6461	10320	14977
1333	3673	6584	10753	15647
1589	5360	11161	17420	25297

X

1	-2	1	0	0
0	1	-2	1	0
0	0	1	-2	1
0	0	0	1	-2
0	0	0	0	1

=

1313	597	1001	858	675
1369	1745	265	2194	2003
1349	642	1130	738	798
1333	1007	571	1258	725
1589	2182	2030	458	1618

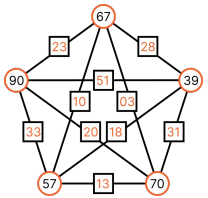
Calcular o inverso da matriz A.

A =

0	28	3	10	23
28	0	31	18	51
3	31	0	13	20
10	18	13	0	33
23	51	20	33	0

A^{-1} =

-2.17E-01	-9.11E-18	1.67E-01	5.00E-02	3.97E-18
-6.53E-18	-1.80E-02	1.31E-17	2.78E-02	9.80E-03
1.67E-01	8.22E-18	-1.92E-01	-2.30E-17	2.50E-02
5.00E-02	2.78E-02	-2.07E-17	-7.78E-02	3.96E-18
7.40E-18	9.80E-03	2.50E-02	-1.02E-33	-1.52E-02



```c

    // Calculate the matrix B
    gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, inverse_A, N, 0.0, B);

    print_matrix(B, "matriz B reconstruída");

    printf("\nMensagem decodificada: ");

    // Print the decoded message
    for(int i = 0; i < n; i++)
        printf("%c", (int)round(gsl_matrix_get(B, i, i)));


    gsl_matrix_free(inverse_K);
    gsl_matrix_free(inverse_A);
}


```

Calcular a matriz B multiplicando N e A^{-1}.

1313	597	1001	858	675
1369	1745	265	2194	2003
1349	642	1130	738	798
1333	1007	571	1258	725
1589	2182	2030	458	1618

X

-2.17E-01	-9.11E-18	1.67E-01	5.00E-02	3.97E-18
-6.53E-18	-1.80E-02	1.31E-17	2.78E-02	9.80E-03
1.67E-01	8.22E-18	-1.92E-01	-2.30E-17	2.50E-02
5.00E-02	2.78E-02	-2.07E-17	-7.78E-02	3.96E-18
7.40E-18	9.80E-03	2.50E-02	-1.02E-33	-1.52E-02

=

7	28	0	0	23
28	18	31	0	0
0	31	1	13	0
0	0	13	6	33
23	0	0	33	15

7 18 1 6 15
G R A F O

ERROR: matrix is singular

O que é uma matriz singular

Uma matriz é singular se seu determinante for zero ,e pela definição A^-1 = adj(A) / det(A), não é possível calcular a inversa de tal matriz.

Usando a propriedade do determinante zero (se o determinante de uma matriz é zero, pelo menos uma linha/coluna dessa matriz é um múltiplo escalar de outra), podemos concluir que se garantirmos que nenhuma linha/coluna seja um múltiplo escalar de outra na matriz, você pode calcular sua inversa.

Exemplos de matriz singular

1	2	3
2	4	6
9	1	1

1	2	1
2	4	1
3	6	1

	G	R	A	F	O	O
G	0	28	3	10	23	23
R	28	0	31	18	51	51
A	3	31	0	13	20	20
F	10	18	13	0	33	33
O	23	51	20	33	0	0
O	23	51	20	33	0	0

0	0	0	0
3	2	7	9
1	2	1	2
9	9	3	3

Segunda versão do código

```
char non_repeated_str[29];
int repeated_str[504];

int nr_size = 0;
int  r_size = 0;
for ( ;n < MAX_TEXT_SIZE && plain_text[n] != '\n'; n++){
    bool is_repeated = false;
    for(int j = 0; j < nr_size; j++){
        if(non_repeated_str[j] == plain_text[n]){
            is_repeated = true;
            break;
        }
    }

    if (is_repeated){
        repeated_str[r_size] = n;
        repeated_str[r_size +1] = plain_text[n] +n;
        r_size += 2;
    } else {
        non_repeated_str[nr_size] = plain_text[n];
        nr_size++;
    }
}
non_repeated_str[nr_size] = '\n';
n = nr_size;

encryption(non_repeated_str);

gsl_matrix_set_all(C1, 0);
gsl_matrix_set_all(N, 0);
gsl_matrix_set_all(B, 0);

decryption(repeated_str, r_size/2);

```

```c
void encryption (char plain_text[MAX_TEXT_SIZE]){

    // Check if the message has 3 distinc characters minimum
    if (n < 3){
        printf("The plain text should have at least 3 distinc characters\n");
        exit(2);
    }

    // Translate the plain_text to an string of numbers
    int coded_plain_text[MAX_TEXT_SIZE];
    for(int i = 0; i < n; i++){

        char letter = plain_text[i];

        // Handle corner cases when the character is space or an dot
        if(letter == ' '){
            coded_plain_text[i] = 105;
            continue;
        } else if(letter == '.'){
            coded_plain_text[i] = 106;
            continue;
        }

        // Get the position of the letter in the alphabet
        int letter_num = letter- 64;

        // Calculate the row and column of the letter according to the especified table
        int row = ceil((double)letter_num /7) +6;
        int column = letter_num -(row -7)*7 -1;

        // Set the final value of the translation [A-Z] -> number
        if (is_consonant(letter)){
            coded_plain_text[i] = column*10 + row;
        } else{
            coded_plain_text[i] = row*10 + column;
        }
    }
```

```c
printf("\nMensagem decodificada: ");

int count_repeated_letters = 0;
// Print the decoded message
for(int i = 0, max = n + size; i < max; i++){

    int idx_value = repeated_letters[count_repeated_letters*2];
    if (i == idx_value){
        printf("%c", repeated_letters[(count_repeated_letters*2)+1] -idx_value);
        count_repeated_letters++;
    }
    else
        printf("%c", (int)round(gsl_matrix_get(B, i-count_repeated_letters, i-count_repeated_letters)));
}
```

Terceira versão do código

```
// Generate matrix A
A = gsl_matrix_alloc(n, n);

for(int i = 0; i < n; i++){
    for(int j = 0; j < i; j++){
        int value = abs(coded_plain_text[i]-coded_plain_text[j]);
        gsl_matrix_set(A, i, j, value);
        gsl_matrix_set(A, j, i, value);
    }

    gsl_matrix_set(A, i, i, PRIMES[i]);
}

print_matrix(A, "matriz A >>>");
```

Referências

- Kumar Gurjar, D. & Krishnaa, A. (2021). Complete Graph and Hamiltonian Cycle in Encryption and Decryption. International Journal of Mathematics Trends and Technology, 67(12), 62-71.
- https://www.gnu.org/software/gsl/doc/html/blas
- https://en.wikipedia.org/wiki/Invertible_matrix
- https://www.cuemath.com/algebra/singular-matrix/