```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

typedef struct Grafo {
  int v; // número de vertices
  double **mat; // matriz de adjacência
} Grafo;

typedef struct Dijkstra { // Representa a tabela usada no Algoritmo de Dijkstra
    double e; // Estimativa
    int p; // Precedente
    int status; // Status: aberto ou fechado
}Dijkstra;

typedef struct Pilha { // Pilha utilizada para armazenar os vértices que levam ao menor caminho encontrado
    int *caminho; // lista com os índices que levam ao menor caminho
    int tamanho; //tamanho da pilha
}Pilha;

Grafo *criaGrafo(int vertices){
    int i;

    Grafo *g = malloc(sizeof(Grafo));
    g->v = vertices;

    g->mat = calloc(g->v, sizeof(double *));
    for (i = 0; i < g->v; ++i)
        g->mat[i] = calloc(g->v, sizeof (double));
    return g;
}

void freeGrafo(Grafo *g){
    int i;
    for (i = 0; i < g->v; ++i)
        free(g->mat[i]);
    free(g->mat);
}

void insereAresta(Grafo *g, int v1, int v2){
    int linha1, linha2, v;

    if(v1 >= 0 && v2 >= 0 && v1 < g->v && v2 < g->v && v1 != v2){ // Verifica se os valores das arestas de
entrada excedem os limites do grafo
        if(v1 > v2){ // Define v1 sempre menor que v2
            v = v1;
            v1 = v2;
            v2 = v;
        }
        if(g->v <= 4){ //Até 4 arestas todos podem se conectar
            if( (v1 == 1 && v2 == 2) || (v1 == 0 && v2 == 3) ){  //verifica se é uma diagonal
                g->mat[v1][v2] = sqrt(2);
                g->mat[v2][v1] = sqrt(2);
            }
            else{
                g->mat[v1][v2] = 1;
                g->mat[v2][v1] = 1;
            }
        }
        else{//Nem todos podem se conectar
            linha1 = ceil((float)g->v/2) - 1; //até onde vai a linha 1

            if(v2 == v1+1 && v2 <= linha1){ // Se v1 e v2 estiverem na primeira linha um do lado do outro,
então podem se conectar
                g->mat[v1][v2] = 1;
                g->mat[v2][v1] = 1;
            }
            else if(v2 == v1+1 && v1 > linha1 && v2 < g->v){ // Se v1 e v2 estiverem na segunda linha um do
lado do outro, então podem se conectar
                g->mat[v1][v2] = 1;
                g->mat[v2][v1] = 1;
            }
```

```
            else if(v2 == v1+linha1+1 && v2 < g->v){ // Se v1 é pralelo a v2 e v2 não excede o limite de vértices, então podem se conectar
                g->mat[v1][v2] = 1;
                g->mat[v2][v1] = 1;
            }
            else if( (v2 == v1+linha1+2 && v2 < g->v) || (v2 == v1+linha1 && v2 > linha1) ){ // Se a diagonal for possível, então podem se conectar
                g->mat[v1][v2] = sqrt(2);
                g->mat[v2][v1] = sqrt(2);
            }
            else
                printf("Valor de entrada incorreto: os vertices %d %d escolhidos nao podem se conectar.\n\n", v1, v2);
        }
    }
    else{
        printf("Valores de entrada incorretos, possiveis erros:\n");
        printf("  - Aresta fora dos limites do grafo.\n");
        printf("  - Aresta ligando vertices iguais.\n");
        printf("  - O vertice %d nao se liga com o vertice %d por uma aresta.\n\n", v1, v2);
    }
}

void removeAresta(Grafo *g, int v1, int v2){ // Remove uma única aresta
    g->mat[v1][v2] = 0;
    g->mat[v2][v1] = 0;
}

void insereArestas(Grafo *g, int vI, int vF){ // Insere mais de uma aresta
    int i;
    for(i=vI; i<vF; i++){
        insereAresta(g, i, i+1);
    }
}

void removeArestas(Grafo *g, int vI, int vF){ // Remove mais de uma aresta
    int i;
    for(i=vI; i<vF; i++){
        removeAresta(g, i, i+1);
    }
}

void imprimiMatriz(Grafo *g){
    int i, j;
    for(i=0; i < g->v; i++){
        for(j=0; j < g->v; j++){
            printf("%.1lf ", g->mat[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void imprimiGrafo(Grafo *g){ // Imprimi o grafo na tela
    int i, j;
    int colunas = ceil((float)g->v/2); // Armazena o número de colunas do grafo

    for(i=0; i < g->v; i++){
        if(i == colunas){
            printf("\n");
            for(j=0; j < (g->v-colunas); j++){
                if(g->mat[j][j+colunas] == 1) // Verifica se a aresta é na vertical
                    printf(" | ", j);
                else
                    printf("   ");
                if(j+colunas+1 < g->v && g->mat[j][j+colunas+1] == sqrt(2) && g->mat[j+1][j+colunas] == sqrt(2)) // Verifica se a aresta é nas duas diagonais
                    printf("X");
                else
                    if(j+colunas+1 < g->v && g->mat[j][j+colunas+1] == sqrt(2)) // Verifica se a aresta é na diagonal
                        printf("\\");
                    else
```

```
                        if(g->mat[j+1][j+colunas] == sqrt(2) && j+1 < colunas) // Verifica se a aresta é na
outra diagonal
                            printf("/");
                        else
                            printf(" ");
                }
                printf("\n");
            }

            printf(" o "); // Imprimi na tela o vértice representado pela letra "o"

            if(g->mat[i][i+1] == 1 && i != colunas-1) // Verifica se a aresta é na horizontal
                printf("-");
            else
                printf(" ");
        }
        printf("\n\n");
    }

    Dijkstra *criaDijkstra(int nVertices){
        int i;
        Dijkstra *d;
        d = calloc(nVertices, sizeof(Dijkstra));
        for (i = 0; i < nVertices; i++){
            d[i].e = 0;
            d[i].p = -1;
            d[i].status = 0;
        }
        return d;
    }

    void imprimiDijkstra(Dijkstra *d, int nVertices){ // Imprimi a tabela de Dijkstra na tela
        int i;

        printf("Vertices:    ");
        for(i=0; i < nVertices; i++){
            printf("%d    ", i);
        }
        printf("\n");

        printf("Estimativas: ");
        for(i=0; i < nVertices; i++){
            printf("%.2lf ", d[i].e);
        }
        printf("\n");

        printf("Precedentes: ");
        for(i=0; i < nVertices; i++){
            if(d[i].p == -1)
                printf("%d   ", d[i].p);
            else
                printf("%d    ", d[i].p);
        }
        printf("\n");

        printf("Fechado:     ");
        for(i=0; i < nVertices; i++){
            printf("%d    ", d[i].status);
        }
        printf("\n");

    }

    Grafo *execDijkstra(Grafo *g, Dijkstra *d, int vI, int vF){
        Grafo *grafoDijkstra;
        grafoDijkstra = criaGrafo(g->v);
        int i, menorPeso=g->v, termino=0, vAtual;

        d[vI].e = 0;
        d[vI].p = vI;
        d[vI].status = 0;
        vAtual = vI;
```

```
    while(termino == 0){// Caso o valor da variável "termino" seja modificado o grafo com a resposta está
pronto para ser retornado

        for(i=0; i < g->v; i++){ //mostra as opções e adiciona na tabela dijkstra
            if(g->mat[vAtual][i] != 0 && vAtual != i ){
                if( (d[i].status == 0 && d[i].p == -1) || (d[i].status == 1 && g->mat[vAtual][i] +
d[vAtual].e < d[i].e) ){
                    d[i].e = g->mat[vAtual][i] + d[vAtual].e;
                    d[i].p = vAtual;
                    d[i].status = 1;
                }
            }
            if(d[i].status == 1){
                menorPeso = i;
            }
        }

        termino=1;
        for(i=0; i < g->v; i++){
            if(d[i].status == 1 && d[i].e < d[menorPeso].e){ // Procura pelo menor peso de aresta a ser
escolhido para fechá-lo
                menorPeso = i;
            }
            if(d[i].status == 1)
                termino = 0;
        }
        if(menorPeso != g->v){
            d[menorPeso].status = 0; // Fecha o menor peso
            insereAresta(grafoDijkstra, d[menorPeso].p, menorPeso); // Insere a aresta no grafo que
representa o resultado da aplicação do Algoritmo de Dijkstra
            vAtual = menorPeso;
        }
    }

    return grafoDijkstra;
}

// Gerenciamento da Pilha, utilizada para armazenar a ordem dos vértices que levam ao menor caminho
Pilha *criaPilha(){
    Pilha *pilha;
    pilha = malloc(sizeof(Pilha));
    pilha->caminho = malloc(sizeof(int));
    pilha->tamanho = 0;
    return pilha;
}
void push(Pilha *p, int valor){
    p->tamanho += 1;
    p->caminho = realloc(p->caminho, sizeof(int)*p->tamanho);
    p->caminho[p->tamanho-1] = valor;
}
void pop(Pilha *p){
    if(p->tamanho != 1)
        p->tamanho -= 1;
    else
        printf("Impossivel remover da pilha, tamanho: %d", p->tamanho);
}
void mostraPilha(Pilha *p){
    int i;
    for(i=0; i < p->tamanho; i++){
        printf("%d ", p->caminho[i]);
    }
    printf("\n\n");
}

Grafo *encontraMenorCaminho(Grafo *grafoDijkstra, int vI, int vF, Grafo *grafoOriginal, Pilha *pilha){
    //após a criação do dijkstra será necessário motrar o menor caminho entre vInicial e vFinal
    Grafo *grafoMenorCaminho;
    grafoMenorCaminho = criaGrafo(grafoDijkstra->v);  //inicializa o grafoMenorCaminho igual ao
grafoDijkstra
    int i, j, aux, achou=0;
    push(pilha, vI);

    do{
```

```
        aux=0; //Aux começa em 0, caso está variável não seja alterada o programa encerra
        for(i = 0; i < grafoDijkstra->v; i++){
            if(grafoDijkstra->mat[vI][i] > 0){
                if(achou != 1){ //Ao achar o vertice final a pilha não pode mais ser alterada pois contém o
menor caminho encontrado
                    push(pilha, i); //Insere o valor i na pilha, onde i se refere a um vertice do menor
caminho
                }
                if(i == vF){
                    achou=1; //Ao achar o vertice final a variavel achou muda para 1 indicando que a pilha
está pronta e não deve ser alterada
                }

                grafoDijkstra->mat[vI][i] *= -1; //Altera as arestas indentificando-as como visitadas
                grafoDijkstra->mat[i][vI] *= -1;
                vI = i;
                aux=1; //Modifica a aux para que o programa continue em loop pois ainda não encontrou o
menor caminho
                i=-1;
            }
        }
        for(i = 0; i < grafoDijkstra->v; i++){
            if(grafoDijkstra->mat[vI][i] < 0){ //Arestas visitadas que estão com um valor negativo devem
ser removidas pois não pertencem ao menor caminho
                removeAresta(grafoDijkstra, vI, i);
                vI = i;
                aux=1; //Modifica a aux para que o programa continue em loop pois ainda não encontrou o
menor caminho
                if(achou != 1)
                    pop(pilha); //Retira o vertice da pilha pois não pertence ao menor caminho
                break;
            }
        }
    }while(aux==1);

    for(i=0; i < grafoDijkstra->v; i++)
        for(j=0; j<grafoDijkstra->v; j++)
            if(grafoDijkstra->mat[i][j] < 0)
                grafoDijkstra->mat[i][j] *= -1; //Desfaz as alterações anteriormente feitas para
identificar uma aresta como visitada

    for(i=0; i<pilha->tamanho-1; i++)
        insereAresta(grafoMenorCaminho, pilha->caminho[i], pilha->caminho[i+1]); //Insere as arestas do
caminho encontrado no gragoMenorCaminho

    if(achou != 1){
        printf("Nao foi possivel encontrar um caminho:\n  - Os vertices nao se comunicam entre
arestas.\n\nGrafo Original:\n");
        imprimiGrafo(grafoOriginal);
    }

    return grafoMenorCaminho; //Retorna um grafo com o menor caminho encontrado
}

int calcManhattan(Grafo *g, int vI, int vF){
    int xVI, xVF, yVI, yVF, res;
    int linha1 = ceil((float)g->v/2);

    //Como o grafo é formado a partir de um mundo de grade 2 colunas por n linha, é preciso definir em que
linha e coluna o vértice se encontra
    //Um grafo com 10 vertices tem 5 linhas e 2 colunas, assim o último vértice se encontra na posição (x=1,
 y=2)
    if(vI < linha1){
        xVI = vI;
        yVI = 0;
    }
    else{
        xVI = vI - linha1;
        yVI = 1;
    }

    if(vF < linha1){
        xVF = vF;
```

```
        yVF = 0;
    }
    else{
        xVF = vF - linha1;
        yVF = 1;
    }

    res = (abs(xVI - xVF) + abs(yVI - yVF)); //Calcula a distância Manhattan

    return res;
}

double calcCustoArestas(Grafo *g){
    int i, j;
    double res=0;

    for(i=0; i<g->v; i++)
        for(j=0; j<g->v; j++)
            if(g->mat[i][j] > 0)
                res += g->mat[i][j]; //Soma o peso das arestas

    return (res/2); // Como a matriz não é ordenada os valores são duplicados e por isso é feita uma
divisão por 2
}


int main() {
    Grafo *grafo, *grafoDijkstra, *grafoMenorCaminho;
    Dijkstra *dijkstra;
    Pilha *pilha;
    int v1, v2, dM, vertices=0;
    double custoArestas;
    char c[20];

    printf("Comandos: \n");
    printf("  - show 0 0   -> Mostra o grafo criado e suas arestas.\n");
    printf("  - add v1 v2  -> Adiciona uma aresta entre dois vertices.\n");
    printf("  - adds v1 v2 -> Adiciona mais de uma aresta entre dois vertices.\n");
    printf("  - rmv v1 v2  -> Remove uma aresta entre dois vertices.\n");
    printf("  - rmvs v1 v2 -> Remove mais de uma aresta entre dois vertices.\n");
    printf("  - show 0 0   -> Mostra o grafo criado e suas arestas.\n");
    printf("  - find v1 v2 -> Encontra o menor caminho entre dois vertices.\n");
    printf("  - end 0 0    -> Encerra o programa.\n\n");

    while(vertices == 0){ // Enquanto o valor digitado for inválido, no caso maior que 20
        printf("Para inicializar digite o numero de vertices: ");
        scanf("%d", &vertices);
        if(vertices > 20){
            printf("Aviso:\n - Valor maximo possivel de 20 arestas!\n\n");
            vertices=0;
        }
    }

    grafo = criaGrafo(vertices); //cria o grafo com n vertices sem arestas -> comandos para inserir arestas
    dijkstra = criaDijkstra(vertices); // comando para criar a tabela dijkstra
    grafoDijkstra = criaGrafo(vertices); // Inicializa o grafo que armazena o resultado do Algoritmo de
Dijkstra
    grafoMenorCaminho = criaGrafo(vertices); // Inicializa o grafo que armazena o menor caminho encontrado
    pilha = criaPilha();

    do{
        printf("Digite um comando: ");
        scanf("%s %d %d", c, &v1, &v2);

        if(strcmp(c, "adds") == 0){
            insereArestas(grafo, v1, v2);
        }
        else if(strcmp(c, "add") == 0){
            insereAresta(grafo, v1, v2);
        }
        else if(strcmp(c, "rmvs") == 0){
            removeArestas(grafo, v1, v2);
        }
```

```
        else if(strcmp(c, "rmv") == 0){
            removeAresta(grafo, v1, v2);
        }
        else if(strcmp(c, "show") == 0)
            imprimiGrafo(grafo);
        else if(strcmp(c, "find") == 0){
            dijkstra = criaDijkstra(vertices);
            grafoDijkstra = execDijkstra(grafo, dijkstra, v1, v2); //comando que executa a heuristica
dijkstra
            pilha = criaPilha();
            grafoMenorCaminho = encontraMenorCaminho(grafoDijkstra, v1, v2, grafo, pilha); //comando que
acha o menor caminho no grafo dijkstra
            //Grafo Original sendo mostrado na tela
            printf("\nMatriz de adjacencia do grafo original: \n");
            imprimiMatriz(grafo);
            printf("Grafo Original:\n");
            imprimiGrafo(grafo);
            //Grafo Menor Caminho sendo mostrado na tela
            printf("Matriz de adjacencia do menor caminho: \n");
            imprimiMatriz(grafoMenorCaminho);
            printf("Grafo com o menor Caminho:\n");
            imprimiGrafo(grafoMenorCaminho);
            /*
            printf("Matriz de adjacencia do Algoritmo de Dijkstra: \n");
            imprimiMatriz(grafoDijkstra);
            printf("Grafo com o Algoritmo de Dijkstra:\n");
            imprimiGrafo(grafoDijkstra);
            */
            printf("Nome de cada vertice percorrido, na sequencia certa, indicando o caminho: ");
            mostraPilha(pilha);
            custoArestas = calcCustoArestas(grafoMenorCaminho);
            printf("Distancia efetivamente percorrida por um agente, do vertice inicial ao vertice final:
%.2lf\n", custoArestas);
            dM = calcManhattan(grafo, v1, v2);
            printf("\nDist. Manhattan: %d\n\n", dM);
            printf("Algoritmo usado:\n");
            printf("  - Algoritmo de Dijkstra para achar os caminhos de custo minimo partindo de um vertice
inicial.\n");
            printf("  - Busca em profundidade para extrair o menor caminho.\n\n");
        }
        else if(strcmp(c, "end") == 0){
            freeGrafo(grafo);
            freeGrafo(grafoDijkstra);
            freeGrafo(grafoMenorCaminho);
            free(pilha->caminho);
            free(dijkstra);
            break;
        }
        else
            printf("Comando invalido\n");
    }while(1);


  return 0;
}
```