Aluno(a):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Matrícula:\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| 1a) | 2,5 |  |
| 2a) | 2,5 |  |
| 3a) | 2,5 |  |
| 4a) | 2,5 |  |
|  | 10,0 |  |

**LEIA COM CUIDADO**

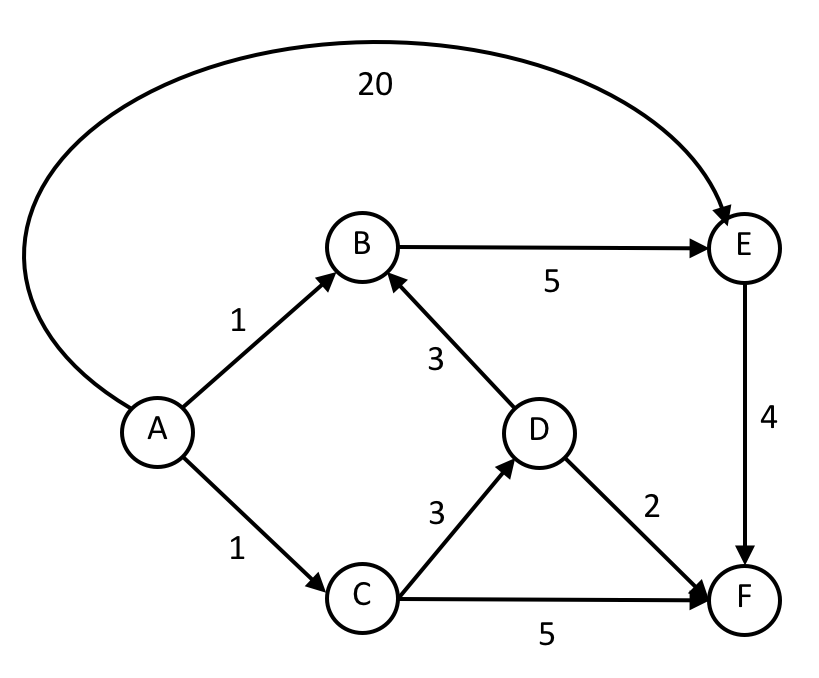
1. A prova é individual e sem consulta.
   1. **Qualquer tentativa de “cola” resultará na anulação da prova do aluno ou dos alunos envolvidos**.
   2. Os aparelhos celulares deverão permanecer desligados e guardados fora do alcance durante toda a prova. **Aparelhos celulares ligados ou de alguma forma visíveis serão tratados como tentativa de “cola”**.
2. A interpretação faz parte da questão.
   1. **Não há perguntas durante a prova.**
   2. Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
3. O tempo de prova é 1:45 h.
4. **Após o início da prova, não será possível sair e voltar à sala.**
5. As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
6. A prova pode ser feita a lápis.

**Questão 1 (2,5 pontos)** Considere o grafo dirigido ponderado da Figura 1.

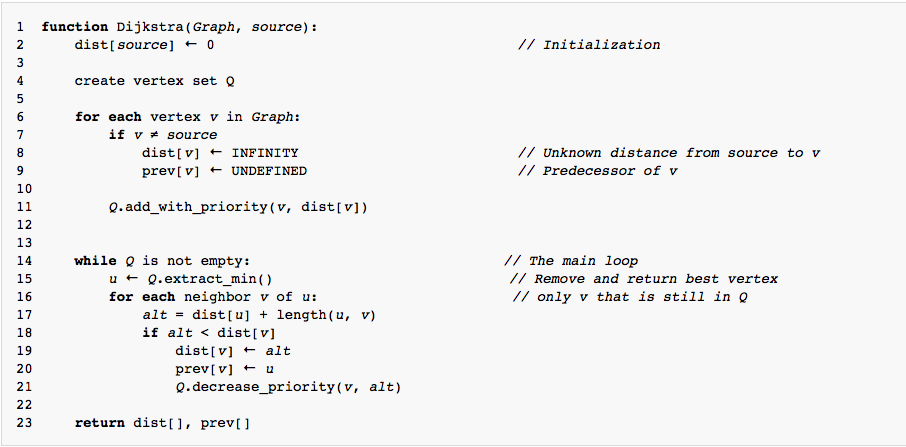
(a) **(1,5 pontos)** Usando o pseudo-código do Algoritmo de Dijkstra apresentado na Figura 2, mostre esquematicamente todos os valores da variável u e dos vetores dist e prev durante a computação do caminho mínimo de A a F para o grafo da Figura 1. Use a Tabela 1 como base para a sua resposta.

(b) **(0,5 ponto)** Mostre como reconstruir o caminho mínimo de A a F para o grafo da Figura 1 a partir das estruturas computadas no item (a).

(c) **(0,5 ponto)** Se acrescentarmos o arco (D,A), com peso -5, o que acontecerá com o algoritmo de Dijkstra?

****

**Figura 1.** Grafo dirigido.



**Figura 2.** Pseudo-código do Algoritmo de Dijkstra.

***Resposta***

a)

**Tabela 1. Estruturas de dados para o Algoritmo de Dijkstra.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | u | dist | prev |  | u | dist | prev |  | u | dist | prev |  | u | dist | prev |  | u | dist | prev |
| A | x |  | 0 |  | X | A | 0 |  | X |  | 0 |  | X |  | 0 |  | X |  | 0 |  |
| B |  |  | ∞ |  |  |  | 1 | A | X | B | 1 | A | X |  | 1 | A | X |  | 1 | A |
| C |  |  | ∞ |  |  |  | 1 | A |  |  | 1 | A | X | C | 1 | A | X |  | 1 | A |
| D |  |  | ∞ |  |  |  | ∞ |  |  |  | ∞ |  |  |  | 4 | C | X | D | 4 | C |
| E |  |  | ∞ |  |  |  | 20 | A |  |  | 6 | B |  |  | 6 | B |  |  | 6 | B |
| F |  |  | ∞ |  |  |  | ∞ |  |  |  | ∞ |  |  |  | 6 | C |  |  | 6 | C |

Inicialização Passo: 1 Passo: 2 Passo: 3 Passo: 4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | u | dist | prev |  | u | dist | prev |  | u | dist | prev |  | u | dist | prev |  | u | dist | prev |
| A | X |  | 0 |  | X |  | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | X |  | 1 | A | X |  | 1 | A |  |  |  |  |  |  |  |  |  |  |  |  |
| C | X |  | 1 | A | X |  | 1 | A |  |  |  |  |  |  |  |  |  |  |  |  |
| D | X |  | 4 | C | X |  | 4 | C |  |  |  |  |  |  |  |  |  |  |  |  |
| E | X | E | 6 | B | X |  | 6 | B |  |  |  |  |  |  |  |  |  |  |  |  |
| F |  |  | 6 | C | X | F | 6 | C |  |  |  |  |  |  |  |  |  |  |  |  |

Passo: 5 Passo: 6 Passo: 7 Passo: Passo:

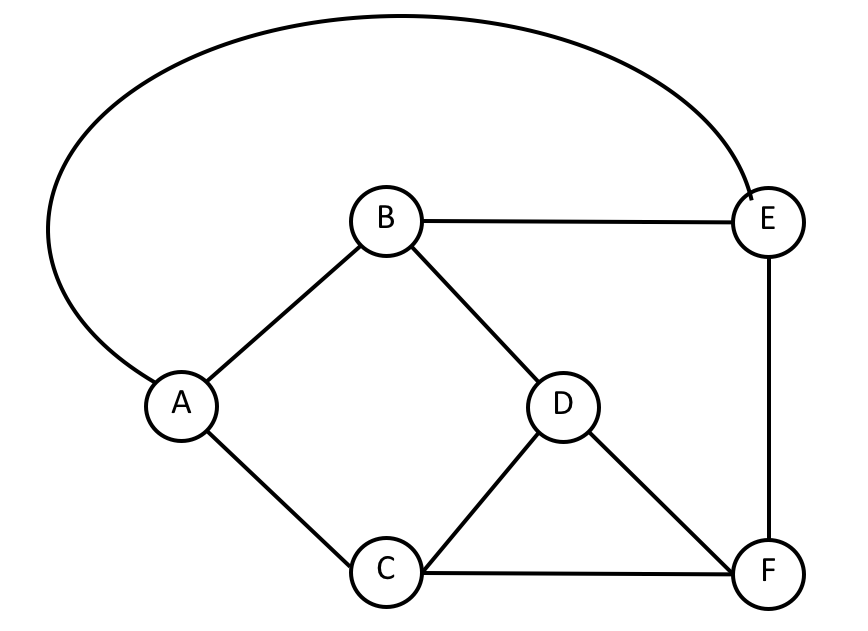
b) Usando ao vetor prev, temos: prev(F) = C caminho = (C,F)

prev(C) = A caminho = (A,C,F)

c) O algoritmo de Dijkstra não poderá ser usado pois não admite pesos negativos.

**Questão 2 (2,5 pontos)**. Seja *G=(N,E)* um grafo não dirigido. Um vértice *x é vizinho* de um vértice *y* em *G* sse há uma aresta entre *x* e *y* em *G*. Seja Γ(*x*) o conjunto de vizinho de um vértice *x* em *G* e denote por |*s*| a cardinalidade de um conjunto *s*. Dados dois vértices *x* e *y* em *G*, o *preferential attachment* entre *x* e *y* é definido como:

(a) **(0,5 ponto**) Considere o grafo não dirigido da Figura 3. Compute os conjuntos de vizinhos dos vértices   
*A*, *D*, *E*. Em seguida, compute *pa(A,E)* e *pa(D,A).*



**Figura 3.** Grafo não-dirigido.

(b) **(0,5 ponto**) Explique quais as vantagens e desvantagens de utilizar *preferential attachment* para computar a similaridade entre 2 vértices.

(c) **(1,5 pontos)**. Implemente uma função em C que receba como entrada um grafo não dirigido *G* e um vértice *x* do grafo *G* e devolva o vizinho *y* de *x* com o maior *preferential attachment* com relação a *x*. Considere que o grafo está representado por listas de adjacências:

typedef struct graph Graph;

typedef struct listNode ListNode;

struct listNode {

int vertex;

ListNode\* link;

};

struct graph {

int nv; /\* número de vértices no grafo \*/

ListNode\*\* vv; /\* vv[i] aponta p/ lista de vértices adjacentes a i \*/

};

A função deve ter o seguinte protótipo:

int MaxPA(Graph\* G, int x);

onde x é um inteiro indicando a posição do vértice no vetor vv. A função deve devolver um inteiro indicando a posição no vetor vv do vizinho de x com maior *preferential attachment*, ou -1, se x não tiver vizinhos.

***Resposta***

a) Γ(*A*) = *{B, C, E}*

Γ(*D*) = *{B, C, F}*

Γ(*E*) = *{A, B, F}*

*pa(A,E) =* 3 × 3 = 9

*pa(D,A) =* 3 × 3 = 9

b) Preferencial attachment é simples de computar, mas vértices com muitos vizinhos gerarão um índice muito alto, mesmo que os vértices não tenham nada em comum.

int MaxPA(Graph\* G, int x)

{

int y\_max = -1, y; /\* x não tem vizinhos: retorna -1 \*/

int pa\_max = 0, paxy; /\* pa\_max e pa entre x e y \*/

listNode\* u, v, w;

int gammax = 0, gammay, gammaxy;

for (u=G->vv[x]; u!=NULL; u=u->link) /\* calcula gammax \*/

gammax = gammax + 1;

for (u=G->vv[x]; u!=NULL; u=u->link) /\* computa gammay \*/

{ /\* para cada vizinho y de x \*/

y = u->vertex;

gammay = 0;

for (v=G->vv[y]; v!=NULL; v=v->link)

gammay = gammay + 1; /\* computa gammay \*/

paxy = gammax \* gammay;

if (paxy > pa\_max) { pa\_max = paxy; y\_max = y}

}

return y\_max;

}

**Questão 3 (2,5 pontos)**. O *diâmetro* de um grafo é o maior caminho mínimo dentre todos os possíveis caminhos mínimos entre pares de vértices pertencentes ao grafo.

(a) **(1,5 ponto)** Implemente uma função em C que receba como entrada um grafo não dirigido ponderado *G*, com pelo menos 1 vértice, e retorne o diâmetro do grafo. Assuma que o peso de uma aresta é sempre positivo. A função deve ter o seguinte protótipo:

float diametro(Graph\* G);

Considere que o grafo está representado pela matriz de pesos:

typedef struct graph Graph;

struct graph {

int nv; /\* número de vértices no grafo \*/

float\*\* peso; /\* matriz de pesos do grafo \*/

};

A função deverá se basear no algoritmo de Dijkstra, usado como subrotina e implementado como uma função em C com o seguinte protótipo:

void Dijkstra(Graph\* G, int s, float\* dist);

onde G é o grafo recebido como entrada, representado pela matriz de pesos, s indica o vértice do grafo (um índice de uma linha da matriz de pesos) onde o caminhamento começa, e dist é o vetor de saída do algoritmo de Dijkstra. Não é necessário incluir o código em C do algoritmo de Dijkstra na solução da questão, mas a função diametro deve alocar o vetor dist e passá-lo como parâmetro para a função Dijkstra.

(b) **(0,5 ponto)** Argumente porque a sua implementação da função diametro está correta.

(c) **(0,5 ponto)** Qual a complexidade de tempo, no pior caso, da função diametro, incluindo a execução da função Dijkstra. Explique a sua resposta.

***Resposta***

a)

float diametro(Graph\* G) {

int i, k, n;

float diametro = 0;

float\* dist;

dist = (float\*) malloc(G->nv \* sizeof(float));

for (i=0; i < G->nv; i=i+1) {

Dijkstra(G, s, dist);

for (j=0; j < G->nv; j=j+1)

if (dist[j] > diametro) diametro = dist[j];

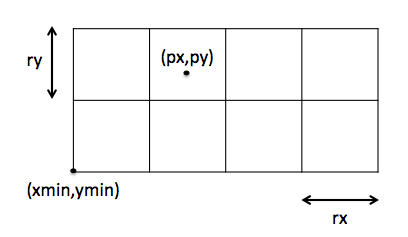
}

return diametro;

}  
b) O algoritmo de Dijkstra retorna em dist o custo do caminho mínimo entre o vértice s passado como entrada e todos os outros vértices do grafo. Portanto, a função diametro está correta pois o duplo loop chama o algoritmo de Dijkstra para cada vértice do grafo, e para cada chamada computa o maior caminho mínimo encontrado, varrendo o vetor dist.

c) O algoritmo de Dijkstra tem complexidade de tempo, no pior caso, O(|*V*|2), onde *V* é o conjunto de vértices do grafo. Logo, como diametro chama o algoritmo de Dijkstra |*V*| vezes, terá complexidade O(|*V*|3).

**Questão 4 (2,5 pontos)**. Considere uma grade regular bidimensional com *NX* células na horizontal e *NY* células na vertical. O tamanho de cada célula é dado por *rx* na horizontal e *ry* na vertical e o ponto inferior esquerdo da grade está localizado na posição (*xmin, ymin*), como apresentado esquematicamente na figura abaixo:



Cada célula da grade armazena uma lista encadeada de pontos. Considere os seguintes tipos que representam a grade:

typedef struct lista Lista;

struct lista {

float x, y; /\* ponto na lista \*/

Lista\* prox; /\* ponteiro para próximo elemento da lista \*/

};

typedef struct grade Grade;

#define NX 200

#define NY 135

struct grade {

float rx; /\* tamanho da dimensão x da célula \*/

float ry; /\* tamanho da dimensão y da célula \*/

float xmin, ymin; /\* posição mínima da grade \*/

Lista\* prim[NX][NY]; /\* lista por célula (inicializada com NULL) \*/

};

(1) **(2,0 pontos)** Implemente em C a função insere, que recebe como entrada uma grade *g*, com a definição acima, e um ponto *(px, py)*, como na figura, e insere o ponto na grade, caso o ponto não ocorra na grade. A função deve retornar 1, se o ponto foi inserido na grade, e 0, se o ponto está fora da área coberta pela grade ou se o ponto já ocorria na grade. A função deve seguir o seguinte protótipo:

int insere(Grade\* g, float px, float py);

(b) **(0,5 ponto)** Argumente porque a sua implementação da função insere está correta.

Resposta:

(a)

int ocorre(Grade\* g, float x, float y)

{

1. int i = floor((x - g->xmin) / g->rx); /\* calcula possível célula \*/

2. int j = floor((y - g->ymin) / g->ry); /\* onde o ponto se localiza \*/

3. Lista\* w, ponto;

4. if (i < 0 || j < 0 || i > NX || j > NY) return 0; /\* ponto fora da grade \*/

5. for(w = g->prim[i][j]; w != NULL; w = w->prox) /\* procura ponto na \*/

6. if (w->x = x && w->y = y) return 0; /\* lista da célula \*/

7. ponto =(lista\*)malloc(sizeof(lista));

8. ponto->x = px; ponto→y = py;

9. ponto->prox = g->prim[i][j]; g->prim[i][j] = ponto;

10. return 1

11.}

(b) As linhas 1 e 2 computam a célula da grade onde o ponto deve ocorrer, lembrando que a grade é regular. A linha 4 testa se o ponto de fato ocorre na área coberta pela grade. As linhas 5 e 6 procuram o ponto na lista associada à célula. A linha 7 aloca um novo ponto. As linhas 8 e 9 acrescentam o ponto à lista da célula apropriada.