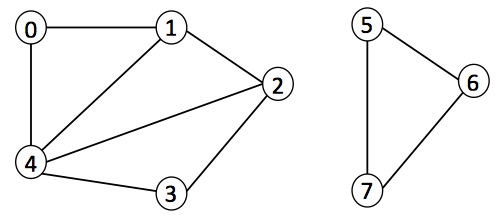
Aluno(a):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Matrícula:\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| 1a) | 2.5 |  |
| 2a) | 2.5 |  |
| 3a) | 2.5 |  |
| 4a) | 2.5 |  |
|  | 10.0 |  |

1. A prova é individual e sem consulta. Qualquer tentativa de “cola” resultará na anulação da prova de ambos os alunos envolvidos.
2. A interpretação faz parte da questão. Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
3. O tempo de prova é 1:45 h.
4. As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
5. A prova pode ser feita a lápis.
6. (2.5 pontos) O *giant coefficient* de um grafo não dirigido é definido como o número de vértices da maior componente conexa do grafo dividido pelo total de vértices do grafo.
   1. (0,5 ponto) Compute o *giant coefficient* do grafo abaixo. Mostre os passos da computação.



Resposta:

1. Compute incialmente as componentes conexas do grafo, contando quantos nós há em cada uma. O grafo possui 2 componentes conexas, com 5 e 3 nós
2. O *giant coeficiente* será então gc=5/8=0,625
   1. (2,0 pontos) Implemente uma função em C que receba como entrada um grafo não dirigido e retorne o *giant coefficient* do grafo. A função deve ter o seguinte protótipo:

float gc(Graph\* G);

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 \*/

int\* vis; /\* vis[i]=0 sse vertice i não foi visitado \*/

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

};

Resposta:

/\*

marca os nós da k-ésima componente conexa e conta o seu número de nós

\*/

void dfs(Graph\* g, int\* c, int N, int k)

{

ListNode\* w;

g->vis[N] = k;

c[k] = c[k]+1;

for (w = g->vv[N]; w != NULL; w = w->link)

if (!g->vis[w->vertex]) dfs(g, c, w->vertex, k);

}

/\*

Computa o giant coeficient

\*/

float gc(Graph\* g) {

int i, k, n; /\* c[k] = número de nós da \*/

int\* c=(int\*)malloc(sizeof(int)\*g->nv); /\* k-ésima component conexa \*/

for (i=0; i<g->nv; i=i+1) /\* (há no máximo nv componentes \*/

{g->vis[i]=-1; /\* nó não visitado marcado com -1 \*/

c[i]=0 } /\* c[k] = 0 incialmente \*/

k=-1;

for (i=0; i<g->nv; i=i+1) /\* marca cada componente conexa e \*/

if (g->vis[i]=-1) {k=k+1; dfs(g,c,i,k)}; /\*conta o seu número de nós \*/

n = c[0];

for (i=1; i<=k; i=i+1) /\* pesquisa a maior componente \*/

if (c[i] > n) n = c[i];

return n/g->nv;

}

1. (2.5 pontos) O *diâmetro* de um grafo não dirigido é definido como o maior valor da menor distância entre quaisquer pares de vértices pertencentes ao grafo.
   1. (2,0 ponto) Usando uma das funções para grafos descritas em sala como subrotina, descreva em pseudo-código um algoritmo para computar o diâmetro de um grafo.

Resposta:

Considere todas as arestas com peso 1. Desta forma, a menor distância entre 2 nós será o caminho de menor custo. Portanto, o algoritmo de Dijkstra determinará a menor distância entre um dado nó e todos os outros (o algoritmo abaixo não é otimizado):

D = 0;

Considere todas as arestas com peso 1;

For each node N do

begin

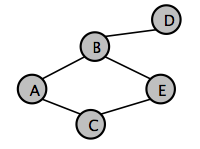
inicialize a estrutura como no algoritmo de Dijkstra;

chame o algoritmo de Dijkstra para computar todos caminhos mínimos de N aos outros nós;

se há um caminho mínimo com comprimento m maior do que D, faça D=m;

end

* 1. (0,5 ponto) Usando o algoritmo definido no item (a), mostre todos os passos para computar o diâmetro do grafo abaixo, inclusive os passos da subrotina.



Resposta:

Seja cmin(x,y) o comprimento do caminho mínimo de x a y.

1) Chamando o algoritmo de Dijkstra tendo como nó inicial A, temos:

cmin(A,B)=cmin(A,C)=1

cmin(A,D)=cmin(A,E)=2

Logo, D=2.

2) Chamando o algoritmo de Dijkstra tendo como nó inicial B, temos:

cmin(B,A)=cmin(B,E)=cmin(B,D)=1

cmin(B,C)=2

Logo, D permanece 2.

3) Chamando o algoritmo de Dijkstra tendo como nó inicial C, temos:

cmin(C,A)=cmin(C,E)=1

cmin(C,B)=2

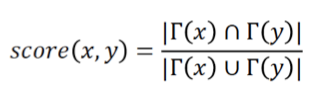
cmin(C,E)=3

Logo, D=3.

(Para o resto dos nós a computação é semelhante)

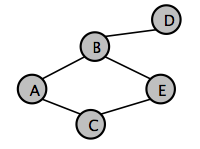
Logo, D=3

1. (2.5 pontos) Dado um grafo não dirigido, o *coeficiente de Jaccard* de dois nós *x* e *y*, *score(x,y)*, é definido como:



onde Γ(*n*) denota o conjunto de nós ligados por arestas a um nó *n* e |*s*| denota a cardinalidade de um conjunto *s*.

1. (0,5 ponto) Compute *score(A,D)* e *score(A,E)* no grafo abaixo.



Resposta:

Γ(*A*) = {*B,C*}

Γ(*D*) = {*B*}

Γ(*E*) = {*B,C*}

*score(A,D)* = |{*B*}| / |{*B,C*}| = 1/2

*score(A,E)* = |{*B,C*}| / |{*B,C*}| = 1

1. (2,0 pontos) Implemente uma função em C que receba como entrada um grafo não dirigido *G* e dois nós, *x* e *y*, do grafo *G* e retorne o coeficiente de Jaccard de *x* e *y*. 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:

float jaccard(Graph\* G, int x, int y);

onde x e y são dois inteiros indicando a posição dos nós no vetor vv.

Resposta:

float jaccard(Graph\* G, int x, int y)

{

int #x = 0; /\* cardinalidade de gamma(x) \*/

int #y = 0; /\* cardinalidade de gamma(y) \*/

int #inter = 0; /\* cardinalidade da interseção de gamma(x) e gamma(y) \*/

listNode\* u=vv[x], v=vv[y];

if (!u && !v) return 0; /\* gamma(x)=gamma(y)=0 implica jaccard = 0 \*/

while(u) /\* computa |gamma(x) inter gamma(y)| \*/

{ while(v)

{if (u->vertex == v->vertex) #inter = #inter + 1;

v = v->link }

#x = #x + 1; /\* computa |gamma(x)| \*/

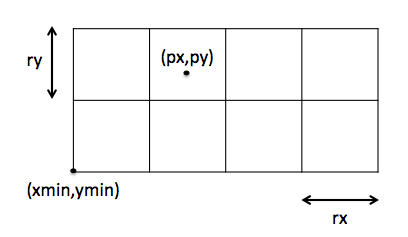
u = u->link }

while(v) {v = v->link; #y = #y + 1}; /\* computa |gamma(y)| \*/

return #inter / (#x + #y - #inter);

}

1. (2,5 ponto): 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) \*/

};

Implemente uma função que receba como entrada uma grade *g* com a definição acima e um ponto *(px, py)*, como na figura, e retorne 1, se o ponto ocorre na grade, e 0, em caso contrário. A função deve seguir o seguinte protótipo:

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

Resposta:

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

{

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

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

Lista\* w;

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

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

if (w->x = x && w->y = y) return 1; /\* lista da célula \*/

return 0

}