Algoritmos e Estruturas de Dados II – SCC-203

Grafos: Caminhos mais Curtos

Gustavo Batista

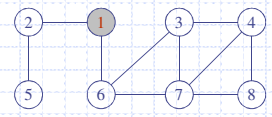
Caminhos Mais Curtos: Busca em Largura

- A busca em largura obtém os caminhos mais curtos entre o vértice de origem da busca e qualquer outro vértice alcançável do grafo, para um grafo cujas arestas possuem o mesmo peso.
- Dizemos que a busca em largura soluciona o problema de caminhos mais curtos de origem única.

Caminhos Mais Curtos: Busca em Largura

- Muitos problemas podem ser resolvidos pelo algoritmo para o problema origem única:
  - Caminhos mais curtos entre um par de vértices: o algoritmo para origem única é a melhor opção conhecida.
  - Caminhos mais curtos entre todos os pares de vértices: resolvido aplicando o algoritmo origem única $|V|$ vezes, uma vez para cada vértice origem.
  - Caminhos mais curtos com destino único: reduzido ao problema origem única invertendo a direção de cada aresta do grafo, ou seja, calculando o grafo transposto e iniciando a busca do vértice destino.

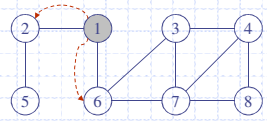
Caminhos Mais Curtos: Busca em Largura



q	1				
	0				

Antecessor	-1	-1	-1	-1	-1	-1	-1	-1
	1	2	3	4	5	6	7	8

Caminhos Mais Curtos: Busca em Largura

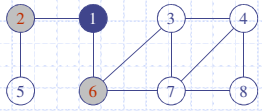


q: 1
0

Antecessor	-1	1	-1	-1	-1	1	-1	-1
	1	2	3	4	5	6	7	8

5

Caminhos Mais Curtos: Busca em Largura

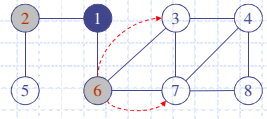


q: 6 2
1 1

Antecessor	-1	1	-1	-1	-1	1	-1	-1
	1	2	3	4	5	6	7	8

6

Caminhos Mais Curtos: Busca em Largura

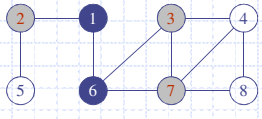


q: 6 2
1 1

Antecessor	-1	1	6	-1	-1	1	6	-1
	1	2	3	4	5	6	7	8

7

Caminhos Mais Curtos: Busca em Largura

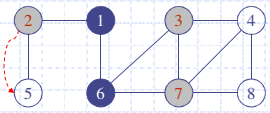


q: 2 3 7
1 2 2

Antecessor	-1	1	6	-1	-1	1	6	-1
	1	2	3	4	5	6	7	8

8

Caminhos Mais Curtos: Busca em Largura

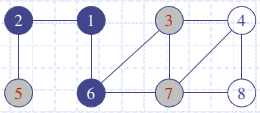


q	2	3	7		
	1	2	2		

Antecessor	-1	1	6	-1	2	1	6	-1
	1	2	3	4	5	6	7	8

9

Caminhos Mais Curtos: Busca em Largura

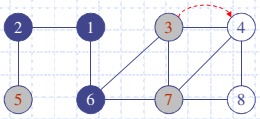


q	3	7	5		
	2	2	2		

Antecessor	-1	1	6	-1	2	1	6	-1
	1	2	3	4	5	6	7	8

10

Caminhos Mais Curtos: Busca em Largura

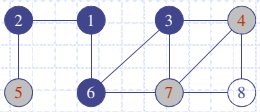


q	3	7	5		
	2	2	2		

Antecessor	-1	1	6	3	2	1	6	-1
	1	2	3	4	5	6	7	8

11

Caminhos Mais Curtos: Busca em Largura

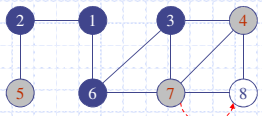


q	7	5	4		
	2	2	3		

Antecessor	-1	1	6	3	2	1	6	-1
	1	2	3	4	5	6	7	8

12

Caminhos Mais Curtos: Busca em Largura

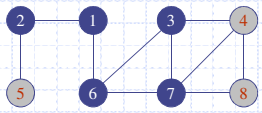


q	~~2~~	5	4		
	2	2	3		

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

13

Caminhos Mais Curtos: Busca em Largura

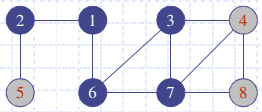


q	5	4	8		
	2	3	3		

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

14

Caminhos Mais Curtos: Busca em Largura

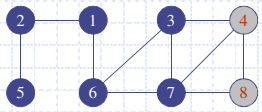


q	~~5~~	4	8		
	2	3	3		

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

15

Caminhos Mais Curtos: Busca em Largura

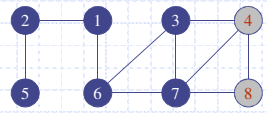


q	4	8			
	3	3			

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

16

Caminhos Mais Curtos: Busca em Largura

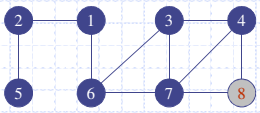


q: ~~4~~ 8
3 3

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

17

Caminhos Mais Curtos: Busca em Largura

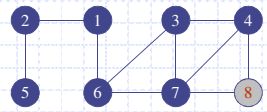


q: 8
3

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

18

Caminhos Mais Curtos: Busca em Largura

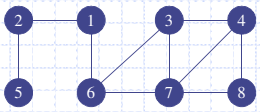


q: ~~8~~
3

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

19

Caminhos Mais Curtos: Busca em Largura



q:

Antecessor	-1	1	6	3	2	1	6	7
	1	2	3	4	5	6	7	8

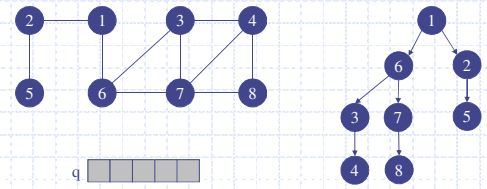
20

Caminhos Mais Curtos: Busca em Largura

- A árvore de busca em largura também representa os caminhos mais curtos entre o vértice de origem e os demais vértices quando todas as arestas possuem o mesmo peso.
- É importante observar que o vetor antecessor é uma segunda maneira de representar a árvore de busca em largura.

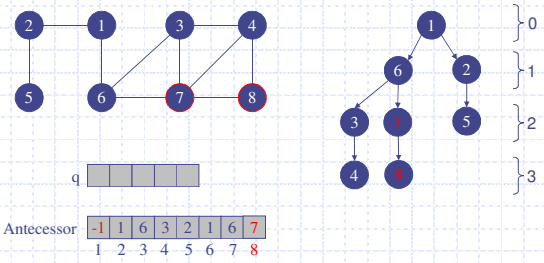
21

Caminhos Mais Curtos: Busca em Largura



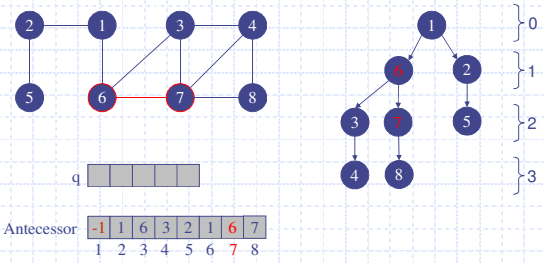
22

Caminhos Mais Curtos: Busca em Largura



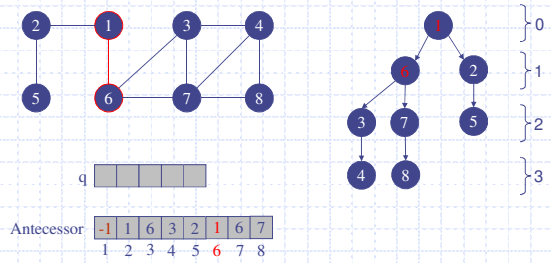
23

Caminhos Mais Curtos: Busca em Largura



24

Caminhos Mais Curtos: Busca em Largura



25

Caminhos Mais Curtos: Busca em Largura

```
#include<queue>
#include<cfloat>

#define MAXNUMVERTICES 100
#define INFINITO DBL_MAX
#define NULO -1
#define BRANCO 0
#define CINZA 1
#define PRETO 2

typedef double tpeso;

struct tgrafo{
    // Definicoes gerais
    tpeso mat[MAXNUMVERTICES][MAXNUMVERTICES];
    int num_vertices;
    // Definicoes somente para caminhos mais curtos
    tvertice antecessor[MAXNUMVERTICES];
    tpeso custo[MAXNUMVERTICES];
};
```

26

Caminhos Mais Curtos: Busca em Largura

```
void busca_largura_cmc(tvertice inicio, tgrafo *grafo) {
    tvertice v;
    int cor[MAXNUMVERTICES];

    for (v = 0; v < grafo->num_vertices; v++) {
        cor[v] = BRANCO;
        grafo->antecessor[v] = NULO;
        grafo->custo[v] = INFINITO;
    }
    visita_bfs_cmc(inicio, cor, grafo);
}
```

27

Caminhos Mais Curtos:

```
void visita_bfs_cmc(tvertice v, int cor[], tgrafo *grafo) {
    tvertice w;
    tapontador p;
    tpeso peso;
    std::queue<tvertice> q;

    grafo->custo[v] = 0;
    cor[v] = CINZA;
    q.push(v);
    while (!q.empty()) {
        v = q.front(); q.pop();
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            if (cor[w] == BRANCO) {
                grafo->antecessor[w] = v;
                grafo->custo[w] = grafo->custo[v] + 1;
                cor[w] = CINZA;
                q.push(w);
            }
            p = proximo_adj(v, p, grafo);
        }
        cor[v] = PRETO;
    }
}
```

Exercício

- Implemente uma sub-rotina que dado um vetor antecessor, o vértice de origem da busca em largura v e o nó destino u imprima o caminho mais curto de v a u.

29

Caminhos Mais Curtos: Algoritmo de Dijkstra

- A busca em largura encontra os caminhos mais curtos somente quando todas as arestas possuem o mesmo peso.
- O **algoritmo de Dijkstra** é capaz de encontrar caminhos mais curtos em um grafo ponderado com pesos diferentes entre as arestas.

30

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Para um grafo ponderado, o peso de um caminho $c = v_0, v_1, \ldots, v_k$ é a soma de todos os pesos das arestas do caminho.
- O caminho mais curto do vértice v_0 para o vértice v_k é definido como o caminho de menor peso de v_0 para v_k.
- Diz-se que um caminho mais curto tem peso infinito se v_k não é alcançável a partir de v_0.

31

Caminhos Mais Curtos: Algoritmo de Dijkstra

- O algoritmo de Dijkstra encontra os caminhos mais curtos para todos os vértices de um grafo G a partir de uma origem única.
- Portanto, o algoritmo de Dijkstra soluciona o problema de caminhos mais curtos de origem única.

32

Princípio de Otimalidade de Bellman

- Alguns problemas obedecem ao princípio de otimalidade de Bellman.
- Esses problemas possuem uma estrutura sub-ótima que permite que soluções ótimas sejam encontradas a partir de soluções prévias ótimas.
- Esses problemas podem ser solucionados utilizando técnicas de programação dinâmica.

33

Caminhos Mais Curtos: Algoritmo de Dijkstra

- A representação de caminhos mais curtos pode ser realizada por um vetor Antecessor, como feito para a busca em largura.
- Ao final do processamento, Antecessor contém uma árvore de caminhos mínimos definidos em termos dos pesos de cada aresta no grafo, ao invés do número de arestas.
- Caminhos mais curtos não são necessariamente únicos.

34

Caminhos Mais Curtos: Algoritmo de Dijkstra

- O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais dos caminhos mais curtos desde a origem já foram determinados. Inicialmente S contém somente o vértice origem.
- O algoritmo de Dijkstra é um algoritmo "guloso" (*greedy*). A cada iteração, um vértice $w \in V - S$ cuja distância ao vértice origem é tão pequena quanto possível é adicionado a S.

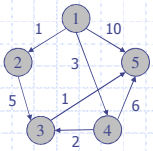
35

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Assumindo que todos os vértices possuem custos não negativos, sempre é possível encontrar um caminho mais curto do vértice origem a w que passa somente por vértices em S.
- A cada iteração, um vetor D armazena o custo do caminho mais curto conhecido até o momento entre o vértice origem e os demais vértices do grafo. Para os vértices em S, D possui o caminho mais curto final.
- O algoritmo termina quando todos os vértices estão em S.

36

Caminhos Mais Curtos: Algoritmo de Dijkstra



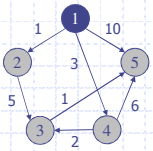
S = ∅

D =

1	2	3	4	5
∞	∞	∞	∞	∞

37

Caminhos Mais Curtos: Algoritmo de Dijkstra



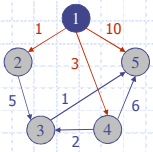
S = {1}

D =

1	2	3	4	5
0	∞	∞	∞	∞

38

Caminhos Mais Curtos: Algoritmo de Dijkstra



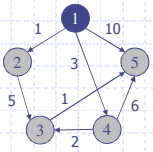
S = {1}

D =

1	2	3	4	5
0	1	∞	3	10

39

Caminhos Mais Curtos: Algoritmo de Dijkstra



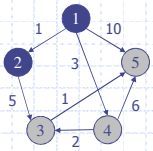
S = {1}

D =

1	2	3	4	5
0	1	∞	3	10

40

Caminhos Mais Curtos: Algoritmo de Dijkstra



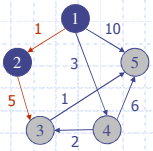
S = {1, 2}

D =

0	1	∞	3	10
1	2	3	4	5

41

Caminhos Mais Curtos: Algoritmo de Dijkstra



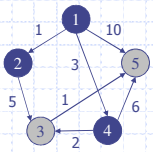
S = {1, 2}

D =

0	1	6	3	10
1	2	3	4	5

42

Caminhos Mais Curtos: Algoritmo de Dijkstra



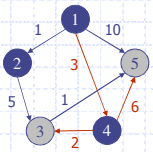
S = {1, 2, 4}

D =

0	1	6	3	10
1	2	3	4	5

43

Caminhos Mais Curtos: Algoritmo de Dijkstra



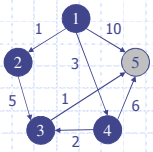
S = {1, 2, 4}

D =

0	1	5	3	9
1	2	3	4	5

44

Caminhos Mais Curtos: Algoritmo de Dijkstra

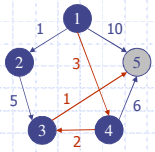


S = {1, 2, 4, 3}

D =

0	1	5	3	9
1	2	3	4	5

Caminhos Mais Curtos: Algoritmo de Dijkstra

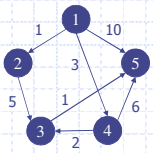


S = {1, 2, 4, 3}

D =

0	1	5	3	6
1	2	3	4	5

Caminhos Mais Curtos: Algoritmo de Dijkstra



S = {1, 2, 4, 3, 5}

D =

0	1	5	3	6
1	2	3	4	5

Caminhos Mais Curtos: Algoritmo de Dijkstra

```
procedimento Dijkstra(origem: TVertice, var G: TGrafo)
variáveis
   D: vetor[TVertice] de TPeso;
   w: TVertice;
   S, V: conjunto de TVertice;
início
   S := {origem};
   V := {todos os vértices de G};
   D[origem] := 0;
   para i:=1 até G.NumVertices faça
   início
        se i != origem e existe a aresta (origem, i) então
                D[i] := Peso da aresta (origem, i)
        senão
                D[i] := ∞;
   fim;
   enquanto S ≠ V faça
   início
        encontre um vértice w ∈ V - S tal que D[w] é mínimo;
        S := S ∪ {w};
        para todo v adjacente a w faça
                D[v] := min(D[v],D[w] + Peso da aresta (w,v));
   fim;
fim;
```

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Caso for necessário reconstruir o caminho mais curto entre o vértice origem e cada vértice, pode-se manter um vetor *Antecessor*, tal que *Antecessor[v]* contém o vértice imediatamente anterior ao vértice *v* no caminho mais curto.
- Essa técnica é similar à utilizada na busca em largura.

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Para entender o funcionamento do algoritmo de Dijkstra, é necessário verificar três fatores importantes:
  - Os vértices adicionados a *S* possuem o seu caminho mínimo definitivo e não precisam mais serem revistos;
  - Não pode haver arestas com custo negativo, uma vez que esse fato faria com que fosse necessário rever os vértices em *S;*
  - *D* possui os caminhos mínimos conhecidos até o momento, as atualizações de *D* a cada novo vértice inserido em *S* se certifica disso.

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Complexidade do algoritmo de Dijkstra:
  - Se uma matriz de adjacências é utilizada para representar o grafo, o laço "para todo" requer $O(|V|)$. Esse laço é executado $|V|-1$ vezes, fornecendo um tempo total de $O(|V|^2)$.
  - Se uma lista de adjacências é utilizada, então pode-se encontrar diretamente os adjacentes a *w*. Ainda, pode-se utilizar uma fila de prioridades para organizar os vértices em $V - S$.

Caminhos Mais Curtos: Algoritmo de Dijkstra

- Complexidade do algoritmo de Dijkstra:
  - Uma fila de prioridades pode implementar uma operação de busca e remoção do novo vértice *w* de menor custo em $O(\log |V|)$.
  - O algoritmo de Dijkstra realiza em sua execução um total de $|A|$ atualizações, cada uma a um custo de $O(\log |V|)$.
  - Portanto o tempo total despedido é $O(|A| \log |V|)$. Esse tempo de execução é consideravelmente melhor que $O(|V|^2)$ se $|A|$ for bem menor que $|V^2|$, i.e. o grafo for esparso.

Caminhos Mais Curtos: Dijkstra em C++-

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

53

Caminhos Mais Curtos: Dijkstra em C++-

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        gra
        marc
    }
    grafo->
    heap.pus
```

```
struct taresta{
    tpeso peso;
    tvertice dest;

    bool operator<(const taresta &a) const {
        return peso>a.peso;
    }
};
```

54

Caminhos Mais Curtos: Dijkstra em C++-

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

55

Caminhos Mais Curtos: Dijkstra em C++-

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpes
    tver
    tapo
    int
    int

    for
```

```
taresta cria_aresta(tpeso peso, tvertice dest) {
    taresta aresta;

    aresta.peso = peso;
    aresta.dest = dest;
    return aresta;
}
```

```
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

56

Caminhos Mais Curtos: Dijkstra em C+-

```
    while (!heap.empty()) {
        v = heap.top().dest; heap.pop();
        if (marc[v] == PRETO) continue;
        marc[v] = PRETO;
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            d = grafo->custo[v] + peso;
            if (cmp(d, grafo->custo[w]) < 0) {
                grafo->custo[w] = d;
                heap.push(cria_aresta(d, w));
                grafo->antecessor[w] = v;
            }
            p = proximo_adj(v, p, grafo);
        }
    }
}
```


Caminhos Mais Curtos: Dijkstra em C+-

```
    while (!heap.empty()) {
        v = heap.top().dest; heap.pop();
        if (marc[v] == PRETO) continue;
        marc[v] = PRETO;
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            d = grafo->custo[v] + peso;
            if (cmp(d, grafo->custo[w]) < 0) {
                grafo->custo[w] = d;
int cmp(double x, double y = 0, double tol = DBL_EPSILON) {
  return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
            }
        }
    }
}
```

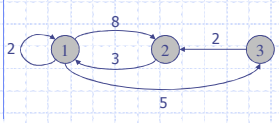

Caminhos Mais Curtos de Todos os Pares

- Suponha que um grafo orientado ponderado representa as possíveis rotas de uma companhia aérea conectando diversas cidades.
- O objetivo é construir uma tabela com os menores caminhos entre todas as cidades.
- Esse e um exemplo de problema que exige encontrar os **caminhos mais curtos para todos os pares de vértices**.

Caminhos Mais Curtos de Todos os Pares

- Uma possível solução é utilizar o algoritmo de Dijkstra utilizando cada vértice como origem alternadamente.
- Uma solução mais direta é utilizar o **algoritmo de Floyd**.
- O algoritmo de Floyd utiliza uma matriz $A\ |V| \times |V|$ para calcular e armazenar os tamanhos dos caminhos mais curtos.

Caminhos Mais Curtos de Todos os Pares: Floyd



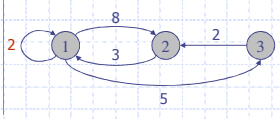
	1	2	3
1			
2			
3			

A

- Inicialmente os custos entre vértices adjacentes são inseridos na tabela A.
- Pesos de self-loops não são considerados.

61

Caminhos Mais Curtos de Todos os Pares: Floyd



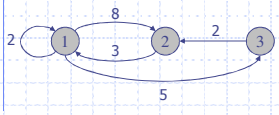
	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A

- Inicialmente os custos entre vértices adjacentes são inseridos na tabela A.
- Pesos de self-loops não são considerados.

62

Caminhos Mais Curtos de Todos os Pares: Floyd



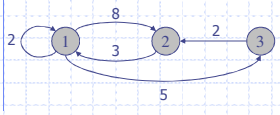
	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A

- A matriz *A* é percorrida */V/* vezes.
- A cada iteração *k*, verifica-se se um caminho entre dos vértices *(v, w)* que passa também pelo vértice *k* é mais curto do que o caminho mais curto conhecido.

63

Caminhos Mais Curtos de Todos os Pares: Floyd



	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

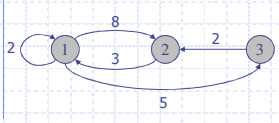
A

Ou seja:

A[v,w] = min(A[v,w], A[v,k] + A[k,w]).

64

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

A[1,1] = min(A[1,1], A[1,1] + A[1,1]).

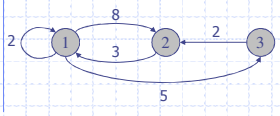
	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A

k = 1

65

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

A[1,2] = min(A[1,2], A[1,1] + A[1,2]).

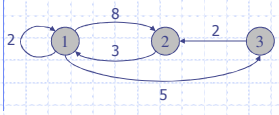
	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A

k = 1

66

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

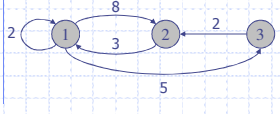
A[1,3] = min(A[1,3], A[1,1] + A[1,3]).

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

k = 1

67

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

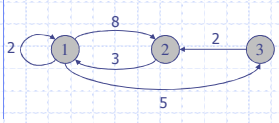
A[2,1] = min(A[2,1], A[2,1] + A[1,1]).

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

k = 1

68

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

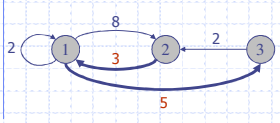
A[2,2] = min(A[2,2], A[2,1] + A[1,2]).

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

k = 1

69

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

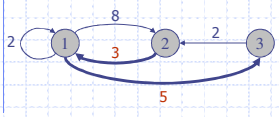
A[2,3] = min(A[2,3], A[2,1] + A[1,3]).

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

k = 1

70

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

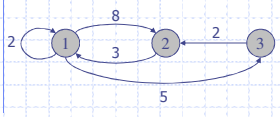
A[2,3] = min(A[2,3], A[2,1] + A[1,3]).

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

k = 1

71

Caminhos Mais Curtos de Todos os Pares: Floyd



Ou seja:

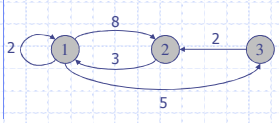
A[3,1] = min(A[3,1], A[3,1] + A[1,3]).

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

k = 1

72

Caminhos Mais Curtos de Todos os Pares: Floyd



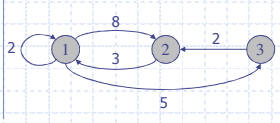
Ou seja:

A[3,2] = min(A[3,2], A[3,1] + A[1,2]).

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

k = 1

Caminhos Mais Curtos de Todos os Pares: Floyd



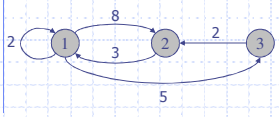
Ou seja:

A[3,3] = min(A[3,3], A[3,1] + A[1,3]).

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

k = 1

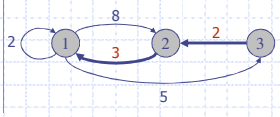
Caminhos Mais Curtos de Todos os Pares: Floyd



- Ao final da iteração *k=1* tem-se todos os caminhos mais curtos entre *v* e *w* que podem passar pelo vértice 1.
- O processo se repete para *k=2* e *k=3*.

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

Caminhos Mais Curtos de Todos os Pares: Floyd

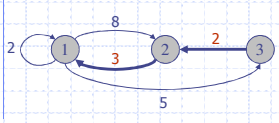


A[3,1] = min(A[3,1], A[3,2] + A[2,1]).

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

k = 2

Caminhos Mais Curtos de Todos os Pares: Floyd



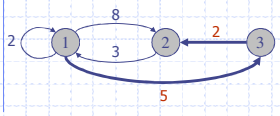
A[3,1] = min(A[3,1], A[3,2] + A[2,1]).

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

k = 2

77

Caminhos Mais Curtos de Todos os Pares: Floyd



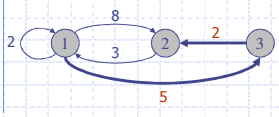
A[1,2] = min(A[1,2], A[1,3] + A[3,2]).

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

k = 3

78

Caminhos Mais Curtos de Todos os Pares: Floyd



A[1,2] = min(A[1,2], A[1,3] + A[3,2]).

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

k = 3

79

Caminhos Mais Curtos de Todos os Pares: Floyd

```
procedimento Floyd(var A: array[TVertice, TVertice]
                   de reais; var G: TGrafo)
variáveis
  v,w,k: TVertice;
início
  para v:=1 até G.NumVertices faça
      para w:=1 até G.NumVertices faça
            se v = w então
                  A[v,w] := 0;
            senão
                  A[v,w] := peso da aresta (v, w);

  para k:=1 até G.NumVertices faça
      para v:=1 até G.NumVertices faça
            para w:=1 até G.NumVertices faça
                  se A[v,k] + A[k,w] < A[v,w] então
                        A[v,w] := A[v,k] + A[k,w];

fim;
```

Caminhos Mais Curtos de Todos os Pares: Floyd

```
struct tgrafo{
    // Definicoes gerais
    tpeso mat[MAXNUMVERTICES][MAXNUMVERTICES];
    int num_vertices;
    // Definicoes somente para caminhos mais curtos de vértice único
    tvertice antecessor[MAXNUMVERTICES];
    tpeso custo[MAXNUMVERTICES];
    // Definicoes somente para caminhos mais curtos a partir de todos
    // os pares de vertices
    tpeso custos_todos[MAXNUMVERTICES][MAXNUMVERTICES];
    tvertice antecessores_todos[MAXNUMVERTICES][MAXNUMVERTICES];
};
```

81

Caminhos Mais Curtos de Todos os Pares: Floyd

```
void floyd_warshall(tgrafo *g) {
    tpeso d;
    tvertice v, w, k;

    for (v = 0; v < g->num_vertices; v++)
        for (w = 0; w < g->num_vertices; w++) {
            if (v == w) {
                g->custos_todos[v][w] = 0;
                g->antecessores_todos[v][w] = NULO;
            } else if (existe_aresta(v, w, g)) {
                g->custos_todos[v][w] = recupera_peso(v, w, g);
                g->antecessores_todos[v][w] = w;
            } else {
                g->custos_todos[v][w] = INFINITO;
                g->antecessores_todos[v][w] = NULO;
            }
        }
```

82

Caminhos Mais Curtos de Todos os Pares: Floyd

```
void floyd_warshall(tgrafo *g) {
    tpeso d;
    tvertice v, w, k;

    for (v = 0; v < g->num_vertices; v++)
        for (w = 0; w < g->num_vertices; w++) {
            if (v == w) {
                g->custos_todos[v][w] = 0;
                g->antecessores_todos[v][w] = NULO;
            } else if (existe_aresta(v, w, g)) {
                g->custos_todos[v][w] = recupera_peso(v, w, g);
                g->antecessores_todos[v][w] = w;
            } else {
                g->custos_todos[v][w] = INFINITO;
                g->antecessores_todos[v][w] = NULO;
            }
        }
tpeso recupera_peso(tvertice v, tvertice u, tgrafo *grafo) {
    return grafo->mat[v][u];
}
```

83

Caminhos Mais Curtos de Todos os Pares: Floyd

```
void floyd_warshall(tgrafo *g) {
    tpeso d;
    tvertice v, w, k;

    for (v = 0; v < g->num_vertices; v++)
        for (w = 0; w < g->num_vertices; w++) {
            if (v == w) {
                g->custos_todos[v][w] = 0;
                g->antecessores_todos[v][w] = NULO;
            } else if (existe_aresta(v, w, g)) {
                g->custos_todos[v][w] = recupera_peso(v, w, g);
                g->antecessores_todos[v][w] = w;
            } else {
                g->custos_todos[v][w] = INFINITO;
                g->antecessores_todos[v][w] = NULO;
            }
        }
```

84

Caminhos Mais Curtos de Todos os Pares: Floyd

```
    for (k = 0; k < g->num_vertices; k++)
        for (v = 0; v < g->num_vertices; v++)
            for (w = 0; w < g->num_vertices; w++) {
                d = g->custos_todos[v][k] + g->custos_todos[k][w];
                if (cmp(d, g->custos_todos[v][w]) < 0) {
                    g->custos_todos[v][w] = d;
                    g->antecessores_todos[v][w] =
                        g->antecessores_todos[k][w];
                }
            }
}
```


Caminhos Mais Curtos de Todos os Pares: Floyd

- O algoritmo de Floyd é $O(|V|^3)$.
- No caso do algoritmo de Dijksta for utilizado para solucionar o problema de caminhos mais curtos entre todos os pares, o algoritmo de Floyd tem desempenho similar ao algoritmo de Dijkstra com matriz de adjacência.

Caminhos Mais Curtos de Todos os Pares: Floyd

- Se o algoritmo de Dijkstra for implementado com listas de adjacências, então sua complexidade para encontrar os caminhos mais curtos entre todos os pares é $O(|V| \, |A| \log |V|)$.
- Essa complexidade pode ser considerada melhor que $O(|V|^3)$ para grafos grandes e esparsos.

Exercício

- Implemente o algoritmo de Floyd utilizando as operações do TAD Grafo.
- Modifique o algoritmo de Floyd de forma a armazenar os caminhos de mais curtos entre cada par de vértices.

Fechamento Transitivo: Algoritmo de Warshall

- Para alguns problemas pode ser interessante simplesmente saber se existe um caminho de qualquer tamanho entre dois vértices.
- O algoritmo de Floyd pode ser especializado para esse problema, resultando no algoritmo de Warshall (o qual é mais antigo que o algoritmo de Floyd).

89

Fechamento Transitivo: Algoritmo de Warshall

- A partir da matriz de adjacência *M*, deseja-se criar uma matriz *A*, tal que *A[v, w] = 1*, se existe um caminho de qualquer tamanho entre *v* e *w*.
- Essa matriz é chamada de fechamento transitivo da matriz de adjacência.

90

Fechamento Transitivo: Algoritmo de Warshall

```
procedimento Warshall(var A: array[TVertice,
                TVertice] de lógico; var G: TGrafo)
variáveis
   v,w,k: TVertice;

início
   para v:=1 até G.NumVertices faça
       para w:=1 até G.NumVertices faça
             A[v,w] := peso da aresta (v, w) > 0;

   para k:=1 até G.NumVertices faça
       para v:=1 até G.NumVertices faça
             para w:=1 até G.NumVertices faça
                   se não A[v,w] então
                         A[v,w] := A[v,k] e A[k,w];

fim;
```

Exercício

- Implemente o algoritmo de Warshall utilizando as operações do TAD Grafo.

92

Problemas com Grafos: Batuíra

Problema: Batuíra

As batuíras são aves migrantes de pequeno porte que fazem seus ninhos no hemisfério norte, na tundra ártica (no Canadá e Groenlândia). Elas permanecem naquela região durante o verão (de maio a junho no hemisfério norte), que é muito curto, e depois migram para o outro lado da Terra. As batuíras-de-peito-vermelho fazem seus ninhos em um raio de até mil quilômetros do Pólo Norte. Quando chega a época de migrarem, fazem longos vôos, em bandos, até o Sudeste dos Estados Unidos. Em seguida, partem para a Patagônia, onde passam o período quente do Hemisfério Sul (de outubro a março). Em março começam sua longa viagem de volta.

93

Problemas com Grafos: Batuíra

Como essas aves se orientam para realizar trajetos tão longos? Não existe uma resposta definitiva, mas apenas hipóteses. Uma das mais prováveis que é elas têm a capacidade de realizar a navegação celestial, isto é, de se guiarem pelos astros. Outra hipótese é que poderiam sentir os campos magnéticos da Terra, utilizando-os como referência.

As batuíras fazem longos vôos, mas precisam parar de vez em quando para recuperar suas forças, em pontos de repouso, normalmente praias ou beiras de lagos. Dependendo do trajeto escolhido e da localização dos pontos de repouso, a viagem migratória das batuíras pode variar em cerca de 10% de ano a ano - o que, no caso, significa uma variação de mil quilômetros!

94

Problemas com Grafos: Batuíra

Tarefa

Você deve escrever um programa que, dadas as distâncias entre possíveis pontos de repouso das batuíras determine o comprimento total do menor trajeto que as aves poderiam seguir em sua viagem do hemisfério norte para o hemisfério sul.

95

Problemas com Grafos: Batuíra

Entrada:

```
6       <= Nr. Paradas
1 2 10 <= Distâncias
1 3 20
5 2 4
3 4 7
6 5 30
2 6 50
4 5 7
4 6 9
0 0 0
```

Saída:

```
36
```

96

Problemas com Grafos: Número de Erdos

Problema: Número de Erdos

O matemático húngaro Paul Erdos (1913-1996), um dos mais brilhantes do século XX, é considerado o mais prolífico matemático da história. Erdos publicou mais de 1500 artigos, em colaboração com cerca de outros 450 matemáticos. Em homenagem a este gênio húngaro, os matemáticos criaram um número, denominado "número de Erdos". Toda pessoa que escreveu um artigo com Erdos tem o número 1. Todos que não possuem número 1, mas escreveram algum artigo juntamente com alguém que possui número 1, possuem número 2. E assim por diante. Quando nenhuma ligação pode ser estabelecida entre Erdos e uma pessoa, diz-se que esta possui número de Erdos infinito.

97

Problemas com Grafos: Número de Erdos

Por exemplo, o número de Erdos de Albert Einstein é 2. E, talvez surpreendentemente, o número de Erdos de Bill Gates é 4.

Tarefa

Sua tarefa é escrever um programa que, a partir de uma lista de autores de artigos, determine o número de Erdos dos autores.

98

Problemas com Grafos: Número de Erdos

Entrada

Smith, M.N., Martin, G., Erdos, P.: *Newtonian forms of prime factors*
Erdos, P., Reisig, W.: *Stuttering in petri nets*
Smith, M.N., Chen, X.: *First order derivates in structured programming*
Jablonski, T., Hsueh, Z.: *Self-stabilizing data structures*

Saída

Smith, M.N. 1
Hsueh, Z. infinity
Chen, X. 2

99

Problemas com Grafos: Número de Erdos

- Problema: caminhos mais curtos de origem única em grafos não ponderados.
- Soluções:
  - Busca em largura;
  - Dijkstra (muito caro/complexo?)

100

Problemas com Grafos: Dengue

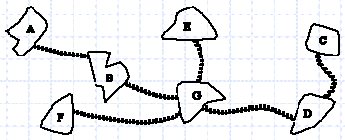
Problema: Dengue

A Costa do Mosquito é um país pequeno mas paradisíaco. Todos os habitantes têm boas moradias, bons empregos, o clima é agradável, e os governantes são justos e incorruptíveis. O sistema de transporte público da Costa do Mosquito é composto de uma rede de linhas de trem. O sistema foi projetado de forma peculiar: existe um único percurso ligando duas quaisquer vilas (esse percurso possivelmente passa por outras vilas). Por exemplo, na figura abaixo, que mostra um trecho do mapa da Costa do Mosquito, há apenas um percurso entre as vilas A e C, passando pelas vilas B, G e D. Uma tarifa fixa de M$ 1,00 é cobrada por cada viagem entre vilas vizinhas; assim, para uma viagem de A a C o usuário gasta M$ 4,00.

101

Problemas com Grafos: Dengue

Devido a um inesperado surto de dengue, o Ministério da Saúde da Costa do Mosquito resolveu montar um Posto de Vacinação. Para evitar que habitantes gastem muito dinheiro para se deslocar até a vila onde ficará o Posto de Vacinação, o Ministério da Saúde decidiu que este deverá ser instalado em uma vila de forma que o gasto com transporte até o Posto, para os habitantes que gastarem mais, seja o menor possível (para o caso da figura abaixo a vila escolhida seria G).



102

Problemas com Grafos: Dengue

Tarefa

Sua tarefa é escrever um programa que determine uma vila onde deve ser instalado o Posto de Vacinação. Esta vila deve ser tal que o custo com transporte, para os habitantes que tiverem maior custo, seja o menor possível. Note que devido à característica peculiar do sistema viário, ou haverá uma única vila que satisfaz essa restrição, ou haverá duas vilas que a satisfazem. No caso de existirem duas vilas apropriadas, qualquer uma delas serve como solução.

103

Problemas com Grafos: Dengue

Entrada

```
7  <= Nr. Vilas
1 2 <= Ligações
2 5
7 4
7 2
4 6
3 4
```

Saída

```
7
```

104

Vértice mais Central de um Grafo

- O problema anterior requer encontrar o vértice mais central (ou centro) de um grafo.
- Para isso, vamos utilizar o conceito de *excentricidade* de um vértice v em um grafo $G = (V, A)$:

$$max\{\text{distância mínima de } w \text{ a } v\}$$

- Sendo $w \in V$.

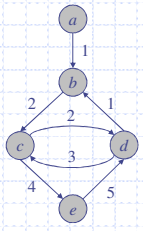
105

Vértice mais Central de um Grafo

- O *centro* de G é o vértice de excentricidade mínima.
- Dessa forma, o centro de um grafo é o seu vértice de menor distância a seu vértice mais distante.
- Por exemplo, para um grafo direcionado $G = (V,A)$.

106

Vértice mais Central de um Grafo



Vértice	Excentricidade
a	∞
b	6
c	8
d	5
e	7

107

Vértice mais Central de um Grafo

- Um algoritmo para encontrar o centro de um grafo G:
  1. Aplicar o algoritmo de Floyd para encontrar os caminhos mais curtos entre todos os pares.
  2. Encontrar o custo máximo em cada coluna i da matriz A.
  3. Encontrar o vértice de menor excentricidade.

108

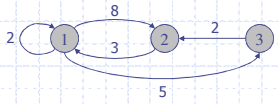
Vértice mais Central de um Grafo

	a	*b*	*c*	*d*	*e*
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	∞	6	8	5	7

109

Vértice mais Central de um Grafo

- Do exemplo passado...



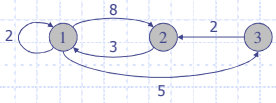
	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

A

110

Vértice mais Central de um Grafo

- Do exemplo passado...



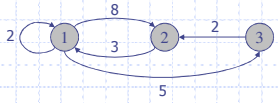
	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

A

111

Vértice mais Central de um Grafo

- Do exemplo passado...



	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

A

112

Problemas com Grafos: Rede Ótica

Problema: Rede Ótica

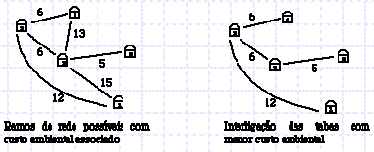
Os caciques da região de Tutuacu pretendem integrar suas tribos à chamada "aldeia global". A primeira providência foi a distribuição de telefones celulares a todos os pajés. Agora, planejam montar uma rede de fibra ótica interligando todas as tabas. Esta empreitada requer que sejam abertas novas picadas na mata, passando por reservas de flora e fauna. Conscientes da necessidade de preservar o máximo possível o meio ambiente, os caciques encomendaram um estudo do impacto ambiental do projeto. Será que você consegue ajudá-los a projetar a rede de fibra ótica?

Problemas com Grafos: Rede Ótica

Tarefa

Vamos denominar uma ligação de fibra ótica entre duas tabas de um ramo de rede. Para possibilitar a comunicação entre todas as tabas é necessário que todas elas estejam interligadas, direta (utilizando um ramo de rede) ou indiretamente (utilizando mais de um ramo). Os caciques conseguiram a informação do impacto ambiental que causará a construção dos ramos. Alguns ramos, no entanto, nem foram considerados no estudo ambiental, pois sua construção é impossível.

Problemas com Grafos: Rede Ótica



Ramos da rede possível com custo ambiental associado

Interligação das tabas com menor custo ambiental

Sua tarefa é escrever um programa para determinar quais ramos devem ser construídos, de forma a possibilitar a comunicação entre todas as tabas, causando o menor impacto ambiental possível.

Problemas com Grafos: Rede Ótica

Entrada

```
5 6      <= Nr. tabas e conexões
1 2 15
1 3 12
2 4 13
2 5 5
3 2 6
3 4 6
```

Saída

```
1 3
2 3
2 5
3 4
```

Árvore Geradora Mínima

- Na próxima aula veremos os algoritmos clássicos para árvores geradoras mínimas em detalhes.

117