Estruturas de Dados

Algoritmos em Grafos

Aula 13

Prof. Felipe A. Louza

UFU

Roteiro

Roteiro

1. Ordenação topológica
2. Caminhos minímos
3. Referências

Realizando tarefas

Queremos realizar várias tarefas, mas existem dependências

Realizando tarefas

Queremos realizar várias tarefas, mas existem dependências

- Ex: `Makefile`, linha de montagem, disciplinas e pré-requisitos, ...

Realizando tarefas

Queremos realizar várias tarefas, mas existem dependências

- Ex: `Makefile`, linha de montagem, disciplinas e pré-requisitos, ...
- Para uma tarefa ser realizada, precisamos antes realizar todas as tarefas das quais ela depende

Realizando tarefas

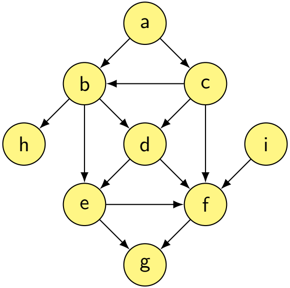
Queremos realizar várias tarefas, mas existem dependências

- Ex: `Makefile`, linha de montagem, disciplinas e pré-requisitos, ...
- Para uma tarefa ser realizada, precisamos antes realizar todas as tarefas das quais ela depende
- Vamos modelar usando um digrafo (grafo direcionado)

Realizando tarefas

Queremos realizar várias tarefas, mas existem dependências

- Ex: `Makefile`, linha de montagem, disciplinas e pré-requisitos, ...
- Para uma tarefa ser realizada, precisamos antes realizar todas as tarefas das quais ela depende
- Vamos modelar usando um digrafo (grafo direcionado)

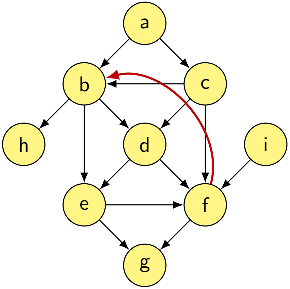


---

a depende de *b* e de *c*, *b* depende de *h*, *e* e de *d*, ...

Ciclos e DAGs

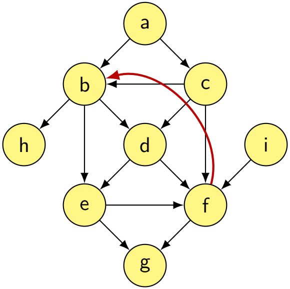
É possível realizar as tarefas de acordo com as dependências deste digrafo?



b depende de d, d depende de f e f depende de b.

Ciclos e DAGs

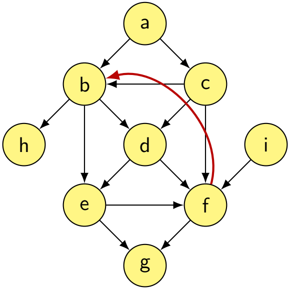
É possível realizar as tarefas de acordo com as dependências deste digrafo?



Um digrafo acíclico (DAG - directed acyclic graph) é um digrafo que **não contém** ciclos

Ciclos e DAGs

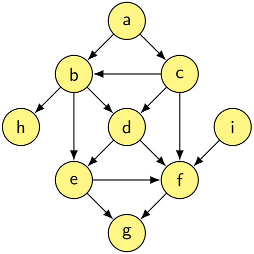
É possível realizar as tarefas de acordo com as dependências deste digrafo?



Um digrafo acíclico (DAG - directed acyclic graph) é um digrafo que **não contém** ciclos

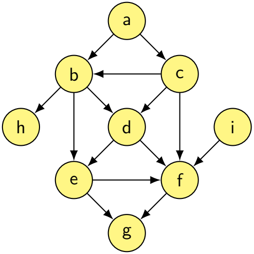
- As tarefas podem ser realizadas se e somente se o **digrafo de dependências** das tarefas é um DAG

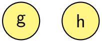
Ordem para realizar tarefas



Em qual ordem devemos realizar essas tarefas?

Ordem para realizar tarefas

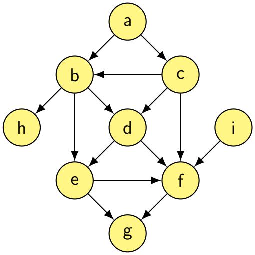


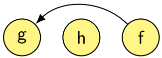


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa

Ordem para realizar tarefas

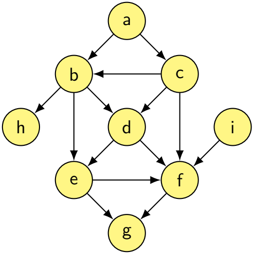


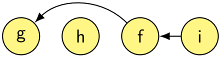


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g

Ordem para realizar tarefas

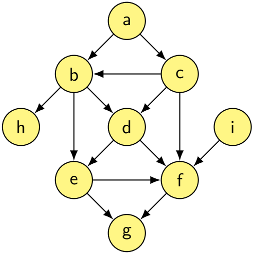


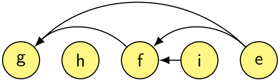


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f

Ordem para realizar tarefas

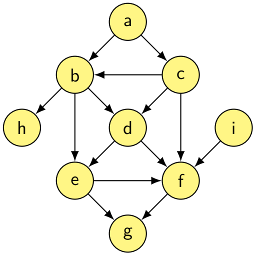


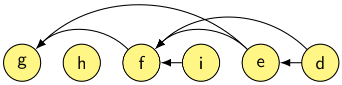


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f
- e depende apenas de f e g

Ordem para realizar tarefas

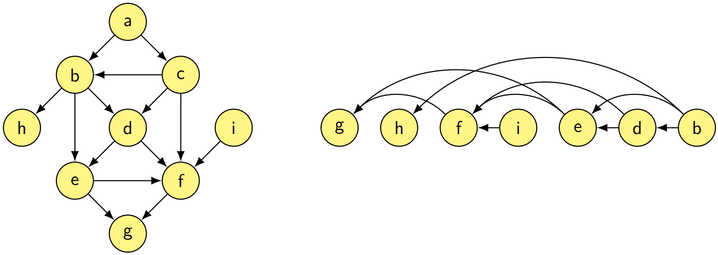




Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f
- e depende apenas de f e g
- d depende apenas de e e f

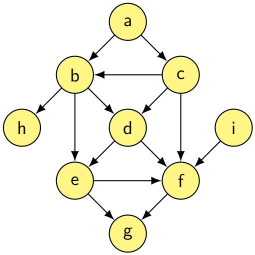
Ordem para realizar tarefas

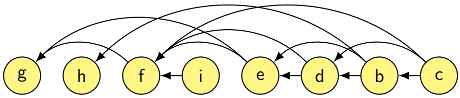


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f
- e depende apenas de f e g
- d depende apenas de e e f
- b depende apenas de h, e e d

Ordem para realizar tarefas

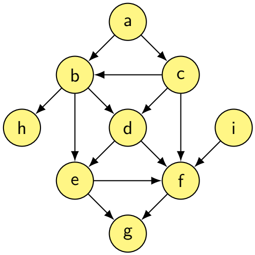


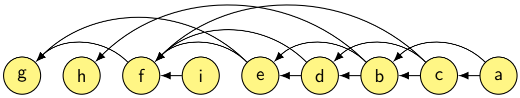


Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f
- e depende apenas de f e g
- d depende apenas de e e f
- b depende apenas de h, e e d
- c depende apenas de b, d e f

Ordem para realizar tarefas

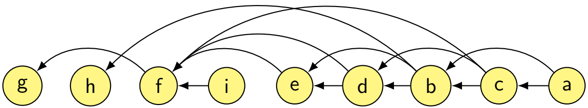




Em qual ordem devemos realizar essas tarefas?

- g e h **não dependem** de outra tarefa
- f depende apenas de g
- i depende apenas de f
- e depende apenas de f e g
- d depende apenas de e e f
- b depende apenas de h, e e d
- c depende apenas de b, d e f
- a depende apenas de b e c

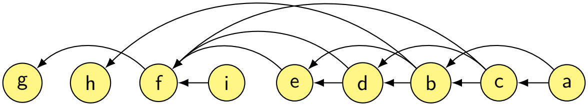
Ordenação Topológica



Uma ordenação topológica (reversa) de um DAG é:

- Uma ordenação dos vértices em que **um vértice** v que aparece na posição i
  - tem arcos apenas para vértices em $\{0, 1, \ldots, i-1\}$

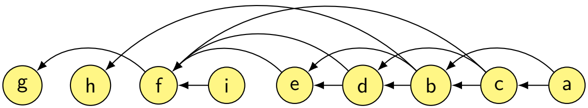
Ordenação Topológica



Uma ordenação topológica (reversa) de um DAG é:

- Uma ordenação dos vértices em que **um vértice** v que aparece na posição i
  - tem arcos apenas para vértices em $\{0, 1, \ldots, i-1\}$
- Na figura, os arcos vão apenas da direita para a esquerda

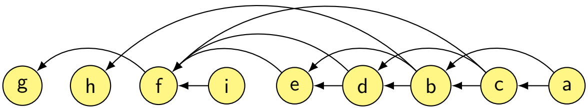
Ordenação Topológica



Uma ordenação topológica (reversa) de um DAG é:

- Uma ordenação dos vértices em que **um vértice** v que aparece na posição i
  - tem arcos apenas para vértices em $\{0, 1, \ldots, i-1\}$
- Na figura, os arcos vão apenas da direita para a esquerda
  - i.e., podemos realizar as tarefas na ordem dada

Ordenação Topológica



Obs.: podem existir **mais de uma** ordenação topológica

- Ex: g, f, e, d, h, b, c, a, i

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

- Todo v tal que (u, v) é um arco deve aparecer antes u
- Todo vértice w tal que existe arco (v, w) e arco (u, v) deve aparecer antes de u
- E assim por diante

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

- Todo v tal que (u, v) é um arco deve aparecer antes u
- Todo vértice w tal que existe arco (v, w) e arco (u, v) deve aparecer antes de u
- E assim por diante

Devemos considerar todos os w tal que existe caminho de u para w antes de considerar u

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

- Todo v tal que (u, v) é um arco deve aparecer antes u
- Todo vértice w tal que existe arco (v, w) e arco (u, v) deve aparecer antes de u
- E assim por diante

Devemos considerar todos os w tal que existe caminho de u para w antes de considerar u

Como encontrar todo w tal que existe caminho de u para w?

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

- Todo v tal que (u, v) é um arco deve aparecer antes u
- Todo vértice w tal que existe arco (v, w) e arco (u, v) deve aparecer antes de u
- E assim por diante

Devemos considerar todos os w tal que existe caminho de u para w antes de considerar u

Como encontrar todo w tal que existe caminho de u para w?

- Busca em profundidade

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

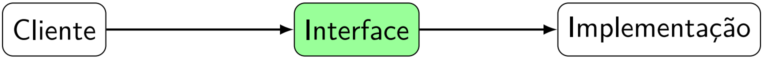
- Todo v tal que (u, v) é um arco deve aparecer antes u
- Todo vértice w tal que existe arco (v, w) e arco (u, v) deve aparecer antes de u
- E assim por diante

Devemos considerar todos os w tal que existe caminho de u para w antes de considerar u

Como encontrar todo w tal que existe caminho de u para w?

- Busca em profundidade (ou em largura)

TAD - Interface (lista de adjacências)



digrafo_lista.h

```c
#ifndef DIGRAFO_LISTA_H
#define DIGRAFO_LISTA_H
#include "lista.h"

//Dados
typedef struct {
  No **L; //Lista de adjacências
  int n;
} Grafo;

//Funções
Grafo* criar_digrafo(int n);
void destruir_digrafo(Grafo *p);

void inserir_arco(Grafo *p, int u, int v);
void remover_arco(Grafo *p, int u, int v);

int tem_arco(Grafo *p, int u, int v);
void imprimir_arcos(Grafo *g);
int* ordenacao_topologica(Grafo *p);

#endif
```

Digrafo - Manipulando arestas

digrafo_lista.c

```c
void inserir_arco(Grafo *p, int u, int v) {
  p->L[u] = inserir_na_lista(p->L[u], v);
}
```

```c
void remover_arco(Grafo *p, int u, int v) {
  p->L[u] = remover_da_lista(p->L[u], v);
}
```

Digrafo - Manipulando arestas

digrafo_lista.c

```c
void inserir_arco(Grafo *p, int u, int v) {
  p->L[u] = inserir_na_lista(p->L[u], v);
}
```

```c
void remover_arco(Grafo *p, int u, int v) {
  p->L[u] = remover_da_lista(p->L[u], v);
}
```

- Custo computacional: $O(|V|)/O(1)$ para inserir/remover[1]

[1] ou o contrário, dependendo do TAD de lista ligada.

Digrafo - Imprimindo arestas

digrafo_lista.c

```
void imprimir_arcos(Grafo *p) {
  int u;
  for (u = 0; u < p->n; u++){
      printf("%c: ", 'a'+u); //imprime como um char
      imprimir_lista(p->L[u]);
  }
}
```

lista.c

```
int imprimir_lista(No *p) {
  while(p != NULL){
    printf("%c ", 'a'+p->v); //imprime como um char
    if(p->prox!=NULL) printf("-> ");
    p = p->prox;
  }
  printf("\n");
return 0;
}
```

Cliente

Cliente → Interface → Implementação

exemplo1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "digrafo_lista.h"
```

```c
int main() {
  int n, m, i, u, v;
  scanf("%d %d", &n, &m);
  Grafo *G = criar_digrafo(n);
  for (i = 0; i < m; i++) {
    scanf("%d %d", &u, &v);
    inserir_arco(G, u, v);
  }
  imprimir_arcos(G);
  destruir_digrafo(G);
return 0;
}
```

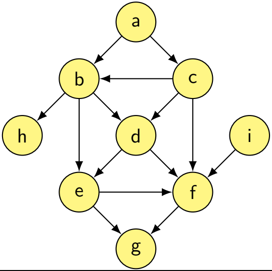
Makefile

Vamos usar o Makefile para compilar:

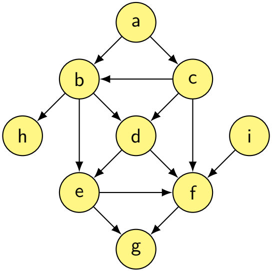
```
exemplo1: exemplo1.c digrafo_lista.o lista.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo1 < teste1.in
a: b -> c
b: d -> e -> h
c: b -> d -> f
d: e -> f
e: f -> g
f: g
g:
h:
i: f
```



Ordenação Topológica



Ordenação topológica com busca em profundidade (ideia):

- Para cada vértice (não visitado) u do DAG:
  1. Percorremos o caminho $(u, v_1), (v_1, v_2), \ldots, (v_k, w)$ (vértices não visitado)
  2. Adicionamos w na ordem i (i é incrementado)
  3. Retornamos para o vértice v_k, repetimos o **Passo 1**.

Ex: g, f, e, d, h, b, c, a, i

Ordenação Topológica - Implementação

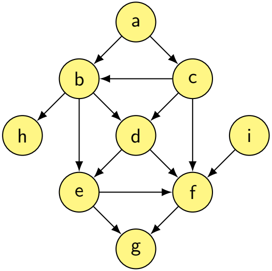
digrafo_lista.c

```c
int* ordenacao_topologica(Grafo *p) {
  int *ordem = (int*) malloc(p->n * sizeof(int));
  int i = 0;
  int *visitado = (int*) malloc(p->n * sizeof(int));
  int u;
  for (u = 0; u < p->n; u++)
    visitado[u] = 0;
  for (u = 0; u < p->n; u++){
    if (!visitado[u])
      busca_recursiva(p, visitado, u, ordem, &i);
  }
  free(visitado);
return ordem;
}
```

Ordenação Topológica - Implementação

digrafo_lista.c

```c
//busca em profundidade
void busca_recursiva(Grafo *p, int *visitado, int v, int *ordem, int *i){
  No *t;//nó da lista ligada
  visitado[v] = 1;
  for (t = p->L[v]; t != NULL; t = t->prox){
    if (!visitado[t->v])
      busca_recursiva(p, visitado, t->v, ordem, i);
  }
  ordem[*i] = v;
  *i += 1;
}
```



0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8

Cliente

exemplo2.c

```c
int main() {
  int n, m, i, u, v;
  scanf("%d %d", &n, &m);
  Grafo *G = criar_digrafo(n);
  for (i = 0; i < m; i++) {
    scanf("%d %d", &u, &v);
    inserir_arco(G, u, v);
  }
  imprimir_arcos(G);
  printf("Ordem topologica:\n");
  int *ordem = ordenacao_topologica(G);
  for (i = 0; i < n; i++)
    printf("%c ", 'a'+ordem[i]);
  printf("\n");
  free(ordem);
  destruir_digrafo(G);
return 0;
}
```

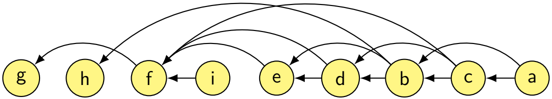
Makefile

Vamos usar o Makefile para compilar:

```
exemplo2: exemplo2.c digrafo_lista.o lista.o
  gcc $^ -o $@
```

Vamos executar:

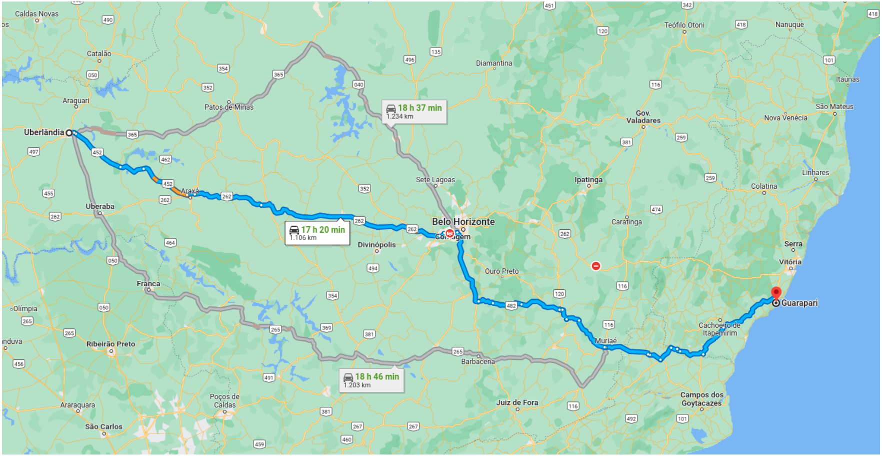
```
$ ./exemplo2 < teste1.in
a: b -> c
b: d -> e -> h
c: b -> d -> f
d: e -> f
e: f -> g
f: g
g:
h:
i: f
Ordem topologica:
g f e d h b c a i
```



Roteiro

Encontrando o menor caminho

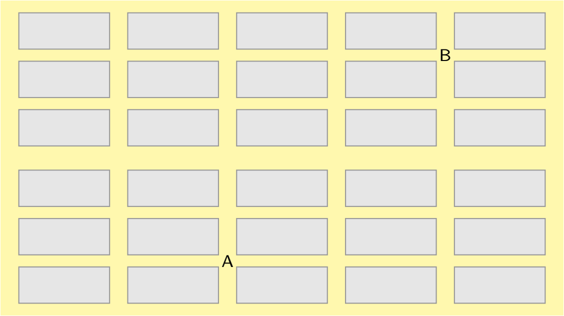
Como encontrar o menor tempo para ir de A para B?



Obs.: o menor tempo pode não ser o menor caminho.

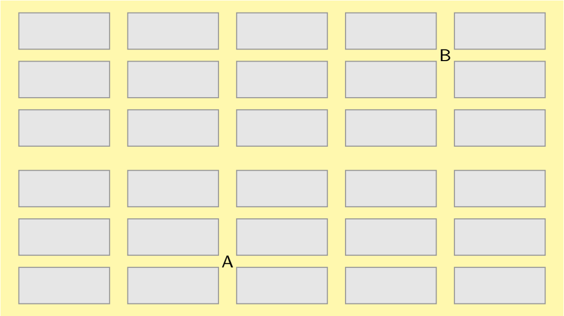
Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



Encontrando o menor caminho

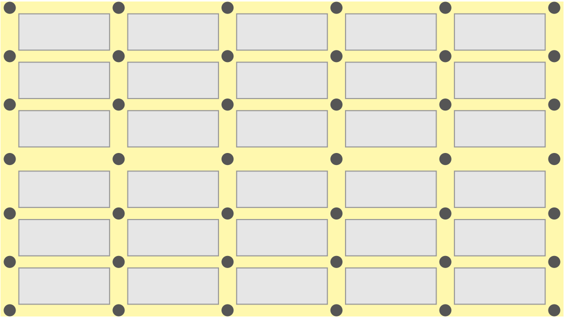
Como encontrar o menor tempo para ir de A para B?



Modelamos como um digrafo com pesos nos arcos:

Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

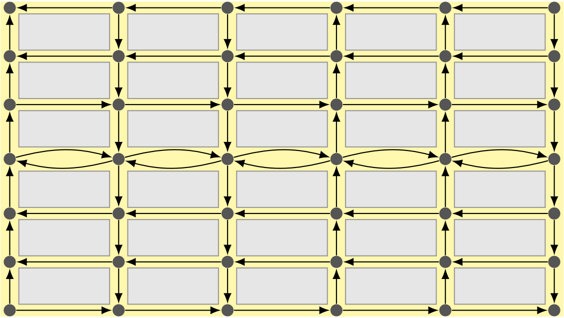


Modelamos como um digrafo com pesos nos arcos:

- Um vértice em cada cruzamento

Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

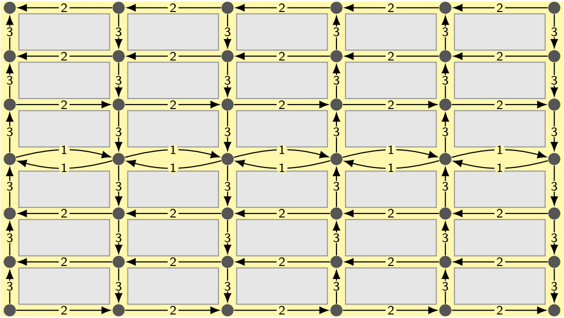


Modelamos como um digrafo com pesos nos arcos:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos

Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

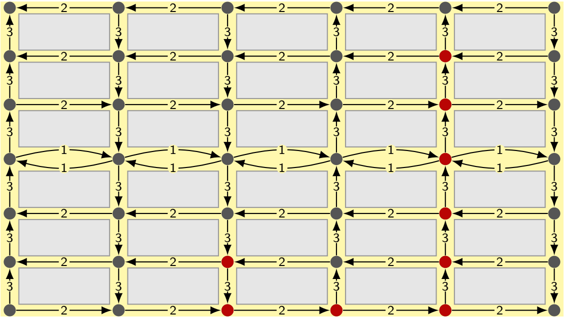


Modelamos como um digrafo com pesos nos arcos:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco (u, v) é o tempo de viagem de u para v

Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



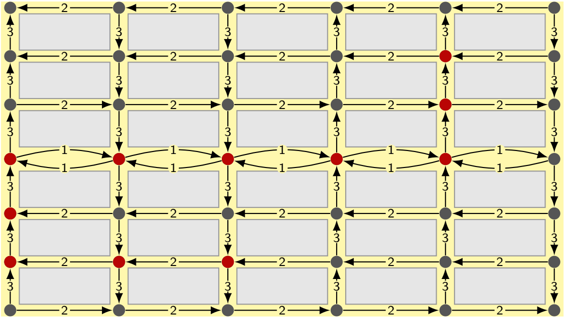
Modelamos como um digrafo com pesos nos arcos:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco (u, v) é o tempo de viagem de u para v

Ex: tempo de percurso do caminho: 22

Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



Modelamos como um digrafo com pesos nos arcos:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco (u, v) é o tempo de viagem de u para v

Ex: tempo de percurso do caminho: 20

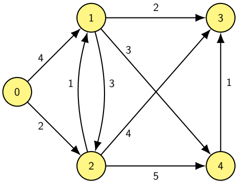
Como representar grafos com pesos nas arestas?

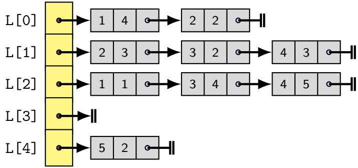
Listas de Adjacência:

- Basta adicionar um campo peso no Nó da lista ligada

lista.h

```
//Dados
typedef struct No {
  int v;
  int peso;
  struct No *prox;
} No;
```





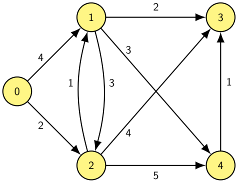
Como representar grafos com pesos nas arestas?

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso 0, -1 ou INT_MAX

digrafo_matriz.c

```
void inserir_arco(Grafo *p, int u, int v, int peso) {
  p->M[u][v] = peso;
}
```



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

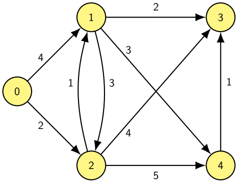
Como representar grafos com pesos nas arestas?

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso 0, -1 ou INT_MAX

digrafo_matriz.c

```
void inserir_arco(Grafo *p, int u, int v, int peso) {
  p->M[u][v] = peso;
}
```



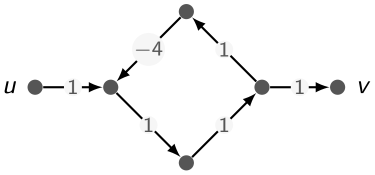
	0	1	2	3	4
0	0	4	2	0	0
1	0	0	3	2	3
2	0	1	0	4	5
3	0	0	0	0	0
4	0	0	0	1	0

- Ou fazemos uma struct com dois campos
  - um indica se há arco ou não, outro denota o peso do arco

Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de u para v

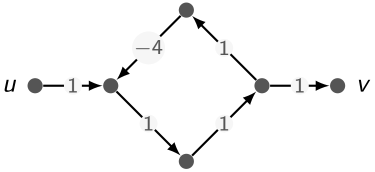
- Consideramos que **não temos** pesos negativos
- Se não, poderiamos percorrer um ciclo negativo infinitas vezes ...



Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de u para v

- Consideramos que **não temos** pesos negativos
- Se não, poderiamos percorrer um ciclo negativo infinitas vezes ...



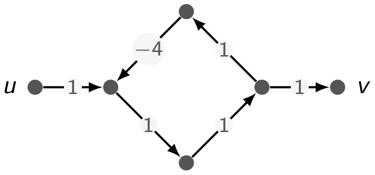
Como é **um** caminho mínimo de u para v?

- Ou u é vizinho de v

Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de u para v

- Consideramos que **não temos** pesos negativos
- Se não, poderiamos percorrer um ciclo negativo infinitas vezes ...



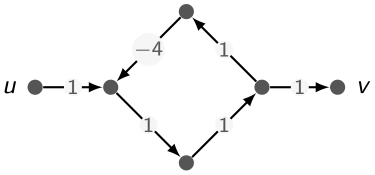
Como é **um** caminho mínimo de u para v?

- Ou u é vizinho de v
- Ou o caminho passa por um vizinho w de v

Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de u para v

- Consideramos que **não temos** pesos negativos
- Se não, poderiamos percorrer um ciclo negativo infinitas vezes ...



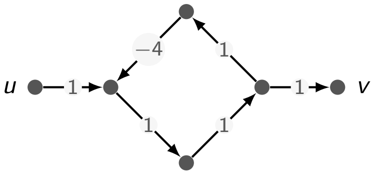
Como é **um** caminho mínimo de u para v?

- Ou u é vizinho de v
- Ou o caminho passa por um vizinho w de v
  - Soma do peso do caminho de u para w e de (w, v) é mínima

Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de u para v

- Consideramos que **não temos** pesos negativos
- Se não, poderiamos percorrer um ciclo negativo infinitas vezes ...



Como é **um** caminho mínimo de u para v?

- Ou u é vizinho de v
- Ou o caminho passa por um vizinho w de v
  - Soma do peso do caminho de u para w e de (w, v) é mínima
  - Este caminho de u a w tem que ter peso mínimo

Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de u):

Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de u):

- Dado u, vamos obter uma árvore enraizada em u

Árvore de Caminhos mínimos

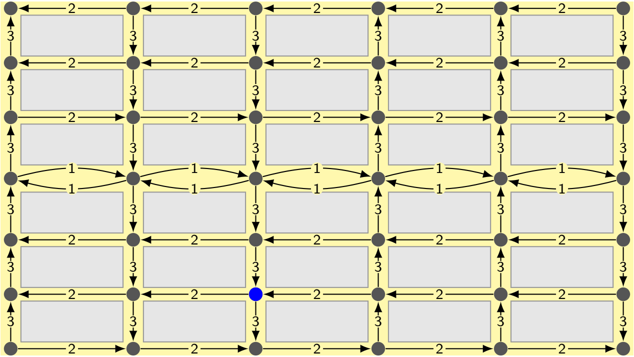
Árvore de caminhos mínimos (a partir de u):

- Dado u, vamos obter uma árvore enraizada em u
- De forma que o caminho de u para v na árvore seja **um** caminho mínimo de u para v no digrafo

Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de u):

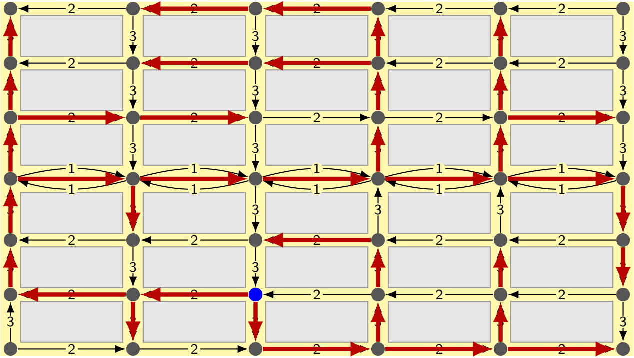
- Dado u, vamos obter uma árvore enraizada em u
- De forma que o caminho de u para v na árvore seja **um** caminho mínimo de u para v no digrafo



Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de u):

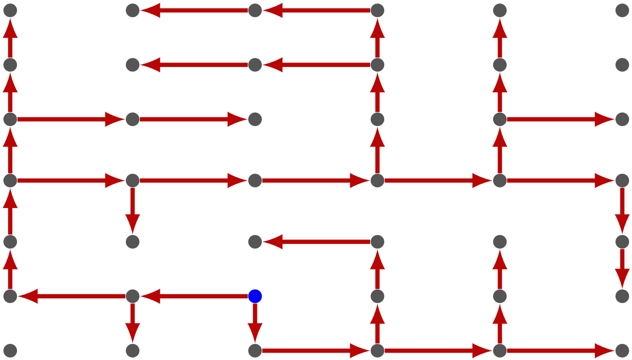
- Dado u, vamos obter uma árvore enraizada em u
- De forma que o caminho de u para v na árvore seja **um** caminho mínimo de u para v no digrafo



Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de u):

- Dado u, vamos obter uma árvore enraizada em u
- De forma que o caminho de u para v na árvore seja **um** caminho mínimo de u para v no digrafo



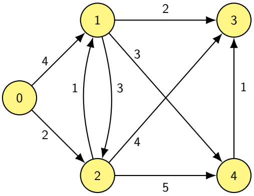
Caminhos mínimos

Um algoritmo conhecido para este problema foi proposto por Edsger W. Dijkstra em 1959.

https://pt.wikipedia.org/wiki/Edsger_Dijkstra

O algoritmo de Dijkstra se parece muito com uma busca em largura.

- Só que em vez de uma Fila, usamos uma Fila de prioridades.
- Dessa forma, priorizamos o próximo vértice com o caminho com menor custo.

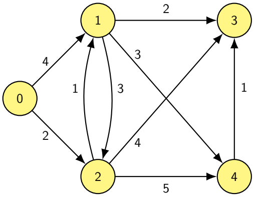


Pode ser visto como um algoritmo guloso!

Algoritmo de Dijkstra

Algorithm: DIJKSTRA

```
for all( v ∈ V )
    dist[v] ← ∞
dist[u] ← 0
R ← ∅                    // Região conhecida
while R ≠ V do
    v ← dist.EXTRACT-MIN()
    R ← R ∪ {v} // Adiciona v em R
    for all( (v, w) ∧ w ∉ R )
        if( dist[v] + c(v, w) < dist[w] )
            // Atualiza distância
            dist[w] ← dist[v] + c(v, w)
```

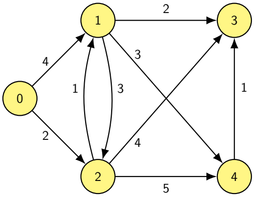


dist =

0	1	2	3	4
∞	∞	∞	∞	∞

Algorithm: DIJKSTRA

```
for all( v ∈ V )
    dist[v] ← ∞
dist[u] ← 0
R ← ∅                    // Região conhecida
while R ≠ V do
    v ← dist.EXTRACT-MIN()
    R ← R ∪ {v} // Adiciona v em R
    for all( (v, w) ∧ w ∉ R )
        if( dist[v] + c(v, w) < dist[w] )
            // Atualiza distância
            dist[w] ← dist[v] + c(v, w)
```



dist =

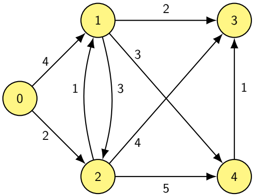
0	1	2	3	4
∞	∞	∞	∞	∞

Custo computacional:

- $O(|V| + |E|)$ vezes o tempo da operação dist.EXTRACT-MIN().

Algorithm: DIJKSTRA

```
for all( v ∈ V )
    dist[v] ← ∞
dist[u] ← 0
R ← ∅                    // Região conhecida
while R ≠ V do
    v ← dist.EXTRACT-MIN()
    R ← R ∪ {v}  // Adiciona v em R
    for all( (v, w) ∧ w ∉ R )
        if( dist[v] + c(v, w) < dist[w] )
            // Atualiza distância
            dist[w] ← dist[v] + c(v, w)
```



dist =

0	1	2	3	4
∞	∞	∞	∞	∞

Custo computacional:

- $O(|V| + |E|)$ vezes o tempo da operação dist.EXTRACT-MIN().
- Com uma Fila de Prioridades (heap binário): $O((|V| + |E|)\lg|V|)$

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

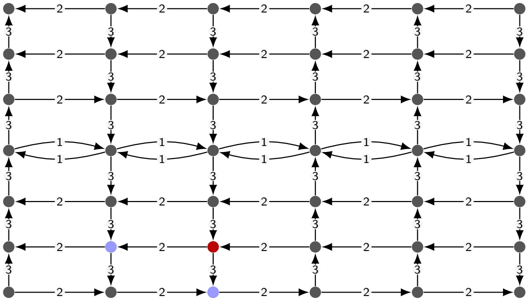
Em seguida, pegamos o vértice **na borda** **mais próximo** de u

Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \rightarrow w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

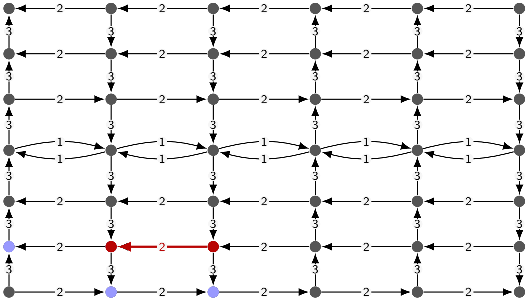


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

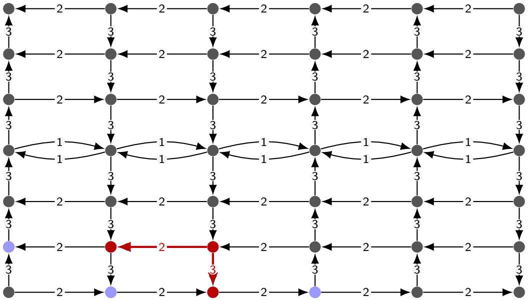


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

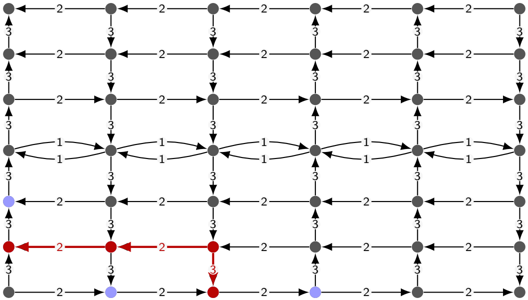


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

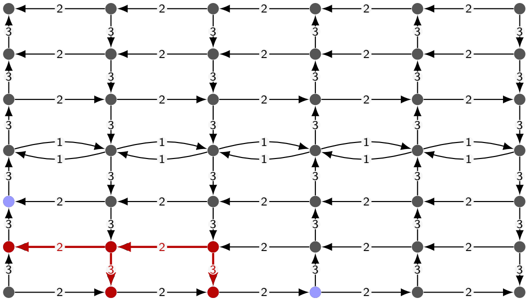


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

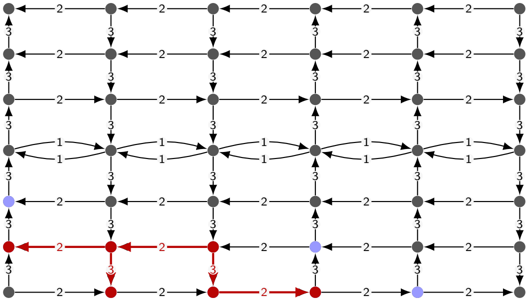


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]`$= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

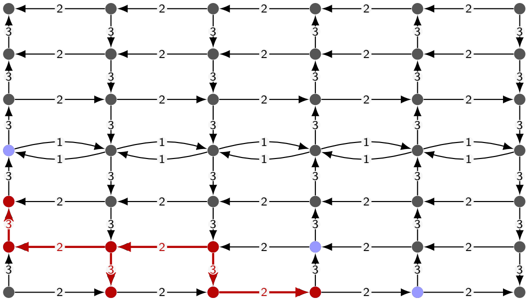


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

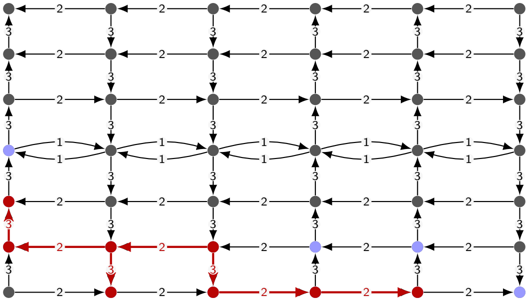


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

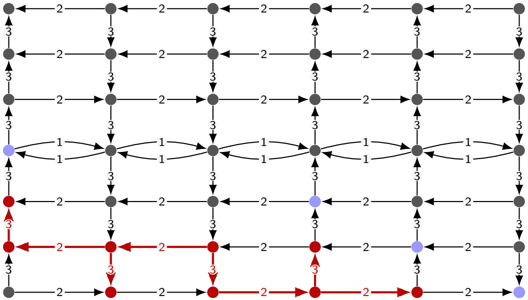


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u

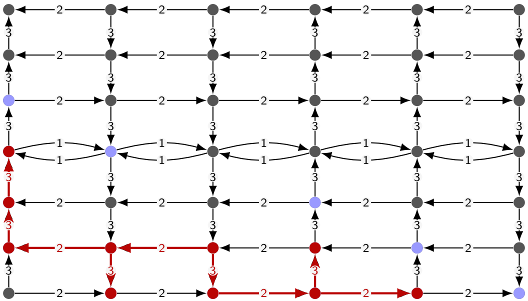


Algoritmo de Dijkstra

No final de cada iteração do loop **while** $R \neq V$:

1. Todos os vértices $v \in R$ estão à uma distância mínima de u.
2. Para cada $w \notin R$, `dist[w]` é a menor distância de $u \to w$ passando por vértices em R
   - Se esse caminho **não existe**, `dist[w]` $= \infty$

Em seguida, pegamos o vértice **na borda** **mais próximo** de u



TAD - Interface (lista de adjacências)

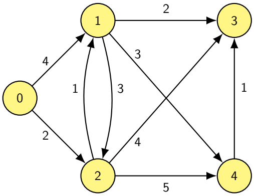
`digrafo_lista_peso.h`

```c
#ifndef DIGRAFO_LISTA_PESO_H
#define DIGRAFO_LISTA_PESO_H
#include "lista_peso.h"
#include "pq_heap.h"

//Dados
typedef struct {
  No **L; //Lista de adjacências
  int n;
} Grafo;

//Funções
Grafo* criar_digrafo(int n);
void destruir_digrafo(Grafo *p);

void inserir_arco(Grafo *p, int u, int v, int w);
void remover_arco(Grafo *p, int u, int v);

int tem_arco(Grafo *p, int u, int v);
void imprimir_arcos(Grafo *g);
int* dijkstra(Grafo *p, int u);

#endif
```

Algoritmo de Dijkstra - Implementação

grafo_lista_peso.c

```c
int* dijkstra(Grafo *p, int u){ //O(|V|+|E| lg(|V|))
  int *pai = (int*) malloc(p->n * sizeof(int));
  PQ *Fila = pq_criar(p->n);//Heap de MIN
  int v;
  for(v = 0; v < p->n; v++){//O(|V|lg|V|)
    pai[v] = -1;
    pq_adicionar(Fila, v, INT_MAX); //O(lg|V|)
  }
  pai[u] = u;
  muda_prioridade(Fila, u, 0);
  while(!pq_vazia(Fila)){            //O(|E|lg|V|)
    t_item aux = pq_extrai_minimo(Fila);
    int v = aux.idx;
    int dist_v = prioridade(Fila, v);//dist[v]
    if(dist_v == INT_MAX) break;
    No* t;
    for(t = p->L[v]; t!=NULL; t = t->prox){
      //atualiza as distancias para cada vizinho de v
      int w = t->v; //valor na lista
      int peso = t->peso; //c(v,w)
      if(dist_v + peso < prioridade(Fila, w)){//dist[w]
        muda_prioridade(Fila, w, dist_v+peso);//O(lg|V|)
        pai[w] = v;
      }
    }
  }
  pq_destruir(&Fila);
return pai;
}
```



dist =

0	1	2	3	4
∞	∞	∞	∞	∞

pai =

0	1	2	3	4
-1	-1	-1	-1	-1

Cliente

`exemplo3.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "digrafo_lista_peso.h"

int main() {
  int n, m, i, u, v, w;
  scanf("%d %d", &n, &m);
  Grafo *G = criar_digrafo(n);
  for (i = 0; i < m; i++) {
    scanf("%d %d %d", &u, &v, &w);
    inserir_arco(G, u, v, w);
  }
  int *pai = dijkstra(G, 0);
  for (i = 0; i < n; i++){
    printf("%d: ", i);
    imprimir_caminho(G, pai, i);
    printf("\n");
  }
  free(pai);
  destruir_digrafo(G);
return 0;
}
```

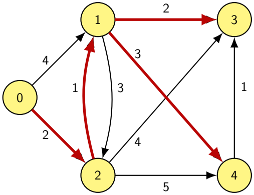
Makefile

Vamos usar o Makefile para compilar:

```
exemplo3: exemplo3.c digrafo_lista_peso.o lista_peso.o pq_heap.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo3 < teste3.in
0:
1: 2 (2) 1 (1)
2: 2 (2)
3: 2 (2) 1 (1) 3 (2)
4: 2 (2) 1 (1) 4 (3)
~
~
```

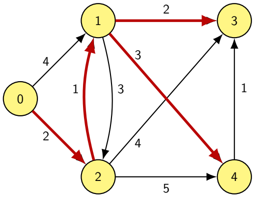


Vamos usar o Makefile para compilar:

```
exemplo3: exemplo3.c digrafo_lista_peso.o lista_peso.o pq_heap.o
  gcc $^ -o $@
```

Vamos executar:

```
$ ./exemplo3 < teste3.in
0:
1: 2 (2) 1 (1)
2: 2 (2)
3: 2 (2) 1 (1) 3 (2)
4: 2 (2) 1 (1) 4 (3)
~
~
```



Custo computacional:

- Tempo: $\mathrm{O}((|V| + |E|)\lg|V|)$

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
2. R. Sedgewick, "*Algorithms in C - Parts 5 - Third Edition*" (Seções 19.1 a 19.6, 21.1 e 21.2)