

MC-202

Algoritmos em Grafos

Rafael C. S. Schouery
rafael@ic.unicamp.br

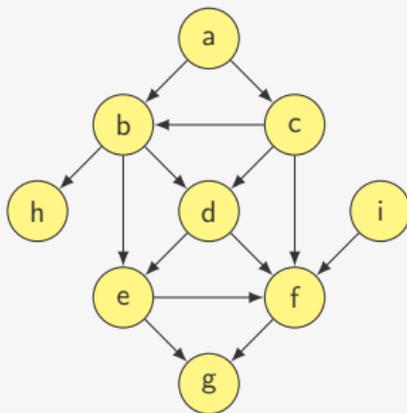
Universidade Estadual de Campinas

Atualizado em: 2023-08-02 10:42

Realizando tarefas

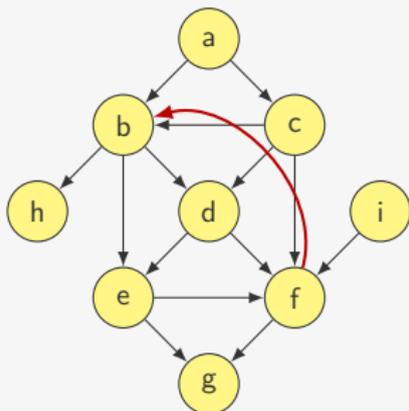
Queremos realizar várias tarefas, mas existem dependências

- Ex: Makefile
- Para uma tarefa ser realizada, precisamos primeiro realizar todas as tarefas das quais elas dependem
- Vamos modelar usando um digrafo



Ciclos e DAGs

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

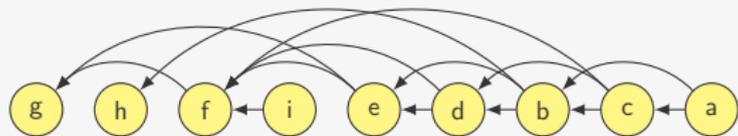
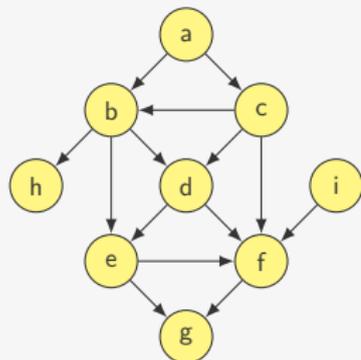


Um **digrafo acíclico** (**DAG** - *directed acyclic graph*) é um digrafo que não contém ciclos dirigidos

- Porém, ele pode ter ciclos não-dirigidos (ex: **a, b, d, c, a**)

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

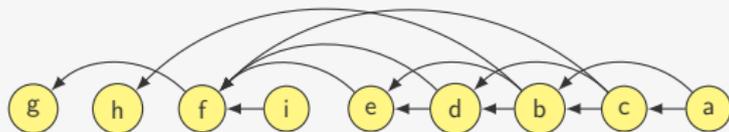
Ordem para realizar tarefas



Em qual ordem devemos realizar essas tarefas?

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

Como encontrar um ordenação topológica?

Considere um vértice u do DAG:

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

Devemos considerar todos w para os quais existe caminho de u para w antes de considerar u

- Lembra uma pós-ordem em árvores binárias...

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

- Busca em profundidade

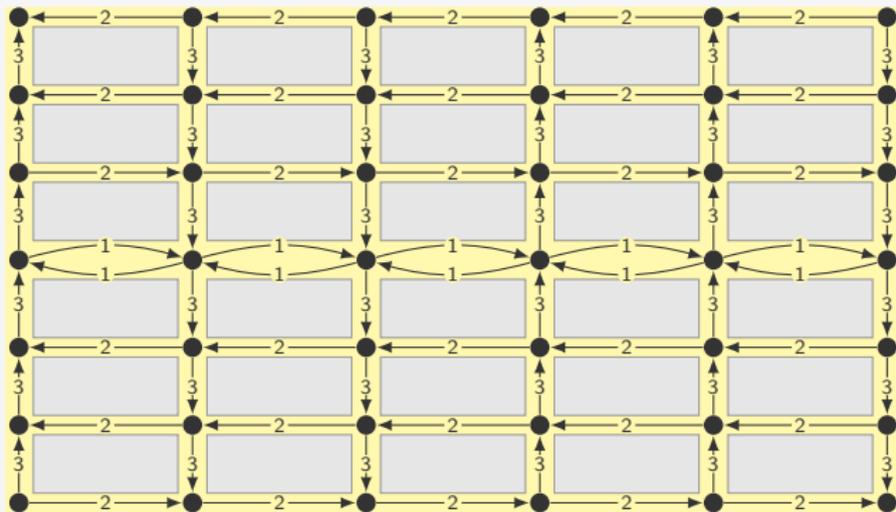
Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
4     for (t = g->adj[v]; t != NULL; t = t->prox)
5         if (!visitado[t->v])
6             visita_rec(g, visitado, t->v);
7     printf("%d ", v);
8 }
```


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

Tempo de percurso do caminho: **20**

Digrafos com pesos nas arestas — Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo `peso` no Nó da lista ligada

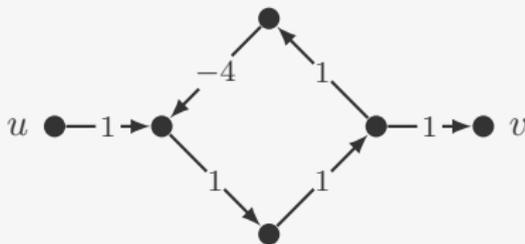
Matriz de Adjacências:

- Podemos indicar que não há arco usando peso `0`
 - Isso nem sempre é uma boa opção
 - Podemos trocar por `-1` ou então `INT_MAX`
- 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 no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



Como é o 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):

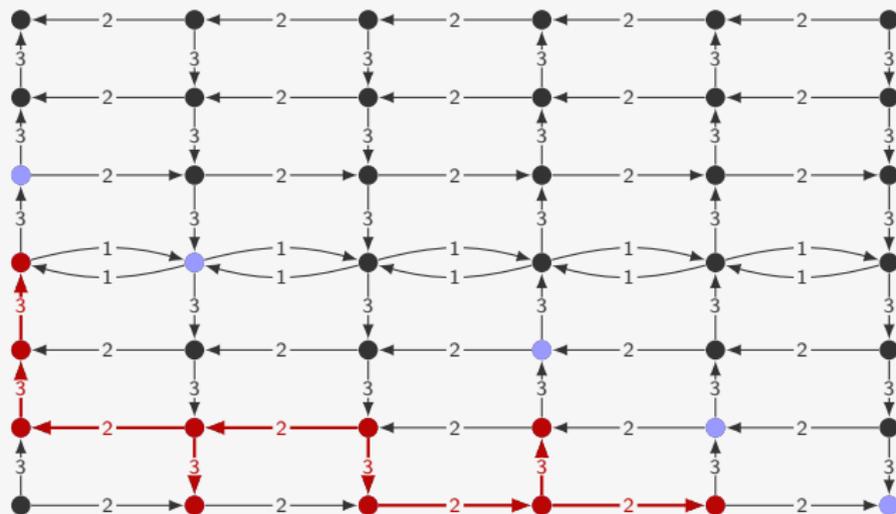
- Dado u , o algoritmo encontra uma árvore enraizada em u
- De forma que o caminho de v para u na árvore seja um caminho mínimo de u para v no digrafo



Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de u



Implementação

Grafo

```
1 typedef struct no *p_no;
2
3 struct no {
4     int v;
5     int peso;
6     p_no prox;
7 };
8
9 typedef struct grafo *p_grafo;
10
11 struct grafo {
12     int n;
13     p_no *adj;
14 };
```

Heap binário

```
1 typedef struct {
2     int prioridade;
3     int vertice;
4 } Item;
5
6 typedef struct {
7     Item *v;
8     int *indice;
9     int n, tamanho;
10 } FP;
11
12 typedef FP * p_fp;
```

Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
17                    pai[t->v] = v;
18                }
19    }
20    return pai;
21 }
```

Tempo: $O(|E| \lg |V|)$