MAC0323 Algoritmos de Estruturas de Dados II

Prova 3 – 3 de julho de 2018

Nome: _____

Assinatura: _____

Nº USP: _____

Instruções:

1. Não destaque as folhas deste caderno. A prova pode ser feita a lápis.
2. A prova consta de 10 questões. Verifique antes de começar a prova se o seu caderno está completo.
3. Cuidado com a legibilidade.
4. Não é permitido o uso de folhas avulsas para rascunho, a consulta a livros, apontamentos, colegas ou equipamentos eletrônicos. Desligue o seu celular e qualquer equipamento que possa perturbar o andamento da prova.

DURAÇÃO DA PROVA: 100 minutos

Questão	Valor	Nota
1	1,0	
2	1,0	
3	1,0	
4	1,0	
5	1,0	
6	1,0	
7	1,0	
8	1,0	
9	1,0	
10	1,0	
Total	10,0	

1. Burrows-Wheeler (1 ponto)

(a) Qual é o resultado da transformada de Burrows-Wheeler da string `"baracadabra"`.

0	a	b	a							r
1	a	b	r							d
2	a	c								r
3	a	d								c
4	a	r								b
5	b	a	r	a	c	a	d	a	b	r	a
6	b	r								a
7	c									a
8	d	a								a
9	r	a	b							b
10	r	a	c							a

Resposta: `"5 rdrcbaaaaba"`

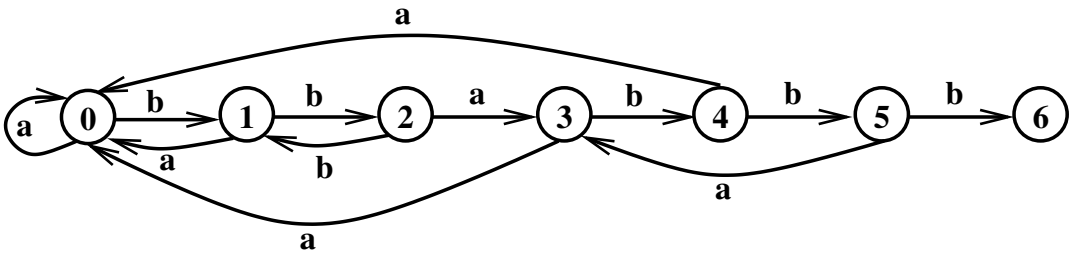
(b) Qual é a inversa da transfoma de Burrows-Wheeler de `"7 rdrbcaaaaab"`.

											next
0	a									r	5
1	a									d	6
2	a									r	7
3	a									b	8
4	a									c	9
5	b									a	3
6	b									a	10
7	c									a	4
8	d									a	1
9	r									a	0
10	r									b	2

Resposta: `"carabadabra"`

2. DFA (1 ponto)

O diagrama a seguir é de um autômato (`DFA` = *deterministic finite-state automaton*) que pretende aceitar precisamente as strings sobre o alfabeto `{ a, b }` que contém a string `"b b a b b b"`. O estado inicial é `0` e o estado de aceitação é `6`.



(a) Mostre a matriz `dfa[][]` que representa esse autômato.

`dfa:`

	b	b	a	b	b	b
	0	1	2	3	4	5
a	0	0	3	0	0	3
b	1	2	1	4	5	6

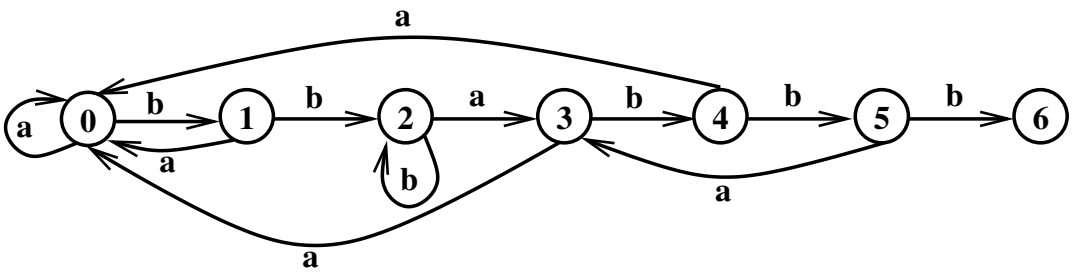
(b) Existe alguma string que **contém** `"b b a b b b"` que é **rejeitada**?
Se existe, apresente uma dessas strings que seja o mais curta possível.

Resposta: Sim, `"b b b a b b b"`

(c) Existe alguma string que **não contém** `"b b a b b b"` que é **aceita**?
Se existe, apresente uma dessas strings que seja o mais curta possível.

Resposta: Não.

(d) Como modificar esse autômato para que reconheça as strings pretendidas?

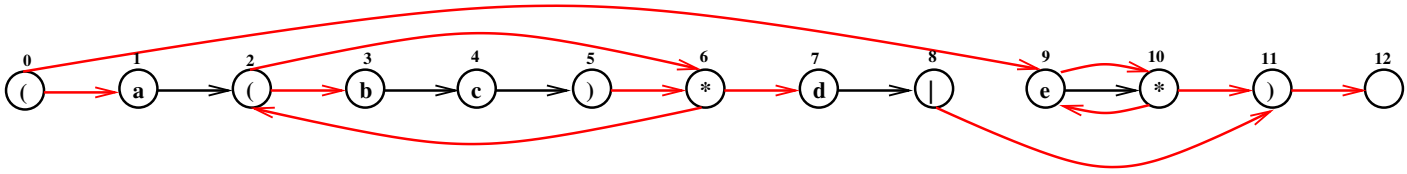


3. NFA (1 ponto)

Mostre a estrutura de dados (= vetor e digrafo) que representa o autômato não-determinístico (`NFA` = *nondeterministic finite-state automaton*) que reconhece a linguagem dada pela expressão regular `"( a ( b c ) * d | e * )"`.

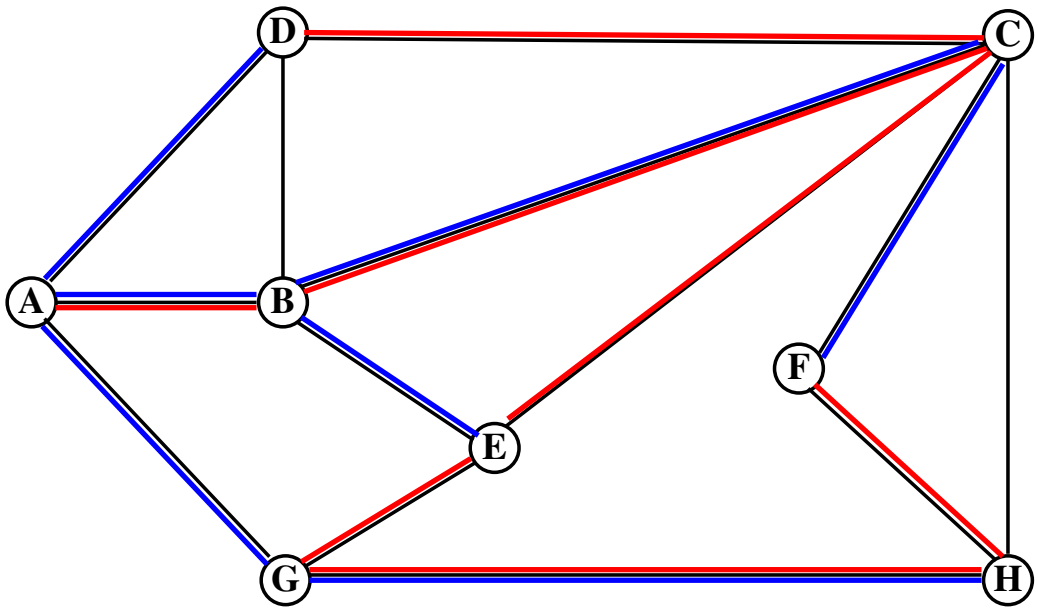
	0	1	2	3	4	5	6	7	8	9	10	11
re	(	a	(	b	c	)	*	d	|	e	*	)

Digrafo em que arcos em **vermelho** representam ϵ-transições. Esse digrafo também é chamado de digrafo das ϵ-transições. O estado 12 é de aceitação. Os arcos em **preto** não fazem parte do digrafo.



4. DFS e BFS em grafos (1 ponto)

Suponha que no vetor de listas de adjacência que representa o grafo abaixo as listas estão em ordem lexicográfica. Por exemplo, quando examinando o vértice `A`, considere a aresta `A-B` antes da aresta `A-D` antes ...



(a) Execute uma **busca em profundidade** (=*depth-first search*) no grafo a partir do vértice `A`.

Liste os vértices em **pré-ordem**: A B C D E G H F

Liste os vértices em **pós-ordem**: D F H G E C B A

(b) Execute uma **busca em largura** (=*breadth-first search*) no grafo a partir do vértice `A`.

Liste os vértices na ordem em que são enfileirados:

A B D G C E H F

(c) Cite um problema em que **devemos** usar **busca em largura** em vez de **busca em profundidade**.

Resposta: caminhos de comprimento mínimo.

(d) Cite um problema em que **devemos** usar **busca em profundidade** em vez de **busca em largura**.

Resposta: ordem topológica de um digrafo.

5. LZW (1 ponto)

Considere o algoritmo `LZW` para codificar strings sobre o alfabeto do `DNA` que possui `R = 4` símbolos: `'A'`, `'C'`, `'G'` e `'T'`. Suponha que cada código `LZW` é de `W = 4` bits. Utilizando esse algoritmo `LZW` para codificar uma string obtivemos a codificação `0 0 1 5 3 5 1 3 4`. Obtenha a string original e complete o dicionário `LZW` de códigos.

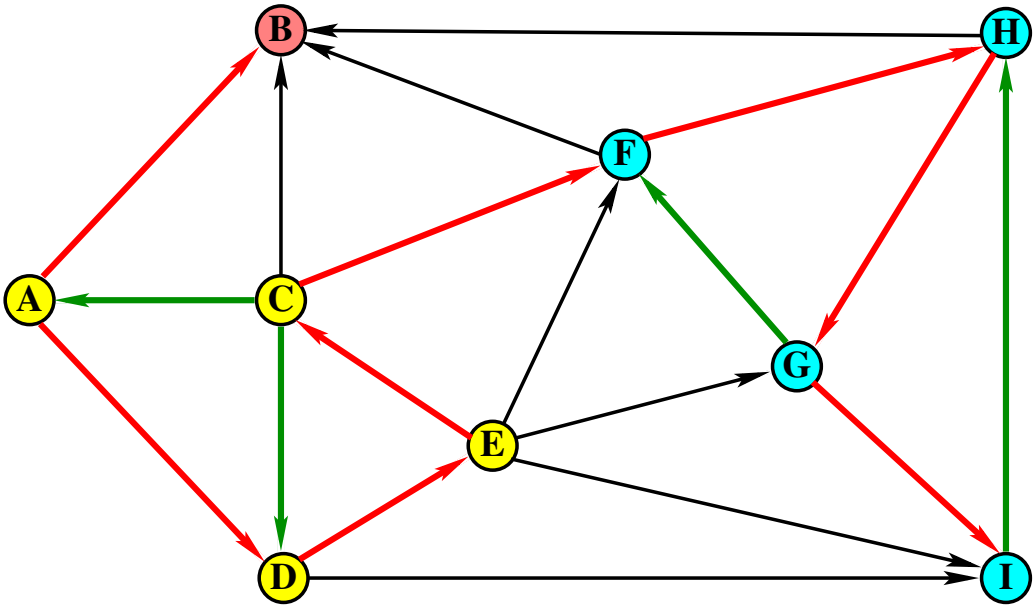
dicionário

string	código
A	0
C	1
G	2
T	3
EOF	4 (= R)
AA	5
AC	6
CA	7
AAT	8
TA	9
AAC	10
CT	11

codificação	0	0	1	5	3	5	1	3	4
string	A	A	C	A A	T	A A	C	T	EOF

6. DFS e componentes fortes (1 ponto)

Suponha que no vetor de listas de adjacência que representa o digrafo a seguir as listas estão em ordem lexicográfica. Por exemplo, quando examinando o vértice `A`, considere a arco `A-B` antes da arco `A-C`.



Liste os arcos do digrafo que estão na árvore de busca em profundidade com raiz no vértice `A`:

A-B, A-D, D-E, E-C, C-F, F-H, H-G, G-I

Liste os **arcos de retorno** da árvore da busca em profundidade obtida:

C-A, C-D, G-F, I-H

Quantos componentes fortemente conexos possui o digrafo? 3

Liste os vértices em cada um desses componentes:

componente 0: A, C, D, E

componente 1: B

componente 2: F, G, H, I

7. Ordenação de strings (1 ponto)

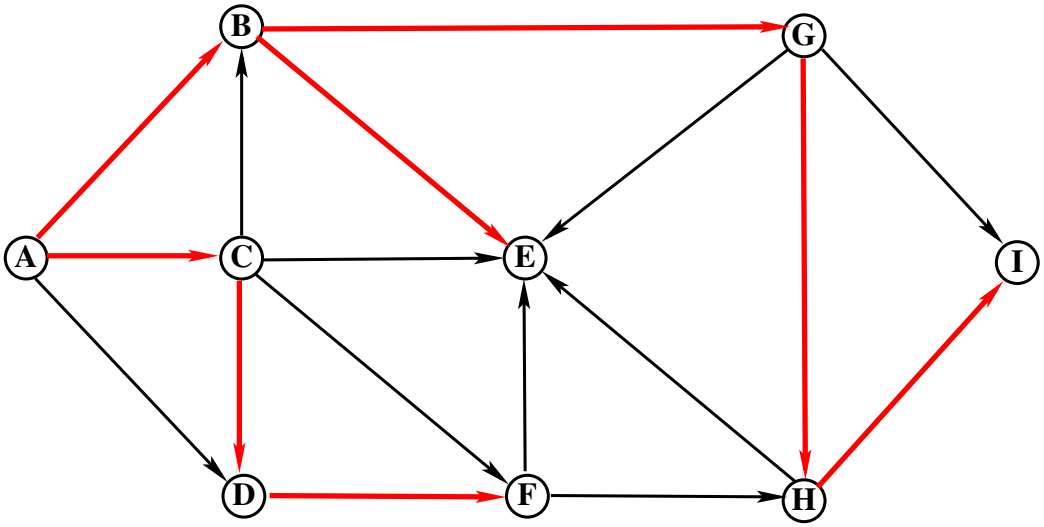
A coluna 0, mais à esquerda, contém strings a serem ordenadas. A colune 9, mais à direita. contém as strings já ordenadas. As colunas de 1 a 8 contém as strings em um passo intermediário de algum algoritmo. Para cada um dos algoritmos listados a seguir **indique uma e apenas uma coluna** entre 1 e 8 que corresponda ao estado das strings em um de seus passos intermediários.

algoritmo	coluna
`LSD radix sort`	6
`MSD radix sort`	5
`3-way radix quicksort`	3
`Quicksort`	2

Jane	Adam	Anna	Abby	Will	Adam	Jada	Abby	Adam	Abby
Adam	Alex	Adam	Cole	Seth	Alex	Emma	Adam	Dave	Adam
Mary	Cole	Abby	Alex	Ryan	Abby	Ella	Alex	Erik	Alex
Jeff	Dave	Ella	Anna	Sean	Anna	Maya	Anna	Erin	Anna
Erik	Erik	Emma	Adam	Mark	Cole	Anna	Cole	Evan	Cole
Dave	Erin	Dave	Dave	Noah	Dave	Sara	Dave	Jack	Dave
Evan	Evan	Alex	Erin	Owen	Erik	Eric	Ella	Jada	Ella
Sean	Jack	Cole	Emma	Sara	Evan	Jane	Emma	Jane	Emma
Erin	Jada	Eric	Ella	Hart	Erin	Dave	Eric	Jeff	Eric
Jada	Jane	Jada	Eric	Joey	Emma	Luke	Erik	Mary	Erik
Jack	Jeff	Jack	Erik	Jack	Ella	Kyle	Erin	Noah	Erin
Noah	Kyle	Noah	Evan	Maya	Eric	Cole	Evan	Sean	Evan
Luke	Luke	Luke	Luke	Luke	Hart	Jake	Luke	Luke	Hart
Kyle	Mary	Kyle	Kyle	Kyle	Jane	Jeff	Kyle	Kyle	Jack
Owen	Noah	Owen	Owen	Mary	Jeff	Noah	Owen	Owen	Jada
Seth	Owen	Seth	Seth	Jeff	Jada	Seth	Seth	Seth	Jake
Cole	Sean	Sean	Sean	Eric	Jack	Leah	Jada	Cole	Jane
Alex	Seth	Evan	Sara	Alex	Joey	Josh	Mary	Alex	Jeff
Hart	Abby	Hart	Hart	Erin	John	Erik	Hart	Hart	Joey
Mark	Anna	Mark	Mark	Jada	Josh	Jack	Mark	Mark	John
Joey	Ella	Joey	Joey	Erik	Jake	Mark	Joey	Joey	Josh
Emma	Emma	Erik	Will	Emma	Kyle	Will	Sean	Emma	Kyle
Ella	Eric	Jeff	Jeff	Ella	Luke	Adam	Noah	Ella	Leah
Lily	Hart	Lily	Lily	Lily	Lily	Evan	Lily	Lily	Lily
Maya	Jake	Maya	Maya	Dave	Leah	Sean	Maya	Maya	Luke
Leah	Joey	Leah	Leah	Leah	Mary	Erin	Leah	Leah	Mark
Abby	John	Mary	Mary	Abby	Mark	Owen	Jane	Abby	Mary
Anna	Josh	Jane	Jane	Anna	Maya	John	Jeff	Anna	Maya
John	Leah	John	John	John	Noah	Ryan	John	John	Noah
Ryan	Lily	Ryan	Ryan	Evan	Owen	Hart	Ryan	Ryan	Owen
Josh	Mark	Josh	Josh	Josh	Ryan	Alex	Josh	Josh	Ryan
Jake	Maya	Jake	Jake	Jake	Sean	Mary	Jake	Jake	Sara
Sara	Ryan	Sara	Noah	Jane	Seth	Joey	Sara	Sara	Sean
Will	Sara	Will	Jack	Cole	Sara	Lily	Will	Will	Seth
Eric	Will	Erin	Jada	Adam	Will	Abby	Jack	Eric	Will
0	1	2	3	4	5	6	7	8	9

8. Ordenação topológica (1 ponto)

Considere o seguinte digrafo.



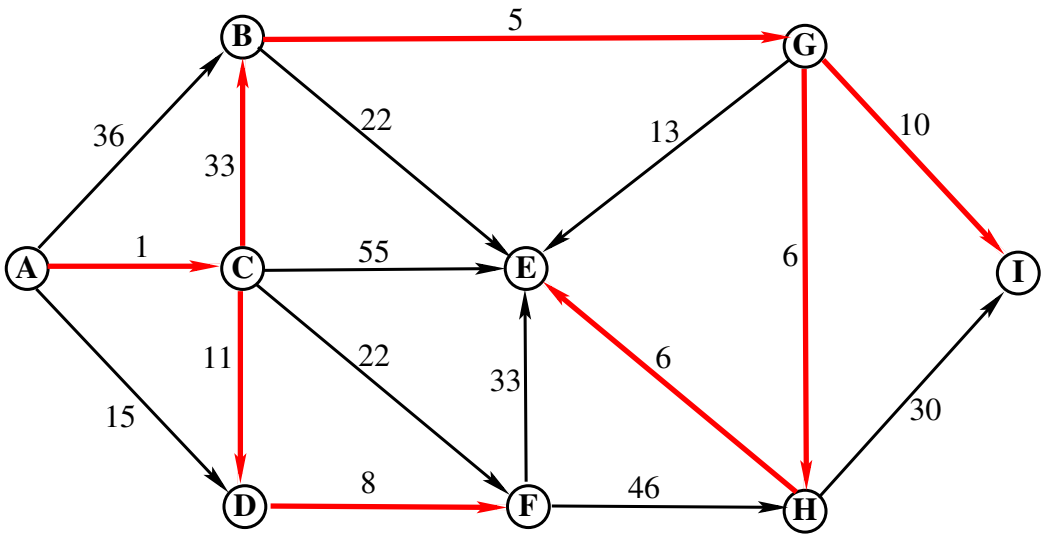
Apresente uma ordem topológica dos vértices do digrafo:

A C D F B G H I E

9. Caminhos mínimos em DAGs (1 ponto)

Suponha que no vetor de listas de adjacência que representa o digrafo acíclico (DAG) a seguir as listas estão em ordem lexicográfica. Por exemplo, quando examinando o vértice `A`, considere a arco `A-B` antes da arco `A-C`.

No digrafo simule o **algoritmo para encontrar caminhos mínimos em digrafos acíclicos** a partir do vértice `A`.



Na tabela a seguir cada linha corresponde a uma iteração do algoritmo. Para cada iteração mostre o vértice examinado e os vetores `distTo[]` e `edgeTo[]` após o vértice ter sido examinado. Na linha com `início` mostre o estado inicial dos vetores. Na linha com `fim` mostre o estado final dos vetores.

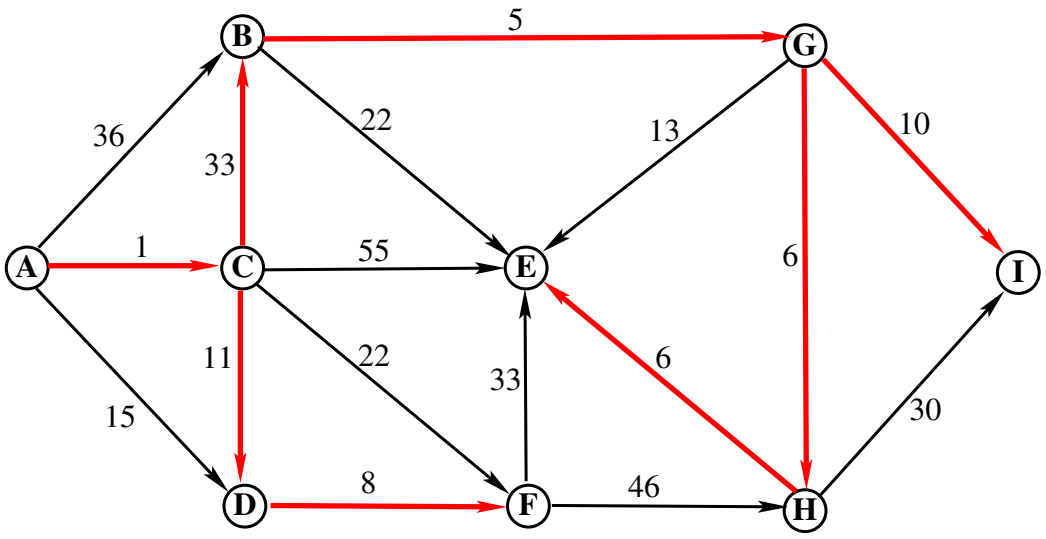
vert.	distTo[]									edgeTo[]								
exam.	A	B	C	D	E	F	G	H	I	A	B	C	D	E	F	G	H	I
início	0	∞	∞	∞	∞	∞	∞	∞	∞	null	null	null	null	null	null	null	null	null
A	0	36	1	15	∞	∞	∞	∞	∞	null	A-B	A-C	A-D	null	null	null	null	null
C	0	34	1	12	56	23	∞	∞	∞	null	C-B	A-C	C-D	C-E	C-F	null	null	null
D	0	34	1	12	56	20	∞	∞	∞	null	C-B	A-C	C-D	C-E	D-F	null	null	null
F	0	34	1	12	53	20	∞	66	∞	null	C-B	A-C	C-D	F-E	D-F	null	F-H	null
B	0	34	1	12	53	20	39	66	∞	null	C-B	A-C	C-D	F-E	D-F	B-G	F-H	null
G	0	34	1	12	52	20	39	45	49	null	C-B	A-C	C-D	G-E	D-F	B-G	G-H	G-I
H	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
I	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
E	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
fim	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I

Nota: Os valores alterados estão em **vermelho**. O algoritmo é iterativo. Em cada iteração um vértice é examinado. Os vértices são examinados em ordem topológica. O consumo de tempo do pré-processamento para determinar a ordem topológica é proporcional a `V+E`. O consumo de tempo de cada iteração é proporcional ao número de arcos saindo do vértice sendo examinado. Portanto, o consumo de tempo total do algoritmo é proporcional a `V+E`, supondo que o digrafo seja representado através de vetor de listas de adjacência. O espaço extra usado pelo algoritmo é proporcional a `V`. Esse espaço é necessário para determinar e armazenar a ordem topológica dos vértices.

10. Dijkstra (1 ponto)

Suponha que no vetor de listas de adjacência que representa o digrafo a seguir as listas estão em ordem lexicográfica. Por exemplo, quando examinando o vértice `A`, considere a arco `A-B` antes da arco `A-C`.

No digrafo simule o **algoritmo de Dijkstra** a partir do vértice `A`.



Na tabela a seguir cada linha corresponde a uma iteração do algoritmo. Para cada iteração mostre o vértice examinado e os vetores `distTo[]` e `edgeTo[]` após o vértice ter sido examinado. Na linha com `início` mostre o estado inicial dos vetores. Na linha com `fim` mostre o estado final dos vetores.

vert.	distTo[]									edgeTo[]								
exam.	A	B	C	D	E	F	G	H	I	A	B	C	D	E	F	G	H	I
início	0	∞	∞	∞	∞	∞	∞	∞	∞	null	null	null	null	null	null	null	null	null
A	0	36	1	15	∞	∞	∞	∞	∞	null	A-B	A-C	A-D	null	null	null	null	null
C	0	34	1	12	56	23	∞	∞	∞	null	C-B	A-C	C-D	C-E	C-F	null	null	null
D	0	34	1	12	56	20	∞	∞	∞	null	C-B	A-C	C-D	C-E	D-F	null	null	null
F	0	34	1	12	53	20	∞	66	∞	null	C-B	A-C	C-D	F-E	D-F	null	F-H	null
B	0	34	1	12	53	20	39	66	∞	null	C-B	A-C	C-D	F-E	D-F	B-G	F-H	null
G	0	34	1	12	52	20	39	45	49	null	C-B	A-C	C-D	G-E	D-F	B-G	G-H	G-I
H	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
I	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
E	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I
fim	0	34	1	12	51	20	39	45	49	null	C-B	A-C	C-D	H-E	D-F	B-G	G-H	G-I

Nota: Os valores alterados estão em **vermelho**. O algoritmo é iterativo. Em cada iteração o vértice um vértice é examinado. O algoritmo mantém uma fila priorizada dos vértices que ainda não foram examinados e já foram visitados (isso varia de implementação para implementação). A prioridade de cada vértice `v` é o menor peso de um caminho que utiliza apenas vértices examinados e termina em `v`. Os vértices de menor prioridade são examinados primeiro. O consumo de tempo de cada iteração é proporcional ao número de arcos saindo do vértices o tempo para atualizar a fila de prioridade. Se usarmos `IndexMinPQ` o consumo de tempo do algoritmo no pior caso será proporcional $V + E \lg V$, supondo que o digrafo é representado através de vetor de listas de adjacência. O espaço extra usado pelo algoritmo é no pior caso proporcional a `V`. Esse espaço é necessário para determinar a fila priorizada (`IndexMinPQ`). Na simulação, os valores sublinhados correspondem aos valores dos vértices e na fila priorizada. Os valores em **azul** são dos vértices sendo examindos. Foi uma mera coincidência que os vértices foram examinados em ordem topológica, como na questão 9.