A tabela final (todos marcados com *) nos permite recuperar as distâncias e os percursos. Por exemplo, para ir de **A** até **B** devemos passar por **E**; mas para passar por **E** devemos passar por **C**; finalmente o antecessor de **C** é **A**. O percurso é **A-C-E-B**.

Três observações:

- Cada vértice que fechamos tem uma ***distância de fechamento*** (que é a ***distância mínima*** desde a origem até ele) ***igual ou maior*** que a do vértice fechado antes dele. Observe as sucessivas tabelas obtidas.

- Este algoritmo ***não garante um resultado correto*** se algum arco tiver ***valor negativo***: pode acontecer que, em uma dada etapa, este arco esteja ***unindo nosso vértice-base*** a um vértice ***já fechado***. Neste caso o ***novo custo*** deste vértice já fechado será ***menor que o anterior*** – o que invalida a afirmação de que um vértice fechado não pode ter seu custo melhorado. O algoritmo não descobrirá o erro, porque não reexamina vértices fechados, logo não mais poderemos ter confiança nos fechamentos, nem nos custos obtidos a partir deles.

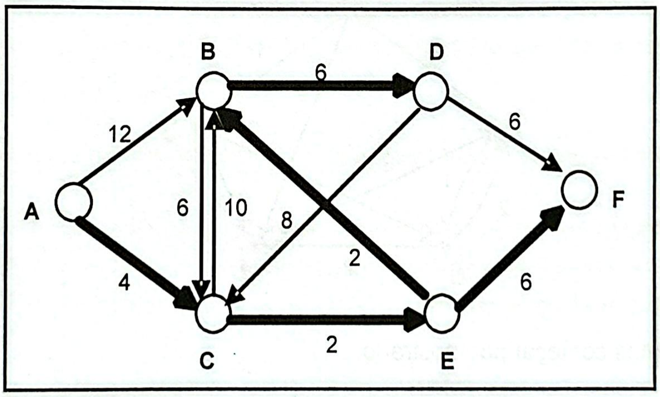
- **Muito importante:** Falar em ***distância correta*** nos traz a necessidade de definir formalmente o que é distância em um grafo, o que ainda não fizemos. Portanto:

  - ✓ Diremos que a *distância* d_{ij} do vértice *i* ao vértice *j* é infinita se não existir no grafo um caminho de *i* para *j* e é finita se ele existir; neste caso, ela será igual ao valor do menor caminho entre *i* e *j* (nula se $i = j$).

Podemos mostrar o resultado final em forma de arborescência (ver o ***Capítulo 4***). Os arcos que pertencem à arborescência que contém os caminhos mínimos estão realçados (mais grossos) na figura abaixo.

Observe que, se somarmos os valores dos arcos da arborescência que levam de **A** a cada um dos demais vértices, obteremos exatamente os valores indicados na última tabela.

Cabe notar que este processo funciona de forma análoga com grafos não orientados: apenas teremos que levar em conta os dois sentidos em cada aresta.



Para terminar, vamos ver como fica o nosso algoritmo, expresso formalmente. Ele usa um vetor chamado ***anterior***, para dar conta da última linha das tabelas que construimos. O conjunto A (***Aberto***) contém todos os vértices no início e o F (***Fechado***) é vazio. (Usamos índices numéricos para os vértices, ao invés de letras, em ordem correspondente à ordem alfabética).

Algoritmo

início

$d_{11} \leftarrow 0$; $d_{1i} \leftarrow \infty \ \forall\, i \in V - \{1\}$; [origem-origem zero; distâncias infinitas a partir da origem]

$A \leftarrow V$; $F \leftarrow \varnothing$; ***anterior*** $(i) \leftarrow 0 \ \forall\, i$;

enquanto $A \neq \varnothing$ **fazer**

  início

$r \leftarrow v \in V \mid d_{1r} = \min_{i \in A}[d_{ij}]$ [acha o vértice mais próximo da origem]

$F \leftarrow F \cup \{r\}; A \leftarrow A - \{r\};$ [o vértice r sai de *Aberto* para *Fechado*]

$S \leftarrow A \cap N^{+}(r)$ [S são os ***sucessores*** de r ***ainda abertos***]

para todo $i \in S$ **fazer**

início

$p \leftarrow \min\ [d_{1i}^{k-1}, (d_{1r} + v_{ri})]$ [compara o valor anterior com a nova soma]

se $p < d_{1i}^{k-1}$ **então**

início

$d_{1i}^{k} \leftarrow p$; ***anterior*** (i) $\leftarrow$ r; [ganhou a nova distância !]

fim;

fim;

fim;

fim.

O problema do algoritmo de Dijkstra com os arcos negativos nos leva a apresentar outro capaz de apresentar resultados corretos em presença desses arcos. Em compensação, ele não é tão rápido quanto o de Dijkstra.

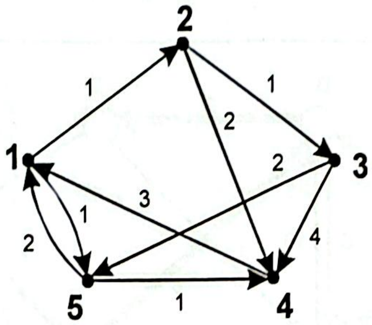
Paciência... ele nos vai ser útil, ao lidarmos com fluxos (***Capítulo 7***).

3.2.2 O algoritmo de Bellmann-Ford

Este algoritmo trabalha com os ***arcos*** do grafo – ou seja, vai procurá-los, um após o outro, em uma ordem dada, para ver se algum deles melhora algum caminho da origem até a chegada do arco.

Ele aceita arcos de valor negativo, porque nada fica sem reexame e termina quando uma rodada com todos os arcos não mostrar nenhuma melhora.

Exemplo: seja o grafo abaixo.



O algoritmo é muito simples: vamos começar por mostrá-lo.

Algoritmo

início

$d_{11} \leftarrow 0$; $d_{1i} \leftarrow \infty \ \forall i \in V - \{1\}$; ***anterior*** (i) $\leftarrow$ 0 $\forall$ i;

enquanto $\exists$ (j,i) $\in$ A | $d_{1i} > d_{1j} + v_{ji}$ **fazer** [varre todos os arcos aplicando o critério]

início

$d_{1i} \leftarrow d_{1j} + v_{ji}$; anterior (i) $\leftarrow$ j;

fim;

fim.

Vamos aplicar o algoritmo, mostrando os valores obtidos e suas modificações quando ocorrerem.

(1,2): d_{12} (∞) > d_{11} (0) + v_{12} (1) $\rightarrow$ d_{12} = 1 ***anterior*** (2) = 1

(1,5): d_{15} (∞) > d_{11} (0) + v_{15} (1) $\rightarrow$ d_{15} = 1 ***anterior*** (5) = 1

Podemos ver, por exemplo (em D^5), que o valor do menor caminho entre 5 e 3 é 4.

Olhando a posição (5,3) em R^5 encontramos 1: logo, 1 é o vértice seguinte a 5 nesse caminho.

Então procuramos (1,3) e achamos 2, que é o vértice seguinte a 1. Vamos então para (2,3) e achamos 3, que é o nosso objetivo.

Logo, o menor caminho entre 5 e 3 é (5,1,2,3).

Finalmente, nossa formalização:

```
início <dados G = (V,E); matriz de valores V(G); matriz de roteamento R = [rij];
    rij ← j  ∀i; D0 = [dij] ← V(G);
    para k = 1, ..., n fazer    [ k é o vértice-base da iteração ]
        início
            para todo i, j = 1, ..., n fazer
                se dik + dkj < dij então
                    início
                        dij ← dik + dkj;
                        rij ← rik;
                    fim;
        fim;
fim.
```

Falta apenas aplicar a notação O(.) ao algoritmo. Podemos observar que, para cada vértice-base, ele inspeciona as n^2 casas da matriz; portanto, a sua complexidade é $O(n^3)$, o que é natural neste tipo de algoritmo.

3.3 Uma aplicação a problemas de localização

Os algorítmos de caminho mínimo permitem a resposta a uma pergunta comum: se temos que escolher um local em uma cidade, ou em uma área rural, para ali colocar uma dada instalação de serviço destinada a atender a parte ou toda a cidade, ou à área que consideramos, exatamente onde essa instalação deverá ser localizada de modo a que ela funcione o melhor possível para nós e para os nossos clientes?

É claro que o uso desses algorítmos – especialmente o de Floyd, por achar diretamente os custos envolvendo todos os pares de vértices – deverá ser considerado na resolução desse problema. Há, no entanto, dois detalhes a observar:

1 em princípio, poderemos pensar, apenas, em localizar uma única instalação. Isto porque as considerações sobre as distâncias a percorrer estarão ligadas a um único vértice (a ser determinado). Com p instalações ($p > 1$) a complexidade combinatória do problema aumenta, porque teremos de achar a menor distância entre um dado conjunto de p vértices e os $n - p$ restantes. Se soubermos o valor de k (quantas instalações queremos), teremos $C_{n,p}$ possibilidades a examinar (o que já aumenta bastante o trabalho), mas se não o conhecermos e quisermos achá-lo, o problema se tornará muito mais complexo (eventualmente de <u>*complexidade exponencial*</u>: lembre que a <u>*soma das combinações*</u> $C_{n,k}$, para todo k, é uma potência de 2, como se vê na fórmula do <u>*binômio de Newton*</u>).

2 além disso, há o critério a ser utilizado, que depende da natureza da instalação. Há dois critérios mais comuns, que são:

- o *critério de lucro*, aplicável a instalações cuja prestação de serviços envolve um ganho financeiro, como fábricas, armazéns etc.. Neste caso, queremos <u>*minimizar*</u> o custo total do transporte a partir do local a ser escolhido. Neste caso, levaremos em consideração as somas de distâncias a partir de cada vértice. O problema é conhecido como um problema de *mediana* ou de *mini-soma*.