PAA - Aula 20
Caminhos Mínimos, Algoritmo de Bellman-Ford

No problema dos Caminhos Mínimos de um para todos recebemos:

- um grafo orientado (ou dirigido) G=(V, E),
- um vértice origem ‘s’ em V,
- e uma função de custos c : E -> R.
    - Note que o custo c(e) pode ser negativo, para qualquer ‘e’ em E.

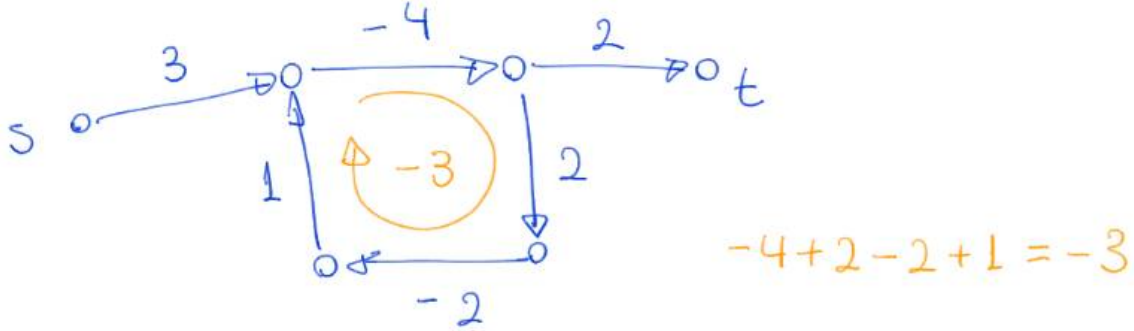
Queremos encontrar o valor do caminho mínimo de ‘s’ até cada vértice ‘v’ em V.

- Também gostaríamos que esses caminhos fossem devolvidos.

Lembre que o algoritmo guloso de Dijkstra leva tempo O(m log n),

- mas não garante a resposta correta na presença de custos negativos.
- Note que, embora arestas de custo negativo não pareçam fazer sentido
    - quando representamos grandezas físicas como distância ou tempo,
- o problema de caminhos mínimos modela cenários mais gerais,
    - i.e., a sequência de decisões de menor custo até um objetivo,
- onde as decisões correspondem a ganhos ou perdas de algum recurso,
    - como em operações financeiras, manutenção de energia, etc.

Temos que definir mais precisamente o que é um caminho mínimo
    - quando existem circuitos de custo negativo.
- Isso porque, se circuitos puderem fazer parte do caminho,
    - o caminho mínimo até um vértice pode ser arbitrariamente “pequeno”.
- Basta percorrer o circuito inúmeras vezes antes de ir ao destino.



Por outro lado, se não permitirmos que circuitos façam parte do caminho,
    - o que é bem razoável, o problema se torna NP-Difícil.
- Isso significa que ele não é tratável, i.e.,
    - não existe algoritmo polinomial para ele a menos que P = NP.
        - Veremos a relevância dessa questão no fim da disciplina.
- Só pela curiosidade, para demonstrar que tal problema é NP-Difícil,
    - podemos reduzir o problema do Caminho Hamiltoniano,
        - que é NP-Completo, para ele.

Nossa definição temporária para o problema de caminhos mínimos

- irá permitir que circuitos façam parte de caminhos,
    - pois assim o problema continua tratável,
- mas vamos supor que o grafo de entrada não possui circuitos negativos,
    - pois nesse caso um circuito nunca fará parte de um caminho mínimo.

Para perceber isso, suponha que um caminho mínimo

- contém um circuito não negativo. Note que, removendo este circuito
    - o caminho continua conectando a origem ao destino
        - e seu custo não aumentou.

No final, a suposição de que não existem circuitos negativos não será necessária,

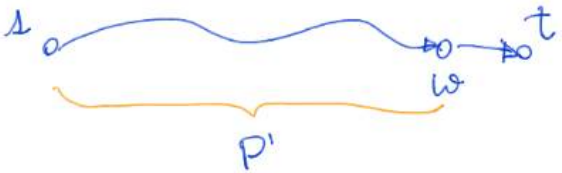
- pois nosso algoritmo será capaz de identificar a presença destes,
    - caso o grafo de entrada os contenha.

Subestrutura ótima:

Começamos imaginando uma solução ótima. Neste problema queremos encontrar
    - o caminho mínimo do vértice ‘s’ para todos os demais vértices em V,
- mas vamos começar imaginando um caminho mínimo
    - de ‘s’ até um vértice ‘t’ específico.

A sequência dos vértices do caminho nos dá uma ordem implícita para seguir.

- Assim, olhemos para o último vértice do caminho
    - e foquemos na última aresta do mesmo.
- O último vértice é ‘t’ e digamos que a última aresta é (w, t)



$$c(P) = c(P') + c(w, t)$$

Assim, temos que nosso caminho P de ‘s’ a ‘t’ é composto por:

    - um caminho P' de ‘s’ a ‘w’ e uma aresta (w, t).
- Portanto, a relação de custo é c(P) = c(P') + c(w, t)

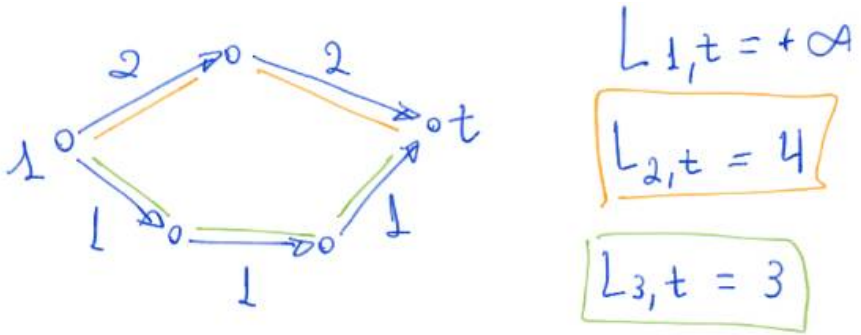
Normalmente, seguiríamos provando que P' é uma solução ótima do subproblema,

- mas vamos parar antes porque esta primeira tentativa tem uma falha.

Na subestrutura ótima, queremos definir/construir a solução ótima
    - em função de subproblemas menores.
- No entanto, não temos qualquer medida para indicar
    - que o problema do caminho mínimo de ‘s’ a ‘w’
        - é menor que o problema do caminho mínimo de ‘s’ a ‘t’.

De fato, essa é uma dificuldade recorrente ao usar programação dinâmica
- para problemas em grafos, já que grafos são definidos por conjuntos,
    - eles não têm ordens bem definidas
        - que tornem óbvio qual problema é maior e qual é menor.

A solução para esse problema, usada pelo algoritmo de Bellman-Ford, é associar
- o tamanho do subproblema ao número de arestas permitidas no caminho.



- Sendo L_i,t o custo do caminho mínimo até ‘t’ com no máximo ‘i’ arestas.

Usando esta ideia, temos uma subestrutura ótima do problema.

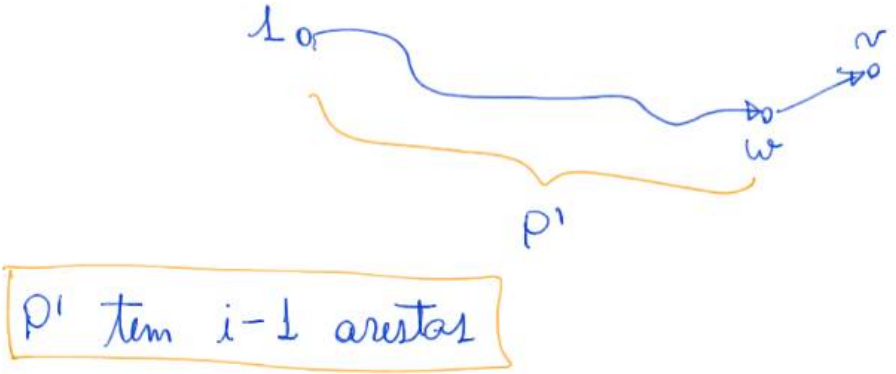
- Para cada vértice ‘v’ em V, e inteiro ‘i’ em {1, 2, ...}, seja
  - P o caminho mínimo de ‘s’ a ‘v’ com no máximo ‘i’ arestas.
- Temos dois caso.

Caso 1) Se P tem menos que ‘i’ arestas, então

- P é uma solução ótima do subproblema do caminho mínimo
  - de ‘s’ a ‘v’ com no máximo ‘i - 1’ arestas.

Caso 2) Se P tem ‘i’ arestas, seja (w, v) a última aresta de P.

- Neste caso, seja P' a parte de P que vai de ‘s’ até ‘w’.



- Assim, P' é uma solução ótima do subproblema do caminho mínimo
  - de ‘s’ a ‘w’ com no máximo ‘i - 1’ arestas.

Demonstração da otimalidade da subestrutura ótima:

Caso 1) P tem menos que ‘i’ arestas. Supondo, por contradição,

- que P não é ótimo para o subproblema com no máximo i - 1 arestas,
    - então existe P* que é solução para este problema e $c(P^*) < c(P)$.
- Como P* também é solução para o problema do caminho mínimo
    - com no máximo ‘i’ arestas, temos uma contradição
        - com a hipótese de que P era ótimo.

Caso 2) P tem ‘i’ arestas. Suponha, por contradição,

- que P' não é ótimo para o subproblema do caminho mínimo
    - de ‘s’ a ‘w’ com no máximo i - 1 arestas.
- Seja P* um caminho de ‘s’ a ‘w’ com no máximo i - 1 aresta e $c(P^*) < c(P')$.
    - Concatenando P* com (w, v) temos um caminho com
        - no máximo ‘i’ arestas
            - e custo $c(P^*) + c(w,v) < c(P') + c(w,v) = c(P)$,
    - o que contraria a hipótese de que P era ótimo.

Recorrência:

Pela subestrutura ótima, nossa recorrência terá dois parâmetros:

- número ‘i’ indicando o máximo de arestas no caminho mínimo,
- vértice ‘v’ que é o destino do caminho.

Na recorrência será escolhido o mínimo entre:

1. custo do caminho mínimo até o próprio vértice destino ‘v’,
    - mas com número máximo de arestas i - 1.
2. custo do caminho mínimo com no máximo i - 1 arestas
    - até um vizinho de ‘v’ que tenha aresta incidindo em ‘v’.

Note que, não sabemos de qual vizinho virá o melhor caminho.

- Por isso temos que verificar todos.

- Sendo \delta_in(v) o conjunto de arestas incidindo (entrando) no vértice ‘v’,
    - o caso 2 na verdade são |\delta_in(v)| casos,
        - um para cada uma dessas arestas.

Desse modo, para todo ‘v’ em V e ‘i’ em {1, 2, ...}, nossa recorrência será:

- A[i, v] = min { A[i-1, v] , min_{(w,v) \in E} {A[i-1, w] + c(w,v) } }
    - com o primeiro termo do mínimo externo correspondendo ao Caso 1,
    - e o segundo termo (o mínimo interno) correspondendo ao Caso 2.

A princípio não definimos para quantos valores de ‘i’

- vamos ter de calcular a recorrência.

- No entanto, note que se não temos circuitos negativos
    - sempre temos caminhos mínimos com no máximo n - 1 arestas.

Isso porque qualquer caminho com ‘n’ ou mais arestas

- tem que repetir vértices e, portanto, forma pelo menos um circuito.

- Como os circuitos são não negativos,
    - o mesmo pode ser removido do caminho e o custo desse não piora.
- Assim, basta calcularmos nossa recorrência para i = 1, …, n - 1
    - para encontrar todos o caminhos mínimos.

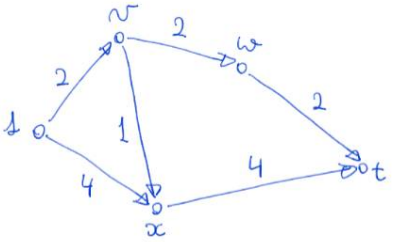
Pseudocódigo do algoritmo de Bellmand-Ford

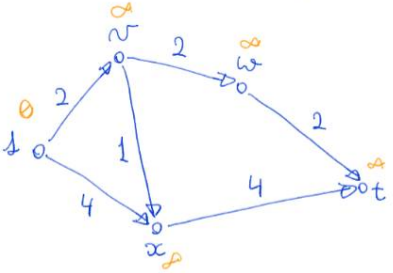
```
algBellman-Ford(G=(V,E), c, s):
    para v em V:
        A[0, v] = +\infinity
    A[0, s] = 0
    para i = 1 até n-1:
        para v em V:
            A[i, v] = min { A[i-1, v] , min_{(w,v) \in E} {A[i-1, w] + c(w,v) } }
    valores em A[n-1, v]
```

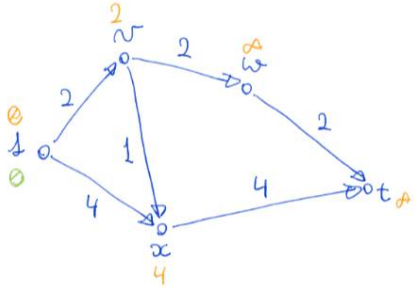
Eficiência: Note que são \Theta(n^2) subproblemas por resolver,

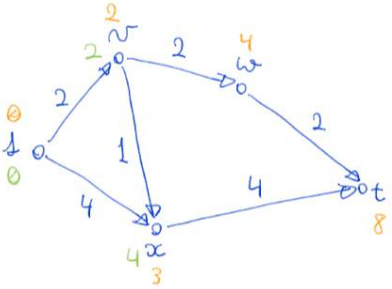
- mas o custo para resolver cada subproblema não é constante.
- Esse custo é \Theta(\delta_in(v)) para o vértice ‘v’.
  - Isso porque é necessário consultar um número de posições da matriz
    - proporcional ao número de arestas incidentes em v
- Focando em uma iteração do laço mais externo, temos que
  - o número de operações na execução do laço interno será da ordem
    - \sum_{v em V} (|\delta_in(v)| + 1) = \Theta(m).
- Como o laço mais interno é executado \Theta(n) vezes,
  - o tempo total do algoritmo é da ordem \Theta(nm).

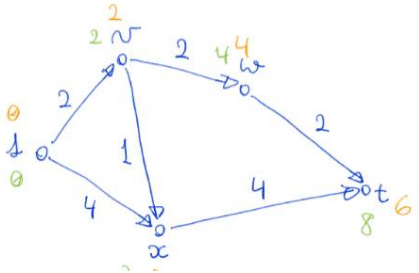
Exemplo do algoritmo de Bellman-Ford











Detectando circuito negativo:

Para que o algoritmo de Bellman-Ford detecte se o grafo de entrada
    - possui circuitos negativos, basta fazer as seguintes modificações:
- Executar o laço mais externo do algoritmo até i = n e
    - verificar se o valor A[i, v] muda para algum ‘v’ na i-ésima iteração.
- Se ocorrer alguma mudança é porque existem circuitos negativos.
    - Mais precisamente, circuitos negativos alcançáveis a partir de s.

A intuição do porque isso funciona é que na iteração n - 1
    - todos os caminhos mínimos com até n - 1 arestas já estão calculados.
- Assim, se algum caminho reduz o custo na iteração ‘n’ é por usar ‘n’ arestas.
- Sabemos que esse caminho repete vértices, pois um caminho
    - sempre usa um vértice a mais que seu número de arestas.
- Se ele repete vértices, forma algum circuito.
    - Como o custo diminuiu, esse circuito tem custo negativo.
- De fato, para encontrar tal circuito basta seguir o caminho de trás pra frente
    - a partir do vértice cujo custo foi reduzido
        - usando a ideia de reconstruir a solução a partir da matriz A.

Para mostrar formalmente que o resultado vale, vamos provar o seguinte lema:

- G não tem circuitos negativos se, e somente se,
    - no algoritmo de Bellman-Ford A[n-1, v] = A[n, v] para todo ‘v’ em V.

Prova: (→) A ida segue da corretude do algoritmo de Bellman-Ford

- e do fato que, na ausência de circuitos negativos
    - todo caminho mínimo tem no máximo n - 1 arestas.
- Assim, A[n-1, v] possui o valor do caminho mínimo de ‘s’ a ‘v’ para todo ‘v’,
    - e permitir que se use uma aresta a mais de nada vai ajudar.

(←) Na volta, considere um circuito C e vamos mostrar que

- ele não tem custo negativo. Note que, para cada vértice ‘v’ em C, temos
    - A[n-1, v] = A[n, v]
        - = min { A[n-1, v] , min_{(w,v) \in E} {A[n-1, w] + c(w,v) } }
        - <= A[n-1, w] + c(w,v),
- onde a primeira igualdade vem da hipótese da volta,
        - a segunda igualdade vem da recorrência do algoritmo,
    - e a desigualdade vale pela definição do mínimo
        - e para qualquer w tal que (w,v) está em E.

Sendo ‘w’ o vértice que precede ‘v’ em C, temos

- A[n-1, v] <= A[n-1, w] + c(w,v) → A[n-1, v] - A[n-1, w] <= c(w,v)

Finalmente, somando o custo de todas as arestas em C temos

- \sum_{(v,w) \in C} c(v,w) >= \sum_{(v,w) \in C} A[n-1, v] - A[n-1, w]
    - >= 0,
- sendo que a desigualdade segue do cancelamento dos termos A[n-1, v],
    - já que cada um aparece duas vezes no somatório,
        - uma vez com o sinal positivo e outra com o sinal negativo.

Assim, concluímos que o custo de qualquer circuito C é não negativo

- quando os valores não mudam entre as iterações n-1 e ‘n’.

Duas melhorias interessantes no algoritmo de Bellman-Ford são:

1) Parar assim que os valores A[i, v] não mudarem de uma iteração para outra.

2) Utilizar vetores A[v] e B[v], para guardar o valor do caminho mínimo até ‘v’
- e o predecessor de ‘v’ nesse caminho.

```
algMelhoradoBellman-Ford(G=(V,E), c, s):
        para v em V:
                A[v] = +\infinity
        A[s] = 0
        para i = 1 até n:
                para v em V:
                        A[v] = min { A[v] , min_{(w,v) \in E} {A[w] + c(w,v) } }
                        B[v] ← w, se w foi o vértice que minimizou a recorrência
                se não houve mudança em A nesta iteração, devolva A e B
        devolva existe ciclo negativo
```