INF05010 – Algoritmos avançados
Notas de aula

Marcus Ritt

2025-05-05

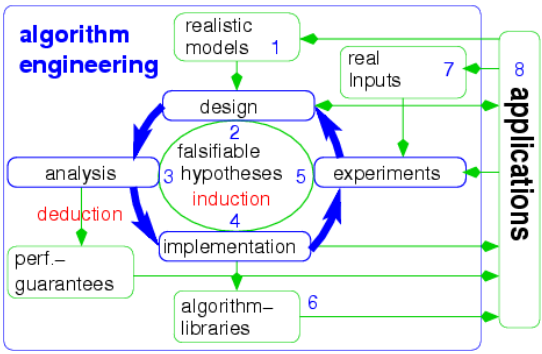
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Departamento de Informática Teórica

Agradecimentos Agradeço os estudantes dessa disciplina por críticas e comentários e em particular o Rafael de Santiago por diversas correções e sugestões.

Conteúdo

Introdução

A disciplina "Algoritmos avançados" foi criada para combinar a teoria e a prática de algoritmos. Muitas vezes a teoria de algoritmos e a prática de implementações eficientes é ensinado separadamente, em particular no caso de algoritmos avançados. Porém a experiência mostra que encontramos muitos obstáculos no caminho de um algoritmo teoricamente eficiente para uma implementação eficiente. Além disso, o projeto de algoritmos novos não termina com uma implementação eficiente, mas é alimentado pelos resultados experimentais para produzir melhores algoritmos. A figura abaixo mostra o ciclo típico da área emergente de *engenharia de algoritmos*.



{fig:ea}

Engenharia de algoritmos (*Algorithm Engineering* s.d.).

Seguindo essa filosofia, o nosso objetivo e tanto entender a teoria de algoritmos, demonstrado a sua corretude e analisando a sua complexidade, quanto dominar a prática de algoritmos, a sua implementação e avaliação experimental. Isso é refletido numa sequência alternada de aulas teóricas a praticas.

Organization

- Theoretical and practical session.

Prática: Como fazer experimentos

Note: Better to leave this for the practical part. Just lecture hint.

- Perform newsworthy experiments.
- Tie your paper to the literature.

- Use instance testbeds that can support general conclusions.
- Use efficient and effective experimental designs.
- Use reasonably efficient implementations.
- Ensure reproducibility.
- Ensure comparability.
- Report the full story.
- Draw well-justified conclusions and look for explanations.
- Present your data in informative ways.

Leitura: A Theoretician's Guide to the Experimental Analysis of Algorithms (Johnson, 2002).

Practice: Planning issues

- What kind of language we will be using?
- We need graph algorithms: I suggest C++ or python.

Prática: Ferramentas

- Profiling: gprof.
- Coverage analysis: gcov.
- Performance counters: perfctr, PAPI, perfsuite, valgrind.
- But: check also the downsides of sampling-based profiling.

There's an talk of Andrescu about engineering sorting algorithms, which has also nices lesson. The simpler ones: i) more predictable code is better; in particular push conditional statements into arithmetic, whenever you can, and ii) more regular access is better. These address branch predicition and caches. His central finding then was: the right empirical complexity model (namely in his case a weighted sum of compares, moves

and average access distance) is important. That what we observe in this lecture all the time: we can't even predict the simplest things without going very much into detail. This also extends to my work on automatic generation of algorithms.

1. Algoritmos em grafos

1.1. Representação de grafos

Um grafo pode ser representado diretamente de acordo com a sua definição por n estruturas que representam os vértices, m estruturas que representam os arcos e ponteiros entre as estruturas. Um vértice possui ponteiros para todo arco incidente sainte ou entrante, e um arco possui ponteiros para o início e término. A representação direta possui várias desvantagens. Por exemplo não temos acesso direto aos vértices para inserir um arco.
Duas representações simples são listas (ou vetores) não-ordenadas de vértices ou arestas. Uma outra representação simples de um grafo G com n vértices é uma *matriz de adjacência* $M = (m_{ij}) \in \mathbb{B}^{n \times n}$. Para vértices u, v o elemento $m_{uv} = 1$ caso existe uma arco entre u e v. Para representar grafos não-direcionados mantemos $m_{uv} = m_{vu}$, i.e., M é simétrico. A representação permite um teste de adjacência em $O(1)$. Percorrer todos vizinhos de um dado vértice v custa $O(n)$. O custo alto de espaço de $\Theta(n^2)$ restringe o uso de uma matriz de adjacência para grafos pequenos[1].
Uma representação mais eficiente é por *listas* ou *vetores* de adjacência. Neste caso armazenamos para cada vértice os vizinhos em uma lista ou um vetor. As listas ou vetores mesmos podem ser armazenados em uma lista ou um vetor global. Com isso a representação ocupa espaço $\Theta(n + m)$ para m arestas.
Uma escolha comum é um vetor de vértices que armazena listas de vizinhos. Esse estrutura permite uma inserção e deleção simples de arcos. Para facilitar a deleção de um vértice em grafos não-direcionados, podemos armazenar junto com o vizinho u do vértice v a posição do vizinho v do vértice u. A representação dos vizinhos por vetores é mais eficiente, e por isso preferível caso a estrutura do grafo é estático (Black Jr. e Martel, 1998; Park et al., 2004).
Caso escolhemos armazenar os vértices em uma lista dupla, que armazena uma lista dupla de vizinhos, em que os vizinhos são representados por posições da primeira lista, obtemos uma *lista dupla de arcos* (ingl. doubly connected arc list, DCAL). Essa estrutura permite uma inserção e remoção tanto de vértices quanto de arcos.

[1] Ainda mais espaço consuma uma *matrix de incidência* entre vértices e arestas em $\mathbb{B}^{n \times m}$.

tab:opcom}

Tabela 1.1.: Operações típicas em grafos.

Operação	Lista de arestas	Lista de vértices	Matriz de adjacência	Lista de adjacência
Inserir aresta	$O(1)$	$O(n+m)$	$O(1)$	$O(1)$ ou $O(n)$
Remover aresta	$O(m)$	$O(n+m)$	$O(1)$	$O(n)$
Inserir vértice	$O(1)$	$O(1)$	$O(n^2)$	$O(1)$
Remover vértice	$O(m)$	$O(n+m)$	$O(n^2)$	$O(n+m)$
Teste $uv \in E$	$O(m)$	$O(n+m)$	$O(1)$	$O(\Delta)$
Percorrer vizinhos	$O(m)$	$O(\Delta)$	$O(n)$	$O(\Delta)$
Grau de um vértice	$O(m)$	$O(\Delta)$	$O(n)$	$O(1)$

TBD: Figura.

Supõe que $V = [n]$. Uma outra representação compacta e eficiente conhecido como *forward star* para grafos estáticos usa um *vetor de arcos* $a_1, \ldots, a_m$. Mantemos a lista de arestas ordenado pelo começo do arco. Uma permutação σ nos dá as arestas em ordem do término. (O uso de uma permutação serve para reduzir o consumo de memória.) Para percorrer eficientemente os vizinhos de um vértice armazenamos o índice s_v do primeiro arco sainte na lista de arestas ordenado pelo começo e o índice e_v do primeiro arco entrante na lista de arestas ordenado pelo término com $s_{n+1} = e_{n+1} = m + 1$ por definição. Com isso temos $N^+(v) = \{a_{s_v}, \ldots, a_{s_{v+1}-1}\}$ com $\delta_v^+ = s_{v+1} - s_v$, e $N^-(v) = \{a_{\sigma(e_v)}, \ldots, a_{\sigma(e_{v+1}-1)}\}$ com $\delta_v^- = e_{v+1} - e_v$. A representação precisa espaço $O(n+m)$.
Tabela 1.1 mostra a complexidade de operações típicas nas diferentes representações.

1.1.1. Amostragem de grafos aleatórios

Um modelo elementar de grafos aleatórios é de Erdős e Rényi. Na variante $G_{n,p}$ temos um grafo com n vertices, e cada uma dss possíveis $M = \binom{n}{2}$ arestas é gerada com probabilidade p; na variante $G_{n,m}$ cada uma das $\binom{M}{m}$ seleções de m das M arestas tem a mesma probabilidade. (Todo que segue funciona também no caso de grafos direcionados, tomando $M = n(n-1)$.)
Para amostrar de acordo com $G_{n,p}$ podemos simplesmente percorrer todos M arestas candidatas e adicionar cada uma com probabilidade p em tempo $O(M)$ e espaço $O(m)$. Neste caso o número de arestas é variável, de acordo com uma distribuição binomial $B(m, p)$ com valor esperado de arestas $m = pM$ e

desvio padrão de $\sqrt{mp(1-p)} = \Theta(n\sqrt{p(1-p)}) = \Theta(n)$. Uma alternativa mais rápida pode ser amostrar o número de arestas de acordo com $B(m, p)$ e depois usar o modelo $G_{n,m}$.
Para amostrar de acordo com $G_{n,m}$ podemos usar um algoritmo de amostragem sem reposição (ver 6.2.1) para selecionar as arestas em tempo e espaço $O(m)$. (Uma forma simples, mas menos eficiente é aplicar a *amostragem por rejeição*: repetidamente selecionar uma aresta aleatória dos M e rejeitar arestas já selecionadas. O tempo esperado de amostrar a i-ésima aresta é $M/(M-i)$ e logo o tempo esperado é

$$\begin{aligned} E[T] &= \frac{M}{M-0} + \frac{M}{M-1} + \cdots + \frac{M}{M-m+1} \\ &= M(H_M - H_{M-m}) \leq M(\ln M - \ln(M-m)). \end{aligned}$$

Com $m = pM$ obtemos $M - m = (1-p)M$ e logo $E[T] = M \ln \frac{1}{1-p}$.)

(Older, simpler estimate.) We focus first on ER with fixed density $\rho \in [0, 1]$, and thus samples $m = \lceil \rho n^2 \rceil$ edges. If we store all edges, this can be done by rejection sampling. (This is also what Knuth in the SGB does.) In the worst case, we need $(1-\rho)^{-1}$ samples per edge, and thus time $\rho(1-\rho)^{-1}n^2$. This diverges for $\rho \to 1$, but if we accept a factor of, say, 2, then up to $\rho = 1/2$ we are good. We will also have to store all edges in memory. So for $\rho > 0.5$ we can opt to store the left out edges. Then the output takes time n^2, since we have to loop over all edges, but since ρ is high we again are at most a factor of 2 slower.

Gallo e Pallottino (1988) is a rather old, but broad survey of shortest path algorithms. It has a good experimental comparison that shows that Dial's algorithm and a list search algorithm perform well in practice (but we have to consider that these experiments were done when the importance of few memory accesses was lesser).

Mais operações:

- Contração de uma aresta.
- Contração de um par de vértices.

See Harold N. Gabow et al. (1989).

1.2. Caminhos e ciclos Eulerianos

Um *caminho Euleriano* passa por toda arestas de grafo exatamente uma vez. Um caminho Euleriano fechado é um ciclo Euleriano. Um grafo é *Euleriano* caso ele possui um ciclo Euleriano que passa por cada vértice (pelo menos uma vez).

Proposição 1.1
Uma grafo não-direcionado $G = (V, E)$ é Euleriano sse G é conectado e cada vértice tem grau par.

Prova. Por indução sobre o número de arestas. A base da indução é um grafo com um vértice e nenhuma aresta que satisfaz a proposição. Supõe que os grafos com $\leq m$ arestas satisfazem a proposição e temos um grafo G com $m+1$ arestas. Começa por um vértice v arbitrário e procura um caminho que nunca passa duas vezes por uma aresta até voltar para v. Isso sempre é possível porque o grau de cada vértice é par: entrando num vértice sempre podemos sair. Removendo este caminho do grafo, obtemos uma coleção de componentes conectados com menos que m arestas, e pela hipótese da indução existem ciclos Eulerianos em cada componente. Podemos obter um ciclo Euleriano para o grafo original pela concatençaõ desses ciclos Eulerianos. ■

Pela prova temos o seguinte algoritmo com complexidade $O(|E|)$ para encontrar um ciclo Euleriano na componente de $G = (V, E)$ que contém $v \in V$:

:hierholzer}

Algoritmo 1.1 (Caminho Euleriano)

```
1  Euler(G = (V,E), v ∈ V) :=
2    if |E| = 0 return v
3    procura um caminho começando em v
4      sem repetir arestas voltando para v
5    seja v = v_1, v_2, ..., v_n = v esse caminho
6    remove as arestas v_1v_2, v_2v_3, ..., v_{n-1}v_n de G
7      para obter G_1
8      return Euler(G_1, v_1) + ··· + Euler(G_{n-1}, v_{n-1}) + v_n
9
10   // Usamos + para concatenação de caminhos.
11   // G_i é G_{i-1} com as arestas do
12   //  caminho Euler(G_{i-1}, v_{i-1}) removidos, i.e
13   //  G_i := (V, E(G_{i-1}) \ E(Euler(G_{i-1}, v_{i-1})))
```

Algoritmo 1.1 é de Hierholzer (1873).

1.3. Árvores geradores

Exemplo 1.1
Árvore geradora mínima através do algoritmo de Prim.

Algoritmo 1.2 (Árvore geradora mínima)
Entrada Um grafo conexo não-direcionado ponderado $G = (V, E, c)$

Saída Uma árvore $T \subseteq E$ de menor custo total.

```
1  V' := {v0} para um v0 ∈ V
2  T := ∅
3  while V' ≠ V do
4    escolhe e = {u,v} de custo mínimo
5      entre V' e V \ V' (com u ∈ V', v ∈ V \ V')
6    V' := V' ∪ {v}
7    T := T ∪ {e}
8  end while
```

{alg:prim}

Algoritmo 1.3 (Prim refinado)
Implementação mais concreta:

```
1  T := ∅
2  for u ∈ V \ {v} do
3    if u ∈ N(v) then
4      value(u) := c_uv
5      pred(u) := v
6    else
7      value(u) := ∞
8    end if
9    insert(Q, (value(u), u)) { pares (chave,elemento) }
10 end for
11 while Q ≠ ∅ do
12   v := deletemin(Q)
13   T := T ∪ {pred(v)v}
14   for u ∈ N(v) do
15   if u ∈ Q e c_vu < value(u) then
16         value(u) := c_uv
17         pred(u) := v
18         update(Q, u, c_vu)
```

```
19       end if
20     end for
21   end while
```

Custo? $n \times$ insert $+ n \times$ deletemin $+ m \times$ update.

$\Diamond$

Observação 1.1
Implementação com vetor de distâncias: insert $= O(1)$[2], deletemin $= O(n)$, update $= O(1)$, e temos custo $O(n + n^2 + m) = O(n^2 + m)$. Isso é assintoticamente ótimo para grafos densos, i.e. $m = \Omega(n^2)$. $\Diamond$

Observação 1.2
Implementação com lista ordenada: insert $= O(n)$, deletemin $= O(1)$, update $= O(n)$, e temos custo $O(n^2 + n + mn) = O(mn)$[3]. $\Diamond$

Observação 1.3
Implementação com uma lista de $\sqrt{n}$ blocos de $\sqrt{n}$ elementos, insert, deletemin e update podem ser implementados em tempo $O(\sqrt{n})$, logo o algoritmo de Prim e de Dijkstra tem complexidade $O(m\sqrt{n})$. $\Diamond$

We look at the problem of keeping a *dynamic minimum spanning tree* under vertex deletion – but with connectivity guaranteed – and vertex addition.
A simple solution. Deleting a leaf is not a problem, when deleting and inner vertex we rebuild from the resulting components (with a Kruskal step that searches for the cheapest reconnecting edge); when adding we connect cheapest. Does this work?
Cattaneo et al. (2010) keep a splay tree for the MST and a binary search tree (ACL) for remaining edges, and do

```
1  add(e) :=
2    if e enters the solution
3      add to MST
4      remove costliest cycle edge
5    else
6      add to BSt
7    end
```

in amortized time $O(\log n)$, and

[2] Com chaves compactas $[1, n]$.
[3] Na hipótese razoável que $m \geq n$.

```
remove(e) :=
  if e in MST
    remove from MST
    scan BST for replacement with 2 × findroot in O(m log n)
  else
    remove from BST
  end
```

They additionally cache calls to findroot.

1.4. Caminhos mais curtos

Um problema fundamental em grafos é encontrar caminhos mais curtos entre pares de vértices. O algoritmo de Dijkstra resolve o problema das distância de um vértice origem para todos demais em grafos com distâncias não-negativas.

Exemplo 1.2
Caminhos mais curtos com o algoritmo de Dijkstra

{alg:dijkstr

Algoritmo 1.4 (Dijkstra)
Entrada Um grafo direcionado $G = (V, A)$ com pesos $d_e \geq 0$ nos arcos arestas $a \in A$, e um vértice $s \in V$.

Saída A distância mínima d_v entre s e cada vértice $v \in V$.

```
1  d_s := 0; d_v := ∞, ∀v ∈ V \ {s}
2  visited(v) := false, ∀v ∈ V
3  Q := ∅
4  insert(Q, (s, 0))
5  while Q ≠ ∅ do
6    v := deletemin(Q)
7    visited(v) := true
8    for u ∈ N⁺(v) do
9      if not visited(u) then
10       if d_u = ∞ then
11         d_u := d_v + d_vu
12         insert(Q, (u, d_u))
13       else if d_v + d_vu < d_u
14         d_u := d_v + d_vu
15         update(Q, (u, d_u))
16       end if
17     end if
```

```
18     end for
19   end while
```

Observação 1.4
A fila de prioridade contém pares de vértices e distâncias. O algoritmo se aplica igualmente a um grafo não-direcionado. ◊

Proposição 1.2
O algoritmo de Dijkstra possui complexidade

$$O(n) + n \times \text{deletemin} + n \times \text{insert} + m \times \text{update}.$$

Prova. O pré-processamento (1-3) tem custo $O(n)$. O laço principal é dominado por no máximo n operações insert, n operações deletemin, e m operações update. A complexidade concreta depende da implementação desses operações. ■

racorrect}

Proposição 1.3
O algoritmo de Dijkstra é correto.

Prova. Seja $\text{dist}(s, x)$ a menor distância entre s e x. Provaremos por indução que para cada vértice v selecionado na linha 6 do algoritmo $d_v = \text{dist}(s, x)$. Como base isso é correto para $v = s$. Seja $v \neq s$ um vértice selecionado na linha 6, e supõe que existe um caminho $P = s \cdots xy \cdots v$ de comprimento menor que d_v, tal que y é o primeiro vértice que não foi processado (i.e. selecionado na linha 6) ainda. (É possível que $y = v$.) Sabemos que

$$\begin{aligned}
d_y &\leq d_x + d_{xy} && \text{porque } x \text{ já foi processado} \\
&= \text{dist}(s, x) + d_{xy} && \text{pela hipótese } d_x = \text{dist}(s, x) \\
&\leq d(P) && \text{dist}(s, x) \leq d_P(s, x) \text{ e } P \text{ passa por } xy \\
&< d_v, && \text{pela hipótese}
\end{aligned}$$

uma contradição com a minimalidade do elemento extraído na linha 6. (Notação: $d(P)$: distância total do caminho P; $d_P(s, x)$: distância entre s e x no caminho P.) ■ ◊

ompheaplb}

Observação 1.5
Podemos ordenar n elementos usando um heap com n operações "insert" e n operações "deletemin". Pelo limite de $\Omega(n \log n)$ para ordenação via comparação, podemos concluir que o custo de "insert" mais "deletemin" é $\Omega(\log n)$. Portanto, pelo menos uma das operações é $\Omega(\log n)$. ◊

O caso médio do algoritmo de Dijkstra Dado um grafo $G = (V, E)$ e um vértice inicial arbitrário supõe que temos um conjunto $C(v)$ de pesos positivos com $|C(v)| = |N^-(v)|$ para cada $v \in V$. Atribuiremos permutações dos pesos em $C(v)$ aleatoriamente para os arcos entrantes em v.

Proposição 1.4 (Noshita (1985))
O algoritmo de Dijkstra chama update em média $n \log(m/n)$ vezes neste modelo.

Prova. Para um vértice v os arcos que podem levar a uma operação update em v são de forma (u, v) com $\text{dist}(s, u) \leq \text{dist}(s, v)$. Supõe que existem k arcos $(u_1, v), \ldots, (u_k, v)$ desse tipo, ordenado por $\text{dist}(s, u_i)$ não-decrescente. Independente da atribuição dos pesos aos arcos, a ordem de processamento sempre é $1, 2, \ldots, k$. O arco (u_i, v) leva a uma operação update caso

$$\begin{aligned}\text{dist}(s, u_i) + d_{u_i v} &< \min_{j:j<i}(\text{dist}(s, u_j) + d_{u_j v}).\\ &< \min_{j:j<i}(\text{dist}(s, u_i) + d_{u_j v}).\\ &< \text{dist}(s, u_i) + \min_{j:j<i} d_{u_j v}.\end{aligned}$$

Com isso temos $d_{u_i v} < \min_{j:j<i} d_{u_j v}$, i.e., $d_{u_i v}$ é um mínimo local na sequência dos pesos dos k arcos. O número esperado de máximos locais de uma permutação aleatória é $H_k - 1 \leq \ln k$ e considerando as permutações inversas, temos o mesmo número de mínimos locais. Como $k \leq \delta^-(v)$ temos um limite superior para o número de operações update em todos vértices de

$$\sum_{v \in V} \ln \delta^-(v) = n \sum_{v \in V} (1/n) \ln \delta^-(v) \leq n \ln \sum_{v \in V} (1/n)\delta^-(v) = n \ln m/n.$$

A desigualdade é justificada pela equação (A.6) observando que $\ln n$ é concava. ■

Com isso complexidade média do algoritmo de Dijkstra é

$$O(m + n \times \text{deletemin} + n \times \text{insert} + n \ln(m/n) \times \text{update}).$$

Usando uma fila de prioridade implementada por um heap binário que executa todas operações em $O(\log n)$ a complexidade média do algoritmo de Dijkstra é $O(m + n \log m/n \log n)$.

http://www.macfreek.nl/memory/Disjoint_Path_Finding

1.4.1. Tópicos

Fast marching method

A equação Eikonal (grego eikon, imagem)

$$\begin{aligned} \|\nabla T(x)\| F(x) &= 1, && x \in \Omega, \\ T|_{\partial\Omega} &= 0, \end{aligned}$$

define o tempo de chegada de uma superfície que inicia no tempo 0 na fronteira $\partial\Omega$ de um subconjunto aberto $\Omega \subseteq \mathbb{R}^3$ e se propaga com velocidade $F(x) > 0$ na direção normal[4]. O fast marching method resolve a equação Eikonal por discretizar o espaço regularmente, aproximar as derivadas do gradiente $\|\nabla T\|$ por diferenças finitas e propagar os valores com um método igual ao algoritmo de Dijkstra.
Com

$$\nabla T = (\partial T/\partial x - 1, \partial T/\partial x_2, \partial T/\partial x_3)$$

temos

$$\|\nabla T\|^2 = (\partial T/\partial x_1)^2 + (\partial T/\partial x_2)^2 + (\partial T/\partial x_3)^2 = 1/F^2.$$

Definindo as diferenças finitas

$$D^{+x_1} T = T(x_1 + 1, x_2, x_3) - T(x); \qquad D^{-x_1} T = T(x) - T(x_1 - 1, x_2, x_3)$$

podemos aproximar

$$\partial T/\partial x_1 \approx T_{x_1} = \max\{D^{-x_1} T, -D^{+x_1} T, 0\}$$

> The sign of the finite differences is explained since the unknown $T(x)$ has a later time than those of its neighbors.

e com aproximações similares para as direções y e z obtemos uma equação quadrática em $T(x)$

q:Tupdate} $$\|\nabla T\|^2 \approx T_{x_1}^2 + T_{x_2}^2 + T_{x_3}^2 = 1/F^2 \tag{1.1}$$

Na solução dessa equação valores ainda desconhecidos de T são ignorados. O fast marching method define $T = 0$ para os pontos iniciais em $\partial\Omega$ e coloca-os

[4] O método também funciona para $F(x) < 0$, mas não para $F(x)$ com sinais diferentes.

numa fila de prioridade. Repetidamente o ponto de menor tempo é extraído da fila, os vizinhos ainda não visitados são atualizados de acordo com (1.1) e entram na fila, caso ainda não fazem parte. (Na terminologia do fast marching method, os pontos com distância já conhecida são "vivos" (*alive*), os pontos na fila formam a "faixa estreita" (*narrow band*), os restantes pontos são "distantes" (*far away*).)

> See Sethian's page. Seems to complicated for 2014/2, because we want also to do A^* (see below).
> My above description: Baerentzen, On the implementation of fast marching methods for 3D lattices.

Busca informada

O algoritmo de Dijkstra encontra o caminho mais curto de um vértice origem $s \in V$ para todos os outros vértices num grafo ponderado $G = (V, E, d)$. Caso estamos interessados somente no caminho mais curto para um único vértice destino $t \in T$, podemos parar o algoritmo depois de processar t. Isso é uma aplicação muito comum, por exemplo na busca da rota mais curta em sistemas de navegação. Uma *busca informada* processa vértices que estimadamente são mais próximos do destino com preferência. O objetivo é processar menos vértices antes de encontrar o destino. Um dos algoritmos mais conhecidos de busca informada é o algoritmo A^*. Para cada vértice $v \in V$ com distância $g(v)$ da origem s, ele usa uma função heurística $h : V \to \mathbb{R}_{\geq 0}$ que estima a distância para o destino t e processa os vértices em ordem crescente do custo total estimado

$$f(v) = g(v) + h(v). \tag{1.2}$$

O desempenho do algoritmo A^* depende da qualidade de heurística h. Ele pode, diferente do algoritmo de Dijkstra, processar vértices múltiplas vezes, case ele descobre um caminho mais curto para um vértice já processado. Isso é a principal diferença com o algoritmo de Dijkstra. Uma outra modificação é que substituímos o campo "visited" usando no algoritmo Dijkstra 1.4 por um conjunto V de vértices já visitados, porque o A^* é frequentemente aplicado em grafos com um número grande de vértices, que são explorados passo a passo sem armazenar todos vértices do grafo na memoria.

```
1  g(s) := 0
2  f(s) := g(s) + h(s)
3  C := ∅  { vértices já visitados }
4  Q := ∅
```

```
5  insert(Q, (s, f(s)))
6  while Q ≠ ∅ do
7    v := deletemin(Q)
8    C := C ∪ {v}
9    if v = t          { destino encontrado }
10     return x
11   for u ∈ N⁺(v) do
12     if u ∈ Q then { ainda aberto: atualiza }
13       g(u) := min(g(v) + d_vu, g(u))
14       f(u) := g(u) + h(u)
15       update(Q, (u, f(u)))
16     else if u ∈ C then
17       if g(v) + d_vu < g(u) then
18         { caminho menor p/ vértice já processado }
19         C := C \ {u}
20         g(u) := g(v) + d_vu
21         f(u) := g(u) + h(u)
22         insert(Q, (u, f(u)))
23       end if
24     else                        { novo vértice }
25         g(u) := g(v) + d_vu
26         f(u) := g(u) + h(u)
27         insert(Q, (u, f(u)))
28     end if
29   end for
30 end while
```

Observação 1.6
O algoritmo de Dijkstra e a busca A^* funcionam de forma idêntica quando substituímos o vértice destino $t \in V$ por um conjunto de vértices destino $T \subseteq V$. ◊

Existe uma formulação alternativa, equivalente do algoritmo A^*. Ao invés de sempre processar o vértice aberto de menor valor f podemos processar sempre o vértice aberto de menor distância $\hat{g}$ num grafo com pesos modificados $\hat{d}_{uv} = d_{uv} - h(u) + h(v)$. Com pesos modificados obtemos para a distância total de um caminho uv arbitrário P

$$\begin{aligned}\hat{g}(u,v) &= \sum_{(u',v')\in P} \hat{d}_{u'v'} = \sum_{(u',v')\in P} d_{u'v'} - h(u') + h(v') \\ &= h(v) - h(u) + \sum_{(u',v')\in P} d_{u'v'} = h(v) - h(u) + g(u,v).\end{aligned}$$

In particular: shortest paths remain shortest. By the above we have $\hat{g}_P(u,v) \leq \hat{g}_{P'}(u,v)$ iff $g_P(u,v) \leq g_{P'}(u,v)$.

Com $\hat{g}(u) = \hat{g}(s,u)$ obtemos

$$\begin{aligned} f(u) \leq f(v) &\iff g(u) + h(u) \leq g(v) + h(v) \\ &\iff \hat{g}(u) + h(s) \leq \hat{g}(v) + h(s) \\ &\iff \hat{g}(u) \leq \hat{g}(v). \end{aligned}$$

Logo a ordem de processamento por menor $\hat{g}$ ou por menor valor f é equivalente.
Para garantir a otimalidade de uma solução a heurística h tem que ser *admissível*. Caso h é *consistente* o algoritmo A^* não somente retorna a solução ótima, mas processa cada vértice somente uma vez.

These topics can be found in Edelkamp & Schrödl (ch. 2, p. 58), Russell & Norvig (ch. 4, p. 99). ES are quite detailed, but the book is somewhat sloppy (f.ex. uses "invariance" and "invariant" in Lemma 2.2), and it is not clear why the invariant (I) in Lemma 2.2 is so complicated: the second part of it seems never to be used. RN, on the other hand, are very simple: correctness follows almost in the same way as that of Dijkstra's algorithm. Also nice, and maybe best: Pearl, ch. 3.1. On the question of goal-awareness, the latter just states that $h(t) = 0$ for goal states.

Definição 1.1 (Admissibilidade e consistência)
xxx Seja $\mathrm{dist}(v,t)$ a distância mínima do vértice v ao destino t. Uma heurística h é *admissível* caso h é um limitante inferior à distância mínima, i.e.

$$h(v) \leq \mathrm{dist}(v,t). \tag{1.3}$$ {rel:dr}

Uma heurística é consistente caso o seu valor diminui de acordo com o pesos do grafo: para um arco $(u,v) \in A$

$$h(v) \geq h(u) - d_{uv}. \tag{1.4}$$ {def:cons}

The simplest way of understanding consistency: h-values are relaxed. Note that shortest path are usually distance from a source vertex v, so arc uv is relaxed if $d_v \leq d_u + d_{uv}$. Since h-values are distance to a goal t, they are relaxed if $d_u \leq d_v + d_{uv}$.
Other ways of looking at consistency: if we get more distance, the estimated distance h increases by at most the real distance; if we get closer, the estimated distance h does not decrease more than the real distance.

In inconsistency, in that sense, is a too sharp drop: g increases less than h decreases. And so f is not monotone.
For a consistent heuristic, on the other hand, we have

$$f(v) = g(v) + h(v) = g(u) + d_{uv} + h(v) \geq g(u) + h(u) = f(u)$$

if we come over uv so the total estimate is monotone.

Na representação alternativa (1.3), o critério de consistência (1.4) é equivalente com $\hat{d}_{uv} = d_{uv} - h(u) + h(v) \geq 0$. Com isso temos diretamente o

Teorema 1.1
Caso h é consistente o algoritmo A^* nunca processa um vértice mais que uma vez.

Prova. Neste caso $\hat{d}_{uv} \geq 0$. Logo todas distâncias são positivas é o algoritmo A^* é equivalente com o algoritmo de Dijkstra. Por um argumento similar ao da proposição (1.3) o A^* nunca processa um vértice duas vezes. ■

Lema 1.1
Caso h é consistente e $h(t) = 0$ (i.e reconhece o destino t), h é admissível.

Prova. Seja $P = v_0 v_1 \dots v_k$ um caminho de $v_0 = u$ a $v_k = t$. Então

$$d(P) = \sum_{i \in [k]} d_{v_{i-1}, v_i} \overset{(1.4)}{\geq} \sum_{i \in [k]} h(v_{i-1}) - h(v_i) = h(u) - h(t) = h(u).$$

Em particular, para um caminho P^* ótimo de u a t temos $h(u) \leq d(P^*) = \delta(u)$. ■

Teorema 1.2
Caso existe uma solução mínima e h é admissível o algoritmo A^* encontra a solução mínima.

Prova. Seja $P^* = v_0 v_1 \dots v_k$ um caminho ótimo de $v_0 = s$ a $v_k = t$. Caso A^* não terminou, t ainda não foi explorado. Logo existe um vértice aberto de menor índice v_i em P^*. Agora supõe que o próximo vértice explorado é t, mas o valor de t não é ótimo, i.e. $f(t) > d(P^*)$. Mas então $f(v_i) = g(v_i) + h(v_i) \leq g(v_i) + \delta(v_i) = d(P^*) < f(t)$, porque h é admissível, em contradição com a exploração de t. ■

Punchy summary is this:

1. Consistence (t-relaxed h) $\rightarrow$ Non-negative distances $\hat{d}$ $\rightarrow$ No repetition of vertices.

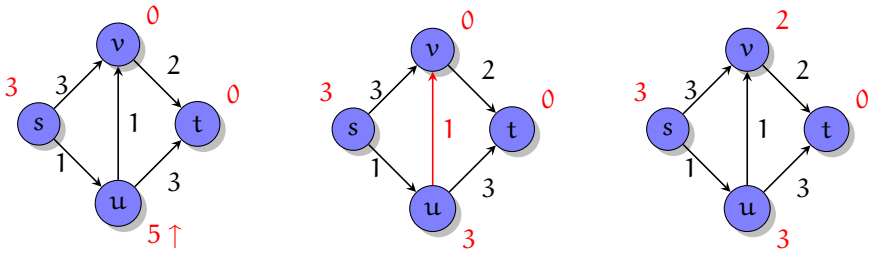


Figura 1.1.: Esquerda: Heurística não-admissível. A^* produz o valor sub-ótimo 5. Centro: Heurística admissível, mas inconsistente (arco vermelho). A^* visita v duas vezes. Direita: Heurística admissível e consistente. A^* visita cada vértice somente uma vez.

{fig:ex:as

2. Consistence & Goal-awareness → Admissibility → Correctness.
3. Not admissible: may return sub-optimal solution.
4. Only admissible: keeps correctness, but must re-open vertices. See 1.1.

Exemplo 1.3
Figure 1.1 mostra uma grafo com três funções heurísticos h diferentes. A heurística no grafo da esquerda não é admissível em u (marcado por ↑). O A^* expande s, v e depois t e termina com a distância sub-ótima 5 para chegar em t. A heurística no grafo do meio é admissível, mas não consistente: $h(u) \leq h(v)+1$ não é satisfeito. O A^* expande s, v, u, v, t, i.e. o vértice v é processado duas vezes. Finalmente a heurística no grafo da direita é consistente (e por isso admissível). O A^* expande cada vértice uma vez, na ordem s, u, t (ou s, u, v, t).

◊

Exemplo 1.4
A Figura 1.2 compara uma busca com o algoritmo de Dijkstra com uma busca com o A^* num grafo geométrico com 5000 vértices e uma aresta entre vértices de distância no máximo 0.02. Vértices não explorados são pretos, vértices explorados claros. A claridade corresponde com a ordem de exploração.

◊

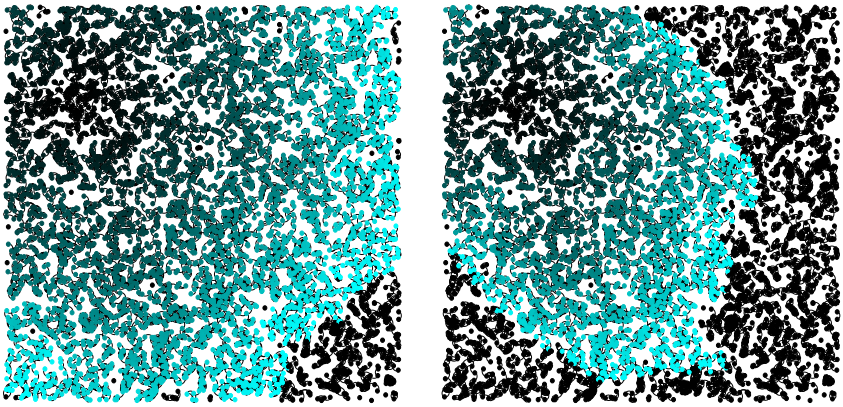


Figura 1.2.: Comparação de uma busca com o algoritmo de Dijkstra (esquerda) e o A* (direita).

{fig

1.4.2. Mais sobre caminhos mais curtos

Define um arco $a = uv \in A$ como *relaxado* caso $d_v \leq d_u + d_{uv}$, senão *tenso*. Para relaxar temos a operação

```
1   relax(a) := d_v := min{d_v, d_u + d_uv}.
```

Similarmente, define $t_v := \min_{u \in N^-(v)} d_u + d_{uv}$. Podemos definir um vértice v como *relaxado* caso $d_v \leq t_v$, e senão *tenso*. Para relaxar podemos aplicar

```
1   relax(v) := d_v := t_v.
```

> I changed the definition of a relaxed vertex from $d_v \geq t_v$ to $d_v \leq t_v$, since I could not figure out the sense of the former definition. This is more consistent with super-estimators, and we can reformulate Bellman-Ford below as relaxing each vertex per round and having $n - 1$ rounds. This now also means that a vertex is relaxed, if all incoming arcs are relaxed. The change is inconsequential for the rest of the text, since I don't use vertex relaxation anywhere else. I suppose the came from Karczmarz e Łącki (2005); but there they keep under-estimators $0 \leq d_v \leq t_v$ and update to t_v, which is different.

Com isso temos dois algoritmos simples que melhoram (super-)estimativas d_v das distâncias $\text{dist}(s, v)$, inicialmente $d_s = 0$, e $d_v = \infty$, para todos $v \neq s$.

- Dijkstra: em ordem de d_v, relaxa $a \in N^+(v)$; tempo $O(n \log n + m)$.

- Bellman-Ford: repete ate n vezes: relaxa todos $a \in A$; tempo $O(nm)$.

O algoritmo de Bellman-Ford também funciona para pesos negativos, na ausência de ciclos negativos. (E neste caso é um dos melhores algoritmos atualmente para todas distâncias de uma origem.)

I follow here mainly Schrijver, Chapter 8.

Potenciais Chama p_v, $v \in V$ um potencial caso

$$d_{uv} \geq p_v - p_u, \qquad a = uv \in A. \tag{1.5}$$ {cond:pot}

Compare to consistent heuristics: the idea is the same, the sign is different: $d_{uv} \geq h_u - h_v$. In some sense a potential models a "distance-landscape", such that the distances over p are always shorter; a consistent heuristic models a – in my view – effective potential, that shortens distances according to the potential difference (but never so much that they turn negative).

Teorema 1.3
Um potencial existe sse todo circuito (ciclo direcionado) tem comprimento não-negativo.

Prova. "⇒": Considere o circuito $C = (v_0, v_1, \ldots, v_m)$, $v_m = v_0$. Então

$$d(C) = \sum_{i \in [m]} d_{v_{i-1}, v_i} \geq \sum_{i \in [m]} p_{v_i} - p_{v_{i-1}} = p_m - p_0 = 0.$$

"⇐": seleciona algum $s \in V$, define $p_v := \mathrm{dist}(s, v)$. Isso claramente satisfaz (1.5). ■
Logo: podemos definir

$$\tilde{d}_{uv} := d_{uv} - (p_v - p_u) \geq 0, \tag{1.6}$$ {transform

uma transformação que mantém caminhos mais curtos.
Agora: como podemos encontrar circuitos negativos?

Teorema 1.4
Um circuito negativo pode ser encontrado em tempo $O(nm)$.

Prova. Roda Bellman-Ford para obter distâncias $d^0, d^1, \ldots, d^n$. Assume $d^{n-1} \neq d^n$, com testemunha $t \in V$, i.e. $d^n_t < d^{n-1}_t$. Logo existe uma st-caminhada P de distância $d(P) = d^n_t$ e de comprimento $|P| = n$. Como ela tem n arcos, contém um circuito C. Remove C de P para obter uma caminhada P' com menos que n arcos. Como

$$d(P') \geq d^{n-1}(t) > d^n(t) = d(P),$$

temos $d(C) < 0$. Caso $d^{n-1} = d^n$ nenhum circuito negativo é alcançável. ■

Teorema 1.5
Um potencial pode ser encontrado em tempo $O(nm)$ caso não tem circuitos negativos.

Prova. Adiciona um vértice s e arcos sv para todo $v \in V$ com $d_{sv} = 0$, roda Bellman-Ford e define $p_v := d_v$. Como não tem circuitos negativos $d_v = \text{dist}(s, v)$ e logo $d_{uv} \geq \text{dist}(s, v) - \text{dist}(s, u) = p_v - p_u$. ■

Caminhos mais curtos entre todos pares de vértices. Seja $d_k(s, t)$ a distância entre s e t usando somente vértices $\{s, t, v_1, \ldots, v_k\}$ para alguma ordem de vértices $v_1, v_2, \ldots, v_n$ e define $d_0(s, t) = d_{st}$ caso $st \in A$ e ∞ caso contrário. O algoritmo de *Floyd-Warshall* computa

$$d_{k+1}(s, t) := \min\{d_k(s, t), d_k(s, v_{k+1}) + d_k(v_{k+1}, t)\};$$

isso custa tempo $O(n^2)$ por iteração, logo não mais que $O(n^3)$ em total.
Com potenciais, podemos melhorar a complexidade (Johnson 1973): encontra um potencial p, aplica a transformação (1.6) e roda o algoritmo de Dijkstra n vezes. Isso custa somente $O(n(n \log n + m)) = O(nm + n^2 \log n)$ e caso o grafo tem $m = \Omega(n \log m)$ arcos temos custo $O(nm)$.

O método de Dial Assume distâncias inteiras e que temos um limite superior $\Delta \geq \max_{v \in V} \text{dist}(s, v)$. Neste caso podemos substituir a fila de prioridade no algoritmo de Dijkstra por $\Delta + 1$ "baldes" $L_0, \ldots, L_\Delta$ (implementados como listas) onde balde L_i contém os vértices de distância $d_v = i$. Mantendo o número do menor balde não-vazio μ, é simples de ver que

- podemos atualizar μ em tempo amortizado $O(1)$ sobre todas n iterações (porque μ só aumenta para pesos não negativos);
- podemos atualizar os baldes em tempo constante sobre atualizações de distâncias.

Logo: temos uma complexidade de $O(m + \Delta) = O(m + nD)$, e caso $D := \max_{a \in A} d_a$.

I follow here mainly Karczmarz e Łącki (2005).

Atualizações Considera grafos parcialmente dinâmicos, ou

- *incremental*: inserção de arcos, ou diminuição de distâncias; ou
- *decremental*: deleção de arcos, ou aumento de distâncias,

e uma sequência de no máximo Δ atualizações, no seguinte scenário:

- caminhos mais curtos de um $s \in V$ para todos demais;
- não há circuitos negativos;
- somente distâncias até L são interessantes;
- todos vértices são alcançáveis de s (adiciona arcos auxiliares sv com $d_{sv} = L+1$, para $v \neq s$: o peso desses arcos não pode ser alterado);
- pesos $d_{uv} \in \mathbb{Z}_+$.

Agora faça, no caso incremental:

```
1  updateArc(a,d):=
2    A := A ∪ {a}
3    d_a := d
4    relax(a)
5
6  relax(a = uv):=
7    if d_v ≤ d_u + d_uv: return
8    d_v := d_u + d_uv
9    for w | vw ∈ A: relax(vw)
```

{th:incremen

Teorema 1.6
O algoritmo `updateArc` é correto.

Chama d um *superestimador relaxado* caso: i) $d_s = 0$, ii) todos arcos são relaxados ($d_v \leq d_u + d_{uv}$), iii) as distâncias são superestimadas: $d_v \geq \text{dist}(s,v)$.

{lem:suprel}

Lema 1.2
Um superestimador relaxado está correto, i.e $d_v = \text{dist}(s,v)$.

Prova. (Semelhante ao Dijkstra). Vamos demonstrar $d_v \leq \text{dist}(s,v)$. Suponha, para fins de contradição, $d_v > \text{dist}(s,v)$, e seja P_v um caminho sv mais curto. Escolha um vértice v de modo que $|P_v|$ seja mínimo. Como não temos um circuito negativo $\text{dist}(s,s) = 0 = d_s$, logo $v \neq s$, e $|P_v| > 0$. Assume $P_v = s \ldots uv$. Logo $P_u = s \ldots u$ satisfaz $d_u = \text{dist}(s,u)$ pela minimalidade de $|P_v|$ e

$$d_v > \text{dist}(s,v) = \text{dist}(s,u) + d_{uv} = d_u + d_{uv}$$

portanto, uv não é relaxado, o que é uma contradição. ■

{lem:r

Lema 1.3
A função `relax` termina e retorna um superestimador relaxado.

Prova.

a) As distâncias d_v permanecem não-negativas (porque $d_v = d_u + d_{uv} \geq 0+0 = 0$). Uma chamada recursiva de `relax` ocorre somente após uma redução de pelo menos 1, portanto, as chamadas são finitas.

b) Os arcos relaxados não ficam tensos durante `relax`, por indução sobre a profundidade da recursão: um arco vw fica tenso somente quando d_v diminui: isso leva a relaxamentos recursivos.

c) O arco uv é relaxado: ele fica relaxado no início e, por b), e permanece assim.

d) Como d_v sempre representa o comprimento de algum caminho sv, $d_v \geq \text{dist}(s,v)$, ou seja, superestima.

■

Prova. (Do teorema 1.6.) Pelo lema 1.2 um superestimador relaxado é correto, pelo lema (1.3) `relax` termina e retorna um superestimador relaxado. ■

Teorema 1.7
Chamar `updateArc` Δ vezes custa $O(mL+\Delta)$ tempo total de atualização e precisa espaço $O(n)$.

Prova. As chamadas não-recursivas de `relax` custam $O(\Delta)$. As chamadas recursivas de `relax`(uv): cada vértice custa δ_v^+ e é chamado no máximo $L+1$ vezes, portanto, no total $\sum_{v\in V}\delta_v^+(L+1) = O(mL)$. ■

Similarmente, faça, no caso decremental (onde vamos supor ainda que não existem circuitos de comprimento 0):

```
1  update(v):=
2    if v = s or d_v = t_v: return
3    d_v := t_v
4    for vw ∈ A: update(w)
5
6  updateArc(a = uv, d):=
7    d_a := d
8    update(v)
```

onde $t_v := \min_{u|uv\in A} d_u + d_{uv}$.

Teorema 1.8
Algoritmo `updateArc` é correto. {th:decremen

Chama d de estimador relaxado (de vértices) se a) $d_s = 0$ e b) $d_v = t_v$, para todos os $v \neq s$.

Lema 1.4
Uma estimador relaxado está correto, i.e $d_v = \text{dist}(s, v)$. {lem:estrel}

Prova. Para $v \neq s$, deixe p_v ser o predecessor do que testemunha b). Ele existe, pois cada vértice $v \neq s$ é acessível a partir de s e, portanto, tem $\delta^-(v) \geq 1$.
Considere todos os arcos $p_v v$. Eles são acíclicos, pois para um ciclo $C = (v_1 v_2 \dots v_k)$ com $v_k = v_1$ temos

$$\begin{aligned} d_1 = d_k &= d_{k-1} + d_{k-1,k} = d_{k-2} + d_{k-2,k-1} + d_{k-1,k} \\ &= \cdots = d_1 + \sum_{i\in[k-1]} d_{i,i+1} = d_1 + d(C), \end{aligned}$$

portanto, $d(C) = 0$, o que contradiz a exclusão de ciclos de comprimento 0.
Assim, a árvore $T = (V, \{p_v v \mid v \neq s\})$ é uma árvore com raiz s (um *out-tree*), pois cada $v \neq s$ tem $\delta^-(v) = 1$ em T. Em T, temos $\text{dist}(s, v) = d_v$ e existe um caminho sv para todos os v.

Portanto, em G temos $d_v \geq \text{dist}(s, v)$, pois temos mais arcos, e todos os arcos são relaxados pela definição de t_v, ou seja, $d_v = t_v \leq d_u + d_{uv}$ para todos os $u \in N^-(v)$. Portanto, o Lema 1.2 se aplica e d está correto. ■

Lema 1.5
Algoritmo `updateArc` retorna um estimador relaxado. {lem:u

Prova. Primeiro, dê uma olhada em `update`(v). Se o invariante

$$0 \leq d_v \leq t_v, d(s) = 0, \qquad \text{(Inv)} \quad \{\texttt{ua:in}$$

é correta, mantemos $d_u = t_u$ onde for válido (pelas chamadas recursivas) e obtemos $d_v = t_v$.
(Inv) é válido por indução. a) É válido no início, pois t_v só pode aumentar. b) Se aumentarmos algum d_u, devemos ter $d_u < t_u$ e, após o aumento, ainda teremos $d_u = t_u \leq t_u$. Além disso, para todos os v, $uv \in A$, t_v pode aumentar, mas como $d_v \leq t_v$ antes, o invariante ainda se mantém após o aumento de d_u para t_u.
Além disso, como d_v só aumenta, as chamadas recursivas são limitadas a $D + 1$ alterações, portanto, `update` termina. Como sempre que t_u aumenta, `update`(u) é chamado e atualiza $d_u = t_u$, mantemos $d_u = t_u$. Segundo, antes de `updateArc`$(a = uv)$, temos $d_v = t_v$ para todos os $v \neq s$ e $d_s = 0$. Então, possivelmente $d_v < t_v$, mas pelo exposto acima, temos $d_v = t_v$ para $v \neq s$ e $d_s = 0$ após `update`(v). ■

Prova. (Do teorema 1.8.) Pelo lema (1.5) `updateArc` retorna um estimador relaxado, que pelo lema (1.4) está correto. ■

Teorema 1.9
Isso pode ser implementando de forma que o tempo total de atualização é $O(mL + \Delta)$ em espaço $O(n)$.

Prova. To realize this, we need to work harder. First, for vertex v, let $N^-(v) = (u_1, u_2, \ldots, u_{\delta_v^-})$ be its ordered neighborhood. We compute t_v in that order, and call the index of the first minimum β_v.
So we have: $d_{u_i} + d_{u_i,v} > t_v$ for $i < \beta_v$.
Now we do this: we maintain for each vertex the previous t_v, called t'_v and the current β_v. Initially we set $t'_v = \infty$ and $\beta_v = \delta_v^-$. Now define

```
1 computeMin(v) :=
2   T := {k | β_v ≤ k ≤ δ_v^- | d_{u_k} + d_{u_k,v} = t'_v}
3   // a witness of the previous minimum
```

```
4    if T ≠ ∅
5      β_v := min T
6    else
7      t'_v := t_v
8      β_v := 1
9    return t'_v
```

(This is done to compute t_v faster.) Note that `computeMin` works, since t_v can only increase. So if we find a witness of the previous minimum, we're done. Otherwise we recompute t_v.
With this we can make `update` faster.

```
1  update(v):=
2    if v = s or d_v = t_v: return
3    d_v :=computeMin(v)
4    for vw ∈ A: update(w)
```

With this in place we turn to the analysis. We have Δ non-recursive calls to `update` and at most $(L+1)\delta_v^-$ recursive ones, since each predecessor updates at most $L+1$ times. This makes $O(mL+\Delta)$ calls to `update` or `computeMin`. Naïvely, each call could cost $O(\delta_v^-)$, so we could end up with cost $\Delta + \sum_{v\in V}(L+1)\delta_v^{-2} = O(\Delta + Lmn)$. But, by maintaing β_v, t'_v we have

a) cost $j - \beta_v + 1$ when another minimum is found; here j is the next witness;

a) cost δ_v^- otherwise; but then t_v increases.

So the amortized time over the non-increasing case is $O(\delta_v^-)$, and the increasing case costs the same. In summary, then, we have at most $O(\delta_v^-)$ per increase, and thus $O(mL)$ overall, plus $O(mL+\Delta)$ for the calls. So we have $O(\Delta + mL)$ overall, and $O(n)$ space for β_v and t'_v. ■
We next turn to approximate distances for SSSP. We accept real distances $\{0\}\cup[1, D]$ and, as before, exclude zero-length circuits for the decremental case. Additionally: we limit our interest to $h^* < n$ hops, and just want an estimator d' such that

$$\mathrm{dist}(s,v) \le d'_v \le (1+\epsilon)^h \mathrm{dist}^h(s,v),$$

for $\epsilon \in (0,1)$ and $\mathrm{dist}^h(s,v)$ the shortest $\le h$-hop sv-path.
We keep an auxiliary graph: there are arcs (s,v), $v \ne s$, with distances $d_{sv} = nW$, and note that this way $\mathrm{dist}(s,v) \ne \infty$ iff $\mathrm{dist}((,s),v) < nW$.

The idea: let $\mathrm{exprnd}_a(x) = a^{\lceil \log_a x \rceil}$. Now relax to

$$d_v := \mathrm{exprnd}_{1+\epsilon} d_u + d_{uv}$$

in the incremental case and

$$t_v := \min_{u|uv\in A} \mathrm{exprnd}_{1+\epsilon} d_u + d_{uv}$$

in the decremental case.
Problem: reachability needs extra effort. Solution: maintain reachable vertices R_h (can't be seen by distances, since we overestimate!).
The new algorithms for the incremental case are this.

```
1  relax(a = uv):=
2    if  d_v ≤ exprnd_{1+ε} d_u + d_uv: return
3    d_v := exprnd_{1+ε} d_u + d_uv
4    for  w | vw ∈ A: relax(vw)
5
6  updateArc(a,d):=
7    A := A ∪ {a}
8    d_a := d
9    relax(a)
10
11 estimate(v) :=
12   return d_v if  v ∈ R_h  else  ∞
```

And for the decremental case we have this.

```
1  init() :=
2    t'_v := ∞,  d_v := 0,  β_v := δ⁻_v ,  v ∈ v
3    ∀v ∈ V : update(v)
4
5  update(v):=
6    if  v = s  or  d_v = t_v: return
7    d_v :=computeMin(v)
8    for  vw ∈ A: update(w)
9
10 updateArc(a = uv,d):=
11   d_a := d
12   if   d = ∞:  A := A \ {a}
13   update(v)
14
```

```
estimate(v) :=
  return d_v if v ∈ R_h else ∞
```

Here `computeMin` applies `exprnd`.
Fact: This works, and costs:

- Incremental case: $O(m \log(nW)/\epsilon + \Delta)$ time and $O(n)$ space;
- Decremental case: $O(m \log(nW)/\epsilon + mH^* + \Delta)$ time and $O(n)$ space.

The main idea: $D = O(\log(nW)/\epsilon)$, since distances are 0 or powers of $1+\epsilon$, but never more than $(1+\epsilon)nW$. So: $\log_{1+\epsilon} nW = 1+\log_{1+\epsilon} nW = O(\log(nW)/\epsilon)$. [Use $\log nW/\log 1+\epsilon$ and the fact that $\log 1+\epsilon \approx \epsilon$ for small ϵ.]
To maintain the reachable vertices R_h (in at most h hops). Incremental case: BFS, keep state, continue after arc insertions, total cost $O(m)$. Decremental case: use the exact algorithm with distance limit h^*: cost $O(mh^*$.
Further fact; APSP is also possible with

$$\text{dist}(u,v) \le d_{uv} \le (1+\epsilon)^{\lceil \log_2 h \rceil + 1} \text{dist}(u,v)$$

in time $O(n^3 \log(nW/\epsilon + \Delta)$.

1.4.3. Arborescências

We start with two definitions.

{def:arb:1}
Definição 1.2
A is an arborescence rooted in r if the underlying undirected graph T is a spanning tree, and for every vertex $v \in V$ a directed rv-path exists.

{def:arb:2}
Definição 1.3
A is an arborescence rooted in r if A is cycle-free, $\delta(r)^- = 0$, and all vertices $v \neq r$ have $\delta^-(v) = 1$.

Proposição 1.5
Definitions (1.2) and (1.3) are equivalent.

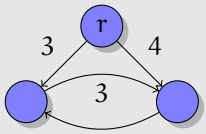
Prova. (1.2) $\rightarrow$ (1.3)**:** Since T is a spanning tree it has $n-1$ edges. Furthermore, by the existence of rv-paths we have $\delta^-(v) \ge 1$ for $v \neq r$.

This accounts for $n-1$ arcs. So, since $|T| = |A|$ we have $\delta^-(v) = 1$ for $v \neq r$, and $\delta^-(r) = 0$. Also since T has no cycles, neither has A.
(1.3) $\rightarrow$ (1.2)**:** Since $\delta^-(v) = 1$ for $v \neq r$, we can trace a path back from every $v \neq r$, and since A is cycle-free it must end in r. So there's an rv-path. We also have $|A| = n-1 = |T|$, and the paths show connectivity. Thus T is a spanning tree. ■
We futher observe

Proposição 1.6
An arborescence exists iff there's and rv-path for all $v \in V$.

Prova. Sufficiency is by definition, necessity follows from running BFS starting at r: the BFS tree is an arborescence. ■

Exercício 1.1
Why is it necessary to reduce the weights in Edmonds algorithm for spanning arborescences?
Here is a small example that shows the difference:



The basic algorithmic scheme is simple. First we observe that it is possible to subtract a constant from the in-arcs of every vertex, such that the lightest in-arc has weight 0. In the formulation of Kleinberg e Tardos (2005) we always use this transformation, and proceed as follows.

1) Form the graph obtained by selecting a 0-weight in-arc for every non-root vertex.

2) If this graph is cycle-free, it is optimal, since every non-root has a predecessor, so the chain of predecessor must terminate at the root, which shows that there is a path to each vertex.

3) Otherwise: contract a cycle, recursively find a minimum cost arborescence in the reduced graph, the expand the cycle, and remove the cycle arc that enters the single vertex of in-degre 2. This relies on the fact (that needs a proof!) that we can always find an optimal solution that enters only once into the contracted cycle.

In practice, we do not need to transform the weights to always have a 0-weight in-arc. Equivalently, we can:

1. select among the lightest in-arcs, and
2. on contracting subtract the difference from the current cycle arc to the lightest, from newly created in-arcs. This is required to "level" the different arcs. Concretely, suppose an oriented cycle $C = (v_0, \ldots, v_n)$ with arcs $a_0 = (v_n, v_0)$ and $a_i = (v_{i-1}, v_i)$. Let a^* be the lightest cycle arc. Then when contracting C to a single new vertex v, for every arc $a = (u, v_i)$ create a new arc (u, v) of weight $w_{a_i} - w_{a^*}$. Outgoing arcs (v_i, u) are transformed to (v, u) keeping the weight.

It is sufficient to contract one cycle, say, the largest, or the first we find. We need to following operations:

- Contract a cycle. This can be done in different ways, even naively by creating a new graph.
- In the new graph we need to remember the contracted vertex v to expand it.
- We need to be able to map in- and out-arcs from v to arcs in the original graph.

For this reason, a simple strategy is:

- to maintain a union-find structure on the vertices, and translate arcs (u, v) to current arcs $(\mathtt{find}(u), \mathtt{find}(v))$; we use a simple, explicit structure, where on each contraction a vertex is linked to its representing vertex in the cycle; the representing vertex is the one which has the smallest in-arc in the cycle.
- to maintain, for every vertex a weight adjustment for entering arcs; on calling $\mathtt{find}(v)$, for every non-root vertex traversed, we subtract the weight, accordingly;

1.4.4. Notes on available material

Schrijver (1997) is very short, and presents only an $O(nm)$ algorithm and refers to some others. Of these Papadimitriou e Steiglitz (1982) does not seem to contain anything about this topic, Gondran e Minoux (1984)

is also rather superficial (and contains probably a wrong theorem that claims that every optimal arborescence has to have a single entry into a contracted cycle, which is not true). I did not check Minieka (1978) since the book is even older. An open question is, if any textbook besides Kleinberg e Tardos (2005) contains a readable presentation of minimum cost arborescences. The original paper of Tarjan (1977) is readable, but required thorough studying. It has the disadvantage that the algorithm computes a maximum cost branching.

1.4.5. Notas

O algoritmo (assintoticamente) mais rápido para árvores geradoras mínimas usa *soft heaps* é possui complexidade $O(m\alpha(m,n))$, com α a função inversa de Ackermann (Chazelle, 2000; Kaplan e Zwick, 2009).
Karger propôs uma variante de heaps de Fibonacci que substituem a marca "cut" usado nos cortes em cascata por uma decisão randômica: com probabilidade 0.5 continua cortando, senão para. Além disso o heap é construído novamente com probabilidade $1/n$ depois de cada operação. Com isso "deletemin" possui complexidade esperada amortizada $\Theta(\log^2 n/\log\log n)$ (Li e Peebles, 2015).
Armazenar e atravessar árvores em ordem de van Emde Boas usando índices, similar ao ordem por busca em largura é possível (Brodal et al., 2001). O consumo de memoria das árvores de van Emde Boas pode ser reduzido para $O(n)$ (Dementiev et al., 2004; Cormen et al., 2009).
Mais sobre o fast marching method se encontra em Sethian (1999). Uma aplicação interessante é a solução do caixeiro viajante contínuo (Andrews e Sethian, 2007).

A minha apresentação da caminhos mais curtos em grafos dinâmicos segue Karczmarz e Łącki (2005).

1.4.6. Dynamic connectivity

Idea: maintain an undirected graph with n nodes, under `insert`(e) (incremental case) `delete`(e) (decremental case), or both (fully dynamic case) and allow queres `path`(x,y)?
We have:

incremental : union-find, $\alpha(n)$ amortized;

decremental Even & Shiloach (1981): $O(n)$ amortized delete, $O(1)$ find.

fully : Kapron et al. (2015): $O(\log^4 n)$ insert worst-case, $O(\log^5 n)$ delete worst-case, $O(\log n / \log\log n)$ worst-case path query, with high probability.

fully : Thorup (2000): $O(\log n(\log\log n)^3)$ expected amortized update, $O(\log n / \log\log\log n)$ path query.

The case of directed graphs: dynamic reachability, under same operations. Same as: transitive closure, $O(n^3)$, brute force update $O(n^2)$.
Incremental: $\approx$ union-find. DS of Italiano (1986). Plus: searchpath.
Idea: Maintain: 1) for each node a tree of successors, 2) a matrix of connectivity `index(i,j)`. There are conveniently combined: `index(i,h)` points to node `j` in `i`'s tree.
Example: see Italiano.

```
1  searchpath(i,j) :=
2    if index(i,j) = null return ∅
3    T := {j}
4    while parent(j) ≠ null
5      j := parent(j)
6      T := T ∪ {j}
7
8  path(i,j) := return index(i,j) ≠ null
9
10 insert(i,j) :=
11   if index(i,j) ≠ null return
12   for v ∈ [n] do
13    if index(v,i) ≠ null ∧ index(v,j) = null then
14      // new path from v to j
15      meld(v,j,i,j)
16
17 meld(i,j,u,v) :=
18   // i: destination tree; j: source tree
19   // (u,v): new edge
20   create node index(i,v)
21   insert as child of u (in i)
22   for all children w of v do
23     if index(i,w) = null then meld(i,j,v,w)
```

Now for the analysis of the above. Define a potential $\varphi = \sum_{v \in V} \varphi_v$, and let $\varphi_v = -|\mathrm{vis}(v)| - 3|\mathrm{desc}(v)|$ where we have visible edges

$$\mathrm{vis}(v) = \{(w, x) \mid w \text{ is descendant}\},$$

and $\mathrm{desc}(v)$ are all descendants of v.
It remains to analyze $\mathrm{meld}(v, j, i, j)$. Operation meld examines h_1 arcs in $\mathrm{desc}(j)$ and adds h_2 arcs to $\mathrm{desc}(v)$, where $h_2 \leq h_1 + 1 \leq n$. Furthermore,

1) h_1 arcs enter into visibility; no visible arc is examined;

2) the number of descendants increases by h_2.

Thus, the potential goes down by $h_1 + 3h_2$, and the real cost is $h_1 + 3h_2$, so we get amortized cost $O(1)$ for meld, i.e. $O(n)$ amortized cost for insert.
Survey:

incremental Italiano, $O(n)$ amortized insert, $O(1)$ query;

decremental Italiano (1988), DAGs: $O(n)$ amortized delete, $O(1)$ query;

decremental Roddity & Zwick (2002): ditto for general graphs;

fully King (1999): $O(n^2 \log n)$ amortized update, $O(1)$ query;

fully Roddity (2003): $O(n^2)$ amortized update, $O(1)$ query: this is the best with query $O(1)$.

1.5. Filas de prioridade e heaps

Uma fila de prioridade mantem um conjunto de chaves com prioridades de forma que a atualizar prioridades e acessar o elemento de menor prioridade é eficiente. Ela possui aplicações em algoritmos para calcular árvores geradores mínimas, caminhos mais curtos de um vértice para todos outros (algoritmo de Dijkstra) e em algoritmos de ordenaçao (heapsort).

1.5.1. Heaps binários

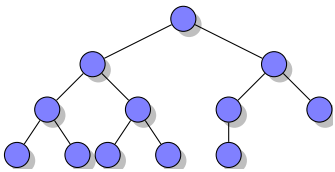
{sec:binhe

Teorema 1.10
Uma fila de prioridade pode ser implementado com custo insert = $O(\log n)$, deletemin = $O(\log n)$, update = $O(\log n)$. Portanto, uma árvore geradora mínima pode ser calculado em tempo $O(n \log n + m \log n)$.

Um *heap* é uma árvore com chaves nos vértices que satisfazem um critério de ordenação.

- *min-heap*: as chaves dos filhos são maior ou igual que a chave do pai;
- *max-heap*: as chaves dos filhos são menor ou igual que a chave do pai.

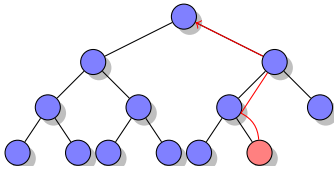
Um *heap* binário é um heap em que cada vértice possui no máximo dois filhos. Implementaremos uma fila de prioridade com um heap binário *completo*. Um heap completo fica organizado de forma que possui folhas somente no último nível, da esquerda para direita. Isso garante uma altura de $O(\log n)$.



Positivo: Achar a chave com valor mínimo (operação findmin) custa $O(1)$. Como implementar a inserção? Idéia: Colocar na última posição e restabelecer a propriedade do min-heap, caso a chave é menor que a do pai.

```
1   insert(H,c) :=
2   insere c na última posição p
3   heapify-up(H,p)
4
5   heapify-up(H,p) :=
6     if root(p) return
```

```
7       if key(parent(p))>key(p) then
8         swap(key(parent(p)),key(p))
9         heapify-up(H,parent(p))
10      end if
```



For revision, to simplify, including ideas of Kleinberg & Tardós. Arguably, this is not much simpler than what I had before, but maybe a bit more structured, and a bit more rigorous at some points. The question is if introducing all these definitions fosters understanding at the end.
We consider a heap H, with elements i, and keys k_i.

For heap H, we write $H[k_i = c]$ for the heap after setting i's key k_i to c. We call H a *quasi-heap* if there is some i and α such that $H[k_i = \alpha]$ is a heap. If H is not a heap we say that H has a violation at i. We further say H is a Δ-quasi-heap if $\alpha \geq k_i$, and a ∇-quasi-heap if $\alpha \leq k_i$. Note that if $\alpha = k_i$ we have a Δ- and ∇-quasi-heap that is also a heap. Further note that when we decrease the key of i, say from c' to c, we obtain a Δ-quasi-heap, since $\alpha = c'$ is a witness; this includes the special case of a insertion, where $c' = \infty$. Similarly, when we increase the key of i from c' to c, we obtain a ∇-quasi-heap.
In the lemmas below we write $l = \mathsf{left}(i)$ and $r = \mathsf{right}(i)$, and write $\mu \in \{l, r\}$ for the smaller of the two keys, assuming key ∞ for a descendant that does not exist. Similarly we write $p = \mathrm{parent}(i)$ and assume key $-\infty$ if i is the root.

Lema 1.6
Let H be a Δ-quasi-heap with violation at p. Then heapify-up(H,p) produces a heap in time $O(k)$, where k is p's depth.

Prova. We first note that (*) if H is a heap, then $H[k_i = k_p]$ is, too. (So parent keys can always be copied down.)
The proof is by induction over depth k. If $k = 1$ then i is the root. But then, since H is a Δ-quasi heap, we have $k_i \leq \alpha \leq \min\{k_l, k_r\}$, so H is a heap. Now consider $k > 1$. Then, either H is a heap, or $k_p > k_i$. Since H is a Δ-quasi-heap, $H[k_i = \alpha]$ is a heap, and by (*)

$H[k_i = \alpha][k_i = k_p] = H[k_i = k_p]$ is, too. But then, since $k_p > k_i$, $H[k_i = k_p][k_p = k_i]$ is a Δ-quasi-heap as witnessed by $\alpha = k_p$. Element p has depth $k-1$, so by the induction hypothesis heapify-up produces a heap.
Furthermore we have at most k calls to heapify-up, each with constant work; thus the cost is $O(k)$. ■

Lema 1.7
Let H be a ∇-quasi-heap with violation at p. Then heapify-down(H,p) produces a heap in time $O(k)$, where k is p's height.

Prova. We first note that (*) if H is a heap, then $H[k_i = k_\mu]$ is, too. (So the smallest key of a child can be copied up.)
The proof is by induction over height k. If $k = 1$ then i is a leaf. Then, since H is a ∇-quasi-heap, we have $k_i \geq \alpha \geq k_p$, so H is a heap. Now consider $k > 1$. Then either H is a heap, or $k_i > k_\mu$. Since H is a ∇-quasi-heap, $H[k_i = \alpha]$ is a heap, and by (*) $H[k_i = \alpha][k_i = k_\mu] = H[k_i = k_\mu]$ is, too. But then, since $k_i > k_\mu$, $H[k_i = k_\mu][k_\mu = k_i]$ is a ∇-quasi-heap with violation at μ as witnessed by k_μ. Element μ has height $k-1$, so by the induction hypothesis heapify-down produces a heap.
Furthermore we have at most k calls to heapify-down, each with constant work; thus the cost is $O(k)$. ■
By consequence: inserting a new element and deleting the root costs $O(\log n)$.

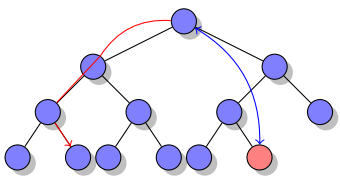
Lema 1.8
Seja T um min-heap. Decremente a chave do nó p. Após heapify-up(T, P) temos novamente um min-heap. A operação custa $O(\log n)$.

Prova. Por indução sobre a profundidade k de p. Caso $k = 1$: p é a raiz, após o decremento já temos um min-heap e heapify-up não altera ele. Caso $k > 1$: Seja c a nova chave de p e d a chave de parent(p). Caso $d \leq c$ já temos um min-heap e heapify-up não altera ele. Caso $d > c$ heapify-up troca c e d e chama heapify-up(T,parent(p)) recursivamente. Podemos separar a troca em dois passos: (i) copia d para p. (ii) copia c para parent(p). Após passo (i) temos um min-heap T' e passo (ii) diminui a chave de parent(p) e como a profundidade de parent(p) é $k-1$ obtemos um min-heap após da chamada recursiva, pela hipótese da indução.
Como a profundidade de T é $O(\log n)$, o número de chamadas recursivas também é, e como cada chamada tem complexidade $O(1)$, heapify-up tem complexidade $O(\log n)$. ■

Como remover? A idéia básica é a mesma: troca a chave com a menor chave

dos filhos. Para manter o heap completo, colocaremos primeiro a chave da última posição na posição do elemento removido.

```
1  delete(H,p):=
2    troca última posição com p
3    heapify-down(H,p)
4
5  heapify-down(H,p):=
6    if p não possui filhos return
7    if p possui um filho then
8      if key(left(p))<key(p)) then swap(key(left(p)),key(p))
9      return
10   end if
11   { p possui dois filhos }
12   if key(p)>key(left(p)) or key(p)>key(right(p)) then
13     if (key(left(p))<key(right(p)) then
14       swap(key(left(p)),key(p))
15       heapify-down(H,left(p))
16     else
17       swap(key(right(p)),key(p))
18       heapify-down(H,right(p))
19     end if
20   end if
```



Lema 1.9
Seja T um min-heap. Incremente a chave do nó p. Após heapify-down(T, p) temos novamente um min-heap. A operação custa $O(\log n)$.

Prova. Por indução sobre a altura k de p. Caso $k = 1$, p é uma folha e após o incremento já temos um min-heap e heapify-down não altera ele. Caso $k > 1$: Seja c a nova chave de p e d a chave do menor filho f. Caso $c \leq d$ já temos um min-heap e heapify-down não altera ele. Caso $c > d$ heapify-down troca c e d e chama heapify-down(T,f) recursivamente. Podemos separar a troca em dois passos: (i) copia d para p. (ii) copia c para f. Após passo (i) temos um min-heap T' e passo (ii) aumenta a chave de f e como a altura de f é $k - 1$, obtemos um min-heap após da chamada recursiva, pela hipótese da indução.

Como a altura de T é $O(\log n)$ o número de chamadas recursivas também, e como a cada chamada tem complexidade $O(1)$, heapify-up tem complexidade $O(\log n)$. ■

Última operação: atualizar a chave.

```
1  update(H,p,v) :=
2    if v < key(p) then
3      key(p):=v
4      heapify-up(H,p)
5    else
6      key(p):=v
7      heapify-down(H,p)
8    end if
```

{bt:implem

Sobre a implementação Uma árvore binária completa pode ser armazenado em um vetor v que contém as chaves. Um pontador p a um elemento é simplesmente o índice no vetor. Caso o vetor contém n elementos e possui índices a partir de 0 podemos definir

```
1  root(p) := return p = 0
2  parent(p) := return ⌊(p − 1)/2⌋
3  key(p) := return v[p]
4  left(p) := return 2p + 1
5  right(p) := return 2p + 2
6  numchildren(p) := return max(min(n − left(p), 2), 0)
```

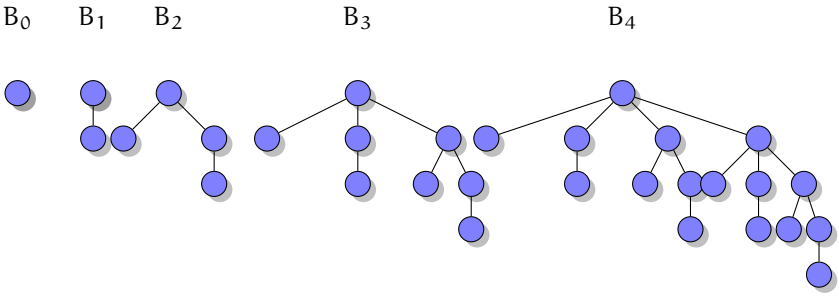
Outras observações:

- Para chamar update, temos que conhecer a posição do elemento no heap. Para um conjunto de chaves compactos $[0, n)$ isso pode ser implementado usando um vetor pos, tal que $\texttt{pos}[c]$ é o índice da chave c no heap.
- A fila de prioridade não possui teste $u \in Q$ (linha 15 do algoritmo 1.3) eficiente. O teste pode ser implementado usando um vetor visited, tal que visited[u] sse $u \notin Q$.

> Often `deletemin` takes more `sift`s, since the key we put from the last position to the root tends to be large. We can't just sift up the smallest child, and sift the "hole" down, because the hole must end up at the last position. But: we can do that as long as the right child is the smaller one, and only then fetch the key in the last position.

1.5.2. Heaps binomiais

Um heap binomial é um coleção de *árvores binomiais* que satisfazem a ordenação de um heap. A árvore binomial B_0 consiste de um único vértice. A árvore binomial B_i possui uma raiz com filhos $B_0, \ldots, B_{i-1}$. O *posto* de B_k é k. Um heap binomial contém no máximo uma árvore binomial de cada posto.



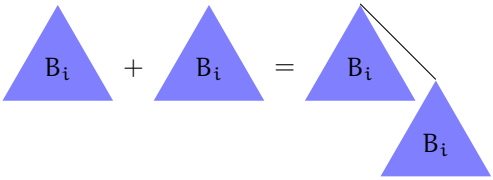
:binotree}

Lema 1.10
Uma árvore binomial tem as seguintes características:

1. B_n possui 2^n vértices, 2^{n-1} folhas (para $n > 0$), e tem altura $n + 1$.
2. O nível k de B_n (a raiz tem nível 0) tem $\binom{n}{k}$ vértices. (Isso explica o nome.)

Prova. Exercício. ■

Observação 1.7
Podemos combinar dois B_i obtendo um B_{i+1} e mantendo a ordenação do heap: Escolhe a árvore com menor chave na raiz, e torna a outra filho da primeira. Chamaremos essa operação "link". Ela tem custo $O(1)$ (veja observações sobre a implementação).



◊

Observação 1.8
Um B_i possui 2^i vértices. Um heap com n chaves consiste em $O(\log n)$ árvores. Isso permite juntar dois heaps binomiais em tempo $O(\log n)$. A operação é

semelhante à soma de dois números binários com "carry". Começa juntar os B_0. Caso tem zero, continua, case tem um, inclui no heap resultante. Caso tem dois o heap resultante não recebe um B_0. Define como "carry" o link dos dois B_0's. Continua com os B_1. Sem tem zero ou um ou dois, procede como no caso dos B_0. Caso tem três, incluindo o "carry", inclui um no resultado, e define como "carry" o link dos dois restantes. Continue desse forma com os restantes árvores. Para heaps h_1, h_2 chamaremos essa operação meld(h_1,h_2). ♢

Com a operação meld, podemos definir as seguintes operações:

- makeheap(c): Retorne um B_0 com chave c. Custo: $O(1)$.
- insert(h,c): meld(h,makeheap(c)). Custo: $O(\log n)$.
- getmin(h): Mantendo um link para a árvore com o menor custo: $O(1)$.
- deletemin(h): Seja B_k a árvore com o menor chave. Remove a raiz. Define dois heaps: h_1 é h sem B_k, h_2 consiste dos filhos de B_k, i.e. $B_0, \ldots, B_{k-1}$. Retorne meld(h_1,h_2). Custo: $O(\log n)$.
- updatekey(h,p,c): Como no caso do heap binário completo com custo $O(\log n)$.
- delete(h,c): decreasekey(h,c,$-\infty$); deletemin(h)

Em comparação com um heap binário completo ganhamos nada no caso pessimista. De fato, a operação insert possui complexidade pessimista $O(1)$ *amortizada*. Um insert individual pode ter custo $O(\log n)$. Do outro lado, isso acontece raramente. Uma análise amortizada mostra que em média sobre uma série de operações, um insert só custa $O(1)$. Observe que isso não é uma análise da complexidade média, mas uma análise da complexidade pessimista de uma série de operações.

Análise amortizada

{ex:contad

Exemplo 1.5

Temos um contador binário com k bits e queremos contar de 0 até 2^k-1. Análise "tradicional": um incremento tem complexidade $O(k)$, porque no caso pior temos que alterar k bits. Portanto todos incrementos custam $O(k2^k)$. Análise amortizada: "Poupamos" operações extras nos incrementos simples, para "gastá-las" nos incrementos caros. Concretamente, setando um bit, gastamos duas operações, uma para setar, outra seria "poupada". Incrementando, usaremos as operações "poupadas" para zerar bits. Desta forma, um incremento custa $O(1)$ e temos custo total $O(2^k)$.

Uma outra forma da análise amortizada é através uma *função potencial* φ, que associa a cada estado de uma estrutura de dados um valor positivo (a "poupança"). O custo amortizado de uma operação que transforma uma estrutura e_1 em uma estrutura e_2 e $c - \varphi(e_1) + \varphi(e_2)$, com c o custo de operação. No exemplo do contador, podemos usar como $\varphi(i)$ o número de bits na representação binário de i. Agora, se temos um estado e_1

$$\underbrace{11\cdots1}_{p \text{ bits um}}\,0\quad\underbrace{\cdots}_{q \text{ bits um}}$$

com $\varphi(e_1) = p + q$, o estado após de um incremento é

$$\underbrace{00\cdots0}_{0}\,1\,\underbrace{\cdots}_{q}$$

com $\varphi(e_2) = 1 + q$. O incremento custa $c = p + 1$ operações e portanto o custo amortizado é

$$c - \varphi(e_1) + \varphi(e_2) = p + 1 - p - q + 1 + q = 2 = O(1).$$

$\Diamond$

Resumindo: Dado um série de chamadas de uma operação com custos $c_1, \ldots, c_n$ o custo amortizado da operação é $\sum_{1 \le i \le n} c_i/n$. Caso temos m operações diferentes, o custo amortizado da operação que ocorre nos índices $J \subseteq [1, m]$ é $\sum_{i \in J} c_i/|J|$.

As somas podem ser difíceis de avaliar diretamente. Um método para simplificar o cálculo do custo amortizado é o *método potencial*. Acha uma *função potencial* φ que atribui cada estrutura de dados antes da operação i um valor não-negativo $\varphi_i \ge 0$ e normaliza ela tal que $\varphi_1 = 0$. Atribui um custo amortizado

$$a_i = c_i - \varphi_i + \varphi_{i+1}$$

a cada operação. A soma dos custos não ultrapassa os custos originais, porque

$$\sum a_i = \sum c_i - \varphi_i + \varphi_{i+1} = \varphi_{n+1} - \varphi_1 + \sum c_i \ge \sum c_i$$

Portanto, podemos atribuir a cada tipo de operação $J \subseteq [1, m]$ o custo amortizado $\sum_{i \in J} a_i/|J|$. Em particular, se cada operação individual $i \in J$ tem custo amortizado $a_i \le F$, o custo amortizado desse tipo de operação é F.

:aa:table}

Exemplo 1.6

Queremos implementar uma tabela dinâmica para um número desconhecido de elementos. Uma estratégia é reserver espaço para n elementos, manter a última posição livre p, e caso $p > n$ alocara uma nova tabela de tamanho maior. Uma implementação dessa ideia é

```
1  insert(x):=
2    if p > n then
3      aloca nova tabela de tamanho t = max{2n, 1}
4      copia os elementos x_i, 1 ≤ i < p para nova tabela
5      n := t
6    end if
7    x_p := x
8    p := p + 1
```

com valores iniciais $n := 0$ e $p := 0$. O custo de insert é $O(1)$ caso existe ainda espaço na tabela, mas $O(n)$ no pior caso.
Uma análise amortizada mostra que a complexidade amortizada de uma operação é $O(1)$. Seja Cn o custo das linhas 3–5 e D o custo das linhas 7–8. Escolhe a função potencial $\varphi(n) = 2Cp - Dn$. A função φ é satisfaz os critérios de um potencial, porque $p \geq n/2$, e inicialmente temos $\varphi(0) = 0$. Com isso o custo amortizado caso tem espaço na tabela é

$$\begin{aligned} a_i &= c_i - \varphi(i-1) + \varphi(i) \\ &= D - (2C(p-1) - Dn) + (2Cp - Dn) = C + 2C = O(1). \end{aligned}$$

Caso temos que alocar uma nova tabela o custo é

$$\begin{aligned} a_i &= c_i - \varphi(i-1) + \varphi(i) = D + Cn - (2C(p-1) - Dn) + (2Cp - 2Dn) \\ &= C + Dn + 2C - Dn = O(1). \end{aligned}$$

$\Diamond$

Custo amortizado do heap binomial Nosso potencial no caso do heap binomial é o número de árvores no heap. O custo de getmin e updatekey não altera o potencial e por isso permanece o mesmo. makeheap cria uma árvore que custa mais uma operação, mas permanece $O(1)$. deletemin pode criar $O(\log n)$ árvores novas, porque o heap contém no máximo um $B_{\lceil \log n \rceil}$ que tem $O(\log n)$ filhos, e permanece também com custo $O(\log n)$. Finalmente, insert reduz o potencial para cada link no meld e portanto agora custa somente $O(1)$ amortizado, com o mesmo argumento que no exemplo 1.5.
Desvantagem: a complexidade (amortizada) assintótica de calcular uma árvore geradora mínima permanece $O(n \log n + m \log n)$.

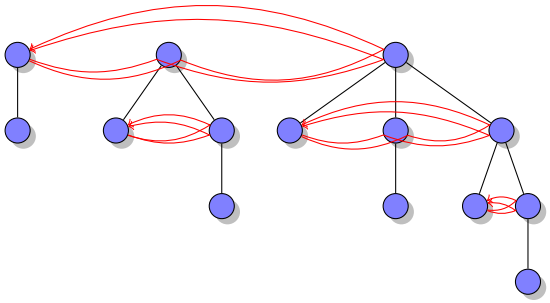
Meld preguiçosa Ao invés de reorganizar os dois heaps em um meld, podemos simplesmente concatená-los em tempo $O(1)$. Isso pode ser implementado sem custo adicional nas outras operações. A única operação que não tem complexidade $O(1)$ é deletemin. Agora temos uma coleção de árvores binomiais

não necessariamente de posto diferente. O deletemin reorganiza o heap, tal que obtemos um heap binomial com árvores de posto único novamente. Para isso, mantemos um vetor com as árvores de cada posto, inicialmente vazio. Sequencialmente, cada árvore no heap, será integrado nesse vetor, executando operações link só for necessário. O tempo amortizado de deletemin permanece $O(\log n)$.
Usaremos um potencial φ que é o dobro do número de árvores. Supondo que antes do deletemin temos t árvores e executamos l operações link, o custo amortizado é

$$(t + l) - 2t + 2(t - l) = t - l.$$

Mas $t - l$ é o número de árvores depois o deletemin, que é $O(\log n)$, porque todas árvores possuem posto diferente.

Sobre a implementação Um forma eficiente de representar heaps binomiais, é em forma de apontadores. Além das apontadores dos filhos para o os pais, cada pai possui um apontador para um filho e os filhos são organizados em uma lista encadeada dupla. Mantemos uma lista encadeada dupla também das raízes. Desta forma, a operação link pode ser implementada em $O(1)$.



1.5.3. Heaps Fibonacci

Um heap Fibonacci é uma modificação de um heap binomial, com uma operação decreasekey de custo $O(1)$. Com isso, uma árvore geradora mínima pode ser calculada em tempo $O(m + n \log n)$. Para conseguir decreasekey em $O(1)$ não podemos mais usar heapify-up, porque heapify-up custa $O(\log n)$.
Primeira tentativa:

- delete(h,p): Corta p de h e executa um meld entre o resto de h e os filhos de p. Uma alternativa é implementar delete(h,p) como decreasekey(h,p,$-\infty$) e deletemin(h).

- decreasekey(h,p): A ordenação do heap pode ser violada. Corta p é execute um meld entre o resto de h e p.

Problema com isso: após de uma série de operações delete ou decreasekey, a árvore pode se tornar "esparso", i.e. o número de vértices não é mais exponencial no posto da árvore. A análise da complexidade das operações como deletemin depende desse fato para garantir que temos $O(\log n)$ árvores no heap. Consequência: Temos que garantir, que uma árvore não fica "podado" demais. Solução: Permitiremos cada vértice perder no máximo dois filhos. Caso o segundo filho é removido, cortaremos o próprio vértice também. Para cuidar dos cortes, cada nó mantém ainda um valor booleana que indica, se já foi cortado um filho. Observe que um corte pode levar a uma série de cortes e por isso se chama de corte em cascatas (ingl. cascading cuts). Um corte em cascata termina na pior hipótese na raiz. A raiz é o único vértice em que permitiremos cortar mais que um filho. Por isso não mantemos flag na raiz.

Implementações Denotamos com h um heap, c uma chave e p um elemento do heap. minroot(h) é o elemento do heap que correspondo com a raiz da chave mínima, e cut(p) é uma marca que verdadeiro, se p já perdeu um filho.

```
1  insert(h, c) :=
2    meld(makeheap(c))
3
4  getmin(h) :=
5    return minroot(h)
6
7  delete(h,p) :=
8    decreasekey(h,p,−∞)
9    deletemin(h)
10
11 meld(h1,h2) :=
12   h := lista com raízes de h1 e h2 (em O(1))
13   minroot(h) :=
14     if key(minroot(h1))<key(minroot(h2)) h1 else h2
15
16 decreasekey(h,p,c) :=
17   key(p):= c
18   if c < key(minRoot(h))
19     minRoot(h) := p
20   if not root(p)
21     if key(parent(p))>key(p)
22       corta p e adiciona na lista de raízes de h
```

```
23        cut(p) := false
24        cascading-cut(h,parent(p))
25
26  cascading-cut(h,p) :=
27    { p perdeu um filho }
28    if root(p)
29      return
30    if (not cut(p)) then
31      cut(p) := true
32    else
33      corta p e adiciona na lista de raízes de h
34      cut(p) := false
35      cascading-cut(h,parent(p))
36    end if
37
38  deletemin(h) :=
39    remover minroot(h)
40    juntar as listas do resto de h e dos filhos de minroot(h)
41    { reorganizar heap }
42    determina o posto máximo M = M(n) de h
43    r_i := undefined para 0 ≤ i ≤ M
44    for toda raíz r do
45      remove r da lista de raízes
46      d := degree(r)
47      while (r_d not undefined) do
48        r := link(r, r_d)
49        r_d := undefined
50        d := d + 1
51      end while
52      r_d := r
53    end for
54    definir a lista de raízes pelas entradas definidas r_i
55    determinar o novo minroot
56
57  link(h_1,h_2) :=
58    if (key(h_1)<key(h_2))
59      h := makechild(h_1,h_2)
60    else
61      h := makechild(h_2,h_1)
62    cut(h_1) := false
63    cut(h_2) := false
```

```
64    return h
```

Para concluir que a implementação tem a complexidade desejada temos que provar que as árvores com no máximo um filho cortado não ficam esparsos demais e analisar o custo amortizado das operações.

Custo amortizado Para análise usaremos um potencial de $c_1 t + c_2 m$ sendo t o número de árvores, m o número de vértices marcados e c_1, c_2 constantes. As operações makeheap, insert, getmin e meld (preguiçoso) possuem complexidade (real) $O(1)$. Para decreasekey temos que considerar o caso em que o corte em cascata remove mais que uma subárvore. Supondo que cortamos n árvores, o número de raízes é $t + n$ após dos cortes. Para todo corte em cascata, a árvore cortada é desmarcada, logo temos no máximo $m - (n - 1)$ marcas depois. Portanto custo amortizado é

$$O(n) - (c_1 t + c_2 m) + (c_1(t + n) + c_2(m - (n - 1))) = c_0 n - (c_2 - c_1)n + c_2$$

e com $c_2 - c_1 \geq c_0$ temos custo amortizado constante $c_2 = O(1)$.
Com posto máximo M, a operação deletemin tem o custo real $O(M + t)$, com as seguintes contribuições

- Linha 43: $O(M)$.
- Linhas 44–51: $O(M + t)$ com t o número inicial de árvores no heap. A lista de raízes contém no máximo as t árvores de h e mais M filhos da raiz removida. O laço total não pode executar mais que $M + t$ operações link, porque cada um reduz o número de raízes por um.
- Linhas 54–55: $O(M)$.

Seja m o número de marcas antes do deletemin e m' o número depois. Como deletemin marca nenhum vértice, temos $m' \leq m$. O número de árvores t' depois de deletemin satisfaz $t' \leq M$ porque deletemin garante que existe no máximo uma árvore de cada posto. Portanto, o potencial depois de deletemin e $\varphi' = c_1 t + c_2 m' \leq c_1 M + c_2 m$, e o custo amortizado é

$$\begin{aligned} O(M + t) - (c_1 t + c_2 m) + \varphi' &\leq O(M + t) - (c_1 t + c_2 m) + (c_1 M + c_2 m) \\ &= (c_0 + c_1)M + (c_0 - c_1)t \end{aligned}$$

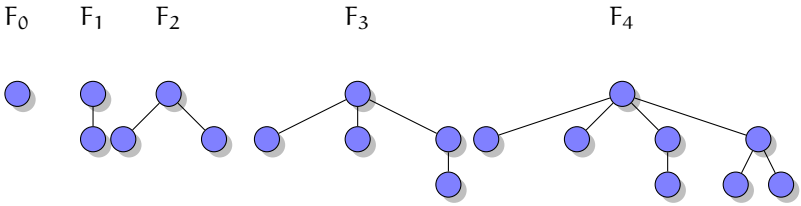
e com $c_1 \geq c_0$ temos custo amortizado $O(M)$.

Um limite para M Para provar que deletemin tem custo amortizado $\log n$, temos que provar que $M = M(n) = O(\log n)$. Esse fato segue da maneira "cautelosa" com que cortamos vértices das árvores.

Lema 1.11
Seja p um vértice arbitrário de um heap Fibonacci. Considerando os filhos na ordem temporal em que eles foram introduzidos, filho i possui pelo menos $i-2$ filhos.

Prova. No instante em que o filho i foi introduzido, p estava com pelo menos $i-1$ filhos. Portanto i estava com pelo menos $i-1$ filhos também. Depois filho i perdeu no máximo um filho, e portanto possui pelo menos $i-2$ filhos. ■

Quais as menores árvores, que satisfazem esse critério?



Lema 1.12
Cada subárvore com uma raiz p com k filhos possui pelo menos F_{k+2} vértices.

Prova. Seja S_k o número mínimo de vértices para uma subárvore cuja raiz possui k filhos. Sabemos que $S_0 = 1$, $S_1 = 2$. Define $S_{-2} = S_{-1} = 1$. Com isso obtemos para $k \geq 1$

$$S_k = \sum_{0 \leq i \leq k} S_{k-2} = S_{k-2} + S_{k-3} + \cdots + S_{-2} = S_{k-2} + S_{k-1}.$$

Comparando S_k com os números Fibonacci

$$F_k = \begin{cases} k & \text{se } 0 \leq k \leq 1 \\ F_{k-2} + F_{k-1} & \text{se } k \geq 2 \end{cases}$$

e observando que $S_0 = F_2$ e $S_1 = F_3$ obtemos $S_k = F_{k+2}$. Usando que $F_n \in \Theta(\Phi^n)$ com $\Phi = (1+\sqrt{5})/2$ (exercício!) conclui a prova. ■

Corolário 1.1
O posto máximo de um heap Fibonacci com n elementos é $O(\log n)$.

Sobre a implementação A implementação da árvore é a mesma que no caso de heaps binomiais. Uma vantagem do heap Fibonacci é que podemos usar os nós como ponteiros – lembre que a operação decreasekey precisa isso, porque os heaps não possuem uma operação de busca eficiente. Isso é possível, porque sem heapify-up e heapify-down, os ponteiros mantem-se válidos.

1.5.4. Rank-pairing heaps

Haeupler et al. (2009) propõem um rank-pairing heap (um heap "emparelhando postos") com as mesmas garantias de complexidade que um heap Fibonacci e uma implementação simplificada e mais eficiente na prática (ver observação 1.11).

Torneios Um *torneio* é uma representação alternativa de heaps. Começando com todos elementos, vamos repetidamente comparar pares de elementos, e promover o vencedor para o próximo nível (Fig. 1.3(a)). Uma desvantagem de representar torneios explicitamente é o espaço para chaves redundantes. Por exemplo, o campeão (i.e. o menor elemento) ocorre $O(\log n)$ vezes. A figura 1.3(b) mostra uma representação sem chaves repetidas. Cada chave é representado somente na comparação mais alta que ele ganhou, as outras comparações ficam vazias. A figura 1.3(c) mostra uma representação compacta em forma de *semi-árvore*. Numa semi-árvore cada elemento possui um filho *ordenado* (na figura o filha da esquerda) e um filho *não-ordenado* (na figura o filho da direita). O filho ordenado é o perdedor da comparação direta com o elemento, enquanto o filho não-ordenado é o perdedor da comparação com o irmão vazio. A raiz possui somente um filho ordenado.
Cada elemento de um torneio possui um *posto*. Por definição, o posto de uma folha é 0. Uma comparação *justa* entre dois elementos do mesmo posto r resulta num elemento com posto $r+1$ no próximo nível. Numa comparação *injusta* entre dois elementos com postos diferentes, o posto do vencedor é definido pelo maior dois postos dos participantes (uma alternativa é que o posto fica o mesmo). O posto de um elemento representa um limite inferior do número de elementos que perderam contra-lo:

{lem:rank}

Lema 1.13
Um torneio com campeão de posto k possui pelo menos 2^k elementos.

Prova. Por indução. Caso um vencedor possui posto k temos duas possibilidades: (i) foi o resultado de uma comparação justa, com dois participantes com posto $k-1$ e pela hipótese da indução com pelo menos 2^{k-1} elementos, tal que o vencedor ganhou contra pelo menos 2^k elementos. (ii) foi resultado

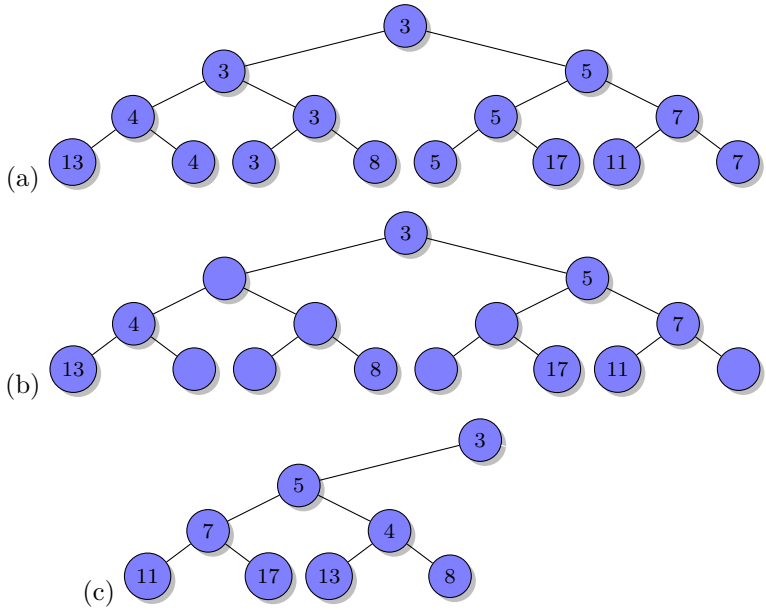


Figura 1.3.: Representações de heaps.

{fig

de uma comparação injusta. Neste caso um dos participantes possuíu posto k e o vencedor novamente ganhou contra pelo menos 2^k elementos. ■

Cada comparação injusta torna o limite inferior dado pelo posto menos preciso. Por isso uma regra na construção de torneios é fazer o maior número de comparações justas possíveis. A representação de um elemento de heap é possui quatro campos para a chave (c), o posto (r), o filho ordenado (o) e o filho não-ordendo (u):

```
1  def Node(c,r,o,u)
```

Podemos implementar as operações de uma fila de prioridade (sem update ou decreasekey) como segue:

```
1  { compara duas árvores }
2  link(t1,t2) :=
3    if t1.c < t2.c then
4      return makechild(t1,t2)
5    else
6      return makechild(t2,t1)
7    end if
8
9  makechild(s,t) :=
10   t.u := s.o
11   s.o := t
12   setrank(t)
13   s.r := s.r + 1
14   return s
15
16 setrank(t) :=
17   if t.o.r = t.u.r
18     t.r = t.o.r + 1
19   else
20     t.r = max(t.o.r,t.u.r)
21   end if
22
23 { cria um heap com um único elemento com chave c }
24 make-heap(c) := return Node(c,0,undefined,undefined)
25
26 { insere chave c no heap }
27 insert(h,c) := link(h,make-heap(c))
28
29 { união de dois heaps }
```

```
30 meld(h1,h2) := link(h1,h2)
31
32 { elemento mínimo do heap }
33 getmin(h) := return h
34
35 { deleção do elemento mínimo do heap }
36 deletemin(h) :=
37   aloca array r0...rh.o.r+1
38   t = h.o
39   while t not undefined do
40     t′ := t.u
41     t.u := undefined
42     register(t,r)
43     t := t′
44   end while
45   h′ := undefined
46   for i = 0,...,h.o.r + 1 do
47     if ri not undefined
48       h′ := link(h′,ri)
49     end if
50   end for
51   return h′
52 end
53
54 register(t,r) :=
55   if rt.o.r+1 is undefined then
56     rt.o.r+1 := t
57   else
58     t:=link(t,rt.o.r+1)
59     rt.o.r+1 := undefined
60     register(t,r)
61   end if
62 end
```

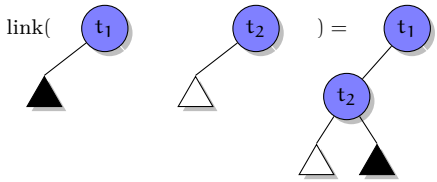
(A figura 1.4 visualiza a operação "link".)

Observação 1.9

Todas comparações de "register" são justas. As comparações injustas ocorrem na construção da árvore final nas linhas 35–39. ♢

Lema 1.14

Num torneio balanceado o custo amortizado de "make-heap", "insert", "meld" e "getmin" é $O(1)$, o custo amortizado de "deletemin" é $O(\log n)$.



ink}

Figura 1.4.: A operação "link" para semi-árvores no caso $t_1.c < t_2.c$.

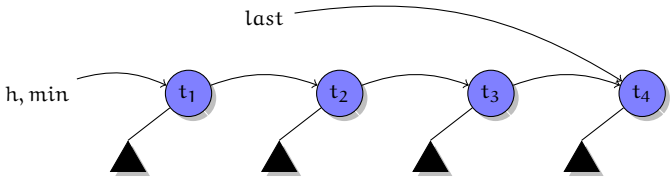


Figura 1.5.: Representação de um heap binomial.

{fig:binom

Prova. Usaremos o número de comparações injustas no torneio como potencial. "make-heap" e "getmin" não alteram o potencial, "insert" e "meld" aumentam o potencial por no máximo um. Portanto a complexidade amortizada dessas operações é $O(1)$. Para analisar "deletemin" da raiz r do torneio vamos supor que houve k comparações injustas com r. Além dessas comparações injustas, r participou em no máximo $\log n$ comparações justas pelo lema 1.13. Em soma vamos liberar no máximo $k + \log n$ árvores, que reduz o potencial por k, e com no máximo $k + \log n$ comparações podemos produzir um novo torneio. Dessas $k+\log n$ comparações no máximo $\log n$ são comparações injustas. Portanto o custo amortizado é $k + \log n - k + \log n = 2 \log n = O(\log n)$. ■

Heaps binomiais com varredura única O custo de representar o heap numa árvore única é permitir comparações injustas. Uma alternativa é permitir somente comparações justas, que implica em manter uma coleção de $O(\log n)$ árvores. A estrutura de dados resultante é similar com os heaps binomiais: manteremos uma lista (simples) de raízes das árvores, junto com um ponteiro para a árvore com a raíz de menor valor. O heap é representado pela raíz de menor valor, ver Fig. 1.5.

```
1  insert(h,c) :=
2    insere make-heap(c) na lista de raizes
3    atualize a árvore mínima
```

```
4
5  meld(h1,h2) :=
6    concatena as listas de h1 e h2
7    atualize a árvore mínima
```

Somente "deletemin" opera diferente agora:

```
1  deletemin(h) :=
2    aloca um array de listas r0...r⌈log n⌉
3    remove a árvore mínima da lista de raizes
4    distribui as restantes árvores sobre r
5
6    t := h.o
7    while t not undefined do
8      t' := t.u
9      t.u := undefined
10     insere t na lista r_{t.o.r+1}
11     t := t'
12   end while
13
14   { executa o maior número possível }
15   { de comparações justas num único passo }
16
17   h := undefined { lista final de raizes }
18   for i = 0,...,⌈log n⌉ do
19     while |r_i| ≥ 2
20       t := link(r_i.head,r_i.head.next)
21       insere t na lista h
22       remove r_i.head,r_i.head.next da lista r_i
23     end if
24     if |r_i| = 1 insere r_i.head na lista h
25   end for
26   return h
```

Observação 1.10
Continuando com comparações justas até sobrar somente uma árvore de cada posto, obteremos um heap binomial. ◊

Lema 1.15
Num heap binomial com varredura única o custo amortizado de "make-heap", "insert", "meld", "getmin" é $O(1)$, o custo amortizado de "deletemin" é $O(\log n)$.

Prova. Usaremos o dobro do número de árvores como potencial. "getmin" não altera o potencial. "make-heap", "insert" e "meld" aumentam o potencial

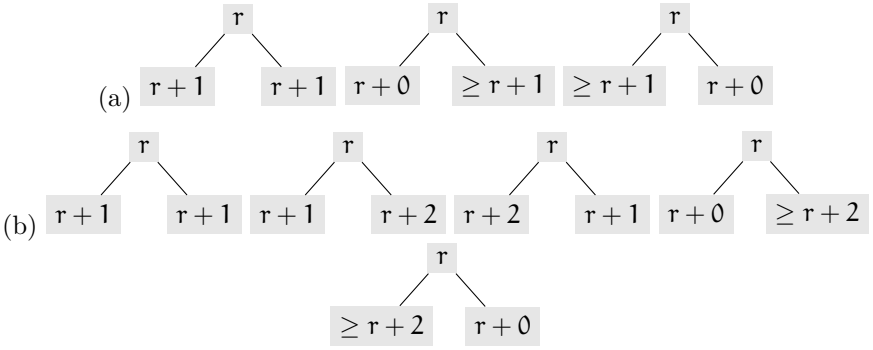


Figura 1.6.: Diferenças no posto de rp-heaps do tipo 1 (a) e tipo 2 (b). {fig:rptyp

por no máximo dois (uma árvore), e portanto possuem custo amortizado $O(1)$. "deletemin" libera no máximo $\log n$ árvores, porque todas comparações foram justas. Com um número total de h árvores, o custo de deletemin é $O(h)$. Sem perda de generalidade vamos supor que o custo é h. A varredura final executa pelo menos $(h - \log n)/2 - 1$ comparações justas, reduzindo o potencial por pelo menos $h - \log n - 2$. Portanto o custo amortizado de "deletemin" é $h - (h - \log n - 2) = \log n + 2 = O(\log n)$. ■

rp-heaps O objetivo do rp-heap é adicionar ao heap binomial de varredura única uma operação "decreasekey" com custo amortizado $O(1)$. A ideía e os problemas são os mesmos do heap Fibonacci: (i) para tornar a operação eficiente, vamos cortar a sub-árvore do elemento cuja chave foi diminuída. (ii) o heap Fibonacci usava cortes em cascata para manter um número suficiente de elementos na árvore; no rp-heap ajustaremos os postos do heap que perde uma sub-árvore. Para poder cortar sub-árvores temos que permitir uma folga nos postos. Num heap binomial a diferença do posto de um elemento com o posto do seu pai (caso existe) sempre é um. Num rp-heap do tipo 1, exigimos somente que os dois filhos de um elemento possuem diferença do posto 1 e 1, ou 0 e ao menos 1. Num rp-heap do tipo 2, exigimos que os dois filhos de um elemento possuem diferença do posto 1 e 1, 1 e 2 ou 0 e pelo menos 2. (Figura 1.6.)
Com isso podemos implementar o "decreasekey" (para rp-heaps do tipo 2) como segue:

```
1  decreasekey(h,e,Δ) :=
2    e.c := e.c − Δ
```

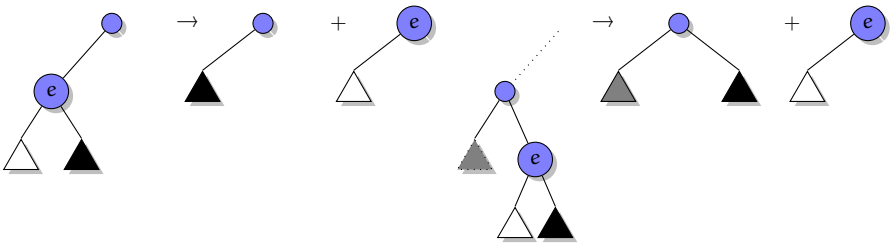


Figura 1.7.: A operação “decreasekey”.

{fig

```
3    if root(e)
4      return
5    if parent(e).o = e then
6      parent(e).o := e.u
7    else
8      parent(e).u := e.u
9    end if
10   parent(e).u := parent(e)
11   e.u := undefined
12   u := parent(e)
13   parent(e) := undefined
14   insere e na lista de raízes de h
15   decreaserank(u)
16
17 rank(e) :=
18   if e is undefined
19     return −1
20   else
21     return e.r
22
23 decreaserank(u) :=
24   if root(u)
25     return
26   if rank(u.o) > rank(u.u)+1 then
27     k := rank(u.o)
28   else if rank(u.u) > rank(u.o)+1 then
29     k := rank(u.u)
30   else
31     k = max(rank(u.o),rank(u.u))+1
32   end if
```

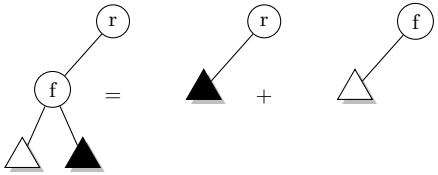


Figura 1.8.: Separar uma semi-árvore de posto k em duas. {fig:split

```
33    if u.r = k then
34      return
35    else
36      u.r := k
37      decreaserank(parent(u))
38
39  delete(h,e) :=
40    decreasekey(h,e,−∞)
41    deletemin(h)
```

{obs:efici

Observação 1.11
Para implementar o rp-heap precisamos além dos ponteiros para o filho ordenado e não-ordenado um ponteiro para o pai do elemento. A (suposta) eficiência do rp-heap vem do fato que o decreasekey altera os postos do heap, e pouco da estrutura dele e do fato que ele usa somente três ponteiros por elemento, e não quatro como o heap Fibonacci. ◊

Lema 1.16
Uma semi-árvore do tipo 2 com posto k contém pelo menos ϕ^k elementos, sendo $\phi = (1+\sqrt{5})/2$ a razão áurea.

Prova. Por indução. Para folhas o lema é válido. Caso a raiz com posto k não é folha podemos obter duas semi-árvores: a primeira é o filho da raiz sem o seu filho não-ordenado, e a segunda é a raiz com o filho não ordenado do seu filho ordenado (ver Fig. 1.8). Pelas regras dos postos de árvores de tipo dois, essas duas árvores possuem postos $k-1$ e $k-1$, ou $k-1$ e $k-2$ ou k e no máximo $k-2$. Portanto, o menor número de elementos n_k contido numa semi-árvore de posto k satisfaz a recorrência

$$n_k = n_{k-1} + n_{k_2}$$

que é a recorrência dos números Fibonacci. ■

Lema 1.17
As operações "decreasekey" e "delete" possuem custo amortizado $O(1)$ e $O(\log n)$

Prova. Ver (Haeupler et al., 2009). ■

1.5.5. Heaps ocos

Introdução

Objetivo: operações com a mesma complexidade amortizada que heaps de Fibonacci. Para um heap h, chave k e elemento e temos as operações:

- make-heap(): $O(1)$
- find-min(h)/getmin(h): $O(1)$
- meld(h_1,h_2): $O(1)$
- insert(e,k,h): $O(1)$
- decrease-key(e,k,h): $O(1)$
- delete(e,h): $O(\log n)$
- delete-min(h): $O(\log n)$

Ideia principal: a operação delete esvazia nós, produzindo nós ocos (ingl. hollow nodes), a operação decrease-key é um delete, seguido por um insert. Teremos duas medidas:

n Número de elementos no heap

N Número de nós no heap = # de elementos + # de nós ocos = # operações insert + # operações decrease-key

Variantes de heaps ocos:

- Heaps ansiosos (ingl. "eager heaps") com múltiplas raízes.
- Heaps ansiosos com uma única raíz.
- Heaps preguiçosos.

```
1 def Node =
2   item // elemento
3   key  // chave
4   fc   // ponteiro para primeiro filho
5   ns   // ponteiro para próximo irmão
6   rank // posto do nó
```

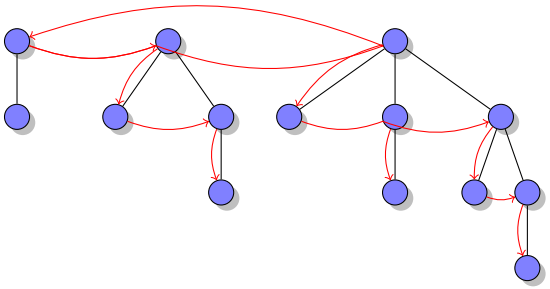
```
7
8  def Item =
9    no   // nó correspondente
10   // mais dados satelites
```

Operação básica: link Um *link* gera um vencedor e um perdedor, que se torna filho do vencedor, e aumenta o posto do vencedor.

```
1    (ranked)link(t1,t2) :=
2      if t1.key ≤ t2.key
3        return makechild(t1,t2)
4      else
5        return makechild(t2,t1)
6
7    makechild(w,l) :=
8      l.ns   := w.fc
9      w.fc   := l
10     w.rank := w.rank+1
11     return w
```

Representação básica

- Lista simples circular de árvores com ordenação do heap, representada por um ponteiro à árvore cuja raíz contém a menor chave (chamada a *raíz mínima*).
- Cada *nó cheia* armazena um item. Podem existir *nós ocos* sem item.
- Nós ocos nunca mais ficam cheias, eles podem somente ser destruídos.
- Filhos ficam armazenados em listas simples, em ordem não-crescente de postos.



```
1   make-heap() := return null
2
3   make-heap(e,k) := return Node(e,k,null,self,0)
4
5   getmin(h) := h
6
7   findmin(h) := return h is not null? h.item : null
8
9   meld(h1,h2) :=
10    if h1 is null return h2
11    if h2 is null return h1
12    swap(h1.ns,h2.ns) // cria uma lista circular simples
13    if h1.key ≤ h2.key return h1 else return h2
14
15  insert(e,k,h) := meld(make-heap(e,k),h)
16
17  decrease-key(e,k,h) :=
18    u = e.node
19    v = make-heap(e,k)
20    v.rank = max{0,u.rank-2}
21    // desloca os filhos de postos 0,...,rank-2 para v
22    if u.rank ≥ 2
23      v.fc := u.fc.ns.ns
24      u.fc.ns.ns := null
25    return meld(v,h)
26
27  delete(e,h) :=
28    e.node.item := null
29    if e.node = h
30      delete-min(h)
31
32  delete-min(h) :=
33    if h is null: return
34    h.node.item := null
35
36    aloca um array R0,R1,...,RM
37    // repetidamente remove raízes ocos e une os heaps
38    r:=h
39    repeat
40      rn := r.ns
41      link-heap(r,R)
```

```
42        r:=rn
43      until r==h
44
45      // reconstrói o heap
46      h:=null
47      for i=0,...,M
48        if R_i is not null
49          R_i.ns := R_i
50          h := meld(h,R_i)
51      return h
52
53    link-heap(h,R) :=
54      if h is hollow
55        r:=h.fc
56        while r is not null
57          rn := r.ns
58          link-heap(r,R)
59          r := rn
60        destroy node h
61      else
62        i := h.rank
63        while R_i is not null
64          h := link(h,R_i)
65          R_i := null
66          i := i + 1
67        end
68        R_i := h
```

Invariantes

1. Ordenação do heap.

2. Invariante do posto: cada nó de posto r possui r filhos com postos $0, \ldots, r-1$, exceto no caso $r \geq 2$ e o nó foi esvaziada por uma operação decrease-key. Neste caso o nó possui dois filhos de postos $r-1$ e $r-2$.

Corretude

Teorema 1.11
Heaps com nós ocos implementam corretamente todas operação e mantém as invariantes.

Prova. Por indução sobre o número de operações. ■
Lembrança: os números de Fibonacci são definidos por $F_0 = 0, F_1 = 1, F_{i+2} = F_i + F_{i+1}$, para $i \geq 0$ e temos $F_{i+2} \geq \Phi^i$, com a razão áurea $\Phi = (1 + \sqrt{5})/2$.

Teorema 1.12
Um nó de posto r possui pelo menos $F_{r+3} - 1$ descendentes (cheios ou ocos), incluindo o próprio nó, na árvore.

Prova. Por indução sobre r. Para $r = 0$, temos $F_3 - 1 = 1$, e para $r = 1$ temos $F_4 - 1 = 2$ e a afirmação está correta, porque para $r < 2$ um nó não perde filhos caso for esvaziado. Para $r \geq 2$ pela invariante do posto temos pelo menos dois filhos com postos $r - 1$ e r_2. Pela hipótese da indução eles tem pelo menos $F_{r+1} - 1$ e $F_{r+2} - 1$ descendentes e logo r possui pelo menos $F_{r+1} - 1 + F_{r+2} - 1 + 1 = F_{r+3} - 1$ descendentes. ■

Corolário 1.2
Depois uma operação delete-min o número de árvores é no máximo $\lceil \log_\Phi N \rceil = O(\log N)$ porque temos no máximo uma árvore por posto. Logo podemos escolher $M = \lceil \log_\Phi N \rceil$ na operação delete-min.

Teorema 1.13
O tempo amortizado por operação num heap oco é $O(1)$, exceto para as operações delete e delete-min, que tem complexidade $O(\log N)$ para um heap com N nós.

Prova. Todas operações exceto a deleção do elemento mínimo possuem tempo $O(1)$ no caso pessimista. O custo de uma deleção é $O(H+T)$ com H o número de nós ocos destruídos, e T o número de árvores antes das operações link. Depois das operações link temos no máximo $\log_\Phi N$ árvores, logo faremos pelo menos $T - \log_\Phi N$ operações link e no máximo $\log_\Phi N$ operações meld. Logo o custo total é $O(1)$ por destruição de um nó oco, e por link, mas $O(\log N)$.
Para contabilizar a destruição do um nó, aumentamos o custo de cada criação (insert, decrease-key) por 1.
Para contabilizar as operações link: define um potencial igual ao número de nós cheias, que não são filho de outro nó cheia (i.e. raízes e filhos de nós ocos). Para todas operações diferente de delete-min e delete, o aumento do potencial é constante (no máximo 1 para insert, 3 para decrease-key, 0 para as demais). Para o delete que remove o elemento mínimo e delete-min, o custo amortizado de cada link é 0, porque um link combina duas raízes cheias, reduzindo o

Tabela 1.2.: Complexidade das operações de uma fila de prioridade. Complexidades em negrito são amortizados. (1): meld preguiçoso.

	insert	getmin	deletemin	update	decreasekey	delete
Vetor	$O(1)$	$O(1)$	$O(n)$	$O(1)$	(update)	$O(1)$
Lista ordenada	$O(n)$	$O(1)$	$O(1)$	$O(n)$	(update)	$O(1)$
Heap binário	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial	$\mathbf{O(1)}$	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial(1)	$O(1)$	$O(1)$	$\mathbf{O(\log n)}$	$O(\log n)$	(update)	$O(\log n)$
Heap Fibonacci	$O(1)$	$O(1)$	$O(\log n)$	-	$\mathbf{O(1)}$	$O(\log n)$
rp-heap	$O(1)$	$O(1)$	$\mathbf{O(\log n)}$	-	$\mathbf{O(1)}$	$\mathbf{O(\log n)}$

{tab:pq}

potencial por 1. Além disso, ao remover um elemento, o potencial aumenta por no máximo $\log_\Phi N$, um por cada filho do novo nó oco. Logo o custo amortizado de delete e delete-min é $O(\log N)$.

■

Re-otimizando o heap A análise acima é em função de N. Caso $\log N = O(\log n)$ temos um heap assintoticamente ótimo. Caso executamos muitas operações decrease-key, temos que reconstruir o heap periodicamente, para garantir $N = O(n)$. O método mais simples é: escolhe uma constante $c > 1$ e para $N > cn$ reconstrói o heap completamente, destruindo os nós ocos, criando heaps de um único nó de todos nós cheios, e aplicando operações meld para unir todos heaps. O custo é $O(N)$ para percorrer todo nó uma vez e pode ser atribuído na análise amortizada para as operações insert e delete-min.

Resumo: Filas de prioridade A tabela 1.2 resume a complexidade das operações para diferentes implementações de uma fila de prioridade.

1.5.6. Árvores de van Emde Boas

Pela observação 1.5 é impossível implementar uma fila de prioridade baseado em comparação de chaves com todas operações em $o(\log n)$. Porém existem algoritmos que ordenam n números em $o(n \log n)$, aproveitando o fato que as chaves são números com k bits, como por exemplo o radix sort que ordena em tempo $O(kn)$, ou aproveitando que as chaves possuem um domínio limitado, como por exemplo o counting sort que ordena n números em $[k]$ em tempo $O(n + k)$.
Uma *árvore de van Emde Boas* (árvore vEB) T realiza as operações

- member(T, e): elemento e pertence a T?
- insert(T, e): insere e em T

- delete(T, e): remove e de T
- min(T) e max(T): elemento mínimo e máximo de T, ou "undefined" caso não existe
- succ(T, e) e pred(T, e): successor e predecessor de e em T; e não precisa pertencer a T

no universo de chaves $[0, u-1]$ em tempo $O(\log\log u)$ e espaço $O(u)$.
Outras operações compostas podem ser implementados, por exemplo

```
1  deletemin(T) :=
2      e := min(T); delete(e); return e
3  deletemax(T) :=
4      e := max(T); delete(e); return e
```

Árvores binárias em ordem vEB Na discussão da implementação de árvores binárias na página 41 discutimos uma representação em ordem da busca por profundidade (BFS order). A ideia da ordem vEB é "cortar" a altura (número de níveis) h de uma árvore binária (que possui $n = 2^h - 1$ nodos e 2^{h-1} folhas) pela metade. Com isso obtemos

- uma árvore superior T_0 de altura $\lfloor h/2 \rfloor$
- e $b = 2^{\lfloor h/2 \rfloor} = \Theta(2^{h/2}) = \Theta(\sqrt{n})$ árvores inferiores $T_1, \ldots, T_b$ de altura $\lceil h/2 \rceil$ e com $2^{\lceil h/2 \rceil} - 1 = \Theta(\sqrt{n})$ nodos.

Os nodos dessa árvore são armazenados em ordem $T_0, T_1, \ldots, T_b$ e toda árvore T_i é ordenado recursivamente da mesma maneira, até chegar numa árvore de altura $h = 1$, como a Figura 1.9 mostra.
Armazenar uma árvore binária em ordem de vEB não altera a complexidade das operações. Uma busca, por exemplo, continua com complexidade $O(h)$. Porém, armazenado em ordem da busca por profundidade, uma busca pode gerar $\Theta(h)$ falhas no *cache*, no pior caso. Na ordem de vEB, a busca sempre atravessa $\Omega(\log_2 B)$ níveis, com B o tamanho de uma linha de cache, antes de gerar uma nova falha no *cache*. Logo uma busca gera somente $O(\log_2 n / \log_2 B) = O(\log_B n)$ falhas no *cache*. O layout se chama *cache oblivious* porque funciona sem conhecer o tamanho de uma linha de cache B.

Árvores vEB A estrutura básica de uma árvore de vEB é

1. Usar uma árvore binária de altura h representar 2^{h-1} elementos nas folhas.

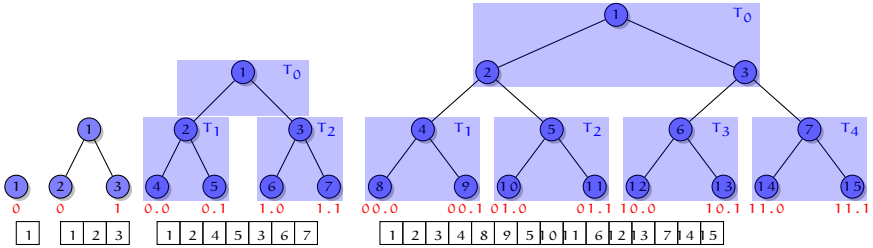


Figura 1.9.: Organização de árvores binárias em ordem de van Emde Boas para $h \in [4]$. As folhas são rotuladas por "cluster.subíndice". Abaixo da árvore a ordem do armazenamento do vértices é dado. Os T_i correspondem com as subárvores do primeiro nível de recursão.

oas}

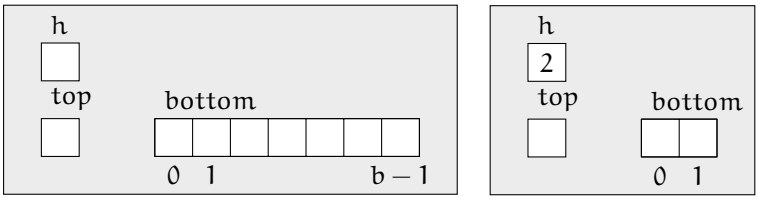


Figura 1.10.: Representação da primeira versão de uma árvore vEB. (a) Forma geral. (b) Caso base.

{fig:vEBr1

2. Cada folha armazena um bit, que é 1 caso o elemento correspondente pertence ao conjunto representado.

3. Os bits internos servem como *resumo* da sub-árvore: eles representam a conjunção dos bits dos filhos, i.e. um bit interno é um, caso na sua sub-árvore existe pelo menos uma folha que pertence ao conjunto representado.

Todas as operações da estrutura acima podem ser implementadas em tempo $O(h) = O(\log u)$. Para melhorar isso, vamos aplicar a mesma ideia da ordem de van Emde Boas: a árvore é separada em uma árvore superior, e uma série de árvores inferiores, cada uma com altura $\approx h/2$. As folhas da árvore superior contém o resumo das raízes das árvores inferiores: por isso a árvore superior possui altura $\lfloor h/2 \rfloor + 1$, uma a mais comparado com a ordem de vEB.
Fig. 1.10 mostra essa representação. A altura da árvore está armazenada no campo h. Além disso temos um ponteiro "top" para a árvore superior, e

um vetor de ponteiros "bottom" de tamanho $b = 2^{\lfloor h/2 \rfloor}$ para as raízes das árvores inferiores. No caso base com $h = 2$, abusaremos os campos "top" e "bottom" para armazenar os bits da raiz e dos dois filhos: um ponteiro arbitrário diferente de undefined representa um bit 1, o ponteiro undefined o bit 0. Para isso servem as funções auxiliares

```
1   set(p) := p := 1
2   clear(p) := p := undefined
3   bit(p)  := return p ≠ undefined
```

Observe que as folhas $0, 1, \ldots, 2^{h-1}-1$ podem ser representadas com $h-1$ bits. Os primeiros $\lfloor h/2 \rfloor$ bits representam o número da sub-árvore que contém a folha, e os últimos $\lceil h/2 \rceil - 1$ bits o índice (relativo) da folha na sua sub-árvore. Isso explica a definição das funções auxiliares

```
1   subtree(e) := e ≫ ⌈h/2⌉ − 1
2   subindex(e) := e&(1 ≪ ⌈h/2⌉ − 1) − 1
3   element(s,i) := (s ≪ ⌈h/2⌉ − 1) | i
```

para extrair de um elemento o número da sub-árvore correspondente, ou o seu índice nesta sub-árvore, e para determinar o índice na árvore atual do i-ésimo elemento da sub-árvore s.

Com isso podemos implementar as operações como segue.

```
1  member(T,e) :=
2    if T.h = 2
3      return bit(T.bottom[e])
4    return member(T.bottom[subtree(e)],subindex(e))
5
6  min(T,e) :=
7    if T.h = 2
8      if bit(T.bottom[0])
9        return 0
10     if bit(T.bottom[1])
11       return 1
12     return undefined
13
14   c := min(T.top)
15   if c = undefined
16     return c
17   return element(c,min(T.bottom[c]))
18
19 succ(T,e) :=
20   if T.h = 2
```

```
21      if e = 0 and bit(T.bottom[1])=1
22        return 1
23      return 0
24
25    s := succ(T.bottom[subtree(e)],subindex(e))
26    if s ≠ undefined
27      return element(subtree(e),s)
28
29    c := succ(T.top,subtree(e))
30    if c = undefined
31      return c
32    return element(c,min(T.bottom[c]))
33
34  insert(T,e) :=
35    if T.h = 2
36      set(T.bottom[e])
37      set(T.top)
38    else
39      insert(T.bottom[subtree(e)],subindex(e))
40      insert(T.top,subtree(e))
41
42  delete(T,e) :=
43    if T.h = 2
44      clear(T.bottom[e])
45      if (bit(T.bottom[1 − e])=0
46        clear(T.top)
47    else
48      delete(T.bottom[subtree(e)],subindex(e))
49      s :=min(T.bottom[subtree(e)])
50      if s = undefined
51        delete(T.top,subtree(e))
```

As complexidades das operações implementadas no caso pessimista são (ver as chamadas recursivas acima em vermelho):

member $T(h) = T(\lceil h/2 \rceil) + O(1) = \Theta(\log h) = \Theta(\log \log u)$.

min $T(h) = T(\lfloor h/2 \rfloor + 1) + T(\lceil h/2 \rceil) + O(1) = 2T(h/2) + O(1) = \Theta(h) = \Theta(\log u)$.

insert $T(h) = T(\lceil h/2 \rceil + T(\lfloor h/2 \rfloor + 1) + O(1) = \Theta(h) = \Theta(\log u)$.

succ/delete $T(h) = T(\lceil h/2 \rceil) + T(\lfloor h/2 \rfloor + 1) + O(h) = 2T(h/2) + O(h) =$

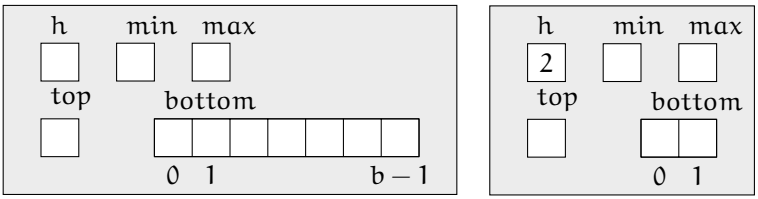


fig:vEBr2} Figura 1.11.: Representação uma árvore vEB. (a) Forma geral. (b) Caso base.

$\Theta(h \log h) = \Theta(\log u \log\log u)$ (com um trabalho extra de $O(h)$ para chamar "min").

Logo todas operações com mais que uma chamada recursiva não possuem a complexidade desejada $O(\log\log u)$. A introdução de dois campos "min" e "max" que armazenam o elemento mínimo e máximo, junto com algumas modificações resolvem este problema.

1. Armazenar somente o mínimo, a operação "min" custa somente $O(1)$ é "insert", "succ" e "delete" consequentemente somente $O(h)$.

2. Armazenado também o máximo, sabemos na operação "succ" se o sucessor está na árvore atual sem buscar, logo a operação "succ" pode ser implementada em $O(\log\log u)$.

3. A última modificação é *não armazenar* o elemento mínimo na sub-árvore correspondente. Com isso a primeira inserção somente modifica a árvore de resumo (top) e a segunda e as demais operações modificam somente a sub-árvore correspondente. A deleção funciona similarmente: ela remove ou um elemento na sub-árvore, ou o último elemento, modificando somente a árvore de resumo (top). Com isso todas operações podem ser implementadas em $O(\log\log u)$.

Na base armazenaremos os elementos somente nos campos "min" e "max". Por convenção setamos "min" maior que "max" numa árvore vazia. As seguintes funções auxiliares permitem remover os elementos de uma árvore base e determinar se uma árvore possui nenhum, um ou mais elementos.

```
1  clear(T) :=
2    T.min:=1; T.max:=0; // convenção
3
4  empty(T) :=
```

```
5    return T.min>T.max
6
7  singleton(T) :=
8    return T.min=T.max
9
10 full(T) :=
11   return T.min<T.max
```

```
1  member(T,e) :=
2    if empty(T)
3      return false
4    if T.min = e or T.max = e
5      return true
6
7    { não é ``min'' nem ``max''? a base não contém o elemento }
8    if T.h = 2
9      return false
10
11   return member(T.bottom[subtree(e)],subindex(e))
12
13 min(T) :=
14   if empty(T)
15     return undefined
16   return T.min
17
18 max(T) :=
19   if empty(T)
20     return undefined
21   return T.max
22
23 succ(T,e) :=
24   if T.h=2
25     if e = 0 and T.max = 1
26       return 1
27     return undefined
28
29   if not empty(T) and e < T.min
30     return T.min
31
32   { sucessor na árvore atual }
33   m:=max(T.bottom[subtree(e)])
```

```
34    if m ≠ undefined and subindex(e)<m
35      return element(subtree(e),
36                     succ(T.bottom[subtree(e)],subindex(e)))
37
38    { mínimo na árvore sucessora }
39    c := succ(T.top,subtree(e))
40    if c = undefined
41      return c
42    return element(c,min(T.bottom[c]))
43
44  pred(T,e) :=
45    if T.h=2
46      if e = 1 and T.min=0
47        return 0
48      return undefined
49
50    if not empty(T) and T.max < e
51      return T.max
52
53    { predecessor na árvore atual }
54    m:=min(T.bottom[subtree(e)])
55    if m ≠ undefined and m <subindex(e)
56      return element(subtree(e),
57                     pred(T.bottom[subtree(e)],subindex(e)))
58
59    { máximo na árvore predecessora }
60    c:= pred(T.top,subtree(e))
61    if c = undefined
62      if not empty(T) and T.min<e
63        return T.min
64      else
65        return undefined
66
67    return element(c,max(T.bottom[c]))
68
69  insert(T,e) :=
70    if empty(T)
71      T.min := T.max := e
72      return
73
```

```
74    { novo mínimo: setar min, insere min anterior }
75    if e < T.min
76      swap(T.min,e)
77
78    { insere recursivamente }
79    if T.h > 2
80      if empty(T.bottom[subtree(e)])                          {line:empt
81        insert (T.top,subtree(e))
82      insert (T.bottom[subtree(e)],subindex(e))               {line:seco
83
84    { novo máximo: atualiza }
85    if T.max < e
86      T.max := e
87
88  delete(T,e) :=
89    if empty(T)
90      return
91
92    if singleton(T)
93      if T.min = e
94        clear(T)
95      return
96
97    { novo mínimo? }
98    if e = T.min
99      T.min := element(min(T.top),min(T.bottom[min(T.top)]))
100     e := T.min
101
102   { remove e da árvore }
103   delete (T.bottom[subtree(e)],subindex(e))                 {line:recu
104
105   if empty(T.bottom[subtree(e)])
106     delete (T.top,subtree(e))                               {line:seco
107     if e = T.max
108       c:=max(T.top)
109       if c = undefined
110         T.max := T.min
111       else
112         T.max := element(c,max(T.bottom[c]))
113   else
```

```
114         T.max := element(subtree(e),max(T.bottom[subtree(e)]))
```

Com essas implementações cada função executa uma chamada recursiva e um trabalho constante a mais e logo precisa tempo $O(\log h)$. Em particular, na função "insert" caso a sub-árvore do elemento é vazia na linha 80 a segunda chamada "insert" na linha 82 precisa tempo constante. Similarmente, ou a deleção recursiva na linha 103 não remove o último elemento, e talvez custa $O(\log h)$, e logo a deleção da linha 106 não é executada, ou ela remove o último elemento e custo somente $O(1)$.

1.5.7. Exercícios

Exercício 1.2
Prove lema 1.10. Dica: Use indução sobre n.

Exercício 1.3
Prove que um heap binomial com n vértices possui $O(\log n)$ árvores. Dica: Por contradição.

Exercício 1.4 (Laboratório 1)

1. Implementa um heap binário. Escolhe casos de teste adequados e verifica o desempenho experimentalmente.

2. Implementa o algoritmo de Prim usando o heap binário. Novamente verifica o desempenho experimentalmente.

Exercício 1.5 (Laboratório 2)

1. Implementa um heap binomial.

2. Verifica o desempenho dele experimentalmente.

3. Verifica o desempenho do algoritmo de Prim com um heap Fibonacci experimentalmente.

Exercício 1.6
A proposição 1.3 continua ser correto para grafos com pesos negativos? Justifique.

1.6. Fluxos em redes

Seja $G = (V, A, c)$ um grafo direcionado e capacitado com capacidades $c : A \to \mathbb{R}$ nos arcos. Uma atribuição de fluxos aos arcos $f : A \to \mathbb{R}$ em G se chama *circulação*, se os fluxos respeitam os limites da capacidade ($f_a \le c_a$) e satisfazem a conservação de fluxo $f(v) = 0$ com

$$f(v) := \sum_{a \in N^+(v)} f_a - \sum_{a \in N^-(v)} f_a \tag{1.7}$$ {eq:totalv}

(ver Fig. 1.12).

Definição 1.4

Para $X, Y \subseteq V$ sejam $A(X,Y) := (X \times Y) \cap A$ os arcos passando de X para Y. O fluxo de X para Y é $f(X,Y) := \sum_{a \in A(X,Y)} f_a$. Ainda estendemos a notação do fluxo total de um vértice (1.7) para conjuntos: $f(X) := f(X, \bar{X}) - f(\bar{X}, X)$ é o fluxo neto do saindo do conjunto X, onde $\bar{X} := V \setminus X$. Analogamente, escrevemos para as capacidades $c(X,Y) := \sum_{a \in A(X,Y)} c_a$.

{lem:xflow}

Lema 1.18

Para qualquer conjunto de vértices $X \subseteq V$ temos $\sum_{v \in X} f(v) = f(X)$.

Prova.

$$\begin{aligned}
\sum_{v \in X} f(v) &= \sum_{v \in X} \left(\sum_{a \in N^+(v)} f_a - \sum_{a \in N^-(v)} f_a \right) \\
&= \left(\sum_{a \in A(X,\bar{X})} f_a + \sum_{a \in A(X,X)} f_a \right) - \left(\sum_{a \in A(\bar{X},X)} f_a + \sum_{a \in A(X,X)} f_a \right) \\
&= \sum_{a \in A(X,\bar{X})} f_a - \sum_{a \in A(\bar{X},X)} f_a = f(X,\bar{X}) - f(\bar{X},X) = f(X).
\end{aligned}$$

■

{lem:flows}

Corolário 1.3

Qualquer atribuição de fluxos f satisfaz $\sum_{v \in V} f(v) = 0$.

Prova.

$$\sum_{v \in V} f(v) = f(V) = f(V, \bar{V}) - f(\bar{V}, V) = 0 - 0 = 0.$$

■

Uma circulação vira um *fluxo*, se o grafo possui alguns vértices que são fontes ou destinos ("sorvedouros") de fluxo, e portanto não satisfazem a conservação de fluxo. Um fluxo s–t possui uma única fonte s e um único destino t. Um objetivo comum (transporte, etc.) é encontrar um fluxo s–t máximo.

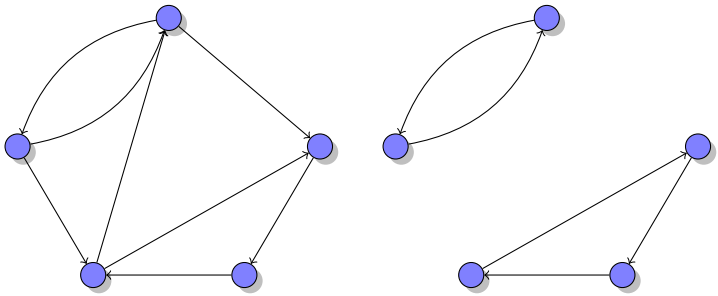


Figura 1.12.: Grafo (esquerda) com circulação (direita) {fig

FLUXO s–t MÁXIMO

Instância Grafo direcionado $G = (V, A, c)$ com capacidades c nos arcos, um vértice origem $s \in V$ e um vértice destino $t \in V$.

Solução Um fluxo f, com $f(v) = 0$, $\forall v \in V \setminus \{s, t\}$.

Objetivo Maximizar o fluxo $f(s)$.

Lema 1.19
Um fluxo s–t satisfaz $f(s) + f(t) = 0$.

Prova. Temos

$$f(s) + f(t) = \sum_{v \in V} f(v) \overset{(1.3)}{=} 0,$$

onde a primeira igualdade vale pela conservação de fluxo nos vértices em $V \setminus \{s, t\}$. ■

Uma formulação como programa linear é

p:maxflow}

$$\begin{aligned} \textbf{maximiza} \quad & f(s) && && (1.8)\\ \textbf{sujeito a} \quad & f(v) = 0, && \forall v \in V \setminus \{s, t\},\\ & 0 \le f_a \le c_a, && \forall a \in A. \end{aligned}$$

Observação 1.12
O programa (1.8) possui uma solução, porque $f_a = 0$ é uma solução viável. O sistema não é ilimitado, porque todas variáveis são limitadas, e por isso possui

uma solução ótima. O problema de encontrar um fluxo s–t máximo pode ser resolvido em tempo polinomial via programação linear. ◊

O problema dual é

$$\begin{aligned}
\textbf{minimiza} \quad & \sum_{a \in A} c_a q_a \\
\textbf{sujeito a} \quad & q_a - p_v \geq 1, && \forall a = (s,v) \in A, v \neq t \\
& q_a + p_u \geq -1, && \forall a = (u,s) \in A, u \neq t \\
& q_a + p_u \geq 0 && \forall a = (u,t) \in A, u \neq s \\
& q_a - p_v \geq 0 && \forall a = (t,v) \in A, v \neq s \\
& q_a \geq 1 && \text{if } (s,t) \in A \\
& q_a \geq -1 && \text{if } (t,s) \in A \\
& q_a + p_u - p_v \geq 0, && \forall a = (u,v) \in A, \\
& p_v \lesseqgtr 0, && \forall v \in V \setminus \{s,t\}, \\
& q_a \geq 0, && \forall a \in A.
\end{aligned}$$

Ou equivalente

$$\begin{aligned}
\text{(MC)} \ \textbf{minimiza} \quad & \sum_{a \in A} c_a q_a && && (1.9) \\
\textbf{sujeito a} \quad & q_a + p_u - p_v \geq 0 && \forall a = (u,v) \in A && (1.10) \\
& p_s = -1, && && (1.11) \\
& p_t = 0, && && (1.12) \\
& p_v \lesseqgtr 0, && \forall v \in V && (1.13) \\
& q_a \geq 0, && \forall a \in A. && (1.14)
\end{aligned}$$

Here the idea is roughly the following. We want to set all $q_a = 0$. But that's not possible, since that implies $p_u \geq p_v$ for all arcs $uv \in A$, so the potential goes only down, but at some point we have to "climb" the hill from $p_s = -1$ and $p_t = 0$. This can be done by arcs of value $q_a = 1$. If we do this minimally, we have to intercept every st-path. So we have a cut. By this reasoning we can see, that every cut can be made a solution of MC, so its value is at most the value of a minimum cut. (This is almost exactly Papadimitriou e Steiglitz (1982, Th. 6.1). Note that we

have fixed p_s and p_t at particular values, but any solution $p + c$ for a constant vector c would also do, when fixing p_s and p_t accordingly.)
It remains to show that MC's value cannot be lower. Papadimitriou e Steiglitz (1982) defer this to the Ford-Fulkerson algorithm, and just remark that complementary slackness implies the condition that for cut $(X, \bar{X})$ arcs in $A(X, \bar{X})$ are saturated, and those in $A(\bar{X}, X)$ are 0.
One could, however, work out that the minimum is obtained by a cut. First look at any given values p. Since $q_a \geq p_v - p_u$, the optimal value is to set $q_a = p_v - p_u$. Now look at any solution (p, q). I claim: we can replace p by p' where $p'_a = c(p_a; [-1, 0])$ with "clamping" function $c(x; [a, b]) = \max\{\min\{x, b\}, a\}$, and then set q as above, to obtain another solution of no larger value. (Note that if we have integer data in the primal, by total unimodularity, the optimal solution is integer, and the same holds for the dual. In this case the "clamping" above means $p \in \{-1, 0\}^n$, and q are exactly the arcs that "climb" from -1 to 0, so for every solution, there exists a cut of the same or better value. But we don't want to invoke total unimodularity here.)
To see the claim, note first that the clamping function c is monotone and 1-Lipschitz, i.e. for x, y, $x \leq y \to c(x; I) \leq c(y; I)$ and $|x - y| \geq |c(x; I) - c(y; I)|$. So, if for any arc uv we have $p_v \leq p_u$ by monotonicity $p'_v \leq p'_u$ so we can choose $q'_a = 0 \leq q_a$. If $p_v - p_u \geq 0$ then $p'_v - p'_u \leq p_v - p_u$, by the Lipschitz condition. From this follows that we can choose $q'_a = p'_v - p'_v \leq p_v - p_u \leq q_a$. Overall, we obtain $c_a q_a \geq c_a q'_a$, since $c_a \geq 0$ for all $a \in A$, and thus $c^t q \geq c^t q'$.
With that in place we would still have to show that the solution can be made integral, without getting worse. This can probably done by induction over the number of fractional arcs (i.e. with non-unit potential difference). We pick one (maybe some minimality here), and show that we can integralize it. (Does that hold? It would be a funny result, since the dual is always integral, while the primal clearly has not to be, when capacities are not integer.)

1.6.1. O algoritmo de Ford-Fulkerson

Nosso objetivo: Achar um algoritmo *combinatorial* mais eficiente. Idéia básica: Começar com um fluxo viável $f_a = 0$ e aumentar ele gradualmente. Observação: caso temos um s–t-caminho $P = (v_0 = s, v_1, \ldots, v_{n-1}, v_n = t)$, podemos aumentar o fluxo atual f um valor que corresponde ao "gargalo"

$$g(f, P) := \min_{\substack{a = (v_{i-1}, v_i) \\ i \in [n]}} c_a - f_a.$$

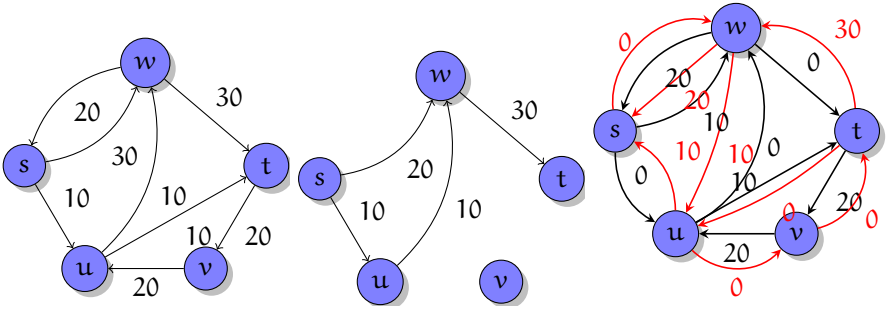


Figura 1.13.: Esquerda: Grafo com capacidades. Centro: Fluxo com valor 30. Direita: O grafo residual correspondente. {fig:simpl

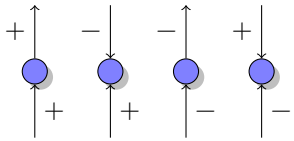


Figura 1.14.: Manter a conservação de fluxo. {fig:mante

Observação 1.13
Repetidamente procurar um caminho de gargalo positivo e aumentar nem sempre produz um fluxo máximo. Na Fig. 1.13 o fluxo máximo possível é 40, obtido pelo aumentos de 10 no caminho $P_1 = (s, u, t)$ e 30 no caminho $P_2 = (s, w, t)$. Mas, se aumentamos 10 no caminho $P_1 = (s, u, w, t)$ e depois 20 no caminho $P_2 = (s, w, t)$ obtemos um fluxo de 30 e o grafo não possui mais caminho que aumenta o fluxo. ◊

Problema no caso acima: para aumentar o fluxo e manter a conservação de fluxo num vértice interno v temos quatro possibilidades: (i) aumentar o fluxo num arco entrante e sainte, (ii) aumentar o fluxo num arco entrante, e diminuir num outro arco entrante, (iii) diminuir o fluxo num arco entrante e diminuir num arco sainte e (iv) diminuir o fluxo num arco entrante e aumentar num arco entrante (ver Fig. 1.14).
Isso é o motivo para definir para um dado fluxo f o *grafo residual* G_f com

- Vértices V

- Arcos para frente ("forward") A com capacidade $c_a - f_a$, caso $f_a < c_a$.
- Arcos para atras ("backward") $A' = \{(v,u) \mid (u,v) \in A\}$ com capacidade $c_{(v,u)} = f_{(u,v)}$, caso $f_{(u,v)} > 0$.

Observe que na Fig. 1.13 o grafo residual possui um caminho $P = (s, w, u, t)$ que aumenta o fluxo por 10. O algoritmo de Ford-Fulkerson (Ford e Fulkerson, 1956) consiste em, repetidamente, aumentar o fluxo num caminho s–t no grafo residual.

Algoritmo 1.5 (Ford-Fulkerson)
Entrada Grafo $G = (V, A, c)$ com capacidades c_a nos arcos.

Saída Um fluxo f.

```
1 for all a ∈ A: f_a := 0
2 while existe um caminho s-t em G_f do
3   Seja P um caminho s-t simples
4   Aumenta o fluxo f um valor g(f,P)
5 end while
6 return f
```

Análise de complexidade Na análise da complexidade, consideraremos somente capacidades em $\mathbb{N}$ (ou equivalente em $\mathbb{Q}$: todas capacidades podem ser multiplicadas pelo menor múltiplo em comum dos denominadores das capacidades.)

egralflow}

Lema 1.20
Para capacidades inteiras, todo fluxo intermediário e as capacidades residuais são inteiros.

Prova. Por indução sobre o número de iterações. Inicialmente $f_a = 0$. Em cada iteração, o "gargalo" $g(f,P)$ é inteiro, porque as capacidades e fluxos são inteiros. Portanto, o fluxo e as capacidades residuais após do aumento são novamente inteiros. ■

Lema 1.21
Em cada iteração, o fluxo aumenta por pelo menos 1.

Prova. O caminho s–t possui por definição do grafo residual uma capacidade "gargalo" $g(f,P) > 0$. O fluxo $f(s)$ aumenta exatamente $g(f,P)$. ■

Lema 1.22
O algoritmo Ford-Fulkerson precisa no máximo $C = \sum_{a \in N^+(s)} c_a$ iterações. Portanto ele tem complexidade $O((n+m)C)$.

Prova. C é um limite superior do fluxo máximo. Como o fluxo inicialmente possui valor 0 e aumenta ao menos 1 por iteração, o algoritmo de Ford-Fulkerson termina em no máximo C iterações. Em cada iteração temos que achar um caminho s–t em G_f. Representando G por listas de adjacência, isso é possível em tempo $O(n+m)$ usando uma busca por profundidade. O aumento do fluxo precisa tempo $O(n)$ e a atualização do grafo residual é possível em $O(m)$, visitando todos arcos. ■

Corretude do algoritmo de Ford-Fulkerson

Definição 1.5
Uma partição $(X, \bar{X})$ de V é um *corte* s–t, se $s \in X$ e $t \in \bar{X}$. Um arco a é *saturado* para um fluxo f, caso $f_a = c_a$.

{lem:outfl

Lema 1.23
Para qualquer corte $(X, \bar{X})$ temos $f(X) = f(s)$.

Prova.

$$f(X) \overset{(1.18)}{=} f(s) + \sum_{v \in X \setminus \{s\}} f(v) = f(s).$$

(O último passo é correto, porque para todo $v \in X, v \neq s$, temos $f(v) = 0$ pela conservação de fluxo.) ■

{lem:flowl

Lema 1.24
O valor $c(X, \bar{X})$ de um corte s–t é um limite superior para um fluxo s–t.

Prova. Seja f um fluxo s–t. Temos

$$f(s) \overset{(1.23)}{=} f(X) = f(X, \bar{X}) - f(\bar{X}, X) \leq f(X, \bar{X}) \leq c(X, \bar{X}).$$

■

Consequência: O fluxo máximo é menor ou igual a o corte mínimo. De fato, a relação entre o fluxo máximo e o corte mínimo é mais forte:

{th:maxflo

Teorema 1.14 (Fluxo máximo – corte mínimo)
O valor do fluxo máximo entre dois vértices s e t é igual ao valor do corte mínimo.

{augment-c

Lema 1.25
Quando o algoritmo de Ford-Fulkerson termina, o valor do fluxo é máximo.

Prova. O algoritmo termina se não existe um caminho entre s e t em G_f. Podemos definir um corte $(X, \bar{X})$, tal que X é o conjunto de vértices alcançáveis

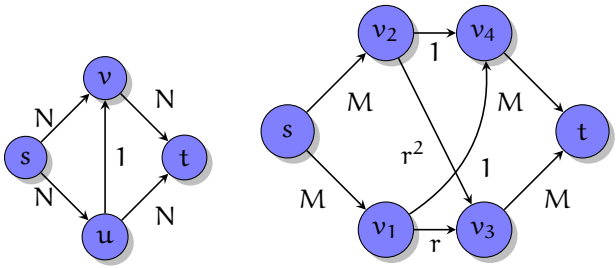


Figura 1.15.: Esquerda: Pior caso para o algoritmo de Ford-Fulkerson com pesos inteiros aumentando o fluxo por 2N vezes por 1 nos caminhos (s,u,v,t) e (s,v,u,t). Direita: Menor grafo com pesos irracionais em que o algoritmo de Ford-Fulkerson falha (Zwick, 1995). $M \geq 3$, e $r = (1+\sqrt{1-4\lambda})/2 \approx 0.682$ com $\lambda \approx 0.217$ a única raiz real de $1 - 5x + 2x^2 - x^3$. Aumentar (s,v_1,v_4,t) e depois repetidamente (s,v_2,v_4,v_1,v_3,t), (s,v_2,v_3,v_1,v_4,t), (s,v_1,v_3,v_2,v_4,t), e (s,v_1,v_4,v_2,v_3,t) converge para o fluxo máximo $2+r+r^2$ sem terminar.

fulkerson}

em G_f a partir de s. Agora considere os arcos entre X e $\bar{X}$. Para um arco $a \in A(X,\bar{X})$ temos $f_a = c_a$, senão G_f terá um arco "forward" a, uma contradição. Para um arco $a = (u,v) \in A(\bar{X},X)$ temos $f_a = 0$, senão G_f terá um arco "backward" $a' = (v,u)$, uma contradição. Logo

$$f(s) = f(X) = f(X,\bar{X}) - f(\bar{X},X) = f(X,\bar{X}) = c(X,\bar{X}).$$

Pelo lema 1.24, o valor de um fluxo arbitrário é menor ou igual que $c(X,\bar{X})$, portanto f é um fluxo máximo. ■

Prova. (Do teorema 1.14) Pela análise do algoritmo de Ford-Fulkerson. ■

Desvantagens do algoritmo de Ford-Fulkerson O algoritmo de Ford-Fulkerson tem duas desvantagens:

(1) O número de iterações C pode ser alto, e existem grafos em que C iterações são necessárias (veja Fig. 1.15). Além disso, o algoritmo com complexidade $O((n+m)C)$ é somente pseudo-polinomial.

(2) É possível que o algoritmo não termina para capacidades reais (veja Fig. 1.15). Usando uma busca por profundidade para achar caminhos s–t ele termina, mas é ineficiente (Dean et al., 2006).

1.6.2. O algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp elimina esses problemas. O princípio dele é simples: Para achar um caminho s–t simples, usa busca por largura, i.e. selecione o caminho mais curto entre s e t. Nos temos

{th:edmond

Teorema 1.15
O algoritmo de Edmonds-Karp precisa $O(nm)$ iterações, e portanto termina em tempo $O(nm^2)$.

{lem:dista

Lema 1.26
Seja $\delta_f(v)$ a distância entre s e v em G_f. Durante a execução do algoritmo de Edmonds-Karp $\delta_f(v)$ cresce monotonicamente para todos vértices em V.

Prova. Para $v = s$ o lema é evidente. Supõe que uma iteração modificando o fluxo f para f' diminuirá o valor de um vértice $v \in V \setminus \{s\}$, i.e., $\delta_f(v) > \delta_{f'}(v)$ ($\circ$). Seja v o vértice de menor distância $\delta_{f'}(v)$ em $G_{f'}$ com essa característica, e $P = (s, \ldots, u, v)$ um caminho mais curto de s para v em $G_{f'}$. Logo $\delta_{f'}(u)+1 = \delta_{f'}(v)$ (Δ). Pela escolha de v, o valor de u não diminuiu nessa iteração, i.e., $\delta_f(u) \leq \delta_{f'}(u)$ (*).
Supondo $uv \in A(G_f)$, temos

$$\delta_f(v) \leq \delta_f(u) + 1 \overset{(*)}{\leq} \delta_{f'}(u) + 1 \overset{(\Delta)}{=} \delta_{f'}(v),$$

uma contradição com a hipótese ($\circ$) que a distância de v diminuiu. Logo o arco uv não existe in G_f, mas $uv \in A(G_{f'})$. Isso só é possível se o fluxo de v para u aumentou nessa iteração. Em particular, vu era parte de um caminho mínimo de s para u e logo $\delta_f(v) + 1 = \delta_f(u)$ ($\dagger$). Para $v = t$ isso é uma contradição imediata. Caso $v \neq t$, temos

$$\delta_f(v) \overset{(\dagger)}{=} \delta_f(u) - 1 \overset{(*)}{\leq} \delta_{f'}(u) - 1 \overset{(\Delta)}{=} \delta_{f'}(v) - 2,$$

novamente uma contradição com a hipótese ($\circ$) que a distância de v diminuiu. Logo, o vértice v não existe. ■

Prova. (do teorema 1.15)
Chama um arco num caminho que aumenta o fluxo com capacidade igual ao gargalo *crítico*. Em cada iteração existe ao menos um arco crítico que desaparece do grafo residual. Provaremos que cada arco pode ser crítico no máximo $n/2 - 1$ vezes, e logo não temos mais que $m(n/2 - 1) = O(mn)$ iterações.
No grafo G_f em que um arco $uv \in A$ é crítico pela primeira vez temos $\delta_f(u) = \delta_f(v)-1$. O arco só aparece novamente no grafo residual caso alguma iteração

posterior diminui o fluxo em uv, i.e., aumenta o fluxo vu. Nessa iteração, com fluxo f', $\delta_{f'}(v) = \delta_{f'}(u) - 1$. Juntamente com o fato de que a distância só aumenta (lema (1.26)) obtemos

$$\delta_{f'}(u) = \delta_{f'}(v) + 1 \overset{(1.26)}{\geq} \delta_f(v) + 1 = \delta_f(u) + 2,$$

i.e., a distância do u entre dois instantes em que uv é crítico aumenta por pelo menos dois. Enquanto u é alcançável por s, a sua distância é no máximo $n-2$, porque o caminho não contém s nem t, e por isso a aresta uv pode ser crítico por no máximo $(n-2)/2 = n/2 - 1$ vezes. ■

Alt: Warum kann man hier nicht gleich $O(n^2)$ Iterationen argumentieren? Antwort: weil nicht in jeder Iteration die Distanz eines Knoten steigt, sonst würde das stimmen. (Siehe Beispiele Zadeh.)
Since $\delta_f(s) = 0$ always, this implies that arcs sv are never critical twice. This makes also sense, since a augmenting path can't go over s.
Also, if u above is not s (t it can't be since it has a successor), we obtain that is distance satisfies $1 \leq \delta_f(u) \leq n-2$, and thus the iteration bound is $1 + 2i \leq n - 2$ or $(n-3)/2$.

Zadeh (1972) apresenta instâncias em que o algoritmo de Edmonds-Karp precisa $\Theta(n^3)$ iterações, logo o resultado do teorema 1.15 é o melhor possível para grafos densos.

1.6.3. O algoritmo "caminho mais gordo" ("fattest path")

Idéia (Edmonds e Karp, 1972): usar o caminho de maior gargalo para aumentar o fluxo. (Exercício 1.7 pede provar que isso é possível com uma modificação do algoritmo de Dijkstra em tempo $O(n \log n + m)$.)

{flow:fp2}

Lema 1.27
Um fluxo f pode ser decomposto em fluxos $f_1, \ldots, f_k$ com $k \leq m$ tal que o fluxo f_i é positivo somente num caminho p_i entre s e t.
Prova. Dado um fluxo f, encontra um caminho p de s para t usando somente arcos com fluxo positivo. Define um fluxo no caminho cujo valor é o valor do menor fluxo de algum arco em p. Subtraindo esse fluxo do fluxo f obtemos um novo fluxo reduzido. Repete até o valor do fluxo f é zero.
Em cada iteração pelo menos um arco com fluxo positivo tem fluxo zero depois da subtração do caminho p. Logo o algoritmo termina em no máximo m iterações. ■

Proof wrong: consider su=2, ut=2, tv=1, vs=1: every flow can be decomposed in paths and circuits! Previous example does not hold for max flow, but there still may be circuits.
We can, however, to the following: if the flow is ayclic, it can be decomposed in that manner. Otherwise: we decompose into paths and circuits. Furthermore, a circulation can be decomposed into (directed) circuits. Then we should be also able to show the following (can we?): any flow can be decomposed into an aycilic flow and a circulation. These, then, can be further decomposed into paths, and circuits, respectively.
Funnily, there are some cyclic flows that can be decomposed into paths (but also into circuits and flows). Take $V = \{s, u, v, t\}$, and two flows suvt and svut: now we have a circuit uvu. We can remove it, and then extract two more flows svt and sut.

{flow:fp1}

Teorema 1.16
O caminho de maior gargalo aumenta o fluxo atual f de valor ν por pelo menos OPT/m, onde OPT é o fluxo máximo no grafo residual G_f.

Prova. Considere um arco crítico a no caminho de maior gargalo, com capacidade c_a no grafo residual G_g. Particiona $V = S \dot\cup T$, onde S contém s e todos vértices alcançáveis por arcos de capacidade maior que c_a. Por construção T contém pelo menos t. O corte (S, T) tem capacidade no máximo mc_a, logo pelo teorema 1.14 $\nu \leq \mathrm{OPT} \leq mc_a$. Por isso o fluxo aumenta por pelo menos $c_a \geq \mathrm{OPT}/m$. ■

Teorema 1.17
A complexidade do algoritmo de Ford-Fulkerson usando o caminho de maior gargalo é $O((n \log n + m)m \log C)$ para um limitante superior C do fluxo máximo.

Prova. Seja f_i o valor do caminho encontrado na i-ésima iteração, G_i o grafo residual após do aumento e OPT_i o fluxo máximo em G_i. Observe que G_0 é o grafo de entrada e $\mathrm{OPT}_0 = \mathrm{OPT}$ o fluxo máximo. Temos

$$\mathrm{OPT}_{i+1} = \mathrm{OPT}_i - f_i \leq \mathrm{OPT}_i - \mathrm{OPT}_i/(2m) = (1 - 1/(2m))\mathrm{OPT}_i.$$

A desigualdade é válida pelo teorema 1.16, considerando que o grafo residual possui no máximo $2m$ arcos. Logo

$$\mathrm{OPT}_i \leq (1 - 1/(2m))^i \mathrm{OPT} \leq e^{-i/(2m)}\mathrm{OPT}.$$

O algoritmo termina caso $\mathrm{OPT}_i < 1$, por isso número de iterações é no máximo $2m \ln \mathrm{OPT} + 1$. Cada iteração custa $O(m + n \log n)$. ■

Para todo $i > 0$ e x temos

$$\left(1 + \frac{x}{i}\right)^i \leq e^x.$$

Logo, com $x = -i/2m$

$$\left(1 - \frac{i/2m}{i}\right)^i \leq e^{-i/2m}.$$

Corolário 1.4
Caso U é um limite superior da capacidade de um arco, o algoritmo termina em no máximo $O(m \log mU)$ passos.

1.6.4. O algoritmo push-relabel

O algoritmo push-relabel representa uma classe de algoritmos que não trabalha com um fluxo e caminhos aumentantes, mas mantém um *pré-fluxo* f que satisfaz

- os limites de capacidade ($0 \leq f_a \leq c_a$)
- e requer somente que o *excesso* $e(v) = -f(v)$ de um vértice $v \neq s$ é não-negativo.

Vértice s pode ter execesso.

Um vértice $v \neq t$ com $e(v) > 0$ é chamado *ativo*. A ideia do algoritmo é que vértices possuem uma "altura" e o fluxo passa para vértices de altura mais baixa ("operação push") ou, caso isso não é possível a altura de um vértice ativo aumenta ("operação relabel"). Concretamente, manteremos um *potencial* ("altura") p_v para cada $v \in V$, tal que,

potential}

$$\begin{aligned} p_s = n; \quad p_t = 0; & \qquad (*) \\ p_v \geq p_u - 1 & \qquad (u,v) \in A(G_f). \end{aligned}$$

O push initial satisfaz (*).

Nota que a segunda parte da condição tem que ser satisfeita somente para arcos no grafo residual.

nce}

Observação 1.14
Pela condição (*), para um caminho $v_0, v_1, \ldots, v_k$ em G_f temos $p_{v_0} \leq p_{v_1} + 1 \leq p_{v_2} + 2 \leq \cdots \leq p_{v_k} + k$. ♢

{lem:poten

Lema 1.28
Condição (*) pode ser satisfeita sse G_f não possui caminho s–t.

Prova. "⇒": Supõe que existe um caminho s–t simples $v_0 = s, v_1, \ldots, v_k = t$. Pela observação (1.14)

$$p_s = p_{v_0} \leq p_{v_k} + k = p_t + k = k < n,$$

uma contradição. "⇐": Sejam X os vértices alcançáveis em G_f a partir de s (incluindo s). Como G_f não possui caminho s–t, $t \in \overline{X}$. Logo setando $p_v = n$ para $v \in X$ e $p_v = 0$ para $v \in \overline{X}$ satisfaz (*). ■

> Logo, o significado da condição: manter as condições de otimalidade (enquanto o algoritmo aumenta factibilidade).

O lema mostra que enquanto algoritmos de caminho aumentante são algoritmos primais, mantendo uma solução factível, até encontrar o ótimo, algoritmos da classe push-relabel podem ser vistos como algoritmos duais: eles mantém o critério de otimalidade (*), até encontrar uma solução factível.
Podemos realizar as operações "push" e "relabel" como segue. A operação "push(u, v)" num arco $(u, v) \in A(G_f)$ manda o fluxo $\min\{c_a, e(v)\}$ de u para v. A operação "relabel(v) aumenta a altura p_v do vértice v por uma unidade.

```
1  push(u,v) :=
2    { pré-condição: u é ativo }
3    { pré-condição: p_v = p_u − 1 }
4    { pré-condição: (u,v) ∈ A(G_f) }
5    aumenta o fluxo em (u,v) por min{c_(u,v), e(u)}
6    { atualiza G_f de acordo }
7  end
8
9  relabel(v) :=
10   { pré-condição: v é ativo }
11   { pré-condição: não existe (u,v) ∈ A(G_f) com p_v = p_u − 1 }
12   p_v := p_v + 1
13 end
```

1. push: done when potential falls by 1, uv disappears, vu satisfies (*), since the potential goes up.
2. relabel: done when neighboring potentials same or higher; a posteriori potential falls by at most 1, so (*)

Observe que as duas operações mantém a condição (*).

Algoritmo 1.6 (Push-relabel)
Entrada Grafo $G = (V, A, c)$ com capacidades c_a no arcos.

Saída Um fluxo f.

```
1  p_s := n;  p_v := 0,        ∀v ∈ V \ {s}
2  f_a := c_a,  ∀a ∈ N⁺(s)  senão  f_a := 0
3  while existe vértice ativo do
4    escolhe o vértice ativo u de maior p_u
5    repete até u é inativo
6      if existe arco (u,v) ∈ G_f com p_v = p_u − 1 then
7        push(u,v)
8      else
9        relabel(u)
10     end if
11   end
12 end while
13 return f
```

Lines 5–11 are called “discharge”.

Lema 1.29
O algoritmo push-relabel é parcialmente correto (i.e. correto caso termina).

Prova. Ao terminar não existe vértice ativo. Logo f é um fluxo. Pelo lema 1.28 não existe caminho s–t em G_f. Logo pelo teorema 1.14 o fluxo é ótimo. ■

A terminação é garantida por

Teorema 1.18
O algoritmo push-relabel executa $O(n^3)$ operações push e $O(n^2)$ operações relabel.

Prova. Um vértice ativo v tem excesso de fluxo, logo existe um caminho v–s em G_f. Por (1.14) $p_v \leq p_s + (n-1) < 2n$, e logo o número de operações relabel é $O(n^2)$. Supõe que uma operação push satura um arco $a = (u, v)$ (i.e. manda fluxo c_a). Para mandar fluxo novamente, temos que mandar primeiramente fluxo de v para u; isso só pode ser feito depois de pelo menos duas operações relabel em v. Logo o número de operações push saturantes é $O(mn)$. Para operações push não-saturantes, podemos observar que entre duas operações relabel temos no máximo n desses operações, porque cada uma torna o vértice de maior p_v inativo (talvez ativando vértices de menor potencial), logo tem no máximo $O(n^3)$ deles. ■

Para garantir uma complexidade de $O(n^3)$ temos que implementar um "push" em $O(1)$ e um "relabel" em $O(n)$. Para este fim, manteremos uma lista dos vértices em ordem do potencial. Para cada vértice manteremos uma lista de arcos candidatos para operações push, i.e. arcos para vizinhos com potencial um a menos com capacidade residual positiva.

Uma busca linear na lista de vértices encontra o vértice de maior potencial. Entre duas operações relabel a busca pode continuar no último ponto e precisa tempo $O(n)$ em total, logo a busca custa no máximo $O(n^3)$ sobre toda execução do algoritmo. Para a operação push podemos simplesmente consultar a lista de candidatos. Para um push saturante, o candidato será removido. Isso custa $O(1)$. Finalmente no caso de um relabel temos que encontrar em $O(n)$ a nova posição do vértice na lista, e reconstruir a lista de candidatos, que também precisa tempo $O(n)$. Logo todas operações relabel custam não mais que $O(n^3)$.

We need an example. Also: make a drawing of the data structures.

Notas on PR (Cherkassky, Goldberg 1994). PR has poor performance. 1) Algorithmic variants. Process active nodes in FIFO order, or highest label (HL) via a bucket list. 2) Heuristics. a) Global relabeling: compute distances to sink, update potential (every n, or every $m/2$ relabelings. b) Gap relabeling: if there's a gap in distances/potential, say no node with $p_v = g$ but some with $g < p_v < n$, then nodes with $g < p_v < n$ can't reach the sink, and thus are relabeled to $p_v = n$. This is combined with the next item. c) Two-phase method: consider only nodes with $p_v < n$ active, all other can't reach the sink. Their excess remains, and is removed in a second phase.

To detect gaps efficiently, keep bins B_i of nodes with $p_v = i$. Whenever some $|B_i| = 0$, we need $|B_j| = 0$ for $i < j < n$, otherwise we can empty

these bins and push their elements to B_n. We can also maintain the lowest empty bin b, and whenever some B_i with $i < b$ drops to 0 empty bins $i+1, \ldots, b-1$ and then update $b := i$.

O algoritmo de Dinitz

O algoritmo de Dinitz (1970) foi um dos primeiros de tempo polinomial, é simples de implementar e eficiente na prática (Dinitz, 2006). Ele compartilha com o algoritmo de Edmonds-Karp um foco em caminhos mais curtos no grafo residual G_f, mas tem complexidade pessimista $O(n^2m)$ melhor.
A ideia central do algoritmo é a seguinte iteração:

1. considere o subgrafo H de G_f com todos arcos que pertencem a um caminho mais curto de s a t,
2. encontra um fluxo f em H, tal que no subgrafo de H sem arcos saturados não existe mais caminho s–t,
3. aumenta o fluxo atual por f.

Em comparação com o algoritmo de Edmonds-Karp que bloqueia um caminho mais curto, o algoritmo de Dinitz bloqueia todos. O grafo do passo 1 é um *grafo em camadas*. Com $\delta_f(v)$ a distância do caminho mais curto em G_f de s a $v \in V$ (em arcos), os vértices com $\delta_v = k$ formam a k-ésima camada, e todos arcos (u, v) em H satisfazem $\delta_u + 1 = \delta_v$. O fluxo do passo 2 é um *fluxo bloqueio*.
A corretude parcial do algoritmo segue diretamente do lema 1.25. A terminação é consequência do

Lema 1.30
As distâncias $\delta_f(v)$ aumentam em cada iteração.

Prova. (TBD.) ■
Como $\delta_f(t) < n$ o algoritmo termina em menos que n iterações.
Ainda temos

Lema 1.31
Um fluxo bloqueio pode ser encontrado em tempo $O(nm)$.

Prova. Repetidamente busca em profundidade até encontrar t, aplica o fluxo, remove todos arcos encontrados durante a bysca e repete, até não existe mais caminho s–t. Cada busca custa $O(n + a)$ com a arcos visitados, e não temos mais que m iterações, logo o custo total é $O(nm)$. ■

Junto com $O(n)$ iterações, o lema garante uma complexidade de $O(n^2 m)$. Usando melhor algoritmo para encontrar um fluxo bloqueio de complexidade $O(m \log n)$ (Sleator, Tarjan 1983) a complexidade pode ser reduzida a $O(nm \log n)$.

Unclear whom I follow here, possibly Schrijver.

1.6.5. Algoritmo de escalonamento

Gabow/1985.
TBD: talk the easy scaling algorithm: say $G_f(\Delta)$ is the residual graph where we permit only edges of residual capacity more or equal to Δ. Then we simply start with $\Delta := 2^k$, where $k = \lfloor \log_2 C^* \rfloor$ and $C^* = \max_{v \in V} c_{sv}$ is the maximum augmentation possible on any s-t-path. If there is no augmenting path in $G_f(\Delta)$, we step down to $\Delta := \Delta/2$. The algorithm terminates after processing with $\Delta = 1$, so we will find all augmenting paths. Advantage: we have only $O(\log C^*)$ phases, and each phase makes at most $2m$ augmentations (TBD: prove it), so we have a (polynomial) $O(m^2 \log C^*)$ algorithm. Kleinberg/Tardos have this in their book.

1.6.6. Variantes do problema

Fontes e destinos múltiplos Para $G = (V, A, c)$ define um conjunto de fontes $S \subseteq V$ e um conjunto de destinos $T \subseteq V$, com $S \cap T = \emptyset$, e considera

$$\begin{aligned} \textbf{maximiza} \quad & f(S) \\ \textbf{sujeito a} \quad & f(v) = 0, \qquad \forall v \in V \setminus (S \cup T), \\ & f_a \le c_a, \qquad \forall a \in A. \end{aligned} \tag{1.15}$$ {multiflow

Tabela 1.3.: Complexidade de diversos algoritmos de fluxo máximo (partes de Schrijver, 2003).

Ano	Referência	Complexidade	Obs
1951	Dantzig	$O(n^2mC)$	Simplex
1955	Ford & Fulkerson	$O(mC) = O(mnU)$	Cam. aument.
1970	Dinitz	$O(nm^2)$	Cam. min. aument.
1972	Edmonds & Karp	$O(m^2 \log C)$	Escalonamento
1973	Dinitz	$O(nm \log C)$	Escalonamento
1974	Karzanov	$O(n^3)$	Preflow-Push
1977	Cherkassky	$O(n^2m^{1/2})$	Preflow-Push
1986	Goldberg & Tarjan	$O(nm \log(n^2/m))$	Push-Relabel
1987	Ahuja & Orlin	$O(nm + n^2 \log C)$	Push-Relabel & Esc.
1990	Cheriyan et al.	$O(n^3/\log n)$	
1990	Alon	$O(nm + n^{8/3} \log n)$	
1992	King et al.	$O(nm + n^{2+\epsilon})$	
1997	Goldberg & Rao	$O(m^{3/2} \log(n^2/m) \log C)$	
		$O(n^{2/3}m \log(n^2/m) \log C)$	
2012	Orlin	$O(nm)$	
2022	Chen et al.	$O(m^{1+o(1)})$	Pontos interiores

O problema (1.15) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', A', c')$ (veja Fig. 1.16(a)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ A' &= A \cup \{s^*\} \times S \cup T \times \{t^*\} \qquad (1.16) \\ c'_a &= \begin{cases} c_a, & a \in A, \\ c(S, \bar{S}), & a = (s^*, s), s \in S, \\ c(\bar{T}, T), & a = (t, t^*), t \in T. \end{cases} \end{aligned}$$

Lema 1.32
Se f' é uma solução máxima de (1.16), a restrição $f = f'|_A$ é uma solução máxima de (1.15) de mesmo valor. Por outro lado, se f é uma solução máxima de (1.15), a extensão

$$f'_a = \begin{cases} f_a, & a \in A, \\ f(s), & a = (s^*, s), s \in S, \\ -f(t), & a = (t, t^*), t \in T, \end{cases} \qquad (1.17)$$

é uma solução máxima de (1.16) de mesmo valor.

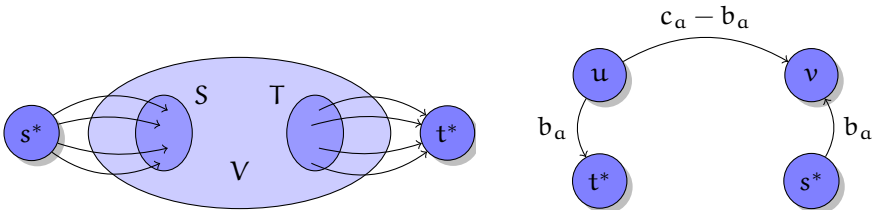


Figura 1.16.: Reduções entre variantes do problema do fluxo máximo. Esquerda: Fontes e destinos múltiplos. Direita: Limite inferior e superior para a capacidade de arcos.

{fig:reduc

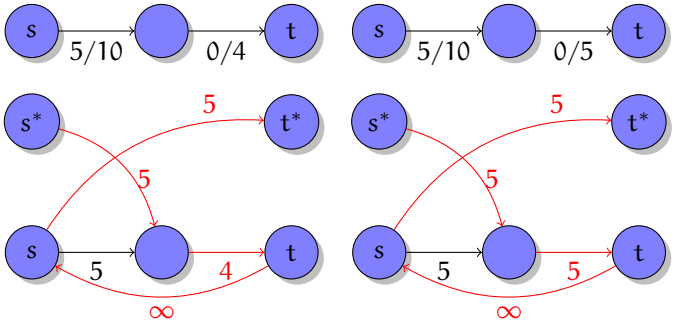


Figura 1.17.: Dois exemplos da transformação do lema 1.33. Esquerda: Grafo sem solução viável e grafo transformado com fluxo máximo 4. Direita: Grafo com solução viável e grafo transformado com fluxo máximo 5.

{fig:limin

Prova. Se f' é solução de (1.16), a restrição $f = f'|_A$ é uma solução de (1.15) de mesmo valor: f é viável porque $f(v) = f'(v) = 0$ para todo $v \in V \setminus S \setminus T$ e $f(S) = \sum_{s \in S} f(s) = f'(s^*)$. Similarmente, dado um fluxo válido f em G, a extensão f' (1.17) é um fluxo válido em G' de mesmo valor: f' é viável porque além de $f'(v) = f(v) = 0$ para todo $v \in V \setminus (S \cup T)$, também $f'(v) = 0$ para $v \in S \cup T$, e $f'(s^*) = \sum_{s \in S} f(s) = f(S)$. ■

On lower limits (or arc demands). Orlin and others seem to solve this problem in a more complicated manner, Kleinberg & Tardos don't have it. A similar construction can be found in Bateni (see material), or Vakken, or Jeffe.

Limites inferiores Para $G = (V, A, b, c)$ com limites inferiores $b : A \to \mathbb{R}$ considere o problema

$$\begin{aligned} \textbf{maximiza} \quad & f(s) \\ \textbf{sujeito a} \quad & f(v) = 0, && \forall v \in V \setminus \{s,t\}, \\ & b_a \le f_a \le c_a, && a \in A. \end{aligned} \tag{1.18}$$

{lim

O problema (1.18) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', A', c')$ (veja Fig. 1.16(b)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ A' &= A \cup \{(u,t^*) \mid (u,v) \in A\} \cup \{(s^*,v) \mid (u,v) \in A\} \cup \{(t,s)\} \\ c'_a &= \begin{cases} c_a - b_a, & a \in A, \\ \sum_{v \in N^+(u)} b_{(u,v)}, & a = (u,t^*), \\ \sum_{u \in N^-(v)} b_{(u,v)}, & a = (s^*,v), \\ \infty, & a = (t,s). \end{cases} \end{aligned} \tag{1.19}$$

ed:liminf}

Chama um fluxo em 1.19 *saturado*, caso ele satura todos arcos auxiliares $\{(u,t^*) \mid (u,v) \in A\} \cup \{(s^*,v) \mid (u,v) \in A\}$.

em:liminf}

Lema 1.33
Problema (1.18) possui um fluxo viável sse (1.19) possui um fluxo saturado.

Prova. Caso f é um fluxo viável em (1.18),

$$f'_a = \begin{cases} f_a - b_a, & a \in A, \\ \sum_{u \in N^+(v)} b_{(v,u)}, & a = (v,t^*), \\ \sum_{u \in N^-(v)} b_{(u,v)}, & a = (s^*,u), \\ f(s), & a = (t,s). \end{cases}$$

é um fluxo saturado de (1.19). Por outro lado, se f' é um fluxo saturado para (1.19), $f_a = f'_a + b_a$ é um fluxo viável em (1.18). ■

One can go into details. First f' is feasible, since $b \le f \le c$ implies $0 \le f' = f - b \le c - b$, and the capacities to $t^{(*}$ and from s^* are saturated. We also have flow conservation: $f'(s) = f(s) - f(s) = 0$, $f'(t) = f(t) + f(s) = -f(s) + f(s) = 0$, and $f'(v) = \sum_{a \in N^+(v)}(f_a - b_a + b_a) + \sum_{a \in N^-(v)}(f_a - b_a + b_a) = f(v) = 0$.

Como um fluxo saturado tem que ser máximo, ele pode ser obtido por um algoritmo de fluxo máximo aplicado a (1.19). Caso o fluxo máximo não satura, não tem solução viável, senão podemos extrair uma solução viável de (1.18) pela construção acima. Para obter um fluxo máximo de (1.18) podemos maximizar o fluxo a partir da solução viável obtida, com qualquer variante do algoritmo de Ford-Fulkerson. Na execução temos que garantir que um fluxo mínimo de b_a é mantido em cada arco $a = (u, v)$. Logo, o grafo residual de um fluxo f tem arcos "backward" $\bar{a} = (v, u)$ de capacidade reduzida $c_{\bar{a}} = f_a - b_a$.

Uma alternativa para obter um fluxo factível com limites inferiores nos arcos é primeiro mandar o limite inferior de cada arco, i.e. setar $f = b$, e depois considerar demandas $d_v = -f(v)$. Uma circulação factível com limites $0 \leq f \leq c - b$ corresponde com um fluxo factível $f + b$ no grafo original.

Existência de uma circulação com demandas nos vértices Para $G = (V, A, c)$ com demandas d_v, com $d_v > 0$ para destinos e $d_v < 0$ para fontes, considere

$$\begin{array}{lll} \textbf{existe} & f & \\ \textbf{s.a} & f(v) = -d_v, & \forall v \in V, \\ & f_a \leq c_a, & a \in A. \end{array} \tag{1.20}$$ {circulati

Evidentemente $\sum_{v \in V} d_v = 0$ é uma condição necessária (lema (1.3)). O problema (1.20) pode ser reduzido para um problema de fluxo máximo em $G' = (V', A')$ com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ A' &= A \cup \{(s^*, v) \mid v \in V, d_v < 0\} \cup \{(v, t^*) \mid v \in V, d_v > 0\} \\ c_a &= \begin{cases} c_a, & a \in A, \\ -d_v, & a = (s^*, v), \\ d_v, & a = (v, t^*). \end{cases} \end{aligned} \tag{1.21}$$ {red:circu

Lema 1.34
Problema (1.20) possui uma solução sse problema (1.21) possui uma solução com fluxo máximo $D = \sum_{v : d_v > 0} d_v$.

Prova. (Exercício.) ■

Circulações com limites inferiores Para $G = (V, A, b, c)$ com limites inferiores e superiores, considere

$$\begin{aligned} \textbf{existe} \quad & f \\ \textbf{s.a} \quad & f(v) = d_v, && \forall v \in V, \\ & b_a \leq f_a \leq c_a, && a \in A. \end{aligned} \tag{1.22}$$ {cir

O problema pode ser reduzido para a existência de uma circulação com somente limites superiores em $G' = (V', A', c', d')$ com

ircliminf}

$$\begin{aligned} V' &= V \\ A' &= A \\ c_a &= c_a - b_a \end{aligned} \tag{1.23}$$

$$d'_v = d_v - \sum_{a \in N^-(v)} b_a + \sum_{a \in N^+(v)} b_a \tag{1.24}$$

Lema 1.35
O problema (1.22) possui solução sse problema (1.23) possui solução.

Prova. (Exercício.) ■

1.6.7. Aplicações

Cobertura mínima em grafos bipartidos Include it here or somewhere in the matching part. Check Trevisan's lecture 14 or the original source.

Projeto de pesquisa de opinião O objetivo é projetar uma pesquisa de opinião, com as restrições

- Cada cliente i recebe ao menos c_i perguntas (para obter informação suficiente) mas no máximo c'_i perguntas (para não cansar ele). As perguntas podem ser feitas somente sobre produtos que o cliente já comprou.
- Para obter informações suficientes sobre um produto, entre p_i e p'_i clientes tem que ser interrogados sobre ele.

Um modelo é um grafo bi-partido entre clientes e produtos, com aresta (c_i, p_j) caso cliente i já comprou produto j. O fluxo de cada aresta possui limite inferior 0 e limite superior 1. Para representar os limites de perguntas por produto e por cliente, introduziremos ainda dois vértices s, e t, com arestas (s, c_i) com fluxo entre c_i e c'_i e arestas (p_j, t) com fluxo entre p_j e p'_j e uma aresta (t, s).

Segmentação de imagens O objetivo é segmentar um imagem em duas partes, por exemplo "foreground" e "background". Supondo que temos uma "probabilidade" a_i de pertencer ao "foreground" e outra "probabilidade" de pertencer ao "background" b_i para cada pixel i, uma abordagem direta é definir que pixels com $a_i > b_i$ são "foreground" e os outros "background". Um exemplo pode ser visto na Fig. 1.19 (b). A desvantagem dessa abordagem é que a separação ignora o contexto de um pixel. Um pixel, "foreground" com todos pixel adjacentes em "background" provavelmente pertence ao "background" também. Portanto obtemos um modelo melhor introduzindo penalidades p_{ij} para separar (atribuir à categorias diferentes) pixels adjacentes i e j. Um partição do conjunto de todos pixels I em $A \dot{\cup} B$ tem um valor de

$$q(A, B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j) \in A \times B} p_{ij}$$

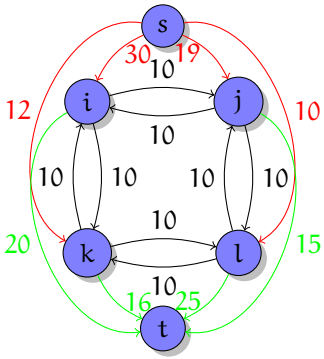
nesse modelo, e o nosso objetivo é achar uma partição que maximiza $q(A, B)$. Isso é equivalente a minimizar

$$\begin{aligned} Q(A, B) &= \sum_{i \in I} a_i + b_i - \sum_{i \in A} a_i - \sum_{i \in B} b_i + \sum_{(i,j) \in A \times B} p_{ij} \\ &= \sum_{i \in B} a_i + \sum_{i \in A} b_i + \sum_{(i,j) \in A \times B} p_{ij}. \end{aligned}$$

A solução mínima de $Q(A, B)$ pode ser visto como corte mínimo num grafo. O grafo possui um vértice para cada pixel e uma aresta com capacidade p_{ij} entre dois pixels adjacentes i e j. Ele possui ainda dois vértices adicionais s e t, arestas (s, i) com capacidade a_i para cada pixel i e arestas (i, t) com capacidade b_i para cada pixel i (ver Fig. 1.18).

Sequenciamento O objetivo é programar um transporte com um número k de veículos disponíveis, dado pares de origem-destino com tempo de saída e chegada. Um exemplo é um conjunto de vôos é

1. Porto Alegre (POA), 6.00 – Florianopolis (FLN), 7.00
2. Florianopolis (FLN), 8.00 – Rio de Janeiro (GIG), 9.00
3. Fortaleza (FOR), 7.00 – João Pessoa (JPA), 8.00
4. São Paulo (GRU), 11.00 – Manaus (MAO), 14.00
5. Manaus (MAO), 14.15 – Belem (BEL), 15.15



	i	j	k	l
a	30	19	12	10
b	20	15	16	25

Figura 1.18.: Exemplo da construção para uma imagem 2×2. Direita: Tabela com valores pele/não-pele. Esquerda: Grafo com penalidade fixa $p_{ij} = 10$. {fig

Figura 1.19.: Segmentação de imagens com diferentes penalidades p. Acima: (a) Imagem original (b) Segmentação somente com probabilidades ($p = 0$) (c) $p = 1000$ (d) $p = 10000$. Abaixo: (a) Walter Gramatté, Selbstbildnis mit rotem Mond, 1926 (b) Segmentação com $p = 10000$. A probabilidade de um pixel representar pele foi determinado conforme Jones e Rehg (1998). {fig

6. Salvador (SSA), 17.00 – Recife (REC), 18.00

O mesmo avião pode ser usado para mais que um par de origem e destino, se o destino do primeiro é o origem do segundo, em tem tempo suficiente entre a chegada e saída (para manutenção, limpeza, etc.) ou tem tempo suficiente para deslocar o avião do destino para o origem.
Podemos representar o problema como grafo direcionado acíclico. Dado pares de origem destino, ainda adicionamos pares de destino-origem que são compatíveis com as regras acimas. A idéia é representar aviões como fluxo: cada aresta origem-destino é obrigatório, e portanto recebe limites inferiores e superiores de 1, enquanto uma aresta destino-origem é facultativa e recebe limite inferior de 0 e superior de 1. Além disso, introduzimos dois vértices s e t, com arcos facultativos de s para qualquer origem e de qualquer destino para t, que representam os começos e finais da viagem completa de um avião. Para decidir se existe um solução com k aviões, finalmente colocamos um arco (t, s) com limite inferior de 0 e superior de k e decidir se existe uma circulação nesse grafo.

O problema $P \mid \text{pmtn}, r_i \mid L_{\max}$ Primeiramente resolveremos um problema mais simples: será que existe um sequenciamento tal que toda tarefa i executa dentro do seu intervalo $[r_i, d_i]$? Equivalentemente, será que existe uma solução com $L_{\max} = 0$?
Seja $\{t_1, t_2, \ldots, t_k\} = \{r_1, r_2, \ldots r_n\} \cup \{d_1, d_2, \ldots, d_n\}$, com $t_1 \leq t_2 \leq \cdots \leq t_k$. (Observe que $k \leq 2n$, e $k < 2n$ no caso de tempos repetidos.) Podemos ver os t_i como eventos em que uma tarefa fica disponível ou tem que terminar o seu processamento. Os t_i definem $k-1$ intervalos $I_i = [t_i, t_{i+1}]$ para $i \in [k-1]$ com duração $S_i = t_{i+1} - t_i$ correspondente. Cada tarefa j pode ser executada no intervalo T_i caso $I_i \subseteq [r_i, d_i]$. Logo podemos modelar o problema via um grafo direcionado bipartido com vértices $T \dot\cup I$, sendo $T = [n]$ o conjunto de tarefas e $I = \{I_i \mid i \in [k-1]\}$ o conjunto de intervalos, e com arcos (j, i) caso tarefa j pode ser executada no intervalo i. Para completar o grafo adicionaremos um arco (s, j) de um vértice origem s para cada tarefa j, e um arco (i, t) de cada intervalo para um vértice destino t. Um fluxo nesse grafo representa tempo, e teremos capacidades p_j entre s e tarefa j, S_i entre tarefa j e intervalo i, e mS_i entre T_i e t, sendo mS_i o tempo total disponível durante o intervalo i. Figura 1.20 mostra a construção completa.
Logo $P \mid \text{pmtn}, r_i \mid L_{\max}$ pode ser resolvido em tempo $O(mn \log \bar{L})$.
Com essa abordagem podemos resolver o problema original por busca binária: para cada valor do $L_{\max}$ entre 0 e $\bar{L}$ testaremos se existe uma solução tal que cada tarefa executa no intervalo $[r_i, d_i + L_{\max}]$. Um limite superior simples é

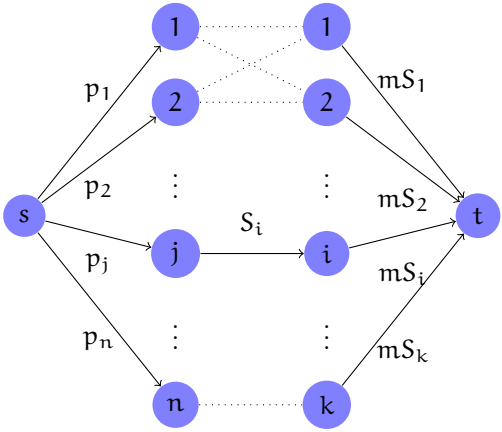


Figura 1.20.: Problema de fluxo para resolver a versão de decisão do problema $P \mid pmtn, r_i \mid L_{max}$. {fig

$\bar{L} = \max_i r_i + \sum_i p_i - \min_i d_i$ executando todas tarefas após a liberação da última numa única máquina em ordem arbitrária.

O problema $1 \mid prec \mid \sum w_j C_j$ Este problema é NP-completo, mas um teorema de Sidney (19xx) mostra que podemos decompor a ordenação parcial em conjuntos inicias de maior densidade $\rho(I) = w(S)/p(S)$. Ordenando os processos em cada componente arbitrariamente, obtemos uma 2-aproximação. O conjunto inicial de maior densidade é obtido por uma busca binária em densidade, resolvendo um problema de fluxo em cada iteração.
Para decidir se a maior densidade é maior ou menor que uma densidade candidata ρ construiremos um grafo $G = ...$ etc. etc.
A 2-aproximação segue pelo
Lema 1.36
TBD: Every such a solution has cost at least $w(S)p(S)/2$ and at most $w(S)p(S)/2$.

O problema $P \mid p_j = 1, r_j \mid \sum f_j$ Suppose can be done like this: a bipartite graph matches jobs to time intervals. Time intervals are defined by release dates. Each job–interval arc has capacity 1, each arc from a time interval to the sink the total number of slots in this time interval

(difference of release times, times number of machines). We want to find a flow of value n, of minimum cost, where the cost of the job–interval arcs corresponds to the actual scheduling cost. For lateness & tardiness costs, we may have to introduce additional events at the due dates.

Agendamento de projetos Suponha que temos n projetos, cada um com lucro $p_i \in \mathbb{Z}$, $i \in [n]$, e um grafo de dependências $G = ([n], A)$ sobre os projetos. Caso $(i,j) \in A$, a execução do projeto i é pré-requisito para a execução do projeto j. Um lucro pode ser negativo, e neste caso representa uma perda. Este problema pode ser reduzido para um problema de fluxo máximo s-t: cria um grafo G' com vértices $V = \{s,t\} \cup [n]$ é

- uma aresta (s,v) para todo $v \in [n]$ com $p_v > 0$, com capacidade p_v,
- uma aresta (v,t) para todo $v \in [n]$ com $p_v < 0$, com capacidade $-p_v$, e
- uma aresta (u,v) para toda dependência $(v,u) \in A$, com capacidade ∞.

(Note que projetos $v \in V$ com $p_v = 0$ não geram arcos (s,v) nem (v,t).)

Lema 1.37
O valor de um corte $(X, \bar{X})$ em G' é mínimo, sse o lucro total dos projetos $S = X \setminus \{s\}$ é máximo. Além disso um corte mínimo em G' corresponde a uma seleção factível de projetos S.

Prova. Cada corte $(X, \bar{X})$ corresponde com uma seleção de projetos $S = X \setminus \{s\}$. Seja $\bar{S} = [n] \setminus S$. Uma seleção de projetos S é válida, caso para todo projeto $p \in S$, ela contém também todos projetos pré-requisitos de p. Logo, o corte correspondente não possui arcos com capacidade ∞. Como o valor do corte $(s, V \setminus \{s\})$ é $\sum_{v \in [n] | p_v > 0} c_{sv}$ o corte mínimo é finito, e logo factível, porque não pode conter um arco entre um projeto selecionado e um projeto pré-requisito não selecionado.
O valor de um corte factível é

$$c(X, \bar{X}) = \sum_{a \in A(X,\bar{X})} c_a = \sum_{v \in \bar{S} | p_v > 0} p_v - \sum_{v \in S | p_v < 0} p_v$$

e nos temos

$$\begin{aligned} \sum_{v \in [n] | p_v > 0} p_v - c(X, \bar{X}) &= \sum_{v \in [n] | p_v > 0} p_v - \sum_{v \in \bar{S} | p_v > 0} p_v + \sum_{v \in S | p_v < 0} p_v \\ &= \sum_{v \in S | p_v > 0} p_v + \sum_{v \in S | p_v < 0} p_v \\ &= \sum_{v \in S} p_v, \end{aligned}$$

i.e. o lucro total da seleção S. Logo o lucro total é máximo sse o valor do corte é mínimo. ■

Vencendo um torneio. Suponha que temos um torneio de n equipes e que elas já ganharam $w_1, \ldots, w_n$ vezes até agora. Para cada par de equipes, ainda temos g_{ij} jogos pela frente (g é simétrico). A equipe 1 ainda pode terminar em primeiro lugar, ou seja, ter o maior número de vitórias?
Para a equipe i, seja $r_i = \sum_j g_{ij}$ o número de jogos restantes. Precisamos que i) a equipe 1 vence todos os seus r_1 jogos restantes, portanto, tem $w_1 + r_1$ vitórias, e ii) todas as outras equipes $i \in T = [2, n]$ vencem no máximo m_i jogos, dado por $w_i + m_i < w_1 + r_1$ i.e. $m_i = w_1 + r_1 - w_i - 1$ (excluindo empates).
Caso algum $m_i < 0$ a equipe 1 já não pode ganhar mais. Caso contrário uma redução para um problema de fluxo é como segue. Cria um grafo com vértices s, $G = \binom{T}{2}$, T, e t e com os seguintes arcos:

- (s, g) para todo $g = (i, j) \in G$ de capacidade g_{ij},
- (g, i) e (g, j) para todo $g = (i, j) \in G$ de capacidade ∞,
- (i, t) para todo $i \in T$ de capacidade m_i.

Nos temos

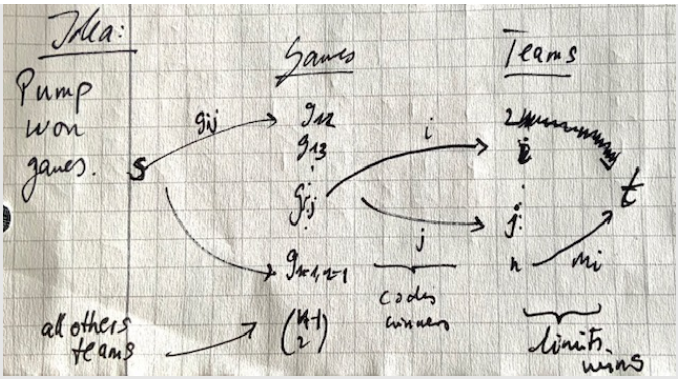
Lema 1.38
Equipe 1 ainda pode vencer sse o grafo acima possui um fluxo st que satura (i.e. de valor $\sum_{(i,j)\in G} g_{ij}$).

Prova. (Exercício. Nota que o que "flui" são jogos, e mandar fluxo em (g, i) ou (g, j) codifica que vence.) ■

Winning a tournament Assume we got n teams, and they have already won $w_1, \ldots, w_n$ times so far. For each pair of teams, we still have g_{ij} games ahead (g is symmetric). Can team 1 still finish first, i.e. have the highest number of wins? For team i, let $r_i = \sum_j g_{ij}$ be the number of remaining games. We need:

- Team 1 winning all its r_1 remaining games, so having $w_1 + r_1$ wins.
- All the other teams: winning at most m_i games, defined by $w_i + m_i \leq w_1 + r_1$ so $m_i = w_1 + r_1 - w_i$ (ignoring ties).

Solution idea: "pump won games". Consider all $\binom{n-1}{2}$ encounters between teams $2, \ldots, n$, and create a graph as follows



where each "encounter node" receives flow at most g_{ij}, sends flow either to i or j, which encodes the wins, and each "team node" can drain at most m_i wins.
Then we can show: Theorem: Team 1 can win iff there's a saturating flow. The time: we have $O(n^2)$ vertices and edges, so we need $O(n^2)$ with the algorithm of Chen et al. (2022).

Disjoint pahths. The maximum number k of disjoint st-paths in directed graphs can be found by a maximum st-flow f with unit capacity. The argument is simple: if we have k disjoints path, we can send a flow of 1 along them, thus $f \geq k$. On the other hand, if we have a flow of f, repeat the following. Find an st-path in f, by following arcs starting at s. This path must exist: due to flow conservation we can also move forward; if we enter a cycle, we can remove, without altering the flow. Thus we must end up in t. Remove this path and repeat. This removes a flow of 1, showing that $k \geq f$.
For undirected graphs, just use the usual transformation to a directed graph with opposing arcs. If any such pair of arcs in the flow is 1, we can remove both. The reduced flow has at most one arcs between any pair of vertices u, v, and thus serves in the undirected case, too.

Done in 2022/2, based on older lecture note of Jeff Erickson.

Binary assignment Consider an assignment between sets X and Y, i.e. we want to select the largest multiset $M \subset X \times Y$, such that

- each $x \in X$ is selected at most $c(x)$ times;
- each $y \in Y$ is selected at most $c(y)$ times;
- each $(x, y) \in E$ is selected at most $c(x, y)$ times.

(A bipartite matching in $G = (X \dot\cup Y, E)$ is a special case of this problem with $c(x) = c(y) = 1$, and $c(x, y) = [xy \in E]$.)
The solution is clearly to create a flow graph from s over X and Y to t, where each (s, x) has capacity $c(x)$, each (x, y) capacity $c(x, y)$ and each (y, t) capacity $c(y)$. Since all capacities are integer, the maximum flow f^* also is, and we can decompose it into f^* xyt-flows of value 1, and add each of the edges.

1.6.8. Outros problemas de fluxo

Obtemos um outro problema de fluxo em redes introduzindo *custos* de transporte por unidade de fluxo:

FLUXO DE MENOR CUSTO

Entrada Grafo direcionado $G = (V, A)$ com capacidades $c \in \mathbb{R}_+^{|E|}$ e custos $k \in \mathbb{R}_+^{|E|}$ nos arcos, um vértice origem $s \in V$, um vértice destino $t \in V$, e valor $v \in \mathbb{R}_+$.

Solução Um fluxo s-t f com valor v, respeitando as capacidades ($f \leq c$).

Objetivo Minimizar o *custo* $\sum_{a \in A} k_a f_a$ do fluxo.

Diferente do problema de menor fluxo, o valor do fluxo é fixo.

(We follow Schrijver here.) Alternatively to finding a minimum cost flow (MCF) we can find

1) a maximum st-flow of least cost (by finding the maximum flow first, and then minimizing the cost), or
2) a minimum cost circulation under arc demands (i.e. flow lower bounds) $d_a \in \mathbb{Q}$, $a \in A$ (MCC).

The reductions are as follows:

1) MCF to MCC: add a back-arc (t, s) of cost $k_{ts} = 0$ and set $d_{ts} =$

$c_{ts} = \nu$ to the desired flow.

2) maximum flow to MCC: add a back-arc (t, s) of cost $k_{ts} = -1$ and set $k \equiv 0$ on all other arcs.

The problem of solving MCC can be broken down in two problems: a) find any feasible circulation, and b) in spirit similar to flow-augmenting methods, the problem of finding a circulation of lower cost. We have discussed before how to solve problem a). For solving problem, we first can observe that if there's a negative cost circuit in the residual graph G_f, we can apply it to reduce the cost. Even better, if there is no such circuit, the circulation is of least cost.
Notation: G_f is the residual graph, as always, but it has forward arcs only if $f_a < c_a$, and backward arcs only if $d_a < f_a$. We set the cost of a backward arc a^{-1} to $k_{a^{-1}} = k_a$ where $uv^{-1} = vu$. For a circuit C, define the characteristic vector

$$\chi^C(a) = \begin{cases} 1 & \text{if the circuit uses } a, \\ -1, & \text{if the circuit uses } a^{-1}, \\ 0, & \text{otherwise.} \end{cases}$$

Teorema 1.19
A feasible circulation f has minimum cost iff each directed circuit in G_f has non-negative cost.

Prova. "⇒": if some circuit C in G_f has negative cost, then for a small enough ϵ flow $f' = f + \epsilon\chi^C$ remains feasible and has lower cost.
"⇐": take any feasible circulation f'. Then $f' - f$ is also a circulation (but not necessarily feasible!), and

$$f' - f = \sum_{j \in [m]} \lambda_j \chi^{C_j}$$

for some directed circuits $C_1, \dots C_m$, and $\lambda_1, \dots, \lambda_m > 0$. (This is not 100% clear: probably every flow can be decomposed in this manner (e.g. take $f = 0$).) Thus

$$k(f') - k(f) = k(f' - f) = \sum_{j \in [m]} \lambda_j k(C_j) \geq 0.$$

So $k(f') \geq k(f)$. ■

There is still one problem: this may take exponential time! Solution: select a circuit of minimum mean cost $k(C)/|C|$. Then:

Teorema 1.20
The above takes at most $4nm^2 \lceil \ln n \rceil$ iterations.

Therefore: MCC can be solved in time $O(n^2 m^3 \log n)$. For integer c, d the circulation is also integer.

Finding minimum mean cycles in time $O(nm)$. (TBD.)

- Discuss parametric/maximum flow over time.
- Discuss minimum cost flow with lower bounds (ps4.ps, two max-flows).
- Discuss Chen et al. (2022), and the popularization Klarreich (2022).

Disser e Skutella (2015) shows that the Simplex algorithm is NP-mighty, in the sense that every problem in NP can be decided transforming its input to an input of Simplex, and then responding "yes" iff a given bit in the input flips during the execution (with Dantzig's rule). They reduce an instance of Partition to a minimum-cost flow problem, such that Simplex (and the successive shortest path algorithm) augment flow on a certain arc iff the Partition instance has a solution.

1.6.9. Exercícios

{flow:ex1}

Exercício 1.7
Mostra como podemos modificar o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois vértices num um grafo para encontrar o caminho com o maior gargalo entre dois vértices. (Dica: Enquanto o algoritmo de Dijkstra procura o caminho com a menor soma de distâncias, estamos procurando o caminho com o maior capacidade mínimo.)

1.7. Emparelhamentos

Dado um grafo não-direcionado $G = (V, A)$, um *emparelhamento* é uma seleção de arestas $M \subseteq A$ tal que todo vértice tem no máximo grau 1 em $G' = (V, M)$. (Notação: $M = \{u_1v_1, u_2v_2, \ldots\}$.) O nosso interesse em emparelhamentos é maximizar o número de arestas selecionados ou, no caso as arestas possuem pesos, maximizar o peso total das arestas selecionados.
Para um grafo com pesos $c : A \to \mathbb{Q}$, seja $c(M) = \sum_{e \in M} c_e$ o *valor* do emparelhamento M.

Emparelhamento máximo (EM)

Entrada Um grafo não-direcionado $G = (V, A)$.

Solução Um emparelhamento $M \subseteq A$, i.e. um conjunto de arestas, tal que para todos vértices v temos $|N(v) \cap M| \leq 1$.

Objetivo Maximiza $|M|$.

Emparelhamento de peso máximo (EPM)

Entrada Um grafo não-direcionado $G = (V, A, c)$ com pesos $c : A \to \mathbb{Q}$ nas arestas.

Solução Um emparelhamento $M \subseteq A$.

Objetivo Maximiza o valor $c(M)$ de M.

Um emparelhamento se chama *perfeito* se todo vértice possui vizinho em M. Uma variação comum do problema é

Emparelhamento perfeito de peso mínimo (EPPM)

Entrada Um grafo não-direcionado $G = (V, A, c)$ com pesos $c : A \to \mathbb{Q}$ nas arestas.

Solução Um emparelhamento perfeito $M \subseteq A$, i.e. um conjunto de arestas, tal que para todos vértices v temos $|N(v) \cap M \models 1$.

Objetivo Minimiza o valor $c(M)$ de M.

Observe que os pesos em todos problemas podem ser negativos. O problema de encontrar um emparelhamento de peso mínimo em $G = (V, A, c)$ é equivalente

com EPM em $-G := (V, A, -c)$ (por quê?). Até EPPM pode ser reduzido para EPM.

Teorema 1.21
EPM e EPPM são problemas equivalentes.

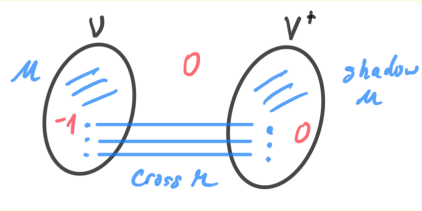
Prova. Seja $G = (V, A, c)$ uma instância de EPM. Define um conjunto de vértices $V' = V \cup V^+$ que contém além de V mais $|V|$ vértices novos $V^+ = \{v^+ \mid v \in V\}$, e um grafo completo $G' = (V', V' \times V', c')$ com

$$c'_a = \begin{cases} -c_a, & \text{caso } a \in A, \\ 0, & \text{caso contrário.} \end{cases}$$

Um emparelhamento M em G de custo $c(M)$ corresponde com um emparelhamento M' em G' como segue. Dado M, define

$$M' = M \cup \{u'v' \mid uv \in M\} \cup \{vv' \mid v \text{ livre em } M\},$$

dado M' define $M = M' \cap V^2$. Ambas construções só adicionam ou removem arestas de custo 0 e o custo das demais arestas é invertido, logo $c'(M') = -c(M)$. Portanto, um EPPM em G' é um EPM em G.



If $|V|$ is even, we can just multiply by -1, remove (now) positive edges, and complete the graph with 0-edges.

Por outro lado, seja $G = (V, A, c)$ uma instância de EPPM. Define $C := 1 + \sum_{a \in A} |c_a|$, novos pesos $c'_e = C - c_e$ e um grafo $G' = (V, A, c')$. Para emparelhamentos M_1 e M_2 em G arbitrários temos

ineq:diff}

$$c(M_2) - c(M_1) \leq \sum_{\substack{a \in A \\ c_a > 0}} c_a - \sum_{\substack{a \in A \\ c_a < 0}} c_a = \sum_{a \in A} |c_a| < C. \qquad (*)$$

Idea: Difference by any number of original edges is always $< C$, so one more new edge is always better.

Portanto, um emparelhamento de peso máximo em G' também é um emparelhamento de cardinalidade máxima: Para $|M_1| < |M_2|$ temos

$$c'(M_1) = C|M_1| - c(M_1) < C|M_1| + C - c(M_2) \leq C|M_2| - c(M_2) = c'(M_2),$$

onde a primeira desigualdade segue por (*). Se existe um emparelhamento perfeito no grafo original G, então o EPM em G' é perfeito e as arestas do EPM em G' definem um EPPM em G. ■

Formulações com programação inteira A formulação do problema do emparelhamento perfeito mínimo para $G = (V, A, c)$ é

$$\begin{aligned} \text{EPPM: } & \textbf{minimiza} && \sum_{a \in A} c_a x_a && && (1.25) \quad \texttt{\{lp:minper} \\ & \textbf{sujeito a} && \sum_{u \in N(v)} x_{uv} = 1, && \forall v \in V, \\ & && x_a \in \mathbb{B}. \end{aligned}$$

A formulação do problema do emparelhamento máximo é

$$\begin{aligned} \text{EPM: } & \textbf{maximiza} && \sum_{a \in A} c_a x_a && && (1.26) \quad \texttt{\{lp:maxmat} \\ & \textbf{sujeito a} && \sum_{u \in N(v)} x_{uv} \leq 1, && \forall v \in V, \\ & && x_a \in \mathbb{B}. \end{aligned}$$

Both linear relaxations are to $x_e \geq 0$, since the upper bound is implicit in the constraints.

{obs:tu}

Observação 1.15
A matriz de coeficientes de (1.25) e (1.26) é totalmente unimodular no caso bipartido (pelo teorema de Hoffman-Kruskal). Portanto: a solução da relaxação linear é inteira. (No caso geral isso não é verdadeiro, K_3 é um contra-exemplo, com solução ótima 3/2.). Observe que isso resolve o caso ponderado sem custo adicional. ♢

Observação 1.16
O dual da relaxação linear de (1.25) é

$$\begin{aligned} \text{CIM: } & \textbf{maximiza} && \sum_{v \in V} y_v && && (1.27) \\ & \textbf{sujeito a} && y_u + y_v \leq c_{uv}, && \forall uv \in A, \\ & && y_v \in \mathbb{R}. \end{aligned}$$

{lp:

e o dual da relaxação linear de (1.26)

lmaxmatch}

$$\begin{aligned} \text{MVC: } & \textbf{minimiza} && \sum_{v \in V} y_v && && (1.28) \\ & \textbf{sujeito a} && y_u + y_v \geq c_{uv}, && \forall uv \in A, \\ & && y_v \in \mathbb{R}_+. \end{aligned}$$

Com pesos unitários $c_{uv} = 1$ e restringindo $y_v \in \mathbb{B}$ o primeiro dual é a formulação do conjunto independente máximo e o segundo da cobertura de vértices mínima. Portanto, a observação 1.15 rende no caso não-ponderado:

{th:berge}

Teorema 1.22 (Berge, 1951)
Em grafos bi-partidos o tamanho da menor cobertura de vértices é igual ao tamanho do emparelhamento máximo.

prop:isvc}

Proposição 1.7
Um subconjunto de vértices $I \subseteq V$ de um grafo não-direcionado $G = (V, A)$ é um conjunto independente sse $V \setminus I$ é um cobertura de vértices. Em particular um conjunto independente máximo I corresponde com uma cobertura de vértices mínima $V \setminus I$.

Prova. (Exercício 1.9.) ■ ◊

1.7.1. Aplicações

Alocação de tarefas Queremos alocar n tarefas a n trabalhadores, tal que cada tarefa é executada, e cada trabalhador executa uma tarefa. O custos de execução dependem do trabalhar e da tarefa. Isso pode ser resolvido como problema de emparelhamento perfeito mínimo.

> Similar: Candidates and jobs (weighted), or persons and chairs (seats in an airplane, maximum cardinality).

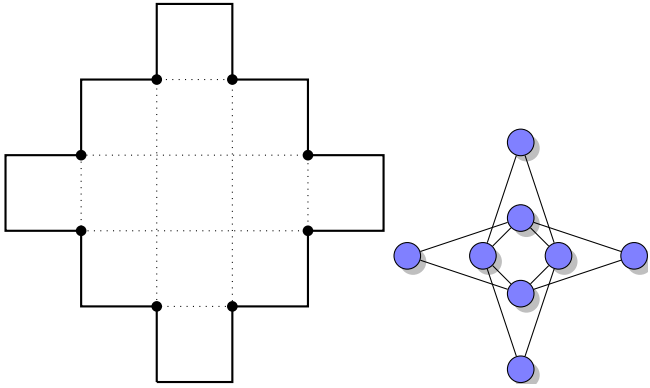


Figura 1.21.: Esquerda: Polígono ortogonal com $n = 8$ vértices de reflexo (pontos), $h = 0$ buracos. As cordas são pontilhadas. Direita: grafo de intersecção. {fig:partp

Heurística para o PCV Match twice and stitch.

Particionamento de polígonos ortogonais

Teorema 1.23 (Sack e Urrutia (2000, cap. 11, Th. 1))
Um polígono ortogonal com n vértices de reflexo (ingl. reflex vertex, i.e., com ângulo interno maior que π), h buracos (ingl. holes) pode ser minimalmente particionado em $n - l - h + 1$ retângulos. A variável l é o número máximo de cordas (diagonais) horizontais ou verticais entre vértices de reflexo sem intersecção.

O número l é o tamanho do conjunto independente máximo no grafo de intersecção das cordas: cada corda é representada por um vértice, e uma aresta representa a duas cordas com interseção. Pela proposição 1.9 podemos obter uma cobertura mínima via um emparelhamento máximo, que é o complemento de um conjunto independente máximo. Podemos achar o emparelhamento em tempo $O(n^{5/2})$ usando o algoritmo de Hopcroft-Karp, porque o grafo de intersecção é bi-partido (por quê?). {ma:pp}

Two chords intersect also, if they share only one endpoint. The maximum number of chords does not define the whole partition. It is possible that there is only one reflex vertex, or several, but without a chord between

them. This must be cut off separately.
Refs: http://code.activestate.com/recipes/123641-hopcroft-karp-bipartite-matching/ Imai and Asano, SIAM J. Computing 15(2):478-494, 1986, for an improvement.

Problemas de agendamento O problema $1 \mid p_j = p \mid \sum w_j T_j$ é resolvido por um emparelhamento perfeito entre as tarefas e os intervalos de execução $[(i-1)p, ip]$, $i \in [n]$. Podemos resolver ainda $1 \mid p_j = 1, r_j \mid \sum w_j T_j$, observando que sempre existe uma solução com as tarefas executando nos intervalos $[t_i, t_i + 1]$, $i \in [n]$, definido por

$$t_0 = -\infty; \qquad t_i = \max\{t_{i-1} + 1; r_i\}$$

e supondo que $r_1 \leq \cdots \leq r_n$.

1.7.2. Grafos bi-partidos

Na formulação como programa inteira a solução do caso bi-partido é mais fácil. Isso também é o caso para algoritmos combinatoriais, e portanto começamos estudar grafos bi-partidos.

These lectures seem to be covered best by chap. 19 and 20 from Kozen.

Redução para o problema do fluxo máximo

pviaflow1}

Teorema 1.24
Um EM em grafos bi-partidos pode ser obtido em tempo $O(mn)$.

Prova. Introduz dois vértices s, t, liga s para todos vértices em V_1, os vértices em V_1 com vértices em V_2 e os vértices em V_2 com t, com todos os pesos unitários. Aplica o algoritmo de Ford-Fulkerson para obter um fluxo máximo. O número de aumentos é limitado por n, cada busca tem complexidade $O(m)$, portanto o algoritmo de Ford-Fulkerson termina em tempo $O(mn)$. ■

Teorema 1.25
O valor do fluxo máximo é igual a cardinalidade de um emparelhamento máximo.

Prova. Dado um emparelhamento máximo $M = \{v_{11}v_{21}, \ldots, v_{1n}v_{2n}\}$, podemos construir um fluxo com arcos sv_{1i}, $v_{1i}v_{2i}$ e $v_{2i}t$ com valor $|M|$.
Dado um fluxo máximo, existe um fluxo integral equivalente (veja lema (1.20)). Na construção acima os arcos possuem fluxo 0 ou 1. Escolhe todos arcos entre

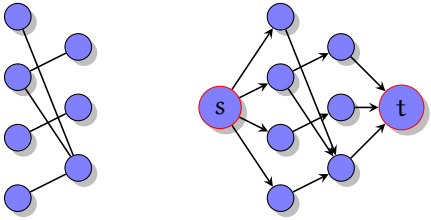


Figura 1.22.: Redução do problema de emparelhamento máximo para o problema do fluxo máximo

ion}

V_1 e V_2 com fluxo 1. Não existe vértice com grau 2, pela conservação de fluxo. Portanto, os arcos formam um emparelhamento cuja cardinalidade é o valor do fluxo. ■

Solução não-ponderada combinatorial Um caminho $P = v_1v_2v_3 \dots v_k$ é *alternante* em relação a M (ou M-alternante) se $v_iv_{i+1} \in M$ sse $v_{i+1}v_{i+2} \notin M$ para todos $1 \leq i \leq k-2$. Um vértice $v \in V$ é *livre* em relação a M se ele tem grau 0 em M, e *emparelhado* caso contrário. Uma aresta $e \in E$ é *livre* em relação a M, se $e \notin M$, e *emparelhado* caso contrário. Escrevemos $|P| = k-1$ pelo *comprimento* do caminho P.

{obs:aumen

Observação 1.17
Caso temos um caminho $P = v_1v_2v_3 \dots v_{2k}$ que é M-alternante com v_1 é v_{2k} livre, podemos obter um emparelhamento $M \setminus (P \cap M) \cup (P \setminus M)$ de tamanho $|M| + k - (k-1) = |M| + 1$. Notação: Diferença simétrica $M \oplus P = (M \setminus P) \cup (P \setminus M)$. A operação $M \oplus P$ é um *aumento* do emparelhamento M. ◊

{th:aument

Teorema 1.26 (Hopcroft e Karp (1973))
Seja M^* um emparelhamento máximo e M um emparelhamento arbitrário. O conjunto $M \oplus M^*$ contém pelo menos $k = |M^*| - |M|$ caminhos M-aumentantes disjuntos (de vértices). Um deles possui comprimento no máximo $|V|/k - 1$.
Prova. Considere os componentes de G em relação às arestas $M \oplus M^*$. Cada vértice possui no máximo grau 2. Portanto, os componentes são vértices livres, caminhos simples ou ciclos, todos disjuntos de vértices, por construção. Os caminhos e ciclos possuem alternadamente ares

$$|M^* \setminus M| = |M^*| - |M^* \cap M| = |M| - |M^* \cap M| + k = |M \setminus M^*| + k$$

e portanto $M \oplus M^*$ contém k arestas mais de M^* que de M. Isso mostra que existem pelo menos $|M^*| - |M|$ caminhos M-aumentantes, porque somente os

caminhos de comprimento ímpar possuem exatamente uma aresta mais de M^*. Pelo menos um desses caminhos tem que ter um comprimento (em arestas) menor ou igual que $|V|/k - 1$, senão cada um possui pelo menos $|V|/k + 1$ vértices, i.e. eles contém em total mais que $|V|$ vértices. ■

Here edge-disjointness implies vertex-disjointness, since two matchings give degree 2, but a joint vertex requires degree more than 2.
An aside on using $M \oplus M'$ (Kozen) versus $M \cup M'$ (Schrijver). The difference are just the joint edges $e \in M \cap M'$, but these are remain single edges in $M \cup M'$, so give rise only to irrelevant components.
Also worth noting: the theorem holds in any graph, not just bipartite ones!

Corolário 1.5 (Berge (1957))
Um emparelhamento é máximo sse não existe um caminho M-aumentante.

Rascunho de um algoritmo:

{alg:em}

Algoritmo 1.7 (Emparelhamento máximo)
Entrada Grafo não-direcionado $G = (V, A)$.

Saída Um emparelhamento máximo M.

```
1  M = ∅
2  while (existe um caminho M-aumentante P) do
3     M := M ⊕ P
4  end while
5  return M
```

Problema: como encontrar caminhos M-aumentantes eficientemente?

Observação 1.18
Um caminho M-aumentante começa num vértice livre em V_1 e termina num vértice livre em V_2. Idéia: começa uma busca por largura com todos vértices livres em V_1. Segue alternadamente arcos livres em M para encontrar vizinhos em V_2 e arcos em M, para encontrar vizinhos em V_1. A busca pára ao encontrar um vértice livre em V_2 ou após de visitar todos os vértices. Ela tem complexidade $O(m + n)$. ◊

Teorema 1.27
O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo $O(mn)$.
Prova. Última observação e o fato que o emparelhamento máximo tem tamanho $O(n)$. ■

Observação 1.19
O último teorema é o mesmo que teorema (1.24). ◊

{obs:findp

Observação 1.20
Pelo teorema (1.26) sabemos que existem vários caminhos M-alternantes disjuntos (de vértices) e nos podemos aumentar M com todos eles em paralelo. Portanto, estruturamos o algoritmo em fases: cada fase procura um conjunto de caminhos aumentantes disjuntos e aplicá-los para obter um novo emparelhamento. Observe que pelo teorema (1.26) um aumento com o maior conjunto de caminhos M-alternantes disjuntos resolve o problema imediatamente, mas não sabemos como encontrar esse conjunto de forma eficiente. Portanto, procuramos somente um conjunto maximal de caminhos M-alternantes disjuntos de menor comprimento.
Podemos encontrar um tal conjunto após uma busca em profundidade usando o DAG (grafo direcionado acíclico) definido pela busca por profundidade. (i) Escolhe um vértice livre em V_2. (ii) Segue os predecessores para encontrar um caminho aumentante. (iii) Coloca todos vértices em uma fila de deleção. (iv) Processa a fila de deleção: Até que a fila esteja vazia, remove um vértice dela. Remove todos arcos adjacentes no DAG. Caso um vértice sucessor após de remoção de um arco possui grau de entrada 0, coloca ele na fila. (v) Repete o procedimento no DAG restante, para encontrar outro caminho, até não existem mais vértices livres em V_2. A nova busca ainda possui complexidade $O(m)$. ◊

O que ganhamos com essa nova busca? Os seguintes dois lemas dão a resposta:

{lem:maisd

Lema 1.39
Em cada fase o comprimento de um caminho aumentante mínimo aumenta por pelo menos dois.

{lem:maxfa

Lema 1.40
O algoritmo termina em no máximo $\sqrt{n}$ fases.

Teorema 1.28
O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo $O(m\sqrt{n})$.
Prova. Pelas lemas 1.39 e 1.40 e a observação que toda fase pode ser completada em $O(m)$. ■

Usaremos outro lema para provar os dois lemas acima.

{lem:mais}

Lema 1.41
Seja M um emparelhamento, P um caminho M-aumentante mínimo, e Q um caminho $M \oplus P$-aumentante. Então $|Q| \geq |P| + 2|P \cap Q|$. ($P \cap Q$ denota as arestas em comum entre P e Q.)

Prova. Caso P e Q não possuem vértices em comum, Q é M-aumentante, $P \cap Q = \emptyset$ e a desigualdade é consequência da minimalidade de P.

> We have $V(P) \cap V(Q) = \emptyset$ which implies $E(P) \cap E(Q) = \emptyset$ and thus $|Q| \geq |P|$, since P is minimal. Otherwise $V(P) \cap V(Q) \neq \emptyset$ implies $E(P) \cap E(Q) \neq \emptyset$: otherwise the joint vertex has a matched edge from P and another from Q, so degree 2 in $M \oplus P \oplus Q$.

Caso contrário, P e Q possuem um vértice em comum, e logo também uma aresta, senão $M \oplus P \oplus Q$ possui um vértice de grau dois. $P \oplus Q$ consiste em dois caminhos, e eventualmente um coleção de ciclos. Os dois caminhos são M-aumentantes, pelas seguintes observações:

1. Cada caminho inicia numa ponta de Q e termina numa ponta de P. Além disso, em M as pontas de P são livres, porque P é M-aumentante; as pontas de Q também são livres em M: são livres $M \oplus P$, e logo não pertencem a P. (Nenhum outro vértice de $P \oplus Q$ é livre em relação a M: P só contém dois vértices livres e Q só contém dois vértices livres em $Q \setminus P$.)

2. Os dois caminhos são M-alternantes. Começando com um vértice livre em Q, a parte do caminho Q em $Q \setminus P$ é M-alternante, porque as arestas livres em $M \oplus P$ são exatamente as arestas livres em M. O caminho entra em P com uma aresta livre, porque todo vértice em P já está emparelhado em $M \oplus P$. A parte de P em $P \oplus Q$ tem que continuar com aresta livre em $M \oplus P$, e logo aresta emparelhada em M. Logo, temos um caminho M-alternante.

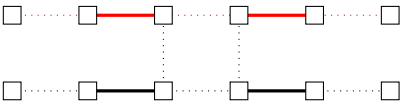
Os dois caminhos M-aumentantes em $P \oplus Q$ tem que ser maiores que $|P|$. Com isso temos $|P \oplus Q| \geq 2|P|$ e

$$|Q| = |P \oplus Q| + 2|P \cap Q| - |P| \geq |P| + 2|P \cap Q|.$$

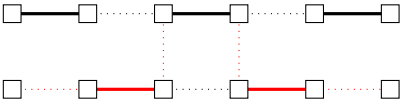
■

> Note: $|P \oplus Q| = |P| + |Q| - 2|P \cap Q|$.

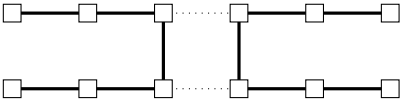
Prova. (do lema 1.39). Seja S o conjunto de caminhos M-aumentantes da fase anterior, e P um caminho aumentante. Caso P é disjunto de todos caminhos em S, ele deve ser mais comprido, porque S é um conjunto máximo de caminhos aumentantes. Caso P possui um vértice em comum com algum caminho em S, ele possui também um arco em comum (por quê?) e podemos aplicar lema 1.41. ■



(a) Grafo com emparelhamento M (em negrito) e um caminho M-aumentante P (em vermelho).



(b) O mesmo grafo com emparelhamento $M \oplus P$ (em negrito) e um caminho $M \oplus P$-aumentante Q (em vermelho).



(c) O conjunto de arestas $P \oplus Q$ (em negrito).

Figura 1.23.: Ilustração do lema 1.41.

ais}

Por quê: same reason as before: otherwise a vertex has degree 2 in the resulting graph.

Prova. (do lema 1.40). Seja M^* um emparelhamento máximo e M o emparelhamento obtido após de $\sqrt{n}/2$ fases. O comprimento de qualquer caminho M-aumentante é no mínimo $\sqrt{n}$, pelo lema 1.39. Pelo teorema 1.26 existem pelo menos $|M^*| - |M|$ caminhos M-aumentantes disjuntos de vértices. Mas então $|M^*| - |M| \leq \sqrt{n}$, porque no caso contrário eles possuem mais que n vértices em total. Como o emparelhamento cresce pelo menos um em cada fase, o algoritmo executa no máximo mais $\sqrt{n}$ fases. Portanto, o número total de fases é no máximo $3/2\sqrt{n} = O(\sqrt{n})$. ■

Proof idea: after $f(n)$ phases we have length $\geq 2f(n)$, therefore $|M^*| - |M| \leq n/2f(n)$ and we terminate in $f(n) + n/2f(n)$ phases. Function $f(n) = \sqrt{n}$ minimizes this, since

$$f(n) = n/2f(n) \iff f(n)^2 = n/2 \iff f(n) = \sqrt{n/2}.$$

(If we use this value above, the number of phases will be at most $\sqrt{2n}$, so still $O(\sqrt{n})$.)

O algoritmo de Hopcroft-Karp é o melhor algoritmo conhecido para encontrar

emparelhamentos máximos em grafos bipartidos não-ponderados esparsos[5]. Para subclasses de grafos bipartidos existem algoritmos melhores. Por exemplo, existe um algoritmo randomizado para grafos bipartidos regulares com complexidade de tempo esperado $O(n \log n)$ (Goel et al., 2010).

> Footnote on dense graphs: say $m = n^\alpha$ with $\alpha \in [1,2]$. Then $\log_n m = \alpha$ and Feder e Motwani (1995) has complexity $O(\sqrt{n}m(2-\alpha))$.

Sobre a implementação A seguir supomos que o conjunto de vértices é $V = [1,n]$ e um grafo $G = (V,A)$ bi-partido com partição $V_1 \dot\cup V_2$. Podemos representar um emparelhamento usando um vetor `mate`, que contém, para cada vértice emparelhado, o índice do vértice vizinho, e 0 caso o vértice é livre.

O núcleo de uma implementação do algoritmo de Hopcroft e Karp é descrito na observação 1.20: ele consiste numa busca por largura até encontrar um ou mais caminhos M-alternantes mínimos e depois uma fase que extrai do DAG definido pela busca um conjunto máximo de caminhos disjuntos (de vértices). A busca por largura começa com todos vértices livres em V_1. Usamos um vetor H para marcar os arcos que fazem parte do DAG definido pela busca por largura[6] e um vetor m para marcar os vértices visitados.

```
1  search_paths(M) :=
2    for all v ∈ V do m_v := false
3
4    U_1 := {v ∈ V_1 | v livre}
5    for all u ∈ U_1 do d_u := 0
6
7    do
8      { determina vizinhos em U_2 via arestas livres}
9      U_2 := ∅
10     for all u ∈ U_1 do
11       m_u := true
12       for all uv ∈ A, uv ∉ M do
13         if not m_v then
14           d_v := d_u + 1
15           U_2 := U_2 ∪ v
16         end if
```

[5] Feder e Motwani (1991) e Feder e Motwani (1995) propuseram um algoritmo em $O(\sqrt{n}m(2-\log_n m))$ que é melhor em grafos densos.

[6] H, porque o DAG se chama *árvore Húngara* na literatura.

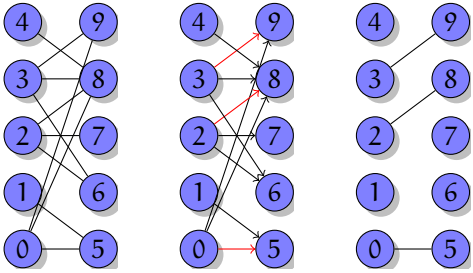
```
17        end for
18      end for
19
20      { determina vizinhos em U1 via arestas emparelhadas }
21      found := false        { pelo menos um caminho encontrado? }
22      U1 := ∅
23      for all u ∈ U2 do
24        mu := true
25        if (u livre) then
26          found := true
27        else
28          v := mate[u]
29          if not mv then
30            dv := du + 1
31            U1 := U1 ∪ v
32          end if
33        end for
34      end for
35    while (not found)
36  end
```

Após da busca, podemos extrair um conjunto máximo de caminhos M-alternantes mínimos disjuntos. Enquanto existe um vértice livre em V_2, nos extraimos um caminho alternante que termina em v como segue:

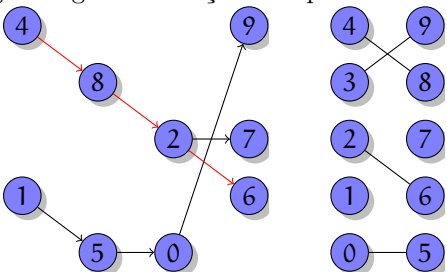
```
1   extract_paths() :=
2     while existe vértice v livre em V2 do
3       aplica um busca em profundidade a partir de v em H
4         (procurando um vértice livre em V1)
5       remove todos vértices visitados durante a busca
6       caso um caminho alternante P foi encontrado: M := M ⊕ P
7     end while
8   end
```

Exemplo 1.7
Segue um exemplo da aplicação do algoritmo de Hopcroft-Karp.

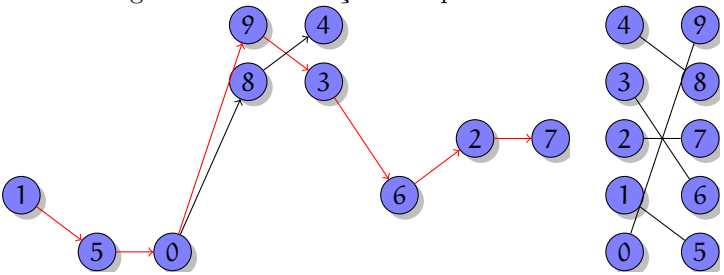
Grafo original, árvore Húngara primeira iteração e emparelhamento resultante:



Árvore Húngara segunda iteração e emparelhamento resultante:



Árvore Húngara terceira iteração e emparelhamento resultante:



◊

Emparelhamentos, coberturas e conjuntos independentes

atchcover}

Proposição 1.8

Seja $G = (S \dot\cup T, A)$ um grafo bipartido e $M \subseteq A$ um emparelhamento em G. Seja R o conjunto de todos vértices livres em S e todos vértices alcançáveis por uma busca na árvore Húngara (i.e. via arestas livres de S para T e arestas emparelhadas de T para S). Então $(S \setminus R) \cup (T \cap R)$ é uma cobertura de vértices em G.

Prova. Seja $uv \in A$ uma aresta não coberta. Logo $u \in S \setminus (S \setminus R) = S \cap R$ e $v \in T \setminus (T \cap R) = T \setminus R$. Caso $uv \notin M$, uv é parte da árvore Húngara é

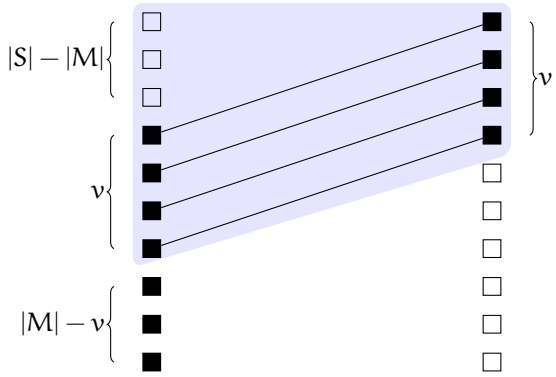


Figura 1.24.: Ilustração da prova da proposição 1.9. {fig:emcm}

$\nu \in R$, uma contradição. Mas caso $u\nu \in M$, νu é parte da árvore Húngara e ν precede u, logo $\nu \in R$, novamente uma contradição. ■

A próxima proposição mostra que no caso de um emparelhamento máximo obtemos uma cobertura mínima. {prop:emcm

Proposição 1.9
Seja $G = (S \dot\cup T, A)$. Caso M é um emparelhamento máximo o conjunto $(S \setminus R) \cup (T \cap R)$ é uma cobertura mínima.

Prova. O tamanho de qualquer emparelhamento M é um limite inferior para o tamanho de qualquer cobertura, porque uma cobertura tem que conter pelo menos um vértice da cada aresta emparelhada. Logo é suficiente demonstrar que $(S \setminus R) \cup (T \cap R)| = |M|$.
Temos $(S \setminus R) \cup (T \cap R)| = |S \setminus R| + |T \cap R|$ porque S e T são disjuntos. Vamos demonstrar que $|T \cap R| = \nu$ implica $|S \setminus R| = |M| - \nu$.
Supõe $|T \cap R| = \nu$. Como M é máximo não existe caminho M-aumentante, e logo $T \cap R$ contém somente vértices emparelhados. Por isso o número de vértices emparelhados em $S \cap R$ também é ν. Além disso $S \cap R$ contém todos $|S| - |M|$ vértices livres em S. Logo $|S \setminus R| = |S| - (|S| - |M|) - \nu = |M| - \nu$. ■

Observação 1.21
O complemento $V \setminus C$ de uma cobertura C é um conjunto independente (por quê?). Logo um emparelhamento M que define um conjunto R de acordo com a proposição (1.8) corresponde com um conjunto independente $(S \cap R) \cup (T \setminus R)$, e caso M é máximo, o conjunto independente também. ◊

This can be extended to a minimal edge cover (of vertices): take a maximum matching, and then cover all uncovered vertices by some incident edge. This is a minimal edge cover, because no further edge can cover two vertices. In other words, in a minimal edge cover, each vertex has an adjacent cover edge; now let the matching be those edges that cover both of its endpoints. Then the total cost is $n - |M|$. Since n is constant minimizing $-|M|$ amount to maximizing $|M|$.
This can be further extended to the weighted version. Here each vertex $v \in V$ can be seen as covered by the lighest incident edge of cost c_v. Now consider a matching of edges that cover both endpoints. An edge $e = uv \in E$ in it has cost $c_e - c_u - c_v$. So we have total cost $\sum_{v \in V} c_v + \sum_{e \in M} c_e - c_u - c_v$ and since the first term is a constant, a minimum weight edge cover corresponds to a maximum weight matching with cost $c_u + c_v - c_e$.

Solução ponderada em grafos bi-partidos Dado um grafo $G = (S \dot\cup T, A)$ bipartido com pesos $c : A \to \mathbb{Q}_+$ queremos achar um emparelhamento de maior peso. Escrevemos $V = S \cup T$ para o conjunto de todos vértices em G.

{obs:wmp}

Observação 1.22
O caso ponderado pode ser restrito para emparelhamentos perfeitos: caso S e T possuem cardinalidade diferente, podemos adicionar vértices, e depois completar todo grafo com arestas de custo 0. O problema de encontrar um emparelhamento perfeito máximo (ou mínimo) em grafos ponderados é conhecido pelo nome "problema de alocação" (ingl. assignment problem). ◊

Observação 1.23
A redução do teorema 1.24 para um problema de fluxo máximo não se aplica no caso ponderado. Mas, com a simplificação da observação 1.22, podemos reduzir o problema no caso ponderado para um problema de fluxo de menor custo: a capacidade de todas arestas é 1, e o custo de transportação são os pesos das arestas. Como o emparelhamento é perfeito, procuramos um fluxo de valor $|V|/2$, de menor custo. ◊

O dual do problema 1.28 é a motivação para

Definição 1.6
Um *rotulamento* é uma atribuição $y : V \to \mathbb{R}_+$. Ele é *viável* caso $y_u + y_v \geq c_a$ para todas arestas $a = \{u, v\}$. (Um rotulamento viável é uma c-*cobertura de vértices*.) Uma aresta é *apertada* (ingl. tight) caso $y_u + y_v = c_a$. O subgrafo de arestas apertadas é $G_y = (V, A', c)$ com $A' = \{a \in A \mid a \text{ apertada em } y\}$.

Pelo teorema forte de dualidade e o fato que a relaxação linear dos sistemas acima possui uma solução integral (ver observação 1.15) temos

Teorema 1.29 (Egerváry (1931))
Para um grafo bi-partido $G = (S \dot\cup T, A, c)$ com pesos não-negativos $c : A \to \mathbb{Q}_+$ nas arestas, o maior peso de um emparelhamento perfeito é igual ao peso da menor c-cobertura de vértices.

O método húngaro Aplicando um caminho M-aumentante $P = (v_1 v_2 \dots v_{2n+1})$ produz um emparelhamento de peso $c(M) + \sum_{i \text{ ímpar}} c_{v_i v_{i+1}} - \sum_{i \text{ par}} c_{v_i v_{i+1}}$. Isso motiva a definição de uma árvore Húngara ponderada. Para um emparelhamento M, seja H_M o grafo direcionado com as arestas $e \in M$ orientadas de T para S com peso $l_e := w_e$, e com as restantes arestas $a \in A \setminus M$ orientadas de S para T com peso $l_a := -w_a$. Com isso a aplicação do caminho M-aumentante P produz um emparelhamento de peso $c(M) - l(P)$ em que $l(P) = \sum_{1 \le i \le 2n} l_{v_i v_{i+1}}$ é o comprimento do caminho P.
Com isso podemos modificar o algoritmo para emparelhamentos máximos para

{alg:epm}

Algoritmo 1.8 (Emparelhamento de peso máximo)
Entrada Um grafo não-direcionado ponderado $G = (V, E, c)$.

Saída Um emparelhamento de maior peso $c(M)$.

```
1  M = ∅
2  while (existe um caminho M-aumentante P) do
3    encontra o caminho M-aumentante mínimo P em H_M
4    caso l(P) ≥ 0: return M;
5    M := M ⊕ P
6  end while
7  return M
```

Chamaremos um emparelhamento M *extremo* caso ele possui o maior peso entre todos emparelhamentos de tamanho $|M|$.

Observação 1.24
O grafo H_M de um emparelhamento extremo M não possui ciclo (par) negativo. Isso seria uma contradição com a maximalidade de M. Portanto podemos encontrar o caminho mínimo no passo 3 do algoritmo usando o algoritmo de Bellman-Ford em tempo $O(mn)$. Com isso a complexidade do algoritmo é $O(mn^2)$. ◊

Observação 1.25
Lembrando Bellman-Ford: Seja $d_k(t)$ a distância mínima entre s e t com um caminho usando no máximo k arcos ou ∞ caso tal caminho não existe. Temos

$$d_{k+1}(t) = \min\{d_k(t), \min_{(u,t)\in A} d_k(u) + l(u,t)\} \forall t \in V,$$

com $d_0(t) = 0$ caso t é um vértice livre em S e $d_0(t) = \infty$ caso contrário. (O algoritmo se aplica igualmente para as distâncias de um conjunto de vértices, como o conjunto de vértices livres em S.) A atualização de k para $k+1$ é possível em $O(m)$ e como $k < n$ o algoritmo possui complexidade $O(nm)$. $\Diamond$

Teorema 1.30
Cada emparelhamento encontrado no Algoritmo 1.8 é extremo.

Prova. Por indução sobre $|M|$. Para $M = \emptyset$ o teorema é correto. Seja M um emparelhamento extremo, P o caminho aumentante encontrado pelo algoritmo 1.8 e N um emparelhamento de tamanho $|M|+1$ arbitrário. Como $|N| > |M|$, pelo teorema (1.26) $M \oplus N$ contém um caminho M-aumentante Q. Sabemos $l(Q) \geq l(P)$ pela minimalidade de P. $N \oplus Q$ é um emparelhamento de cardinalidade $|M|$ (Q é um caminho com arestas em N e M com uma aresta em N a mais), logo $c(N \oplus Q) \leq c(M)$. Com isso temos

$$c(N) = c(N \oplus Q) - l(Q) \leq c(M) - l(P) = c(M \oplus P)$$

(observe que o comprimento $l(Q)$ é definido no emparelhamento M). ■

Proposição 1.10
Caso não existe caminho M-aumentante com comprimento negativo no Algoritmo 1.8, M é máximo.

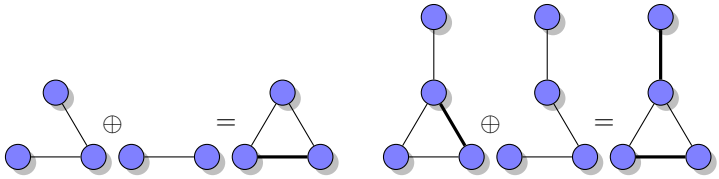
Prova. Supõe que existe um emparelhamento N com $c(N) > c(M)$. Logo $|N| > |M|$ porque M é possui o maior peso entre todos emparelhamentos de cardinalidade no máximo $|M|$. Pelo teorema de Hopcroft-Karp, existem $|N| - |M|$ caminhos M-aumentantes disjuntos de vértices em $N \oplus M$. Nenhum deles tem comprimento negativo, pelo critério de parada do algoritmo. Portanto $c(N) \leq c(M)$, uma contradição. ■

Fato 1.1
É possível encontrar o caminho mínimo no passo 3 em tempo $O(m + n \log n)$ usando uma transformação para distâncias positivas e aplicando o algoritmo de Dijkstra. Com isso um algoritmo em tempo $O(n(m + n \log n))$ é possível.

1.7.3. Emparelhamentos em grafos não-bipartidos

O teorema de Berge 1.22 (ou e de Hopcroft & Karp 1.26) vale em qualquer grafo.

Exemplo 1.8 (Caminhos M-aumentantes em grafos não-bipartidos)
Consequência: dado um caminho M-aumentante, a sua aplicação produz emparelhamentos maiores.



◊

Portanto, o problema central em grafos gerais ainda é

{prob:augm

Problema 1.1 (Encontra um caminho M-aumentante)
Dado um emparelhamento M, retorne um caminho M-aumentante, caso existir.

Dado uma solução em tempo $T(n)$, o algoritmo canônico (inicia com $M = \emptyset$; repetidamente resolve Problema 1.1; caso tem caminho M-aumentante P, $M := M \oplus P$ e repete; senão: para) termina em no máximo $\lfloor n/2 \rfloor = O(n)$ iterações em tempo $O(nT(n))$.

O caso não-ponderado

We start with a classification, following Schrijver 24.1.
If a graph has an odd component (i.e. a component with an odd number of vertices), then at least one is not matched. This gives the simple bound $\nu(G) \leq 1/2(|V| - o(G))$ on the number of edges $\nu(G)$ in a matching, where $o(G)$ is the number of odd components of graph G. Now take some subset $U \subseteq V$. If we suppose U is fully matched, we can bound

$$\begin{aligned} \nu(G) &\leq |U| + \nu(G - U) \\ &\leq |U| + 1/2(|V \setminus U| - o(G - U)) \\ & 1/2(|V| + |U| - o(G - U)) \qquad (*) \end{aligned}$$

{ub:U}

Now the Tutte-Berge theorem states, that the smallest of these bounds is exact.

{th:tutte-be

Teorema 1.31 (Tutte-Berge)

$$\nu(G) = \min_{U \subseteq V} 1/2(|V| + |U| - o(G - U)).$$

Prova. Obviously "$\leq$" follows from (*).
For "$\geq$" we use induction on $|V|$; the base $V = \emptyset$ is trivial. We can assume G to be connected, otherwise we apply the induction hypothesis to the components. We consider two cases: a) some vertex v is in all maximum matchings. b) none is.
Consider case a). Then $\nu(G - v) = \nu(G) - 1$, and we can apply the induction hypothesis to $G - v$ to obtain an U' such that

$$\nu(G - v) = 1/2(|V \setminus \{v\}| + |U'| - o(G - v - U')).$$

But then $U = U' \cup \{v\}$ witnesses equality, viz.

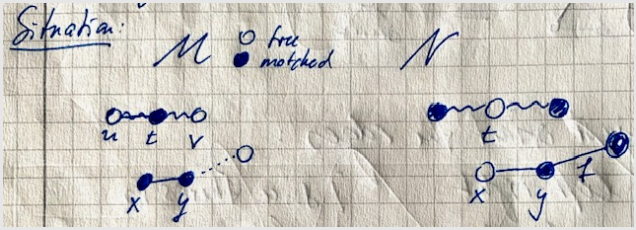
$$\begin{aligned} \nu(G) &= \nu(G - v) + 1 \\ &\quad 1/2(|V \setminus \{v\}| + |U'| - o(G - v - U')) + 1 \\ &\quad 1/2(|V| + |U| - o(G - U)) + 1. \end{aligned}$$

Now consider case b). Then $\nu(G) < |V|/2$, and we claim that there's a matching of size $(|V| - 1)/2$. Given that, have for $U = \emptyset$, and since $o(G) = 1$

$$\begin{aligned} \nu(G) &= (|V| - 1)/2 \\ &= 1/2(|V| + |U| - o(G - U)) + 1. \\ &\geq \min_{U \subseteq V} 1/2(|V| + |U| - o(G - U)), \end{aligned}$$

and we're done.
Now for the claim: assume that in all maximal matchings M at least two vertices $u = u(M)$ and $v = v(M)$ are free; select a matching where u, v have shortest distance $\mathrm{dist}(u, v)$. We can't have $\mathrm{dist}(u, v) = 1$, since otherwise we could add uv to M. Thus there's a vertex t, different from u and v on the shortest uv-path. By assumption there's a matching N where t is free. Choose such a matching of maximal overlap $|M \cap N|$. Note that u and v are matched in N, otherwise we have free vertices u, t with $\mathrm{dist}(u, t) < \mathrm{dist}(u, v)$ or free vertices t, v with $\mathrm{dist}(t, v) < \mathrm{dist}(u, v)$

in contradiction with the selection of u and v. Now since $|M| = |N|$ there must be some $x \neq t$ matched in M but not in N. Let $e = xy$ be the corresponding edge. Vertex y is matched in N, since we otherwise could add xy to N, and some edge $f \in N$ contains y. But now $N \setminus \{f\} \cup \{e\}$ has a larger overlap with M, contradicting the choice of N.



■

This leads finally to

{th:tutte}

Teorema 1.32 (Tutte's 1-factor theorem)
Graph G has a perfect matching iff $G - U$ has $o(G - U) \leq |U|$ for all $U \subseteq V$.

Prova. If G has a perfect matching then $\nu(G) = |V|/2$. So

$$|V|/2 = \min_{U \subseteq V} 1/2(|V| + |U| - o(G - U))$$

and therefore, for all $U \subseteq V$

$$|V|/2 \leq 1/2(|V| + |U| - o(G - U)) \iff \qquad o(G - U) \leq |U|.$$

On the other hand, if for all $U \subseteq V$ we have $o(G - U) \leq |U|$, then $1/2(|V|+|U| \vdash o(G-U) \geq 1/2|V|$, and thus $\nu(G) = |V|/2$ by theorem 1.31.
■

Primeiramente vamos entender porque a abordagem utilizada em grafos bipartidos $G = (S \dot{\cup} T, E)$ falha. Sejam X os vértices livres em G. Em grafos bipartidos encontramos um caminho M-aumentante por uma busca em largura:

{bcma}

Algoritmo 1.9 (Busca caminho M-aumentante)
Inicia em $S_0 = S \cap X$. Dado S_i sejam T_i os vértices ainda não explorados alcançáveis por S_i via arestas livres. Caso T_i contém

um vértice livre, termina, senão sejam S_{i+1} os vértices ainda não explorados alcançáveis por T_i via arestas emparelhadas. Repete.

prop:bcma}

Proposição 1.11
Algoritmo 1.9 sempre encontra pelo menos um caminho mais curto M-aumentante em grafos bipartidos.

Prova. Para todo caminho M-aumentante mais curto $P = (v_0, v_1, \ldots, v_t)$, vértice v_i é encontrado na iteração i. Pela existência do caminho P, é claro que vértice v_i é descoberto em no máximo i iterações. Agora assume v_i é o vértice de menor índice descoberto numa iteração $j < i$, por um caminho alternante $Q = (u_0, u_1, \ldots, u_j = v_i)$ iniciando em u_0 livre. Temos os seguintes casos:

a) j é par, e i é par. Logo $u_{j-1}v_i \in M$, e $v_{i-1}v_i \in M$, e por isso $u_{j-1} = v_{i-1}$ em contradição com a minimalidade de i.

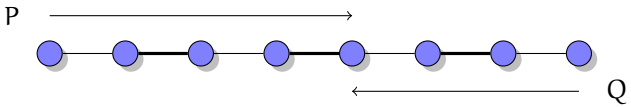
b) j é ímpar, i é ímpar. Logo $u_{j-1}v_i \notin M$, $v_iv_{i+1} \in M$ e Q junto com o caminho $(v_i, v_{i+1}, \ldots, v_t)$ é um caminho M-aumentante de comprimento $j + (t - i) < t$, em contradição com a minimalidade de P.

c) j é par, e i é ímpar. Logo $v_i \in S_{j/2}$ e $v_i \in T_{\lfloor i/2 \rfloor}$, em contradição com G sendo bipartido.

d) j é ímpar, i é par. Similar ao caso c) temos $v_i \in T_{\lfloor j/2 \rfloor}$ e $v_i \in S_{i/2}$, uma contradição.

■

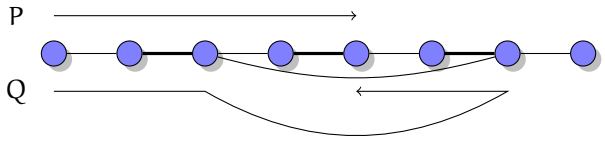
Num grafo geral não temos a partição em S e T. Uma possível alternativa é iniciar a busca em $R_0 = X$ e aplicar a mesma busca alternante para descobrir uma sequência de conjuntos R_i. Mas mesmo em grafos bipartidos, Algoritmo 1.9 então falha: em



os caminhos alternantes P e Q se encontram. Nota que isso corresponde com o caso d) da Proposição 1.11, mas não é mais uma contradição, porque os conjuntos R_i contém vértices de S e T.
Esse problema pode ser resolvido por i) modificar o Algoritmo 1.9 para combinar caminhos encontrados em buscas iniciados em vértices livres diferentes, ou, mais simples, mas menos eficiente, ii) buscar a partir de cada vértice $x \in X$ separadamente.

I won't go into details here, but i) could be done along the lines of Korte e Vygen (2008, p. 10.27). We keep an alternating forest, each tree starting from some $x \in X$. Then, an outer (even) vertex can't have a free neighbor in the same tree, since all free X have their own trees. So if we meet an outer vertex in another tree, we have found an augmenting path. Otherwise the vertex may be not in the tree yet: so it is matched, and we add the vertex and its matched neighbor. What can't happen in the bipartite is the case of a blossom.
For ii) we could check all free vertices with an overhead of n times the search, but this still won't work as the second example shows.

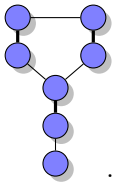
Por ser mais simples considera a solução ii): mesmo procurando a partir de um único vértice $x \in X$ falha em grafos gerais. Por exemplo:



Note que isso corresponde com o caso c) da Proposição 1.11 é não é mais uma contradição, porque em grafos gerais podemos ter laços ímpares.

Sublety: case c) has been excluded by an S, T argument. But if we assume P and Q share the initial vertex, we would find and odd loop, but thus can't happen in bipartite graphs. That's the deeper reason there's no contradiction any more.

O exemplo acima sugere que ciclos ímpares formam o núcleo do Problema 1.1. A árvore de busca do exemplo anterior pode ser visualizado como



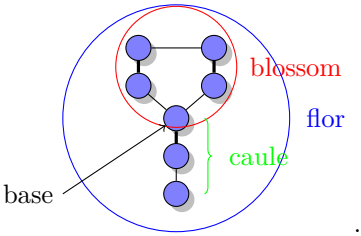
.

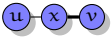
Isso é o motivo para:

{def:flowe

Definição 1.7 (Flor)
Seja $P = (v_0, v_1, \ldots, v_t)$ uma caminhada M-alternante. Caso (i) $v_0 \in X$, (ii) todos vértices $v_0, \ldots, v_{t-1}$ são distintos, (iii) t é ímpar, e (iv) existe um $i < t$,

i par, tal que $v_i = v_t$, P é chamado uma *flor*, com *caule* $(v_0, \ldots, v_i)$, *base* v_i, e *blossom* $B = (v_i, v_{i+1}, \ldots, v_t)$.



Caminhadas M-alternantes. Como encontrar *caminhos* M-alternantes falha, uma outra ideia, que vamos discutir agora, é buscar *caminhadas* M-alternantes. Para conseguir isso, vamos introduzir um grafo direcionado auxiliar $D = (V, A)$, onde $A = \{uv \mid ux \in E, xv \in M \text{ para um } x \in V\}$. A ideia é substituir  por  .
Essa construção tem a seguinte característica

op:xxwalk}

Proposição 1.12
Sejam $N(X)$ todos vértices vizinhos de vértices livres X. Então D possui um caminho X–$N(X)$ sse G possui uma caminhada X–X.

Prova. "⇒": é suficiente expandir os arcos e adicionar uma aresta final para um vértice livre.
"⇐": dado $W = (v_0, \ldots, v_t)$ remove o vértice livre v_t para obter uma caminhada terminando em $N(X)$. Podemos assumir que v_{t-1} é o único vértice em $N(X)$, senão um prefixo de W serve. Contrai arestas $v_{2i}v_{2i+1}$ para arcos e remove eventuais ciclos para obter um caminho. Como o vértice inicial é livre e o vértice final v_{t-1} não tem sucessor, ambos não fazem parte de um ciclo. Logo o caminho resultante é X–$N(X)$. ■
Isso nos permite, em tempo $O(m)$ usar uma busca por profundidade em D iniciando em X e terminando em algum vértice em $N(X)$ para encontrar uma caminhada M-alternante X–X. Porém o seguinte exemplo mostra que as flores ainda são uma fonte de problemas para caminhadas M-alternantes.

matching1}

Exemplo 1.9
Considere o exemplo da Figure 1.25. O caminho $v_1v_3v_5v_7v_9$ corresponde com o caminho M-aumentante $v_1v_2v_3v_4v_5v_6v_7v_8v_9v_a$. Mas caminho $v_1v_8c_6v_5v_7v_9$ que corresponde com o caminhada $v_1v_9v_8v_7v_6v_4v_5v_6v_7v_8v_9v_a$ que não é M-aumentante, mesmo sendo M-alternante entre dois vértices livres. O problema novamente é o laço ímpar $v_6v_4v_5v_6$. ◊

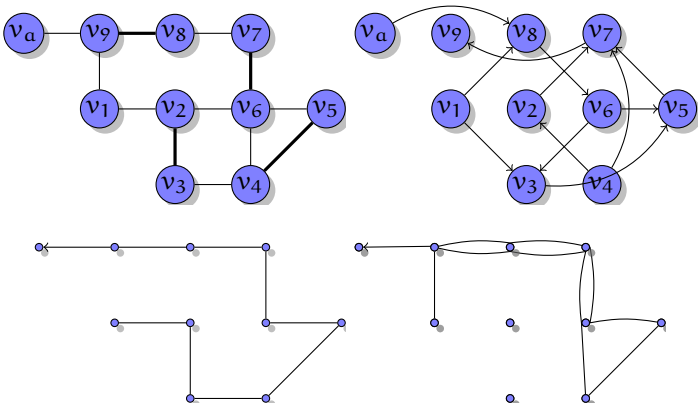
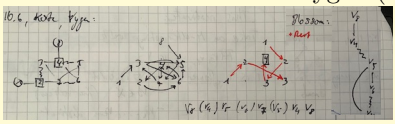


Figura 1.25.: Grafo com emparelhamento, grafo auxiliar e duas caminhadas M-alternantes.

ng1}

Another example I have used: from Korte e Vygen (2008, p. 10.6).



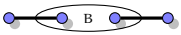
Nota que no Exemplo 1.9 o prefixo $v_1v_9v_8v_7v_6v_4v_5v_6$ da segunda caminhada é uma flor. Isso de fato sempre é o caso:

{prop:path

Proposição 1.13
Seja $P = (v_0, v_1, \ldots, v_t)$ uma caminhada M-alternante X–X mais curta. Então ou (i) P é um caminho M-aumentante, ou (ii) o prefixo $(v_0, v_1, \ldots, v_j)$ para algum $j \leq t$ é uma flor.

Prova. Assume que P não é um caminho. Seleciona $i < j$ tal que $v_i = v_j$ e j mínimo. Então todos vértices $v_0, \ldots, v_{j-1}$ são distintos. A diferença $j - i$ não pode ser par, senão podemos remover $(v_i, \ldots, v_j)$ para obter a caminhada $(v_0, \ldots, v_i)$ M-alternante X–X mais curta que P, em contradição com a minimalidade de P. Ainda, caso i é ímpar e j é par, temos $v_iv_{i+1} \in M$, e $v_{j-1}v_j \in M$ e como $v_i = v_j$ também $v_{i+1} = v_{j-1}$, em contradição com a minimalidade de j. Logo i é par, j é ímpar, e $(v_0, \ldots, v_j)$ satisfaz todos critérios da Definição 1.7 é por isso é uma flor. ■

Lidar com flores. O problema central então é como lidar com flores. Esse problema tem uma solução simples: ao encontrar uma flor, contrai a sua

blossom B. Vamos escrever G/B para o grafo resultante, e assumir que ele tem vértices $G \setminus B \cup \{B\}$ (ou seja o vértice B representa a blossom contraída). Ao contrair, vamos descartar laços. Ainda dado um emparelhamento M, M/B é o emparelhamento após a contração. (Nota que somente caso $|M \cap \delta(B)| \leq 1$ onde $\delta(B) = \{uv \mid u \in B, v \notin B\}$, M/B é um emparelhamento; por exemplo  produz  que não é.)

O seguinte teorema nos garante a corretude dessa estratégia.

th:shrink}

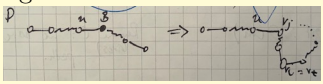
Teorema 1.33

M é um emparelhamento máximo em G sse M/B é um emparelhamento máximo em G/B.

Prova. Seja $B = (v_i, v_{i+1}, \ldots, v_t)$.

"⇒": Assume M/B não é máximo, e seja P um caminho M/B-aumentante. Vamos mostrar que então existe um caminho M-aumentante, logo M também não é máximo. Caso $B \notin P$, P já é M-aumentante. Caso contrário, seja uB a aresta em P que entra em B. Podemos assumir que uB é livre em M/B (senão inverte P). Seja uv_j, $i \leq j \leq t$, a aresta correspondente em G. Caso j é ímpar, podemos expandir B para $v_j, v_{j+1}, \ldots, v_t$ para obter um caminho M-aumentante (nota que $v_j v_{j+1} \in M$) em G. Similarmente, caso j é par, podemos expandir B para $v_j, v_{j-1}, \ldots, v_i$ para obter um caminho M-aumentante.

Problem: clearly, by construction, these paths are M-alternating; but why need they be augmenting? What about the continuation of P beyond B?



"⇐": Assume M não é máximo. Para caule Q, $M \oplus Q$ é um emparelhamento da mesma cardinalidade porque v_i tem índice par, pela Definição 1.7. Então podemos supor que $i = 0$; nota que isso torna v_i livre em M, e logo B é libre em M/B. Dado um caminho M-aumentante $P = (u_0, \ldots, u_s)$, então vamos construir um caminho M/B-aumentante em G/B, mostrando que M/B não é máximo. Caso $P \cap B = \emptyset$, P já é um caminho M/B-aumentante. Caso contrário, podemos assumir $u_0 \notin B$ (senão inverte P). Seja u_j, $j > 0$ o primeiro vértice em P em B. O caminho $(u_0, \ldots, u_j)$ é M/B-alternante, e como B é libre em G/B, também aumentante. ■

Combinando as peças. Com isso podemos resolver o Problema 1.1, como segue.

Algoritmo 1.10 (Busca caminho M-aumentante)

1) Encontra um caminho P M-alternante X–X mais curto. Caso não tenha: para, não existe caminho M-aumentante. (Proposição 1.12).
2) Pela Proposição 1.13 ou
   a) P é um caminho M-aumentante: retorna P; ou
   b) um prefixo de P é uma flor com blossom B: recursivamente encontra um caminho P' M/B-aumentante em G/B. Depois expande P' para um caminho M-aumentante P'' em G, de acordo com Teorema 1.33. retorna P''.

A corretude do algoritmo segue das proposições e teoremas mencionadas. A complexidade de encontrar o caminho P no passo 1, bem como a complexidade da contração para G/B no passo 2c é $O(m)$. Por isso, todas chamadas recursivas não custam mais que $O(nm)$, porque em cada recursão temos pelo menos um vértice a menos. Logo, o algoritmo canônico termina em tempo $O(n^2m)$.

This algorithm is not efficient, but can be made to run in $O(n^3)$ using better data structure. In particular, we don't want to shrink the whole graph all the time.
On the weighted case: the problems presented here and their solutions are valid, also in the weighted case. But: new problems arise. To solve them, we sometimes must expand blossoms again, to find augmenting paths. This puts even more pressure on efficient data structures. A detailed discussion seems at the moment beyond the scope of this lecture.

1.7.4. Tópicos avançados

Sketch of the rest:

- Extract Egerváry's algorithm (Frank, 2004) from the constructive proof above, and discuss its complexity (should be $O(|V|^4C)$?; its not polynomial, and C is some input-dependent constant).
- Discuss Kuhn's idea to increase the duals while searching for an augmenting path in the Hungarian tree. Show his Hungarian algo-

rithm (Kuhn 1955). Discuss its complexity (what is it? $O(|E||V|^2)$?)

- Discuss Munkres improvement (now $O(|V|^3)$?). Or was it Karp? Or Tomizawa? Or Dinits?
- In the historical part, discuss Jacobi's pre 1890 solution: http://www.lix.polytechnique.fr/~ollivier/JACOBI/jacobiEngl.htm, and the contribution of Monge.
- Show (below) how to derive the algorithm from primal-dual theory.

Aplicação do método primal-dual Nessa seção vamos explorar como uma aplicação do método primal-dual resulta em um algoritmos combinatorial para o problema. O problema primal restrito às arestas indicadas pelo teorema de folgas complementares é

$$\begin{aligned}
\textbf{minimiza} \quad & \sum_{v\in V} x_v^a && && \text{(RP)} \quad \texttt{\{mm:rp}\\
\textbf{sujeito a} \quad & \sum_{u\in N(v)} x_{uv} + x_v^a = 1 && \forall v \in V \\
& x_e = 0 && \forall e \in E(G)\setminus E(G_y) \\
& x_e \geq 0 && \forall e \in E(G_y) \\
& x_v^a \geq 0 && \forall v \in V
\end{aligned}$$

com variáveis auxiliares x_v^a para cada vértice. Para simplificar observe que a função objetivo pode ser escrito como

$$\sum_{v\in V} x_v^a = \sum_{v\in V} 1 - \sum_{u\in N(v)} x_{uv} = |V| - 2\sum_{e\in E} x_e$$

e portanto o problema é equivalente com

$$\begin{aligned}
\textbf{maximiza} \quad & \sum_{e\in E} x_e && && \text{(RP')} \quad \texttt{\{mm:rp}\\
\textbf{sujeito a} \quad & \sum_{u\in N(v)} x_{uv} \leq 1 && \forall v \in V \\
& x_e = 0 && \forall e \in E(G)\setminus E(G_y) \\
& x_e \geq 0 && \forall e \in E(G_y).
\end{aligned}$$

Isso é o problema de encontrar um emparelhamento máximo no grafo G_y das arestas apertadas! O dual do problema restrito original é

$$\begin{aligned} \textbf{maximiza} \quad & \sum_{v \in V} y'_v && && \text{(DRP)} \quad \texttt{\{mm:drp\}} \\ \textbf{sujeito a} \quad & y'_u + y'_v \leq 0 && \forall e \in E(G_y) \\ & y'_v \leq 1 && \forall v \in V \\ & y'_v \leqslant 0 && \forall v \in V. \end{aligned}$$

Observe que caso (RP) possui uma solução com valor 0 achamos um emparelhamento perfeito nas arestas apertadas. Esse emparelhamento tem que ser ótimo pelo teorema forte de dualidade. Caso contrário podemos usar a solução de (DRP) para melhorar a solução dual do primal.
What are we going to do from here?

- We apply algorithm 1.7 to (RP'), selecting always one path by BFS. This is conceptually the same as applying the Edmonds-Karp maximum flow.
- If we find a perfect matching, we are done.
- Otherwise: we have a s-t cut in the graph. This is equivalent to the nodes reached in the BFS for an augmenting path. Let the reachable set be $V^* = S^* \dot\cup T^*$.
- We are going to extract a feasible dual for (DRP) from this. The claim is that

$$y_v = \begin{cases} 1 & v \in S^* \\ -1 & v \in T^* \\ -1 & v \in S \setminus S^* \\ 1 & v \in T \setminus T^* \end{cases}$$

does the job. TBD: Check and prove this. Does not work like this: its no solution for the dual of (RP'), but for the dual of (RP).
Prova. (Sketch.) We first show, that we have a feasible solution. Consider some edge $e = (u, v) \in E(G_y)$. The only way (*) could not be satisfied is when $u \in S^*$, but $v \notin T^*$. Therefore e must be part of the matching, and u is not free, otherwise v would be reachable. Then, the only way to reach u is e, but this is impossible.

Tabela 1.4.: Resumo emparelhamentos. Aqui $C = \max_{a \in A} |c_a|$.

	Cardinalidade	Ponderado
Bi-partido	$O(n\sqrt{mn/\log n})$ (Alt et al., 1991) $O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ (Feder e Motwani, 1995)	$O(nm + n^2 \log n)$ (Kuhn, 1955; Munkres, 1957)
Geral	$O(m\sqrt{n})$ (Micali e Vazirani, 1980)	$O(n^3)$ (Edmonds, 1965)
	$O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ (Goldberg e Karzanov, 2004; Fremuth-Paeger e Jungnickel, 2003)	$O(mn + n^2 \log n)$ (H. N. Gabow, 1990)
		$O(m\sqrt{n} \log nC)$ (Duan et al., 2018)

{tab:emp}

Let the value of the maximum matching be m. All matched vertices have opposite duals -1 and 1, and therefore contribute in total 0 to the value of the dual. TBD: now we have to argue that the values of the remaining vertices sum up to m. We have two types of remaining vertices: those free in S and those free in T. We can set them all to 1?! Hmm, something's inverted. ■

- Next, we want to use this dual to improve the current dual of the unrestricted problem.
- Finally, we'd like to show the connection to Kuhn's Hungarian algorithm.

1.7.5. Notas

Duan et al. (2011) apresentam técnicas de aproximação para emparelhamentos.

1.7.6. Exercícios

{ex:ma1}

Exercício 1.8
É possível somar uma constante $c \in \mathbb{R}$ para todos custos de uma instância do EPM ou EPPM, mantendo a otimalidade da solução?

{ex:ma2}

Exercício 1.9
Prove a proposição 1.7.

2. Tabelas hash

Em *hashing* nosso interesse é uma estrutura de dados H para gerenciar um conjunto de chaves sobre um universo U e que oferece as operações de um *dicionário*:

- Inserção de uma chave $c \in U$: insert(c,H)
- Deleção de uma chave $c \in U$: delete(c,H)
- Teste da pertinência: Chave $c \in H$? lookup(c,H)

Uma característica do problema é que tamanho $|U|$ do universo de chaves possíveis pode ser grande, por exemplo o conjunto de todos strings ou todos números inteiros. Portanto usar a chave como índice de um vetor de booleano não é uma opção. Uma tabela hash é um alternativa para outros estruturas de dados de dicionários, p.ex. árvores. O princípio de tabelas hash: aloca uma tabela de tamanho m e usa uma *função hash* $h : U \to [m]$ para calcular a posição de uma chave na tabela.
Como o tamanho da tabela hash é menor que o número de chaves possíveis, existem chaves c_1, c_2 com $h(c_1) = h(c_2)$, que geram *colisões*. Logo uma tabela hash precisa definir um método de *resolução de colisões*. Uma solução é *Hashing perfeito*: escolhe uma função hash, que para um dado conjunto de chaves não tem colisões. Isso é possível se o conjunto de chaves é conhecido e estático.

2.1. Hashing com listas encadeadas

Seja $h : U \to [m]$ uma função hash. Mantemos uma coleção de m listas $l_0, \ldots, l_{m-1}$ tal que a lista l_i contém as chaves c com *valor hash* $h(c) = i$. Supondo que a avaliação de h é possível em $O(1)$, a inserção custa $O(1)$, e o teste é proporcional ao tamanho da lista.
Para obter uma distribuição razoável das chaves nas listas, supomos que h é uma função hash *simples* e *uniforme*:

$$\Pr(h(c) = i) = 1/m. \tag{2.1}$$ {eq:hashun

Seja $n_i := |l_i|$ o tamanho da lista i e c_{ji} a variável aleatória que indica se chave j pertence a lista i. Temos $\Pr(c_{ji} = 1) = \Pr(h(j) = i)$. Ainda $n_i = \sum_{1 \le j \le n} c_{ji}$

e com isso

$$E[n_i] = E[\sum_{1 \le j \le n} c_{ji}] = \sum_{1 \le j \le n} E[c_{ji}] = \sum_{1 \le j \le n} Pr(h(c_j) = i) = n/m.$$

O valor $\alpha := n/m$ é o *fator de ocupação* da tabela hash.

```
1  insert(c,H) :=
2    insert(c,l_h(c))
3
4  lookup(c,H) :=
5    lookup(c,l_h(c))
6
7  delete(c,H) :=
8    delete(c,l_h(c))
```

Teorema 2.1
Uma busca sem sucesso precisa tempo esperado $\Theta(1 + \alpha)$.

Prova. A chave c tem a probabilidade $1/m$ de ter um valor hash i. O tamanho esperado da lista i é α. Uma busca sem sucesso nessa lista precisa tempo $\Theta(\alpha)$. Junto com a avaliação da função hash em $\Theta(1)$, obtemos tempo esperado total $\Theta(1 + \alpha)$. ■

Teorema 2.2
Uma busca com sucesso precisa tempo esperado $\Theta(1 + \alpha)$.

Prova. Supomos que a chave c é uma das chaves na tabela com probabilidade uniforme. Então, a probabilidade de pertencer a lista i (ter valor hash i) é n_i/n. Uma busca com sucesso toma tempo $\Theta(1)$ para avaliação da função hash, e mais um número de operações proporcional à posição p da chave na sua lista. Com isso obtemos tempo esperado $\Theta(1 + E[p])$.
Para determinar a posição esperada na lista, $E[p]$, seja $c_1, \ldots, c_n$ a sequência na qual as chaves foram inseridas. Supondo que inserimos as chaves no inicio da lista, $E[p]$ é um mais que o número de chaves inseridos depois de c na mesma lista.
Seja X_{ij} um variável aleatória que indica se chaves c_i e c_j tem o mesmo valor hash. $E[X_{ij}] = Pr(h(c_i) = h(c_j)) = \sum_{1 \le k \le m} Pr(h(c_i) = k) \, Pr(h(c_j) = k) = 1/m$. Seja p_i a posição da chave c_i na sua lista. Temos

$$E[p_i] = E[1 + \sum_{j:j>i} X_{ij}] = 1 + \sum_{j:j>i} E[X_{ij}] = 1 + (n - i)/m$$

e para uma chave aleatória c

$$\begin{aligned} E[p] &= \sum_{1 \le i \le n} 1/n \, E[p_i] = \sum_{1 \le i \le n} 1/n(1 + (n-i)/m) \\ &= 1 + n/m - (n+1)/(2m) = 1 + \alpha/2 - \alpha/(2n). \end{aligned}$$

Portanto, o tempo esperado de uma busca com sucesso é

$$\Theta(1 + E[p]) = \Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha).$$

■

Seleção de uma função hash Para implementar uma tabela hash, temos que escolher uma função hash, que satisfaz (2.1). Para facilitar isso, supomos que o universo de chaves é um conjunto $U = [u]$ de números inteiros. (Para tratar outros tipos de chaves, costuma-se convertê-los para números inteiros.) Se cada chave ocorre com a mesma probabilidade, $h(i) = i \mod m$ é uma função hash simples e uniforme. Essa abordagem é conhecida como *método de divisão*. O problema com essa função na prática é que não conhecemos a distribuição de chaves, e ela provavelmente não é uniforme. Por exemplo, se m é par, o valor hash de chaves pares é par, e de chaves ímpares é ímpar, e se $m = 2^k$ o valor hash consiste nos primeiros k bits. Uma escolha que funciona na prática é um número primo "suficientemente" distante de uma potência de 2.
O *método de multiplicação* define

$$h(c) = \lfloor m\{Ac\} \rfloor .$$

O método funciona para qualquer valor de m, mas depende de uma escolha adequada de $A \in \mathbb{R}$. Knuth propôs $A \approx (\sqrt{5} - 1)/2$.

Hashing universal Outra idéia: Para qualquer função hash h fixa, sempre existe um conjunto de chaves, tal que essa função hash gera muitas colisões. (Em particular, um "adversário" que conhece a função hash pode escolher chaves $c \in h^{-1}(i)$ para qualquer posição $i \in [m]$, tal que $h(c) = i$ é constante. Para evitar isso podemos escolher uma função hash aleatória de uma família de funções hash.
Uma família $\mathcal{H}$ de funções hash $U \to [m]$ é *universal* se

$$|\{h \in \mathcal{H} \mid h(c_1) = h(c_2)\}| = |H|/m$$

ou equivalente

$$\Pr(h(c_1) = h(c_2)) = 1/m$$

para qualquer par de chaves c_1, c_2.

Teorema 2.3
Se escolhemos uma função hash $h \in \mathcal{H}$ uniformemente, para uma chave arbitrária c o tamanho esperado de $l_{h(c)}$ é

- α, caso $c \notin H$, e
- $1 + \alpha$, caso $c \in H$.

Prova. Para chaves c_1, c_2 seja $X_{ij} = [h(c_1) = h(c_2)]$ e temos

$$E[X_{ij}] = \Pr(X_{ij} = 1) = \Pr(h(c_1) = h(c_2)) = 1/m$$

pela universalidade de $\mathcal{H}$. Para uma chave fixa c seja Y_c o número de colisões.

$$E[Y_c] = E\left[\sum_{\substack{c' \in H \\ c' \neq c}} X_{cc'}\right] = \sum_{\substack{c' \in H \\ c' \neq c}} E[X_{cc'}] \leq \sum_{\substack{c' \in H \\ c' \neq c}} 1/m.$$

Para uma chave $c \notin H$, o tamanho da lista é Y_c, e portanto de tamanho esperado $E[Y_c] \leq n/m = \alpha$. Caso $c \in H$, o tamanho da lista é $1 + Y_c$ e com $E[Y_c] = (n-1)/m$ esperadamente

$$1 + (n-1)/m = 1 + \alpha - 1/m < 1 + \alpha.$$

■

Um exemplo de um conjunto de funções hash universais: Seja $c = (c_0, \ldots, c_r)_m$ uma chave na base m, escolhe $a = (a_0, \ldots, a_r)_m$ randomicamente e define

$$h_a = \sum_{0 \leq i \leq r} c_i a_i \mod m.$$

Hashing perfeito Hashing é *perfeito* sem colisões. Isso podemos garantir somente caso conheçemos as chaves a serem inseridos na tabela. Para uma função aleatória de uma família universal de funções hash para uma tabela hash de tamanho m, o número esperado de colisões é $E[\sum_{i \neq j} X_{ij}] = \sum_{i \neq j} E[X_{ij}] \leq n^2/m$. Portanto, caso escolhemos uma tabela de tamanho $m > n^2$ o número esperado de colisões é menos que um. Em particular, para $m > cn^2$ com $c > 1$ a probabilidade de uma colisão é $\Pr(\sum_{i \neq j} X_{ij} \geq 1) \leq E[\sum_{i \neq j} X_{ij}] \leq n^2/m < 1/c$ onde a primeira desigualdade segue da desigualdade de Markov.

2.2. Hashing com endereçamento aberto

Uma abordagem para resolução de colisões, chamada *endereçamento aberto*, é escolher uma outra posição para armazenar uma chave, caso $h(c)$ é ocupada. Uma estratégia para conseguir isso é procurar uma posição livre numa permutação de todos índices restantes. Assim garantimos que um insert tem sucesso enquanto ainda existe uma posição livre na tabela. Uma função hash $h(c,i)$ com dois argumentos, tal que $h(c,1),\ldots,h(c,m)$ é uma permutação de $[m]$, representa essa estratégia.

```
1  insert(c,H) :=
2    for i in [m]
3      if H[h(c,i)] = free
4        H[h(c,i)]=c
5        return
6
7  lookup(c,H) :=
8    for i in [m]
9      if H[h(c,i)] = free
10       return false
11     if H[h(c,i)] = c
12       return true
13   return false
```

A função $h(c,i)$ é *uniforme*, se a probabilidade de uma chave randômica ter associada uma dada permutação é $1/m!$. A seguir supomos que h é uniforme.

Teorema 2.4
As funções lookup e insert precisam no máximo $1/(1-\alpha)$ testes caso a chave não está na tabela.

Prova. Seja X o número de testes até encontrar uma posição livre. Temos

$$E[X] = \sum_{i\geq 1} i \Pr(X=i) = \sum_{i\geq 1}\sum_{j\geq i} \Pr(X=j) = \sum_{i\geq 1} \Pr(X\geq i).$$

Com T_i o evento que o teste i ocorre e a posição i é ocupada, podemos escrever

$$\Pr(X\geq i) = \Pr(T_1\cap\cdots\cap T_{i-1}) = \Pr(T_1)\Pr(T_2|T_1)\Pr(T_3|T_1,T_2)\cdots\Pr(T_{i-1}|T_1,\ldots,T_{i-2}).$$

Agora $\Pr(T_1) = n/m$, e como h é uniforme $\Pr(T_2|T_1) = n-1/(m-1)$ e em geral

$$\Pr(T_k|T_1,\ldots T_{k-1}) = (n-k+1)/(m-k+1) \leq n/m = \alpha.$$

Portanto $\Pr(X \geq i) \leq \alpha^{i-1}$ e

$$E[X] = \sum_{i \geq 1} \Pr(X \geq i) \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha^i = 1/(1-\alpha).$$

■

Lema 2.1
Para $i < j$, temos $H_j - H_i \leq \ln j - \ln i$.

Prova.

$$H_j - H_i \leq \int_i^j \frac{1}{x} dx = \ln j - \ln i.$$

■

Teorema 2.5
Caso $\alpha < 1$ a função lookup precisa esperadamente $1/\alpha \ln 1/(1-\alpha)$ testes caso a chave esteja na tabela, e cada chave tem a mesma probabilidade de ser procurada.

Prova. Seja c a i-gésima chave inserida. No momento de inserção temos $\alpha = (i-1)/m$ e o número esperado de testes T até encontrar a posição livre foi $1/(1-(i-1)/m) = m/(m-(i-1))$, e portanto o número esperado de testes até encontrar uma chave arbitrária é

$$E[T] = 1/n \sum_{1 \leq i \leq n} m/(m-(i-1)) = 1/\alpha \sum_{0 \leq i < n} 1/(m-i) = 1/\alpha(H_m - H_{m-n})$$

e com $H_m - H_{m-n} \leq \ln(m) - \ln(m-n)$ temos

$$E[T] = 1/\alpha(H_m - H_{m-n}) < 1/\alpha(\ln(m) - \ln(m-n)) = 1/\alpha \ln(1/(1-\alpha)).$$

■

Remover elementos de uma tabela hash com endereçamento aberto é mais difícil, porque a busca para um elemento termina ao encontrar uma posição livre. Para garantir a corretude de lookup, temos que marcar posições como "removidas" e continuar a busca nessas posições. Infelizmente, nesse caso, as garantias da complexidade não mantem-se – após uma série de deleções e inserções toda posição livre será marcada como "removida" tal que delete e lookup precisam n passos. Portanto o endereçamento aberto é favorável somente se temos poucas deleções.

Funções hash para endereçamento aberto

- Linear: $h(c, i) = h(c) + i \mod m$
- Quadrática: $h(c, i) = h(c) + c_1 i + c_2 i^2 \mod m$
- Hashing duplo: $h(c, i) = h_1(c) + i h_2(c) \mod m$

Nenhuma das funções é uniforme, mas o hashing duplo mostra um bom desempenho na prática.

2.3. Cuco hashing

Cuco hashing é outra abordagem que procura posições alternativas na tabela em caso de colisões, com o objetivo de garantir um tempo de acesso constante no pior caso. Para conseguir isso, usamos duas funções hash h_1 e h_2, e inserimos uma chave em uma das duas posições $h_1(c)$ ou $h_2(c)$. Desta forma a busca e a deleção possuem complexidade constante $O(1)$:

```
1  lookup(c,H) :=
2    if H[h1(c)] = c or H[h2(c)] = c
3      return true
4    return false
5
6  delete(c,H) :=
7    if H[h1(c)] = c
8      H[h1(c)] := free
9    if H[h2(c)] = c
10     H[h2(c)] := free
```

Inserir uma chave é simples, caso uma das posições alternativas é livre. No caso contrário, a solução do cuco hashing é comportar-se como um cuco com ovos de outras aves que jogá-los fora do seu "ninho": "insert" ocupa a posição de uma das duas chaves. A chave "jogada fora" será inserida novamente na tabela. Caso a posição alternativa dessa chave é livre, a inserção termina. Caso contrário, o processo se repete. Esse procedimento termina após uma série de reinserções ou entra num laço infinito. Nesse último caso temos que realocar todas chaves com novas funções hash.

```
1  insert(c,H) :=
2    if H[h1(c)] = c or H[h2(c)] = c
3      return
4    p := h1(c)
5    do n vezes
```

```
6       if H[p] = free
7         H[p] := c
8         return
9       swap(c,H[p])
10      { escolhe a outra posição da chave atual }
11      if p = h1(c)
12        p := h2(c)
13      else
14        p := h1(c)
15    rehash(H)
16    insert(c,H)
```

Uma maneira de visualizar uma tabela hash com cuco hashing, é usar o *grafo cuco*: caso foram inseridas as chaves $c_1, \ldots, c_n$ na tabela nas posições $p_1, \ldots, p_n$, o grafo é $G = (V, A)$, com $V = [m]$ é $(p_i, h_2(c_i)) \in A$ caso $h_1(c_i) = p_i$ e $(p_i, h_1(c_i)) \in A$ caso $h_2(c_i) = p_i$, i.e., os arcos apontam para a posição alternativa. O grafo cuco é um grafo direcionado e eventualmente possui ciclos. Uma característica do grafo cuco é que uma posição p é eventualmente analisada na inserção de uma chave c somente se existe um caminho de $h_1(c)$ ou $h_2(c)$ para p. Para a analise é suficiente considerar o grafo cuco não-direcionado.

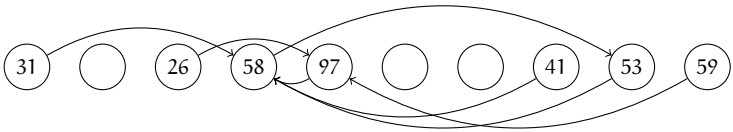
Exemplo 2.1
Para chaves de dois dígitos c_1c_2 seja $h_1(c) = 3c_1 + c_2 \mod m$ e $h_2(c) = 4c_1 + c_2$. Para $m = 10$ obtemos para uma sequencia aleatória de chaves

c	31	41	59	26	53	58	97
$h_1(c)$	0	3	4	2	8	3	4
$h_2(c)$	3	7	9	4	3	8	3

e a seguinte sequencia de tabelas hash

0	1	2	3	4	5	6	7	8	9	
										Inicial
31										Inserção 31
31			41							Inserção 41
31			41	59						Inserção 59
31		26	41	59						Inserção 26
31		26	41	59				53		Inserção 53
31		26	58	59			41	53		Inserção 58
31		26	58	97			41	53	59	Inserção 59

O grafo cuco correspondente é



◊

Lema 2.2
Para posições i e j e um $c > 1$ tal que $m \geq 2cn$, a probabilidade de existir um caminho mínimo de i para j de comprimento $d \geq 1$ é no máximo c^{-d}/m.

Prova. Observe que a probabilidade de um item c ter posições i e j como alternativas é no máximo $\Pr(h_1(c) = i, h_2(c) = j) + \Pr(h_1(c) = j, h_2(c) = i) = 2/m^2$. Portanto a probabilidade de pelo menos uma das n chaves ter posições alternativas i e j é no máximo $2n/m^2 = c^{-1}/m$.
A prova do lema é por indução sobre d. Para $d = 1$ a afirmação está correto pela observação acima. Para $d > 1$ existe um caminho mínimo de comprimento $d - 1$ de i para um k. A probabilidade disso é no máximo $c^{-(d-1)}/m$ e a probabilidade de existir um elemento com posições alternativas k e j no máximo c^{-1}/m. Portanto, para um k fixo, a probabilidade existir um caminho de comprimento d é no máximo c^{-d}/m^2 e considerando todas posições k possíveis no máximo c^{-d}/m. ■
Com isso a probabilidade de existir um caminho entre duas chaves i e j, é igual a probabilidade de existir um caminho começando em $h_1(i)$ ou $h_2(i)$ e terminando em $h_1(j)$ ou $h_2(j)$, que é no máximo $4\sum_{i\geq 1} c^{-i}/m \leq 4/m(c - 1) = O(1/m)$. Logo o número esperado de itens visitados numa inserção é $4n/m(c - 1) = O(1)$, caso não é necessário reconstruir a tabela hash.

2.4. Filtros de Bloom

Um filtro de Bloom armazena um conjunto de n chaves, com as seguintes restrições:

- Não é mais possível remover elementos.
- É possível que o teste de pertinência tem sucesso, sem o elemento fazer parte do conjunto (“false positive”).

Um filtro de Bloom consiste em m bits B_i, $1 \leq i \leq m$, e usa k funções hash $h_1, \ldots, h_k$.

```
1  insert(c,B) :=
2    for i in 1...k
3      b_{h_i(c)} := 1
4    end for
5
6  lookup(c,B) :=
7    for i in 1...k
8      if b_{h_i(c)} = 0
9        return false
10   return true
```

Após de inserir n chaves, um dado bit é ainda 0 com probabilidade

$$p' = \left(1 - \frac{1}{m}\right)^{kn} = \left(1 - \frac{kn/m}{kn}\right)^{kn} \approx e^{-kn/m}$$

que é igual ao valor esperado da fração de bits não setados[1]. Sendo ρ a fração de bits não setados realmente, a probabilidade de erradamente classificar um elemento como membro do conjunto é

$$(1-\rho)^k \approx (1-p')^k \approx \left(1 - e^{-kn/m}\right)^k$$

porque ρ é com alta probabilidade perto do seu valor esperado (Broder e Mitzenmacher, 2003). Broder e Mitzenmacher (2003) também mostram que o número ótimo k de funções hash para dados valores de n, m é $m/n \ln 2$ e com isso temos um erro de classificação $\approx (1/2)^k$.
Aplicações:

1. Hifenação: Manter uma tabela de palavras com hifenação excepcional (que não pode ser determinado pelas regras).

2. Comunicação efetiva de conjuntos, p.ex. seleção em bancos de dados distribuídas. Para calcular um join de dois bancos de dados A, B, primeiramente A filtra os elementos, manda um filtro de Bloom S_A para B e depois B executa o join baseado em S_A. Para eliminação de eventuais elementos classificados erradamente, B manda os resultados para A e A filtra os elementos errados.

- http://en.m.wikipedia.org/wiki/Locality-sensitive_hashing

[1] Lembrando que $e^x \geq (1 + x/n)^n$ para $n > 0$.

Tabela 2.1.: Complexidade das operações em tabelas hash. Complexidades em negrito são amortizados.

	insert	lookup	delete
Listas encadeadas	$\Theta(1)$	$\Theta(1+\alpha)$	$\Theta(1+\alpha)$
Endereçamento aberto	$O(1/(1-\alpha))$	$O(1/(1-\alpha))$	-
(com/sem sucesso)	$O(1/\alpha \ln 1/(1-\alpha))$	$O(1/\alpha \ln 1/(1-\alpha))$	-
Cuco	$\Theta(\mathbf{1})$	$\Theta(1)$	$\Theta(1)$

ash}

3. Algoritmos de aproximação

Para vários problemas não conhecemos um algoritmo eficiente. Para problemas NP-completos, em particular, uma solução eficiente é pouco provável. Um *algoritmo de aproximação* calcula uma solução aproximada para um problema de otimização. Diferente de uma heurística, o algoritmo *garante* a qualidade da aproximação no pior caso. Dado um problema e um algoritmo de aproximação A, escrevemos $A(x) = y$ para a solução aproximada da instância x, $\varphi(x, y)$ para o valor dessa solução, y^* para a solução ótima e $\mathrm{OPT}(x) = \varphi(x, y^*)$ para o valor da solução ótima.

3.1. Problemas, classes e reduções

Definição 3.1
Um *problema de otimização* $\Pi = (\mathcal{P}, \varphi, \mathrm{opt})$ é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$, junto com

- uma função de otimização (função de objetivo) $\varphi : \mathcal{P} \to \mathbb{N}$ (ou $\mathbb{Q}$).
- um objetivo: Encontrar mínimo ou máximo

$$\mathrm{OPT}(x) = \mathrm{opt}\{\varphi(x, y) \mid (x, y) \in \mathcal{P}\}$$

  junto com uma solução y^* tal que $f(x, y^*) = \mathrm{OPT}(x)$.

O par $(x, y) \in \mathcal{P}$ caso y é uma solução para x.

Reductions are as follows.

- From evalution to decision: do a form of binary search (e.g. for maximization double the value until unfeasible, and the binary search) in logarithmic time.
- From construction to evaluation: in general unclear; for many NO problems, in particular self-reducible ones (e.g. SAT): guess solution elements and evaluate; keep them if the best value remains the same.

Uma instância x de um problema de otimização possui soluções $S(x) = \{y \mid (x, y) \in \mathcal{P}\}$.

Convenção 3.1
Escrevemos um problema de otimização na forma

NOME

Instância x

Solução y

Objetivo Minimiza ou maximiza $\varphi(x, y)$.

Com um dado problema de otimização correspondem três problemas:

- Construção: Dado x, encontra a solução ótima y^* e seu valor $\mathrm{OPT}(x)$.
- Avaliação: Dado x, encontra valor ótimo $\mathrm{OPT}(x)$.
- Decisão: Dado x e k, decide se $\mathrm{OPT}(x) \geq k$ (maximização) ou $\mathrm{OPT}(x) \leq k$ (minimização).

f:polimit}

Definição 3.2
Uma relação binária R é *polinomialmente limitada* se

$$\exists p \in \mathsf{poly} : \forall (x, y) \in R : |y| \leq p(|x|).$$

Definição 3.3 (Classes de complexidade)
A classe PO consiste dos problemas de otimização tal que existe um algoritmo polinomial A com $\varphi(x, A(x)) = \mathrm{OPT}(x)$ para $x \in I$.
A classe NPO consiste dos problemas de otimização tal que

(i) As instâncias $x \in I$ são reconhecíveis em tempo polinomial.

(ii) A relação $\mathcal{P}$ é polinomialmente limitada.

(iii) Para y arbitrário, polinomialmente limitado: $(x, y) \in \mathcal{P}$ é decidível em tempo polinomial.

(iv) φ é computável em tempo polinomial.

Definição 3.4
Uma *redução preservando a aproximação* entre dois problemas de minimização Π_1 e Π_2 consiste num par de funções f e g (computáveis em tempo polinomial) tal que para instância x_1 de Π_1, $x_2 := f(x_1)$ é instância de Π_2 com

pa1} $$\mathrm{OPT}_{\Pi_2}(x_2) \leq \mathrm{OPT}_{\Pi_1}(x_1) \tag{3.1}$$

e para uma solução y_2 de Π_2 temos uma solução $y_1 := g(x_1, y_2)$ de Π_1 com

pa2} $$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \tag{3.2}$$

Uma redução preservando a aproximação fornece uma α-aproximação para Π_1 dada uma α-aproximação para Π_2, porque

$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \leq \alpha\mathrm{OPT}_{\Pi_2}(x_2) \leq \alpha\mathrm{OPT}_{\Pi_1}(x_1).$$

Observe que essa definição é vale somente para problemas de minimização. A definição no caso de maximização é semelhante.

[More complexity classes]

- PTAS: $t(n, \epsilon) \approx n^{f(\epsilon)}$.
- EPTAS: $t(n, \epsilon) \approx n^{O(1)f(\epsilon)}$.
- FPTAS: $t(n, \epsilon) \approx \mathrm{poly}(n, 1/\epsilon)$.

We have FPTAS $\subseteq$ EPTAS $\subseteq$ PTAS, and if P $\neq$ NP also FPTAS $\neq$ PTAS and PTAS $\neq$ APX.
We also have: Theorem:

$$\text{FTPAS} \rightarrow \text{pseudo-polynomial} \iff \text{not strongly NP-hard.}$$

[Proof technique] We start with minimization. The main proof technique: find some structure y, usually a relaxation, such that

- $\mathrm{LB}(x) = \phi(x, y) \leq \mathrm{OPT}(x)$, and then
- $\phi(x, A(x)) \leq r\mathrm{LB}(x)$,

from which $\phi(x, A(x)) \leq r\mathrm{OPT}(x)$ follows. Here we design y such that the solution our algorithm finds is easier evaluated compared to y.
For maximization this becomes:

- $\mathrm{UB}(x) = \phi(x, y) \geq \mathrm{OPT}(x)$, and

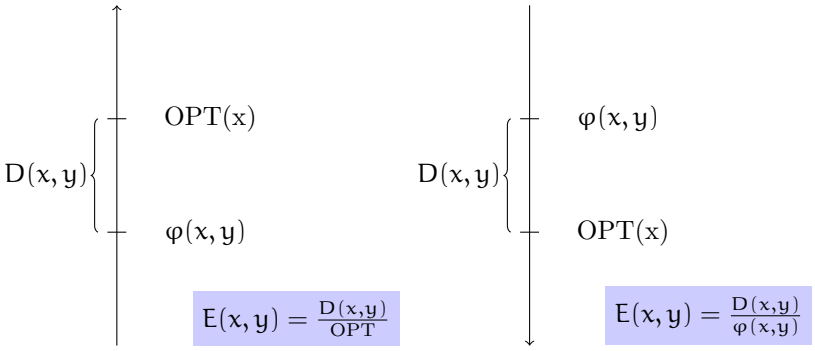
- $\phi(x, A(x)) \geq rUB(x)$.

3.2. Medidas de qualidade

Uma *aproximação absoluta* garante que $D(x, y) = |OPT(x) - \varphi(x, y)| \leq D$ para uma constante D e todo x, enquanto uma *aproximação relativa* garante que o *erro relativo* $E(x, y) = D(x, y)/\max\{OPT(x), \varphi(x, y)\} \leq \epsilon \leq 1$ todos x. Um algoritmo que consegue um aproximação com constante ϵ também se chama ϵ-aproximativo. Tais algoritmos fornecem uma solução que difere no máximo um fator constante da solução ótima. A classe de problemas de otimização que permitem uma ϵ-aproximação em tempo polinomial para uma constante ϵ se chama APX.
Uma definição alternativa é a *taxa de aproximação* $R(x, y) = 1/(1 - E(x, y)) \geq 1$. Um algoritmo com taxa de aproximação r se chama r-aproximativo. (Não tem perigo de confusão com o erro relativo, porque $r \geq 1$.)

Nossa definição segue Ausiello et al. (1999). Ela tem a vantagem, de ser não-ambígua entre o erro relativo e o erro absoluto. Um algoritmo de minimização que garante no máximo um fator 3 da solução ótima ou um algoritmo de maximização que garante no mínimo um terço da solução ótima é 2/3-aproximativo ou 3-aproximativo. A definição tem a desvantagem que ela é pouco intuitivo: seria mais claro, chamar o primeiro algoritmo 3-aproximativo, e o segundo 1/3-aproximativo, usando simplesmente a taxa de aproximação $r = \varphi(x, y)/OPT(x)$. Hromkovič (2001) usa ...

Aproximação relativa



Exemplo 3.1
Coloração de grafos planares e a problema de determinar a árvore geradora e a árvore Steiner de grau mínimo (Fürer e Raghavachari, 1994) permitem uma aproximação absoluta, mas não o problema da mochila.
Os problemas da mochila e do caixeiro viajante métrico permitem uma aproximação absoluta constante, mas não o problema do caixeiro viajante. ◊

4-colorability in $O(n^2)$; decision 3 or 4 colors in NP-complete. Corrob.
Fürer e Raghavachari (1994) show how to get MSTs or Steiner trees whose minimal (maximum) degree is within one of the optimal. Singh e Lau (2007) extend this result for weighted case of MSTs. They show that it is possible to get a MST whose weight is at most the weight of an optimal MST of degree less than k, and whose maximum degree is at most $k+1$.

3.3. Técnicas de aproximação

3.3.1. Algoritmos gulosos

Cobertura de vértices

Algoritmo 3.1 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura de vértices $C \subseteq V$.

```
1  VC-GV(G) :=
2    (C, G) := Reduz(G)
3    if V = ∅ then
4      return C
5    else
6      escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
7      return C ∪ {v} ∪ VC-GV(G − v)
8    end if
```

Proposição 3.1
O algoritmo VC-GV é uma $O(\log |V|)$-aproximação.
Prova. Seja G_i o grafo após iteração i e C^* uma cobertura ótima, i.e., $c := |C^*| = OPT(G)$.
A cobertura ótima C^* todos G_i. Logo, a soma dos graus dos vértices em C^* (contando somente arestas em G_i) é pelo menos o número de arestas em G_i

$$\sum_{v \in C^*} \delta_{G_i}(v) \geq \|G_i\|,$$

e o grau médio dos vértices C^* em G_i satisfaz

$$\sum_{v \in C^*} \delta_{G_i}(v)/c \geq \|G_i\|/c.$$

Como o grau máximo do grafo é pelo menos o grau médio em C^* temos

$$\Delta(G_i) \geq \|G_i\|/c,$$

o que permite estimar

$$\sum_{0 \leq i < c} \Delta(G_i) \geq \sum_{0 \leq i < c} \|G_i\|/c \geq \sum_{0 \leq i < c} \|G_c\|/c = \|G_c\| = \|G\| - \sum_{0 \leq i < c} \Delta(G_i)$$

e logo

$$\sum_{0 \leq i < c} \Delta(G_i) \geq \|G\|/2,$$

i.e. o algoritmo remove em c iterações pelo menos a metade das arestas. Essa estimativa continua a ser válida, logo após

$$c \lceil \lg \|G\| \rceil \leq c \lceil 2 \log |G| \rceil = O(c \log |G|)$$

iterações não tem mais arestas. Como em cada iteração foi escolhido um vértice, a taxa de aproximação é $\log |G|$. ■

Algoritmo 3.2 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Um cobertura de vértices $C \subseteq V$.

```
1  VC-GE(G) :=
2    (C, G) := Reduz(G)
3    if E = ∅ then
4      return C
5    else
6      escolhe e = {u, v} ∈ E
7      return C ∪ {u, v} ∪ VC-GE(G − {u, v})
8    end if
```

Proposição 3.2
Algoritmo VC-GE é uma 2-aproximação para VC.

Prova. Cada cobertura C contém pelo menos um dos dois vértices escolhidos, logo temos $\phi_{\text{VC-GE}}(G) \leq 2|C|$, e no caso particular da solução ótima também $\phi_{\text{VC-GE}}(G) \leq 2\text{OPT}(G)$. ■

Algoritmo 3.3 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura de vértices $C \subseteq V$.

```
1  VC-B(G) :=
2    (C, G) := Reduz(G)
3    if V = ∅ then
4      return C
5    else
6      escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
7      C1 := C ∪ {v} ∪ VC-B(G − v)
8      C2 := C ∪ N(v) ∪ VC-B(G − v − N(v))
9      if |C1| < |C2| then
10       return C1
11     else
12       return C2
13     end if
14   end if
```

Problema da mochila

KNAPSACK

Instância Um número n de itens com valores $v_i \in \mathbb{N}$ e tamanhos $t_i \in \mathbb{N}$, para $i \in [n]$, um limite M, tal que $t_i \leq M$ (todo item cabe na mochila).

Solução Uma seleção $S \subseteq [n]$ tal que $\sum_{i \in S} t_i \leq M$.

Objetivo Maximizar o valor total $\sum_{i \in S} v_i$.

Observação: O problema da mochila é NP-completo. {problem:k

Como aproximar?

- Idéia: Ordene por v_i/t_i ("valor médio") em ordem decrescente e enche o mochila o mais possível nessa ordem.

Abordagem

```
1  K−G(v_i, t_i) :=
2    ordene os itens tal que v_i/t_i ≥ v_j/t_j, ∀i < j.
```

```
3     for i ∈ X do
4       if t_i < M then
5         S := S ∪ {i}
6         M := M − t_i
7       end if
8     end for
9     return S
```

Aproximação boa?

- Considere

$$\begin{aligned} &v_1 = 1, \ldots, v_{n-1} = 1, v_n = M - 1 \\ &t_1 = 1, \ldots, t_{n-1} = 1, t_n = M = kn \qquad k \in \mathbb{N} \text{ arbitrário} \end{aligned}$$

- Então:

$$v_1/t_1 = 1, \ldots, v_{n-1}/t_{n-1} = 1, v_n/t_n = (M-1)/M < 1$$

- K-G acha uma solução com valor $\varphi(x) = n-1$, mas o ótimo é $\mathrm{OPT}(x) = M - 1$.

- Taxa de aproximação:

$$\mathrm{OPT}(x)/\varphi(x) = \frac{M-1}{n-1} = \frac{kn-1}{n-1} \geq \frac{kn-k}{n-1} = k$$

- K-G não possui taxa de aproximação fixa!

- Problema: Não escolhemos o item com o maior valor.

Tentativa 2: Modificação

```
1  K−G'(v_i, t_i) :=
2    S_1 := K−G(v_i, t_i) // solução gulosa
3    v_1 := ∑_{i∈S_1} v_i
4    S_2 := {argmax_i v_i} // maior item
5    v_2 := ∑_{i∈S_2} v_i
6    retorna a maior das duas soluções
```

Aproximação boa?

- O algoritmo melhorou?
- Surpresa

  Proposição 3.3
  K-G' é uma 2-aproximação, i.e. $OPT(x) < 2\varphi_{K\text{-}G'}(x)$.

Prova. Seja j o primeiro item que K-G não coloca na mochila. Nesse ponto temos valor e tamanho

$$\bar{v_j} = \sum_{1 \le i < j} v_i \le \varphi_{K\text{-}G}(x) \tag{3.3}$$

$$\bar{t_j} = \sum_{1 \le i < j} t_i \le M \tag{3.4}$$

Afirmação: $OPT(x) < \bar{v_j} + v_j$. Nesse caso

(a) Seja $v_j \le \bar{v_j}$.

$$OPT(x) < \bar{v_j} + v_j \le 2\bar{v_j} \le 2\varphi_{K\text{-}G}(x) \le 2\varphi_{K\text{-}G'}$$

(b) Seja $v_j > \bar{v_j}$

$$OPT(x) < \bar{v_j} + v_j < 2v_j \le 2v_{max} \le 2\varphi_{K\text{-}G'}$$

Prova da afirmação: No momento em que item j não cabe, temos espaço $M - \bar{t_j} < t_j$ sobrando. Como os itens são ordenados em ordem de densidade decrescente, obtemos um limite superior para a solução ótima preenchendo esse espaço com a densidade v_j/t_j:

$$OPT(x) \le \bar{v_j} + (M - \bar{t_j})\frac{v_j}{t_j} < \bar{v_j} + v_j.$$

■

3.3.2. Aproximações com randomização

Randomização

- Idéia: Permite escolhas randômicas ("joga uma moeda")
- Objetivo: Algoritmos que decidem correta com probabilidade alta.
- Objetivo: Aproximações com *valor esperado* garantido.
- Minimização: $E[\varphi_A(x)] \le 2OPT(x)$
- Maximização: $2E[\varphi_A(x)] \ge OPT(x)$

Randomização: Exemplo

SATISFATIBILIDADE MÁXIMA, MAXIMUM SAT

Instância Uma fórmula $\varphi \in \mathcal{L}(V)$ sobre variáveis $V = \{v_1, \ldots, v_m\}$, $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ em FNC.

Solução Uma atribuição de valores de verdade $a : V \to \mathbb{B}$.

Objetivo Maximiza o número de cláusulas satisfeitas

$$|\{C_i \mid [\![C_i]\!]_a = 1\}|.$$

{alg:satr}

Nossa solução

```
1  SAT−R(φ) :=
2    seja φ = φ(v1,...,vk)
3    for all i ∈ [1,k] do
4      escolhe vi = 1 com probabilidade 1/2
5    end for
```

Observação 3.1
A quantidade $[\![C]\!]_a$ é o valor da cláusula C na atribuição a. ◊

Aproximação?

- Surpresa: Algoritmo SAT−R é 2-aproximação.

Prova. O valor esperado de uma cláusula C com l variáveis é $E[[\![C]\!]] = \Pr([\![C]\!] = 1) = 1 - 2^{-l} \geq 1/2$. Logo o valor esperado do número total $T = \sum_{i\in[n]} [\![C_i]\!]$ de cláusulas satisfeitas é

$$E[T] = E[\sum_{i\in[n]} [\![C_i]\!]] = \sum_{i\in[n]} E[[\![C_i]\!]] \geq n/2 \geq \mathrm{OPT}/2$$

pela linearidade do valor esperado. ■

Outro exemplo

Cobertura de vértices guloso e randomizado.

```
1  VC−RG(G) :=
2    seja w̄ := Σ_{v∈V} deg(v)
3    C := ∅
```

```
4    while E ≠ ∅ do
5      escolhe v ∈ V com probabilidade deg(v)/w̄
6      C := C ∪ {v}
7      G := G − v
8    end while
9    return C ∪ V
```

Resultado: $\mathbb{E}[\phi_{\text{VC-RG}}(x)] \leq 2\text{OPT}(x)$.

3.3.3. Programação linear

Técnicas de programação linear são frequentemente usadas em algoritmo de aproximação. Entre eles são o *arredondamento randomizado* e *algoritmos primais-duais*.

Exemplo 3.2 (Arredondamento para cobertura por conjuntos)
Considere o problema de cobertura por conjuntos

$$\begin{aligned} \textbf{minimiza} \quad & \sum_{i\in[n]} w_i x_i, && && (3.5) \quad \texttt{\{ilp:cpc\}} \\ \textbf{sujeito a} \quad & \sum_{i\in[n]\mid u\in C_i} x_i \geq 1, && \forall u \in U, \\ & x_i \in \{0,1\}, && \forall i \in [n]. \end{aligned}$$

Seja f_e a frequência de um elemento e, i.e. o número de conjuntos que contém e e f a maior frequência. Um algoritmo de arredondamento simples é dado por

Teorema 3.1
A seleção dos conjuntos com $x_i \geq 1/f$ na relaxação linear de (3.5) é uma f-aproximação do problema de cobertura de conjuntos.

Prova. Como $|\{i \in [n] \mid u \in C_i\}| \leq f$, temos $x_i \geq 1/f$ em média sobre esse conjunto. Logo existe, para cada $u \in U$ um conjunto com $x_i \geq 1/f$ que cobre u e a seleção é uma solução válida. O arrendondamento aumenta o custo por no máximo um fator f, logo temos uma f-aproximação. ■ ◊

3.4. Esquemas de aproximação

Novas considerações

- Frequentemente uma r-aproximação não é suficiente. $r = 2$: 100% de erro!

- Existem aproximações melhores? p.ex. para SAT? problema do mochila?
- Desejável: Esquema de aproximação em tempo polinomial (EATP); polynomial time approximation scheme (PTAS)
  - Para cada entrada e taxa de aproximação r:
  - Retorne r-aproximação em tempo polinomial.

Um exemplo: Mochila máxima (Knapsack)

- Problema da mochila (veja página 155):
- Algoritmo MM-PD com programação dinâmica (pág. 226): tempo $O(n \sum_{i\in[n]} v$
- Desvantagem: Pseudo-polinomial.

Denotamos uma instância do problema da mochila com $I = (\{v_i\},\{t_i\})$. Seja $r > 1$ uma qualidade de aproximação desejada.

```
1  MM—PTAS(I,r) :=
2     v_max := max_i{v_i}
3     t := ⌊log_2 (r−1)/r · v_max/n⌋
4     v'_i := ⌊v_i/2^t⌋ para i = 1,...,n
5     Define a nova instância I' = ({v'_i},{t_i})
6     return MM-PD(I')
```

Teorema 3.2
MM-PTAS é uma r-aproximação em tempo $O(rn^3/(r-1))$.

Prova. A complexidade da preparação nas linhas 1–3 é $O(n)$. A chamada para MM-PD custa

$$O\left(n \sum_{i\in[n]} v'_i\right) = O\left(n \sum_{i\in[n]} \frac{v_i}{((r-1)/r)(v_{max}/n)}\right)$$
$$= O\left(\frac{r}{r-1} n^2 \sum_{i\in[n]} v_i/v_{max}\right) = O\left(\frac{r}{r-1} n^3\right).$$

Seja S = MM-PTAS(I) a solução obtida pelo algoritmo e S^* uma solução

ótima.

$$\begin{aligned}
\varphi_{\text{MM-PTAS}}(I,S) = \sum_{i\in S} v_i &\geq \sum_{i\in S} 2^t \left\lfloor v_i/2^t \right\rfloor && \text{definição de } \lfloor\cdot\rfloor \\
&\geq \sum_{i\in S^*} 2^t \left\lfloor v_i/2^t \right\rfloor && \text{otimalidade de MM-PD sobre } v'_i \\
&\geq \sum_{i\in S^*} v_i - 2^t && \text{(A.2)} \\
&= \left(\sum_{i\in S^*} v_i\right) - 2^t|S^*| \\
&\geq \mathrm{OPT}(I) - 2^t n
\end{aligned}$$

Portanto

$$\begin{aligned}
&\mathrm{OPT}(I) \leq \varphi_{\text{MM-PTAS}}(I,S) + 2^t n \leq \varphi_{\text{MM-PTAS}}(I,S) + \frac{\mathrm{OPT}(I)}{v_{\max}} 2^t n \\
&\iff \mathrm{OPT}(I)\left(1 - \frac{2^t n}{v_{\max}}\right) \leq \varphi_{\text{MM-PTAS}}(I,S)
\end{aligned}$$

e com $2^t n/v_{\max} \leq (r-1)/r$

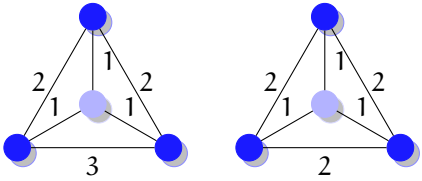
$$\iff \mathrm{OPT}(I) \leq r\varphi_{\text{MM-PTAS}}(I,S).$$

■

Um EATP frequentemente não é suficiente para resolver um problema adequadamente. Por exemplo temos um EATP para

- o problema do caixeiro viajante euclidiano com complexidade $O(n^{3000/\epsilon})$ (Arora, 1996);
- o problema do mochila múltiplo com complexidade $O(n^{12(\log 1/\epsilon)/\epsilon^8})$ (Chekuri, Kanna, 2000);
- o problema do conjunto independente máximo em grafos com complexidade $O(n^{(4/\pi)(1/\epsilon^2+1)^2(1/\epsilon^2+2)^2})$ (Erlebach, 2001).

Para obter uma aproximação com 20% de erro, i.e. $\epsilon = 0.2$ obtemos algoritmos com complexidade $O(n^{15000})$, $O(n^{375000})$ e $O(n^{523804})$, respectivamente!



{fig:gfm}

Figura 3.1.: Grafo com fecho métrico.

3.5. Aproximando o problema da árvore de Steiner mínima

Seja $G = (V, A)$ um grafo completo, não-direcionado com custos $c_a \geq 0$ nos arcos. O problema da árvore Steiner mínima (ASM) consiste em achar o subgrafo conexo mínimo que inclui um dado conjunto de *vértices necessários* ou *terminais* $R \subseteq V$. Esse subgrafo sempre é uma árvore (ex. 3.1). O conjunto $V \setminus R$ forma os *vértices Steiner*. Para um conjunto de arcos A, define o custo $c(A) = \sum_{a \in A} c_a$.

Observação 3.2
ASM é NP-completo. Para um conjunto fixo de vértices Steiner $V' \subseteq V \setminus R$, a melhor solução é a árvore geradora mínima sobre $R \cup V'$. Portanto a dificuldade é a seleção dos vértices Steiner da solução ótima. ◊

Definição 3.5
Os custos são *métricos* se eles satisfazem a desigualdade triangular, i.e.

$$c_{ij} \leq c_{ik} + c_{kj}$$

para qualquer tripla de vértices i, j, k.

Teorema 3.3
Existe uma redução preservando a aproximação de ASM para a versão métrica do problema.

Prova. O fecho métrico de $G = (V, A)$ é um grafo G' completo sobre vértices e com custos $c'_{ij} := d_{ij}$, sendo d_{ij} o comprimento do menor caminho entre i e j em G. Evidentemente $c'_{ij} \leq c_{ij}$ e portanto (3.1) é satisfeita. Para ver que (3.2) é satisfeita, seja T' uma solução de ASM em G'. Define T como união de todos caminhos definidos pelos arcos em T', menos um conjunto de arcos para remover eventuais ciclos. O custo de T é no máximo $c(T')$ porque o custo de todo caminho é no máximo o custo da aresta correspondente em T'. ■
Consequência: Para o problema do ASM é suficiente considerar o caso métrico.

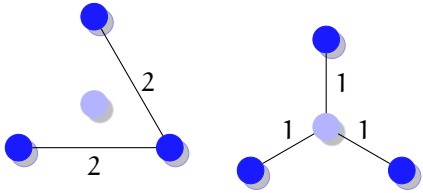


Figura 3.2.: AGM sobre R e melhor solução. ●: vértice em R, ●: vértice Steiner. {fig:agmex

asm}

Teorema 3.4
O AGM sobre R é uma 2-aproximação para o problema do ASM.

Prova. Considere a solução ótima S^* de ASM. Duplica todas arestas[1] tal que todo vértice possui grau par. Encontra um ciclo Euleriano nesse grafo. Remove vértices duplicados nesse caminho. O custo do caminho C obtido dessa forma não é mais que o dobro do custo original: o grafo com todas arestas custa $2c(S^*)$ e a remoção de vértices duplicados não aumenta esse custo, pela metricidade. Como esse caminho é uma árvore geradora, temos $c(A) \leq c(C) \leq 2c(S^*)$ para AGM A. ■

3.6. Aproximando o PCV

Teorema 3.5
Para qualquer função $\alpha(n)$ computável em tempo polinomial o PCV não possui $\alpha(n)$-aproximação em tempo polinomial, caso $P \neq NP$.

Prova. Via redução de HC para PCV. Para uma instância $G = (V, A)$ de HC define um grafo completo G' com

$$c_a = \begin{cases} 1, & a \in A, \\ \alpha(n)n, & \text{caso contrário.} \end{cases}$$

Se G possui um ciclo Hamiltoniano, então o custo da menor rota é n. Caso contrário qualquer rota usa ao menos uma aresta de custo $\alpha(n)n$ e portanto o custo total é $\geq \alpha(n)n$. Portanto, dado uma $\alpha(n)$-aproximação de PCV podemos decidir HC em tempo polinomial. ■

[1] Isso transforma G num multigrafo.

Caso métrico No caso métrico podemos obter uma aproximação melhor. Determina uma rota como segue:

1. Determina uma AGM A de G.
2. Duplica todas arestas de A.
3. Acha um ciclo Euleriano nesse grafo.
4. Remove vértices duplicados.

Teorema 3.6
O algoritmo acima define uma 2-aproximação.

Prova. A melhor solução do PCV menos uma aresta é uma árvore geradora de G. Portanto $c(A) \leq \text{OPT}$. A solução S obtida pelo algoritmo acima satisfaz $c(S) \leq 2c(A)$ e portanto $c(S) \leq 2\text{OPT}$, pelo mesmo argumento da prova do teorema 3.4. ■

O fator 2 dessa aproximação é resultado do passo 2 que duplica todas arestas para garantir a existência de um ciclo Euleriano. Isso pode ser garantido mais barato: A AGM A possui um número par de vértices com grau ímpar (ver exercício 3.2), e portanto podemos calcular um emparelhamento perfeito mínimo E entre esse vértices. O grafo com arestas $A \cup E$ possui somente vértices com grau par e portanto podemos aplicar os restantes passos nesse grafo.

Teorema 3.7 (Cristofides)
A algoritmo usando um emparelhamento perfeito mínimo no passo 2 é uma 3/2-aproximação.

Prova. O valor do emparelhamento E não é mais que OPT/2: remove vértices não emparelhados em E da solução ótima do PCV. O ciclo obtido dessa forma é a união dois emparelhamentos perfeitos E_1 e E_2 formados pelas arestas pares ou ímpares no ciclo. Com E_1 o emparelhamento de menor custo, temos

$$c(E) \leq c(E_1) \leq (c(E_1) + c(E_2))/2 = \text{OPT}/2$$

e portanto

$$c(S) = c(A) + c(E) \leq \text{OPT} + \text{OPT}/2 = 3/2\text{OPT}.$$

■

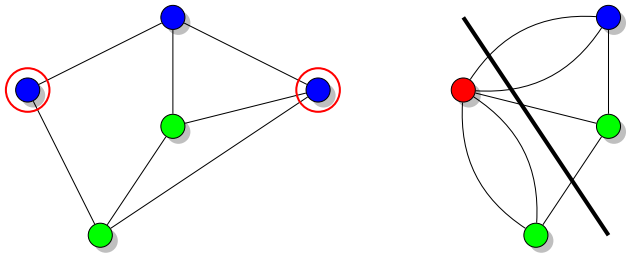


Figura 3.3.: Identificação de dois terminais e um corte no grafo reduzido. Vértices em verde, terminais em azul. O grafo reduzido possui múltiplas arestas entre vértices. {fig:cmm1}

3.7. Aproximando problemas de cortes

Seja $G = (V, A, c)$ um grafo conectado com pesos c nas arestas. Lembramos que um corte C é um conjunto de arestas que separa o grafo em dois partes $S \dot\cup V \setminus S$. Dado dois vértices $s, t \in V$, o problema de achar um corte mínimo que separa s e t pode ser resolvido via fluxo máximo em tempo polinomial. Generalizações desse problema são:

- Corte múltiplo mínimo (CMM): Dado terminais $s_1, \ldots, s_k$ determine o menor corte C que separa todos.
- k-corte mínimo (k-CM): Mesmo problema, sem terminais definidos. (Observe que todos k componentes devem ser não vazios).

Fato 3.1
CMM é NP-difícil para qualquer $k \geq 3$. k-CM possui uma solução polinomial em tempo $O(n^{k^2})$ para qualquer k, mas é NP-difícil, caso k faz parte da entrada (Goldschmidt e Hochbaum, 1988).

Solução de CMM Chamamos um corte que separa um vértice dos outros um *corte isolante*. Idéia: A união de cortes isolantes para todo s_i é um corte múltiplo. Para calcular o corte isolante para um dado terminal s_i, identificamos os restantes terminais em um único vértice S e calculamos um corte mínimo entre s_i e S. (Na identificação de vértices temos que remover self-loops, e somar os pesos de múltiplas arestas.)
Isso leva ao algoritmo

{alg:c

Algoritmo 3.4 (CI)
Entrada Grafo $G = (V, A, c)$ e terminais $s_1, \ldots, s_k$.

Saída Um corte múltiplo que separa os s_i.

```
1  Para cada i ∈ [1,k]: Calcula o corte isolante C_i de s_i.
2  Remove o maior desses cortes e retorne a união dos
       restantes.
```

Teorema 3.8
Algoritmo 3.4 é uma $2 - 2/k$-aproximação.

Prova. Considere o corte mínimo C^*. De acordo com a Fig. 3.4 ele pode ser representado pela união de k cortes que separam os k componentes individualmente:

$$C^* = \bigcup_{i \in [k]} C_i^*.$$

Cada aresta de C^* faz parte das cortes das duas componentes adjacentes, e portanto

$$\sum_{i \in [k]} w(C_i^*) = 2w(C^*)$$

e ainda $w(C_i) \leq w(C_i^*)$ para os cortes C_i do algoritmo 3.4, porque usamos o corte isolante mínimo de cada componente. Logo, para o corte C retornado pelo algoritmo temos

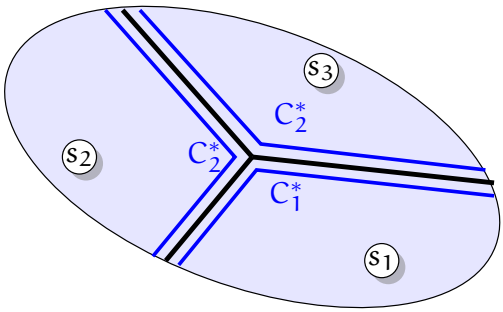
$$w(C) \leq (1 - 1/k) \sum_{i \in [k]} w(C_i) \leq (1 - 1/k) \sum_{i \in [k]} w(C_i^*) \leq 2(1 - 1/k) w(C^*).$$

■

A análise do algoritmo é ótimo, como o exemplo da Fig. 3.5 mostra. O menor corte que separa s_i tem peso $2 - \epsilon$, portanto o algoritmo retorne um corte de peso $(2 - \epsilon)k - (2 - \epsilon) = (k - 1)(2 - \epsilon)$, enquanto o menor corte que separa todos terminais é o ciclo interno de peso k.

Solução de k-CM Problema: Como saber a onde cortar?

Fato 3.2
Existem somente $n-1$ cortes diferentes num grafo. Eles podem ser organizados numa árvore de *Gomory-Hu* (AGH) $T = (V, T)$. Cada aresta dessa árvore define um corte associado em G pelos dois componentes após a sua remoção.



mm2} Figura 3.4.: Corte múltiplo e decomposição em cortes isolantes.

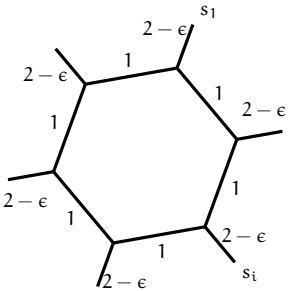


Figura 3.5.: Exemplo de um grafo em que o algoritmo 3.4 retorna uma $2-2/k$-aproximação.

mm3}

1. Para cada $u, v \in V$ o menor corte u–v em G é igual a o menor corte u–v em T (i.e. a aresta de menor peso no caminho único entre u e v em T).

2. Para cada aresta $a \in T$, $w'(a)$ é igual a valor do corte associado.

Por consequência, a AGH codifica o valor de todos cortes em G.
Ele pode ser calculado determinando $n-1$ cortes s–t mínimos:

1. Define um grafo com um único vértice que representa todos vértices do grafo original. Chama um vértice que representa mais que um vértice do grafo original *gordo*.

2. Enquanto existem vértices gordos:
   a) Escolhe um vértice gordo e dois vértices do grafo original que ele representa.
   b) Calcula um corte mínimo entre esses vértices.
   c) Separa o vértice gordo de acordo com o corte mínimo encontrado.

Observação: A união dos cortes definidos por $k-1$ arestas na AGH separa G em pelo menos k componentes. Isso leva ao seguinte algoritmo.

{alg:kcm}

Algoritmo 3.5 (KCM)
Entrada Grafo $G = (V, A, c)$.

Saida Um k-corte.

```
1  Calcula uma AGH T em G.
2  Forma a união dos k−1 cortes mais leves
       definidos por k−1 arestas em T.
```

Teorema 3.9
Algoritmo 3.5 é uma $2 - 2/k$-aproximação.

Prova. Seja $C^* = \bigcup_{i \in [k]} C_i^*$ um corte mínimo, decomposto igual à prova anterior. O nosso objetivo é demonstrar que existem $k-1$ cortes definidos por uma aresta em T que são mais leves que os C_i^*.
Removendo C^* de G gera componentes $V_1, \ldots, V_k$: Define um grafo sobre esses componentes contraindo os vértices de uma componente, com arcos da AGH T entre os componentes, e eventualmente removendo arcos até obter uma nova árvore T'. Seja C_k^* o corte de maior peso, e define V_k como raiz da árvore. Desta forma, cada componente $V_1, \ldots, V_{k-1}$ possui uma aresta associada na direção da raiz. Para cada dessas arestas (u, v) temos

$$w(C_i^*) \geq w'(u, v)$$

porque C_i^* isola o componente V_i do resto do grafo (particularmente separa u e v), e $w'(u,v)$ é o peso do menor corte que separa u e v. Logo

$$w(C) \leq \sum_{a \in T'} w'(a) \leq \sum_{1 \leq i < k} w(C_i^*) \leq (1-1/k) \sum_{i \in [k]} w(C_i^*) = 2(1-1/k)w(C^*).$$

■

3.8. Aproximando empacotamento unidimensional

Dado n itens com tamanhos $s_i \in \mathbb{Z}_+$, $i \in [n]$ e contêineres de capacidade $S \in \mathbb{Z}_+$ o problema do *empacotamento unidimensional* é encontrar o menor número de contêineres em que os itens podem ser empacotados.

EMPACOTAMENTO UNIDIMENSIONAL (MIN-EU) (BIN PACKING)

Entrada Um conjunto de n itens com tamanhos $s_i \in \mathbb{Z}_+$, $i \in [n]$ e o tamanho de um contêiner S.

Solução Uma partição de $[n] = C_1 \cup \cdots \cup C_m$ tal que $\sum_{i \in C_k} s_i \leq S$ para $k \in [m]$.

Objetivo Minimiza o número de partes ("contêineres") m.

A versão de decisão do empacotamento unidimensional (EU) pede decidir se os itens cabem em m contêineres.

Fato 3.3
EU é fortemente NP-completo.

{prop:bp1}

Proposição 3.4
Para um tamanho S fixo EU pode ser resolvido em tempo $O(n^{S^S})$.

Prova. Podemos supor, sem perda de generalidade, que os itens possuem tamanhos $1, 2, \ldots, S-1$. Um padrão de alocação de um contêiner pode ser descrito por uma tupla $(t_1, \ldots, t_{S-1})$ sendo t_i o número de itens de tamanho i. Seja T o conjunto de todos padrões que cabem num contêiner. Como $0 \leq t_i \leq S$ o número total de padrões T é menor que $(S+1)^{S-1} = O(S^S)$. Uma ocupação de m contêineres pode ser descrito por uma tupla $(n_1, \ldots, n_T)$ com n_i sendo o número de contêineres que usam padrão i. O número de contêineres é no máximo n, logo $0 \leq n_i \leq n$ e o número de alocações diferentes é no máximo $(n+1)^T = O(n^T)$. Logo podemos enumerar todas possibilidades em tempo polinomial. ■

{pro

Proposição 3.5
Para um m fixo, EU pode ser resolvido em tempo pseudo-polinomial.

Prova. Seja $B(S_1, \ldots, S_m, i) \in \{\text{falso}, \text{verdadeiro}\}$ a resposta se itens $i, i+1, \ldots, n$ cabem em m contêineres com capacidades $S_1, \ldots, S_m$. B satisfaz

$$B(S_1, \ldots, S_m, i) = \begin{cases} \bigvee_{\substack{1 \le j \le m \\ s_i \le S_j}} B(S_1, \ldots, S_j - s_j, \ldots, S_m, i+1), & i \le n, \\ \text{verdadeiro}, & i > n, \end{cases}$$

e $B(S, \ldots, S, 1)$ é a solução do EU[2]. A tabela B possui no máximo $n(S+1)^m$ entradas, cada uma computável em tempo $O(m)$, logo o tempo total é no máximo $O(mn(S+1)^m)$. ■

Observação 3.3
Com um fator adicional de $O(\log m)$ podemos resolver também MIN-EU, procurando o menor i tal que $B(\underbrace{S, \ldots, S}_{i \text{ vezes}}, 0, \ldots, 0, n)$ é verdadeiro. ◊

A proposição 3.4 pode ser melhorada usando programação dinâmica.

{prop:bp3}

Proposição 3.6
Para um número fixo k de tamanhos diferentes, min-EU pode ser resolvido em tempo $O(n^{2k})$.

Prova. Seja $B(i_1, \ldots, i_k)$ o menor número de contêineres necessário para empacotar i_j itens do j-ésimo tamanho e T o conjunto de todas padrões de alocação de um contêiner. B satisfaz

$$B(i_1, \ldots, i_k) = \begin{cases} 1 + \min_{\substack{t \in T \\ t \le i}} B(i_1 - t_1, \ldots, i_k - t_k), & \text{caso } (i_1, \ldots, i_k) \notin T, \\ 1, & \text{caso contrário,} \end{cases}$$

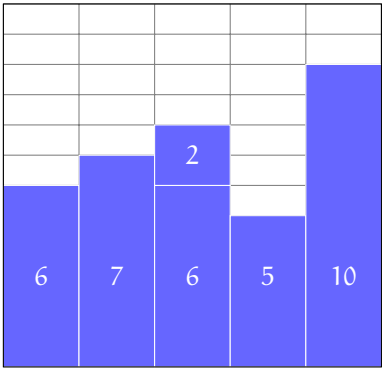
e $B(n_1, \ldots, n_k)$ é a solução do EU, com n_i o número de itens de tamanho i na entrada. A tabela B tem no máximo n^k entradas. Como o número de itens em cada padrão de alocação é no máximo n, temos $|T| \le n^k$ e logo o tempo total para preencher B é no máximo $O(n^{2k})$. ■

Corolário 3.1
Para um tamanho S fixo min-EU pode ser resolvido em tempo $O(n^{2S})$.

[2]Observe que a disjunção vazia é falsa.

Abordagem prática?

- Idéia simples: Próximo que cabe (PrC).
- Por exemplo: Itens $6, 7, 6, 2, 5, 10$ com limite 12.



Aproximação?

- Interessante: PrC é 2-aproximação.
- Observação: PrC é um algoritmo on-line.

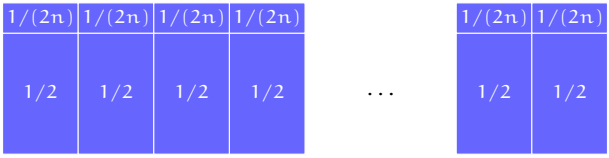
Prova. Seja B o número de contêineres usadas, $V = \sum_{i \in [n]} s_i$. Como dois contêineres consecutivos contém uma soma > 1, temos $\lfloor B/2 \rfloor < V$ e com $B/2 - 1/2 \leq \lfloor B/2 \rfloor$ ainda $B - 1 < 2V$ ou $B \leq 2V$. Mas precisamos pelo menos $\lceil V \rceil$ contêineres, logo $\lceil V \rceil \leq OPT(x)$. Portanto, $\varphi_{PrC}(x) \leq 2V \leq 2\lceil V \rceil \leq 2OPT(x)$. ■

Aproximação melhor?

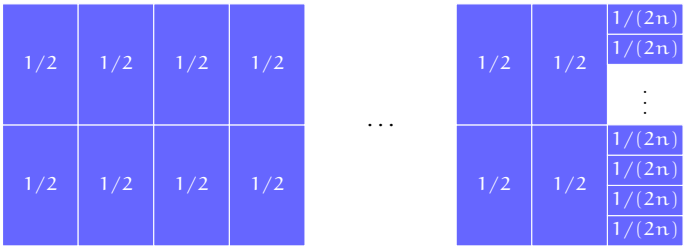
- Isso é a melhor estimativa possível para este algoritmo!
- Considere os $4n$ itens

$$\underbrace{1/2, 1/2n, 1/2, 1/2n, \ldots, 1/2, 1/2n}_{2n \text{ vezes}}$$

- O que faz PrC? $\varphi_{PrC}(x) = 2n$: contêineres com



- Ótimo: n contêineres com dois elementos de $1/2 +$ um com $2n$ elementos de $1/2n$. $\mathrm{OPT}(x) = n = 1$.



- Portanto: Assintoticamente a taxa de aproximação 2 é estrito.

Melhores estratégias

- Primeiro que cabe (PiC), on-line, com "estoque" na memória
- Primeiro que cabe em ordem decrescente: PiCD, off-line.
- Taxa de aproximação?

$$\varphi_{\mathrm{PiC}}(x) \leq \lceil 1.7\mathrm{OPT}(x) \rceil$$
$$\varphi_{\mathrm{PiCD}}(x) \leq 1.5\mathrm{OPT}(x) + 1$$

Prova. (Da segunda taxa de aproximação.) Considere a partição $A \cup B \cup C \cup D = \{v_1, \ldots, v_n\}$ com

$$A = \{v_i \mid v_i > 2/3\}$$
$$B = \{v_i \mid 2/3 \geq v_i > 1/2\}$$
$$C = \{v_i \mid 1/2 \geq v_i > 1/3\}$$
$$D = \{v_i \mid 1/3 \geq v_i\}$$

PiCD primeiro vai abrir $|A|$ contêineres com os itens do tipo A e depois $|B|$ contêineres com os itens do tipo B. Temos que analisar o que acontece com os itens em C e D.

Supondo que um contêiner contém somente itens do tipo D, os outros contêineres tem espaço livre menos que 1/3, senão seria possível distribuir os itens do tipo D para outros contêineres. Portanto, nesse caso

$$B \leq \left\lceil \frac{V}{2/3} \right\rceil \leq 3/2V + 1 \leq 3/2\mathrm{OPT}(x) + 1.$$

Caso contrário (nenhum contêiner contém somente itens tipo D), PiCD encontra a solução ótima. Isso pode ser justificado pelas seguintes observações:

1) O número de contêineres sem itens tipo D é o mesmo (eles são os últimos distribuídos em não abrem um novo contêiner). Logo é suficiente mostrar

$$\varphi_{\mathrm{PiCD}}(x \setminus D) = \mathrm{OPT}(x \setminus D).$$

2) Os itens tipo A não importam: Sem itens D, nenhum outro item cabe junto com um item do tipo A. Logo:

$$\varphi_{\mathrm{PiCD}}(x \setminus D) = |A| + \varphi_{\mathrm{PiCD}}(x \setminus (A \cup D)).$$

3) O melhor caso para os restantes itens são *pares* de elementos em B e C: Nessa situação, PiCD encontra a solução ótima.

■

Garantia ou aproximação melhor?

- Johnson (1973, Tese de doutorado)

$$\varphi_{\mathrm{PiCD}}(x) \leq 11/9\,\mathrm{OPT}(x) + 4$$

- Baker (1985)

$$\varphi_{\mathrm{PiCD}}(x) \leq 11/9\,\mathrm{OPT}(x) + 3$$

- Uma variante de PiCD (Johnson e Garey, 1985):

$$\varphi_{\mathrm{PiCDM}}(x) \leq 71/60\,\mathrm{OPT}(x) + 31/6$$

3.8.1. Um esquema de aproximação assintótico para min-EU

Duas ideias permitem aproximar min-EU em $(1+\epsilon)\mathrm{OPT}(I)+1$ para $\epsilon \in (0, 1]$.

Ideia 1: Arredondamento Para uma instância I, define uma instância R arredondada como segue:

1. Ordene os itens de forma não-decrescente e forma grupos de k itens.
2. Substitui o tamanho de cada item pelo tamanho do maior elemento no seu grupo.

{lem

Lema 3.1
Para uma instância I e a instância R arredondada temos

$$\mathrm{OPT}(R) \leq \mathrm{OPT}(I) + k$$

Prova. Supõe que temos uma solução ótima para I. Os itens do i-ésimo grupo de R cabem nos lugares dos itens do $i+1$-ésimo grupo dessa solução. Para o último grupo de R temos que abrir no máximo k contêineres. ■

:bp-ptas2}

Ideia 2: Descartando itens menores

Lema 3.2
Supõe temos temos um empacotamento para itens de tamanho maior que s_0 em B contêineres. Então existe um empacotamento de todos itens com no máximo

$$\max\Big\{B, \sum_{i\in[n]} s_i/(S-s_0)+1\Big\}$$

contêineres.

Prova. Empacota os itens menores gulosamente no primeiro contêiner com espaço suficiente. Sem abrir um novo contêiner o limite é obviamente correto. Caso contrário, supõe que precisamos B' contêineres. $B'-1$ contêineres contém itens de tamanho total mais que $S-s_0$. A ocupação total W deles tem que ser menor que o tamanho total dos itens, logo

$$(B'-1)(S-s_0) \leq W \leq \sum_{i\in[n]} s_i.$$

■

Juntando as ideias

Teorema 3.10
Para $\epsilon \in (0,1]$ podemos encontrar um empacotamento usando no máximo $(1+\epsilon)\mathrm{OPT}(I)+1$ contêineres em tempo $O(n^{16/\epsilon^2})$.

Prova. O algoritmo tem dois passos:

1. Empacota todos itens de tamanho maior que $s_0 = \lceil \epsilon/2\, S \rceil$ usando arredondamento.

2. Empacota os itens menores depois.

Seja I' a instância com os $n' \leq n$ itens maiores. No primeiro passo, formamos grupos com $\lfloor n'\epsilon^2/4 \rfloor$ itens. Isso resulta em no máximo

$$\frac{n'}{\lfloor n'\epsilon^2/4 \rfloor} \leq \frac{2n'}{n'\epsilon^2/4} = \frac{8}{\epsilon^2}$$

grupos. (A primeira desigualdade usa $\lfloor x \rfloor \geq x/2$ para $x \geq 1$. Podemos supor que $n'\epsilon^2/4 \geq 1$, i.e. $n' \geq 4/\epsilon^2$. Caso contrário podemos empacotar os itens em tempo constante usando a proposição 3.6.)
Arredondando essa instância de acordo com lema 3.1 podemos obter uma solução em tempo $O(n^{16/\epsilon^2})$ pela proposição 3.6. Sabemos que $\mathrm{OPT}(I') \geq n' \lceil \epsilon/2\, S \rceil / S \geq n'\epsilon/2$. Logo temos uma solução com no máximo

$$\mathrm{OPT}(I') + \lfloor n\epsilon^2/4 \rfloor \leq \mathrm{OPT}(I') + n'\epsilon^2/4 \leq (1+\epsilon/2)\mathrm{OPT}(I') \leq (1+\epsilon/2)\mathrm{OPT}(I)$$

contêineres.
O segundo passo, pelo lema 3.2, produz um empacotamento com no máximo

$$\max\left\{(1+\epsilon/2)\mathrm{OPT}(I), \sum_{i\in[n]} s_i/(S-s_0) + 1\right\}$$

contêineres, mas

$$\frac{\sum_{i\in[n]} s_i}{S-s_0} \leq \frac{\sum_{i\in[n]} s_i}{S(1-\epsilon/2)} \leq \frac{\mathrm{OPT}(I)}{1-\epsilon/2} \leq (1+\epsilon)\mathrm{OPT}(I).$$

■

- Give all the examples mentioned on the slides and some extra examples which illustrate the underlying techniques.

- Jeffrey John Hollis and John Kenneth Montague Moody?

- Incorporate some of the stuff in Johnson's "The many limits of approximation" column.

3.9. Aproximando problemas de sequênciamento

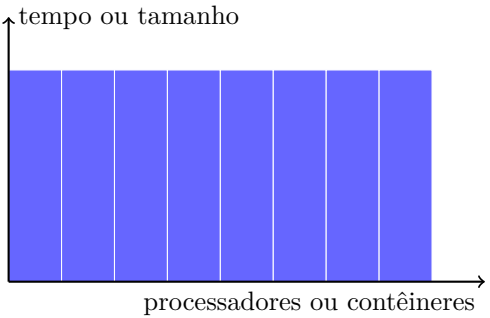
Problemas de sequênciamento recebem nomes da forma

$$\alpha \mid \beta \mid \gamma$$

com campos

Máquina α	
1	Um processador
P	Processadores paralelos
Q	Processadores relacionados
R	Processadores arbitrários
Restrições β	
D_i	Prazo máximo (deadline)
d_i	Prazo previsto (due dates)
r_i	Tempo de liberação (release time)
$p_i = p$	Tempo uniforme p
prec	Precedências
Função objetivo γ	
C_{max}	Maior tempo de término (maximum completion time)
$\sum_i C_i$	Tempo de término total (total completion time)
L_i	Atraso (lateness) $C_i - d_i$
T_i	Tardiness $\max\{L_i, 0\}$

Relação com empacotamento unidimensional:



- Empacotamento unidimensional: Dado C_{max} minimiza o número de processadores.
- $P \parallel C_{max}$: Dado um número de contêineres, minimiza o tamanho dos contêineres.

SEQUENCIAMENTO EM PROCESSORES PARALELOS ($P \parallel C_{max}$)

Entrada O número m de processadores e n tarefas com tempo de execução p_i, $i \in [n]$.

Solução Um *sequenciamento*, definido por uma alocação $M_1 \dot\cup \cdots \dot\cup M_m = [n]$ das tarefas às máquinas.

Objetivo Minimizar o *makespan* (tempo de término) $C_{max} = \max_{j\in[m]} C_j$, com $C_j = \sum_{i\in M_j} p_i$ o tempo de término da máquina j.

Fato 3.4
O problema $P \parallel C_{max}$ é fortemente NP-completo.

Um limite inferior para $C^*_{max} = OPT$ é

$$LB = \max\Big\{\max_{i\in[n]} p_i, \sum_{i\in[n]} p_i/m\Big\}.$$

Uma classe de algoritmos gulosos para este problema são os algoritmos de *sequenciamento em lista* (inglês: list scheduling). Eles processam as tarefas em alguma ordem, e alocam a tarefa atual sempre à máquina de menor tempo de término atual.

{prop:list

Proposição 3.7
Sequenciamento em lista com ordem arbitrária permite uma $2-1/m$-aproximação em tempo $O(n \log n)$.

Prova. Seja C_{max} o resultado do sequenciamento em lista. Considera uma máquina com tempo de término C_{max}. Seja j a última tarefa alocada nessa máquina e C o término da máquina antes de alocar tarefa j. Logo,

$$C_{max} = C + p_j \le \sum_{i\in[j-1]} p_i/m + p_j \le \sum_{i\in[n]} p_i/m - p_j/m + p_j$$
$$\le LB + (1-1/m)LB = (2-1/m)LB \le (2-1/m)C^*_{max}.$$

A primeira desigualdade é correta, porque alocando tarefa j a máquina tem tempo de término mínimo. Usando uma fila de prioridade a máquina com o menor tempo de término pode ser encontrada em tempo $O(\log n)$. ■

{obs:limit

Observação 3.4
Pela prova da proposição 3.7 temos

$$LB \le C^*_{max} \le 2LB.$$

◊

O que podemos ganhar com algoritmos off-line? Uma abordagem é ordenar as tarefas por tempo execução não-crescente e aplicar o algoritmo gulos. Essa abordagem é chamada LPT (largest processing time).

Proposição 3.8
LPT é uma $4/3 - m/3$-aproximação em tempo $O(n \log n)$.

Prova. Seja $p_1 \geq p_2 \geq \cdots \geq p_n$ e supõe que isso é o menor contra-exemplo em que o algoritmo retorne $C_{max} > (4/3 - m/3)C^*_{max}$. Não é possível que a alocação do item $j < n$ resulta numa máquina com tempo de término C_{max}, porque $p_1, \ldots, p_j$ seria um contra-exemplo menor (mesmo C_{max}, menor C^*_{max}). Logo a alocação de p_n define o resultado C_{max}.
Caso $p_n \leq C^*_{max}/3$ pela prova da proposição 3.7 temos $C_{max} \leq (4/3 - m/3)C^*_{max}$, uma contradição. Mas caso $p_n > C^*_{max}/3$ todas tarefas possuem tempo de execução pelo menos $C^*_{max}/3$ e no máximo duas podem ser executadas em cada máquina. Logo $C_{max} \leq 2/3C^*_{max}$, outra contradição. ■

3.9.1. Um esquema de aproximação para $P \parallel C_{max}$

Pela observação 3.4 podemos reduzir o $P \parallel C_{max}$ para o empacotamento unidimensional via uma busca binária no intervalo $[LB, 2LB]$. Pela proposição 3.5 isso é possível em tempo $O(\log LB\ mn(2LB + 1)^m)$.
Com mais cuidado a observação permite um esquema de aproximação em tempo polinomial assintótico: similar com o esquema de aproximação para empacotamento unidimensional, vamos *remover elementos menores* e *arredondar* a instância.

Algoritmo 3.6 (Sequencia)
Entrada Uma instância I de $P \parallel C_{max}$, um término máximo C e um parâmetro de qualidade ϵ.

```
1  Sequencia(I,C,ε):=
2    remove as tarefas menores com p_j < εC, j ∈ [n]
3    arredonda cada p_j ∈ [εC(1+ε)^i, εC(1+ε)^{i+1}) para algum i
         para p'_j = εC(1+ε)^i
4    resolve a instância arredondada com programação
         dinâmica (proposição 3.6)
5    empacota os itens menores gulosamente, usando novas
         máquinas para manter o término (1+ε)C
```

lem:pcmax}

Lema 3.3
O algoritmo Sequencia gera um sequenciamento que termina em no máximo $(1 + \epsilon)C$ em tempo $O(n^{2^{\lceil \log_{1+\epsilon} 1/\epsilon \rceil}})$. Ele não usa mais máquinas que o mínimo necessário para executar as tarefas com término C

Prova. Para cada intervalo válido temos $\epsilon C(1+\epsilon)^i \leq C$, logo o número de intervalos é no máximo $k = \lceil \log_{1+\epsilon} 1/\epsilon \rceil$. O valor k também é um limite para o número de valores p'_j distintos e pela proposição 3.6 o terceiro passo resolve a instância arredondada em tempo $O(n^{2k})$. Essa solução com os itens de tamanho original termina em no máximo $(1+\epsilon)C$, porque $p_j/p'_j < 1+\epsilon$. O número mínimo de máquinas para executar as tarefas em tempo C é o valor $m := \text{min-EU}(C, (p_j)_{j\in[n]})$ do problema de empacotamento unidimensional correspondente. Caso o último passo do algoritmo não usa novas máquinas ele precisa $\leq m$ máquinas, porque a instância arredondada foi resolvida exatamente. Caso contrário, uma tarefa com tempo de execução menor que ϵC não cabe nenhuma máquina, e todas máquinas usadas tem tempo de término mais que C. Logo o empacotamento ótimo com término C tem que usar pelo menos o mesmo número de máquinas. ■

{prop:pcma

Proposição 3.9
O resultado da busca binária usando o algoritmo Sequencia $C_{\max} = \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\}$ é no máximo $C^*_{\max}$.

Prova. Com $\text{Sequencia}(I, C, \epsilon) \leq \text{min-EU}(C, (p_i)_{i\in[n]})$ temos

$$\begin{aligned} C_{\max} &= \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\} \\ &\leq \min\{C \in [LB, 2LB] \mid \text{min-EU}(C, (p_i)_{i\in[n]}) \leq m\} \\ &= C^*_{\max} \end{aligned}$$

■

Teorema 3.11
A busca binária usando o algoritmo Sequencia para determinar determina um sequenciamento em tempo $O(n^{2\lceil \log_{1+\epsilon} 1/\epsilon \rceil} \log LB)$ de término máximo $(1+\epsilon)C^*_{\max}$.

Prova. Pelo lema 3.3 e proposição 3.9. ■

3.10. Programação inteira para aproximação

A programação linear é uma das técnicas mais úteis para construção de algoritmos de aproximação.

- Primal-dual.
- Arredondamento e arredondamento iterado
- “Dual fitting”.

Lembrança Temos um programa linear *primal*

$$\begin{aligned} \textbf{minimiza} \quad & c^t x \\ \textbf{sujeito a} \quad & Ax \leq b, \\ & x \geq 0. \end{aligned} \tag{P}$$

{prima

e um *dual* correspondente

$$\begin{aligned} \textbf{maximiza} \quad & y b^t \\ \textbf{sujeito a} \quad & A^t y \leq c \\ & y \geq 0. \end{aligned} \tag{D}$$

Cada solução do primal é maior que cada solução do dual pelo teorema fraco de dualidade

$$c^t x \leq b^t y,$$

e os valores das soluções ótimas (caso existem) são iguais pelo teorema forte de dualidade

$$c^t x^* = b^t y^*.$$

O teorema de folgas complementares relaciona as variáveis de um sistema com as folgas do outro:

- Condições primais: ou $x_j = 0$ ou $a^{(j)^t} y = c_j$.
- Condições duais: ou $y_i = 0$ ou $a_{(i)} x = b_i$.

Na programação inteira, o teorema forte não é mais válido. Entre o primal (P) e a versão inteira (PI) podemos definir o *gap de integralidade*

$$\sup_I \frac{\mathsf{OPT}_{\mathsf{PI}}(I)}{\mathsf{OPT}(I)}$$

para instâncias $I = (A, b, c)$ no caso de minimização.
Relaxando as condições primais e duais obtemos

- Condições primais α-apertados: ou $x_j = 0$ ou $c_j/\alpha \leq a^{(j)^t} y \leq c_j$ $(c/\alpha \leq A^t y \leq c)$
- Condições duais β-apertados: ou $y_i = 0$ ou $\beta b_i \geq a_{(i)} x \geq b_i$ $(\beta b \geq Ax \geq b)$.

Relaxar as condição é útil porque para um par de soluções primais x e duais y temos $c^t x \leq \alpha\beta b^t y$. **Prova.**

$$c^t x \leq \alpha A^t y x \leq \alpha\beta y b$$

■

Por consequência, x é uma $\alpha\beta$-aproximação do problema (com testemunho y). Isto leva a uma schema generica de um algoritmo de aproximação usando programação linear:

```
1  x := 0 // Primal: inviável
2  y := 0 // Dual: viável
3  do Até o primal é viável
4    // (1) Melhorar dual
5    Incremente alguma variável dual até ela é apertada
6    // (2) Viabilizar o primal
7    Incrementar uma variável primal, indicada pela restrição dua
8  end
```

3.11. Exercícios

{ex:apr1}

Exercício 3.1

Por que um subgrafo conexo de menor custo sempre é uma árvore?

{ex:apr2}

Exercício 3.2

Mostra que o número de vértices com grau ímpar num grafo sempre é par.

{ex:apr3}

Exercício 3.3

Um aluno propõe a seguinte heurística para o empacotamento unidimensional: Ordene os itens em ordem crescente, coloca o item com peso máximo junto com quantas itens de peso mínimo que é possível, e depois continua com o segundo maior item, até todos itens foram colocados em bins. Temos o algoritmo

```
1  ordene itens em ordem crescente
2  m := 1; M := n
3  while (m < M) do
4    abre novo contêiner, coloca v_M, M := M − 1
5    while (v_m cabe e m < M) do
6      coloca v_m no contêiner atual
7      m := m + 1
8    end while
9  end while
```

Qual a qualidade desse algoritmo? É um algoritmo de aproximação? Caso sim, qual a taxa de aproximação dele? Caso não, por quê?

{ex:apr4}

Exercício 3.4

Prof. Rapidez propõe o seguinte pré-processamento para o algoritmo SAT-R de aproximação para MAX-SAT (página 158): Caso a instância contém claúsulas com um único literal, vamos escolher uma delas, definir uma atribuição parcial que satisfazê-la, e eliminar a variável correspondente. Repetindo esse procedimento, obtemos uma instância cujas claúsulas tem 2 ou mais literais. Assim, obtemos $l \geq 2$ na análise do algoritmo, o podemos garantir que $E[X] \geq 3n/4$, i.e. obtemos uma 4/3-aproximação.
Esta análise está correta ou não?

4. Algoritmos randomizados

Um algoritmo randomizado usa *eventos aleatórios* na sua execução. Modelos computacionais adequadas são máquinas de Turing probabilísticas – mais usadas na área de complexidade – ou máquinas RAM com um comando `random(S)` que retorne um elemento aleatório do conjunto S.
Veja alguns exemplos de probabilidades:

- Probabilidade morrer caindo da cama: $1/2 \times 10^6$ (Roach e Pieper, 2007).
- Morrer abanando a máquina de venda automática e ser espancado até a morte: 30 pessoas por ano.
- Probabilidade acertar 6 números de 60 na mega-sena: $1/50063860$.
- Probabilidade que a memória falha: em memória moderna temos 1000 FIT/MBit, i.e. 6×10^{-7} erros por segundo num memória de 256 MB.[1]
- Probabilidade que um meteorito destrói um computador em cada milissegundo: $\geq 2^{-100}$ (supondo que cada milênio ao menos um meteorito destrói uma área de $100\,m^2$).

Portanto, um algoritmo que retorna uma resposta falsa com baixa probabilidade é aceitável. Em retorno um algoritmo randomizado frequentemente é

- mais simples;
- mais eficiente: para alguns problemas, um algoritmo randomizado é o mais eficiente conhecido;
- mais robusto: algoritmos randomizados podem ser menos dependente da distribuição das entradas.
- a única alternativa: para alguns problemas, conhecemos só algoritmos randomizados.

[1] FIT é uma abreviação de "failure-in-time" e é o número de erros cada 10^9 segundos. Para saber mais sobre erros em memória veja (Terrazon, 2004).

Analysis of randomized algorithms. $T(n)$ now is a random variable. So: worst-case expected: $\max_{x||x|=n} E[T(n)]$ where the expectation is over all possible executions (i.e. there is no assumption on the distribution of inputs).

4.1. Teoria de complexidade

Classes de complexidade

TBD: Check where this fits in

A máquina de Turing probabilística Uma máquina de Turing probabilística (MTP, inglês: probabilistic Turing machine, PTM) é uma máquina de Turing com uma *fita aleatória* adicional. A execução da máquina é a mesma que normal, exceto num estado q_r especial. Nesse estado a máquina leia um símbolo da fita aleatória é executa um passo que depende somente desse símbolo e depois avança a cabeça da fita aleatória um para a direita (Arora e Barak, 2009).

Definição 4.1
Seja Σ algum alfabeto e $R(\alpha, \beta)$ a classe de linguagens $L \subseteq \Sigma^*$ tal que existe um algoritmo de decisão em tempo polinomial A que satisfaz

- $x \in L \Rightarrow \Pr(A(x) = \text{sim}) \geq \alpha$.
- $x \notin L \Rightarrow \Pr(A(x) = \text{não}) \geq \beta$.

(A probabilidade é sobre todas sequências de bits aleatórios r. Como o algoritmo executa em tempo polinomial no tamanho da entrada $|x|$, o número de bits aleatórios $|r|$ é polinomial em $|x|$ também.)
Com isso podemos definir

- a classe $RP := R(1/2, 1)$ (randomized polynomial), dos problemas que possuem um algoritmo com erro unilateral (no lado do "sim"); a classe $co - RP = R(1, 1/2)$ consiste dos problemas com erro no lado de "não";
- a classe $ZPP := RP \cap co - RP$ (zero-error probabilistic polynomial) dos problemas que possuem algoritmo randomizado sem erro;
- a classe $PP := \bigcup_{\epsilon \in (0, 1/2]} R(1/2 + \epsilon, 1/2 + \epsilon)$ (probabilistic polynomial), dos problemas com erro $1/2 + \epsilon$ nos dois lados; e

- a classe $\mathsf{BPP} := \mathsf{R}(2/3, 2/3)$ (bounded-error probabilistic polynomial), dos problemas com erro $1/3$ nos dois lados.

> RP may have false negatives, $\mathsf{co}-\mathsf{RP}$ may have false positives.

Algoritmos que respondem corretamente somente com uma certa probabilidade também são chamados do tipo *Monte Carlo*, enquanto algoritmos que usam randomização somente internamente, mas respondem sempre corretamente são do tipo *Las Vegas*.

Exemplo 4.1 (Teste de identidade de polinômios)
Dado dois polinômios $p(x)$ e $q(x)$ de grau máximo d, como saber se $p(x) \equiv q(x)$? Caso temos os dois na forma canônica $p(x) = \sum_{0 \leq i \leq d} p_i x^i$ ou na forma fatorada $p(x) = \prod_{1 \leq i \leq d}(x - r_i)$ isso é simples responder por comparação de coeficientes em tempo $O(n)$. E caso contrário? Converter para a forma canônica pode custar $\Theta(d^2)$ multiplicações. Uma abordagem randomizada é vantajosa, se podemos avaliar o polinômio mais rápido (por exemplo em $O(d)$):

```
1 identico(p,q) :=
2   Seleciona um número aleatório r no intervalo [1,100d].
3   Caso p(r) = q(r) retorne ``sim''.
4   Caso p(r) ≠ q(r) retorne ``não''.
```

Caso $p(x) \equiv q(x)$, o algoritmo responde "sim" com certeza. Caso contrário a resposta pode ser errada, se $p(r) = q(r)$ por acaso. Qual a probabilidade disso? $p(x) - q(x)$ é um polinômio de grau d e possui no máximo d raízes. Portanto, a probabilidade de encontrar um r tal que $p(r) = q(r)$, caso $p \not\equiv q$ é $d/100d = 1/100$. Isso demonstra que o teste de identidade pertence à classe $\mathsf{co}-\mathsf{RP}$. ◊

Observação 4.1
É uma pergunta em aberta se o teste de identidade pertence a P. ◊

> The testing can be formulated more easily as testing for $p \equiv 0$, and the above then is just testing $p - q \equiv 0$.
> The extension to multi-variate polynomials is known as the Schwartz-Zippel lemma: for $p \in F[x_1, \ldots, x_n]$ take a finite subset of F and sample $r_i \in S$ independently and unformly. Then the probability of $p(r_i) = 0$ is $d/|S|$, for total degree d.
> Now consider the question of a bipartite graph having a perfect matching. For any bipartite graph $G = (S \dot\cup T, A)$ the corresponding *Tutte matrix*

T is defined by $t_{ij} = [ij \in A]$. The determinant of any matrix A, by Leibniz' formula is

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i \in [n]} a_{i,\sigma(i)} \tag{4.1}$$ {eq:le

We can see that every permutation $\pi \in S_n$ corresponds to a perfect matching. Now define the following polynomial over variables x_{ij}:

$$D(x_{11}, \ldots, x_{nn}) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i \in [n]} t_{ij} x_{i,\sigma(i)}. \tag{4.2}$$

Since, for every matching that is not perfect the corresponding term in D is 0, we obtain

$$D(x_{11}, \ldots, x_{nn}) = \sum_{\sigma \in P} \operatorname{sgn}(\sigma) \prod_{i \in [n]} x_{i,\sigma(i)}. \tag{4.3}$$ {eq:le

where P are all permutations that correspond to perfect matchings in G. This makes clear that if G has no perfect matching then $D \equiv 0$. On the other hand, if at least one perfect matching exists the corresponding term does not vanish, and since no other term can cancel it, $D \not\equiv 0$. Thus the problem of checking for the existence of a perfect matching can be reduced to polynomial identity testing.

[More concretely, we need to choose some values x_{ij} and compute the resulting determinant. This alone is too hard; Kabanets mentions a parallel algorithm. There's also a point of the numerical stability, but since the degree is at most n, Chawla argues that for prime $p \geq 2n$ we can compute all in Z_p, which is not clear to me, but I suspect is just to bound the possible coefficients. This also makes the problem not easier. Chawla is also interesting since he goes on to show an algorithm to compute the matching based on testing of existence.)

[This is partially based on Kabanets, Lecture 2, which is very sketchy, and also on Chawla, Lecture 3, which is better.

Exemplo 4.2 (Freivalds' algorithm)
For given matrices $A, B, C \in \mathbb{R}^{n \times n}$ we want to test if $AB = C$. With standard matrix multiplication this can be done in time $O(n^3)$, with better algorithms we may get down to $O(n^\omega)$. Freivalds proposal is a Monte Carlo algorithm in $\mathsf{co-RP}$ with probability of $1/2$ for a false

positive. It works as follows.
Let $D = AB - C$, $p = Dr$, and note that $Dr = (AB - C)r = A(Br) - Cr$ can be computed in time $O(n^2)$. Clearly, if there is no error then $D = 0$ and we have $p = 0$ for all r. Otherwise there is some element $d_{ij} \neq 0$, and thus $p_i = d_{i\bullet}r$ could be non-zero. Now choose $r \in \{0,1\}^n$ randomly and write

$$p_i = d_{ij}r_j + \underbrace{\sum_{k \neq j} d_{ik}r_k}_{c},$$

and consider

$$\begin{aligned}\Pr(p_i = 0) &= \underbrace{\Pr(p_i = 0 \mid c = 0)}_{=1/2}\Pr(c = 0) + \underbrace{\Pr(p_i = 0 \mid c \neq 0)}_{\leq 1/2}\Pr(c \neq 0) \\ &\leq 1/2,\end{aligned}$$

where the first equality is because $d_{ij}r_j \neq 0$ iff $r_j = 1$, and the second requires $r_j = 1$, too (but the terms not necessarily cancel, so $1/2$ is an upper bound).
There are many ways to "pimp" Freivals algorithms. First, we can increase the domain of r's elements. Even better, assuming $D \neq 0$ choose

$$r = \begin{pmatrix} 1 \\ s \\ s^2 \\ \vdots \\ s^{n-1} \end{pmatrix} \tag{4.4}$$

for some random $s \in \mathbb{R}$. As above assume $d_{ij} \neq 0$. Then $p_i = d_{i\bullet}r$ can be seen as a polynomial in s of degree at most $n-1$, and thus has at most $n-1$ roots. That means there are at most $n-1$ values $s \in \mathbb{R}$ such that $p_i = 0$. Therefore the probability of a false positive is 0. ◊

Exemplo 4.3 (Welzl's algorithm)
(First introduced in 2022/2 as an example.)
Given a set of points P in $\mathbb{R}^2$, find the smallest enclosing disc $\mathrm{md}(P)$ defined by three points. We define $\mathrm{md}(P) = P$ for $|P| \leq 3$. We do the following. Let $W(P, R)$ be the smallest enclosing disc of P, where R are known to lie on the boundary.

```
1  W(P,R) :=
2    if P = ∅ or |R| = 3: return the solution.
3    choose a random p ∈ P
4    D := W(P − {p}, R) // assume p ∉ R
5    if p ∈ D: return D
6    return W(P − {p}, R ∪ {p}) // p ∈ R
```

◊

Let $n = |P|$ and $3 - j = |R|$. Then the probability of making an error in the first recursive call is j/n. That gives an expected time of

$$t_j(n) = t_j(n-1) + 1 + j/n t_{j-1}(n-1)$$

where we define the base case $t_0(n) = 0$. If is easy to show that $t_j(n) \leq c_j n$ where $c_1 = 1$, $c_2 = 3$, $c_3 = 10$.
[Namely assuming $t_1(n) \leq c_1 n$

$$t_1(n) \leq t_1(n-1) + 1 + \underbrace{1/n t_0(n-1)}_{0} \leq c_1(n-1) + 1 = c_1 n + \underbrace{-c_1 + 1}_{\leq 0}$$

where the last condition gives $c_1 \geq 1$; assuming $t_2(n) \leq c_2 n$

$$\begin{aligned} t_2(n) &\leq t_2(n-1) + 1 + 2/n t_1(n-1) \leq c_2(n-1) + 1 + 2/n c_1(n-1) \\ &\leq c_2 n \underbrace{-c_2 + 1 + 2/n c_1(n-1)}_{\leq 0} \end{aligned}$$

where the last condition is satifisfied for $c_2 \geq 1 + 2/n c_1(n-1)$ which holds for $c_2 \geq 3$, and assuming $t_3(n) \leq c_3 n$

$$\begin{aligned} t_3(n) &\leq t_3(n-1) + 1 + 3/n t_2(n-1) \leq c_3(n-1) + 1 + 3/n c_2(n-1) \\ &\leq c_3 n \underbrace{-c_3 + 1 + 3/n c_2(n-1)}_{\leq 0} \end{aligned}$$

where the last condition is satisfied for $c_3 \geq 1 + 3/n c_2(n-1)$ which holds for $c_3 \geq 10$.]
We can also consider the worst case. If we call the first recursion a zig, the second a zag, the shape of the tree has any number of zigs, but at most three zags. In this case we get

$$t_j(n) \leq t_j(n-1) + 1 + t_{j-1}(n-1)$$

(note that the probability is gone), and we still get $t_1(n) \leq c_1 n$, but

TBD

4.1.1. Amplificação de probabilidades

Caso não estamos satisfeitos com a probabilidade de $1/100$ no exemplo acima, podemos repetir o algoritmo k vezes, e responder "sim" somente se todas k repetições responderam "sim". A probabilidade erradamente responder "não" para polinômios idênticos agora é $(1/100)^k$, i.e. ela diminui exponencialmente com o número de repetições.
Essa técnica é uma *amplificação* da probabilidade de obter a solução correta. Ela pode ser aplicada para melhorar a qualidade de algoritmos em todas classes "Monte Carlo". Com um número constante de repetições, obtemos uma probabilidade baixa nas classes RP, $\mathsf{co-RP}$ e BPP. Isso não se aplica a PP: é possível que ϵ diminui exponencialmente com o tamanho da instância. Um exemplo de amplificação de probabilidade encontra-se na prova do teorema 4.6.

{th:rpinv}

Teorema 4.1
$\mathsf{R}(\alpha, 1) = \mathsf{R}(\beta, 1)$ para $0 < \alpha, \beta < 1$.

Prova. Sem perda de generalidade seja $\alpha < \beta$. Claramente $\mathsf{R}(\beta, 1) \subseteq \mathsf{R}(\alpha, 1)$. Supõe que A é um algoritmo que testemunha $L \in \mathsf{R}(\alpha, 1)$. Execute A no máximo k vezes, respondendo "sim" caso A responde "sim" em alguma iteração e "não" caso contrário. Chama esse algoritmo A'. Caso $x \notin L$ temos $\Pr(A'(x) = \text{"não"}) = 1$. Caso $x \in L$ temos $\Pr(A'(x) = \text{"sim"}) \geq 1 - (1-\alpha)^k$, logo para $k \geq \ln(1-\beta)/\ln(1-\alpha)$, $\Pr(A'(x) = \text{"sim"}) \geq \beta$. ■

$$1 - (1-\alpha)^k \geq \beta \iff 1 - \beta \geq (1-\alpha)^k \iff \ln(1-\beta) \geq k \ln(1-\alpha)$$

Corolário 4.1
$\mathsf{RP} = \mathsf{R}(\alpha, 1)$ para $0 < \alpha < 1$.

{th:bppinv}

Teorema 4.2
$\mathsf{R}(\alpha, \alpha) = \mathsf{R}(\beta, \beta)$ para $1/2 < \alpha, \beta$.

Prova. Sem perda de generalidade seja $\alpha < \beta$. Claramente $\mathsf{R}(\beta, \beta) \subseteq \mathsf{R}(\alpha, \alpha)$.
Supõe que A é um algoritmo que testemunha $L \in \mathsf{R}(\alpha, \alpha)$. Executa A k vezes, responde "sim" caso a maioria de respostas obtidas foi "sim", e "não" caso contrário. Chama esse algoritmo A'. Para $x \in L$ temos

$$\Pr(A'(x) = \text{"sim"}) = \Pr(A(x) = \text{"sim"} \geq \lfloor k/2 \rfloor + 1 \text{ vezes}) \geq 1 - e^{-2k(\alpha - 1/2)^2}$$

e para $k \geq \ln(\beta-1)/2(\alpha-1/2)^2$ temos $\Pr(A'(x) = \text{“sim”}) \geq \beta$. Similarmente, para $x \notin L$ temos $\Pr(A'(x) = \text{“não”}) \geq \beta$. Logo $L \in R(\beta, \beta)$. ■

This result is via Chernoff bounds, example A.4 in CA lecture notes.

Corolário 4.2
$BPP = R(\alpha, \alpha)$ para $1/2 < \alpha$.

Observação 4.2
Os resultados acima são válidos ainda caso o erro dimiui polinomialmente com o tamanho da instância, i.e. $\alpha, \beta \geq n^{-c}$ no caso do teorema 4.1 e $\alpha, \beta \geq 1/2+n^{-c}$ no caso do teorema 4.2 para um constante c (ver por exemplo Arora e Barak (2009)). ◊

4.1.2. Relação entre as classes

Duas caracterizações alternativas de ZPP

Definição 4.2
Um algoritmo A é *honesto* se

i) ele responde ou “sim”, ou “não” ou “não sei”,

ii) $\Pr(A(x) = \text{não sei}) \leq 1/2$, e

iii) no caso ele responde, ele não erra, i.e., para x tal que $A(x) \neq$ “não sei” temos $A(x) = \text{“sim”} \iff x \in L$.

Uma linguagem é honesta caso ela possui um algoritmo honesto. Com isso também podemos falar da classe das linguagens honestas.

h:zppalt1}
▶ ZPP ⊆ H

Teorema 4.3
ZPP é a classe das linguagens honestas.

Lema 4.1
Caso $L \in ZPP$ existe um algoritmo um algoritmo honesto para L.

Prova. Para $L \in ZPP$ existem dois algoritmos $A_1 \in RP$ e $A_2 \in co-RP$. Vamos construir um algoritmo

```
1 if A1(x) = A2(x) then
2   return A1(x)
3 else if A1(x) = ``não'' e A2(x) = ``sim'' then
4   return ``não sei''
5 else if A1(x) = ``sim'' e A2(x) = ``não'' then
6   { caso impossível }
7 end if
```

O algoritmo responde corretamente "sim" e "não", porque um dos dois algoritmos não erra. Qual a probabilidade do segundo caso? Para $x \in L$, $\Pr(A_1(x) = \text{"não"} \wedge A_2(x) = \text{"sim"}) \leq 1/2 \times 1 = 1/2$. Similarmente, para $x \notin L$, $\Pr(A_1(x) = \text{"não"} \wedge A_2(x) = \text{"sim"}) \leq 1 \times 1/2 = 1/2$. ■

{lem:hones
▶ H ⊆ ZPP

Lema 4.2
Caso L possui um algoritmo honesto $L \in \mathsf{RP}$ e $L \in \mathsf{co}-\mathsf{RP}$.

Prova. Seja A um algoritmo honesto. Constrói outro algoritmo que sempre responde "não" caso A responde "não sei", e senão responde igual. No caso de $\mathsf{co}-\mathsf{RP}$ analogamente constrói um algoritmos que responde "sim" nos casos "não sei" de A. ■

Definição 4.3
Um algoritmo A é *sem falha* se ele sempre responde "sim" ou "não" corretamente em *tempo polinomial esperado*. Com isso podemos também falar de linguagens sem falha e a classe das linguagens sem falha.

{th:zppalt

Teorema 4.4
ZPP é a classe das linguagens sem falha.

Lema 4.3
▶ ZPP ⊆ SF
Caso $L \in \mathsf{ZPP}$ existe um algoritmo sem falha para L.

Prova. Sabemos que existe um algoritmo honesto para L. Repete o algoritmo honesto até encontrar um "sim" ou "não". Como o algoritmo honesto executa em tempo polinomial $p(n)$, o tempo esperado desse algoritmo ainda é polinomial:

$$\sum_{k>0} k2^{-k}p(n) \leq 2p(n)$$

■

$1/2+2/4+3/8+4/16+) \leq 2$ follows from (A.37) in the CA lecture notes.

Lema 4.4
▶ SF ⊆ ZPP
Caso L possui um algoritmo A sem falha, $L \in \mathsf{RP}$ e $L \in \mathsf{co}-\mathsf{RP}$.

Prova. Caso A tem tempo esperado $p(n)$ executa ele para um tempo $2p(n)$. Caso o algoritmo responde, temos a resposta certa. Caso contrário, responde "não sei". Pela desigualdade de Markov temos uma resposta com probabilidade $\Pr(T \geq 2p(n)) \leq p(n)/2p(n) = 1/2$. Isso mostra que existe um algoritmo honesto para L, e pelo lema 4.2 $L \in \mathsf{RP}$. O argumento para $L \in \mathsf{co}-\mathsf{RP}$ é similar. ■

Markov: $\Pr(|X| \geq a] \leq \mathbb{E}[|X|] \leq a$, see A.8 in the CA lecture notes.

Mais relações

Teorema 4.5
$\mathsf{RP} \subseteq \mathsf{NP}$ e $\mathsf{co}-\mathsf{RP} \subseteq \mathsf{co}-\mathsf{NP}$

Prova. Supõe que temos um algoritmo em RP para algum problema L. Podemos, não-deterministicamente, gerar todas sequências r de bits aleatórios e responder "sim" caso alguma execução encontra "sim". O algoritmo é correto, porque caso para um $x \notin L$, não existe uma sequência aleatória r tal que o algoritmo responde "sim". A prova do segundo caso é similar. ■

h:rpinbpp}

Teorema 4.6
$\mathsf{RP} \subseteq \mathsf{BPP}$ e $\mathsf{co}-\mathsf{RP} \subseteq \mathsf{BPP}$.

Prova. Seja A um algoritmo para $L \in \mathsf{RP}$. Constrói um algoritmo A'

```
1  if A(x) = ``não'' e A(x) = ``não'' then
2    return ``não''
3  else
4    return ``sim''
5  end if
```

Caso $x \notin L$, $\Pr(A'(x) = \text{“não”}) = \Pr(A(x) = \text{“não”} \wedge A(x) = \text{“não”}) = 1 \times 1 = 1$. Caso $x \in L$,

$$\Pr(A'(x) = \text{“sim”}) = 1 - \Pr(A'(x) = \text{“não”}) = 1 - \Pr(A(x) = \text{“não”} \wedge A(x) = \text{“não”}$$
$$\geq 1 - 1/2 \times 1/2 = 3/4 > 2/3.$$

(Observe que para k repetições de A obtemos $\Pr(A'(x) = \text{“sim”}) \geq 1 - 1/2^k$, i.e., o erro diminui exponencialmente com o número de repetições.) O argumento para $\mathsf{co}-\mathsf{RP}$ é similar. ■

Relação com a classe NP e abundância de testemunhas Lembramos que a classe NP contém problemas que permitem uma verificação de uma solução em tempo polinomial. Não-determinísticamente podemos "chutar" uma solução e verificá-la. Se o número de soluções positivas de cada instância é mais que a metade do número total de soluções, o problema pertence a RP: podemos gerar uma solução aleatória e testar se ela possui a característica desejada. Um problema desse tipo possui uma *abundância de testemunhas*. Isso demonstra a importância de algoritmos randomizados. O teste de equivalência de polinômios acima é um exemplo de abundância de testemunhas.

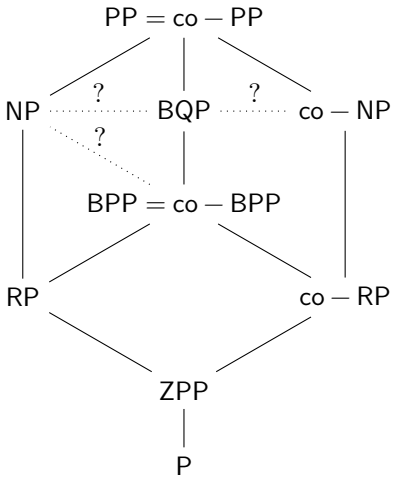


Figura 4.1.: Relações entre classes de complexidade para algoritmos randomizados.

cls}

I guess the relation of BPP to co-NP is the same: show it! Also NP ⊆ BPP is improbable.

4.2. Seleção

O algoritmo determinístico para selecionar o k-ésimo maior elemento de uma sequência não ordenada $x_1, \ldots, x_n$ discutido na seção A.1 (página 228) pode ser simplificado usando randomização: escolheremos um elemento pivô $m = x_i$ aleatório. Com isso o algoritmo A.1 fica mais simples:

{alg:selecti

Algoritmo 4.1 (Seleção randomizada)
Entrada Números $x_1, \ldots, x_n$, posição k.

Saída O k-ésimo maior número.

```
1  S(k,{x_1,...,x_n}) :=
2    if n ≤ 1
3      calcula e retorna o k-ésimo elemento
4    end if
```

```
5    m := x_i para um i ∈ [n] aleatório
6    L := {x_i | x_i < m, 1 ≤ i ≤ n}
7    R := {x_i | x_i ≥ m, 1 ≤ i ≤ n}
8    i := |L| + 1
9    if i = k then
10     return m
11   else if i > k then
12     return S(k, L)
13   else
14     return S(k − i, R)
15   end if
```

Para determinar a complexidade podemos observar que com probabilidade $1/n$ temos $|L| = i$ e $|R| = n - i$ e o caso pessimista é uma chamada recursiva com $\max\{i, n-i\}$ elementos. Logo, com custo cn para particionar o conjunto e os testes temos

$$\begin{aligned} T(n) &\leq cn + \sum_{i\in[0,n]} 1/n T(\max\{n-i, i\}) \\ &= cn + 1/n\left(\sum_{i\in[0,k]} T(n-i) + \sum_{i\in[\lceil n/2\rceil, n]} T(i)\right) \\ &= cn + 2/n \sum_{i\in[0,k]} T(n-i), \end{aligned}$$

onde usamos $k = \lfloor n/2 \rfloor$. Separando o termo $T(n)$ do lado direito obtemos

$$\begin{aligned} (1-2/n)T(n) &\leq cn + 2/n \sum_{i\in[k]} T(n-i) \\ \iff T(n) &\leq \frac{1}{n-2}\left(cn^2 + 2\sum_{i\in[k]} T(n-i)\right). \end{aligned}$$

Provaremos por indução que $T(n) \leq c'n$ para uma constante c'. Para um $n \leq n_0$ o problema pode ser claramente resolvido em tempo constante (por exemplo em $O(n_0 \log n_0)$ via ordenação). Logo, supõe que $T(i) \leq c'i$ para

$i < n$. Demonstraremos que $T(n) \leq c'n$. Temos

$$\begin{aligned} T(n) &\leq \frac{1}{n-2}\left(cn^2 + 2\sum_{i\in[k]} T(n-i)\right) \\ &\leq \frac{1}{n-2}\left(cn^2 + 2c'\sum_{i\in[k]} n-i\right) \\ &= \frac{1}{n-2}\left(cn^2 + 2c'(2n-k-1)k/2\right) \end{aligned}$$

e com $2n-k-1 = 2n - \lfloor n/2\rfloor - 1 \leq 3/2n$

$$\leq \frac{1}{n-2}(cn^2 + 3/4c'n^2) = (c + 3/4c')\frac{n^2}{n-2}$$

Para $n \geq n_0 := 16$ temos $n/(n-2) \leq 8/7$ e com um $c' > 8c$ temos

$$T(n) \leq c'(1/8 + 3/4)8/7n = c'n.$$

4.3. Corte mínimo

Corte mínimo

Entrada Grafo não-direcionado $G = (V, A)$ com pesos $c : A \to \mathbb{Z}_+$ nas arestas.

Solução Uma partição $V = S \cup \bar{S}$ onde $\bar{S} = V \setminus S$.

Objetivo Minimizar o peso do corte $\sum_{a\in A(S,\bar{S})} c_a$.

Soluções determinísticas:

- Calcular a árvore de Gomory-Hu: a aresta de menor peso define o corte mínimo.
- Calcular o corte mínimo (via fluxo máximo) entre um vértice fixo $s \in V$ e todos outros vértices: o menor corte encontrado é o corte mínimo.

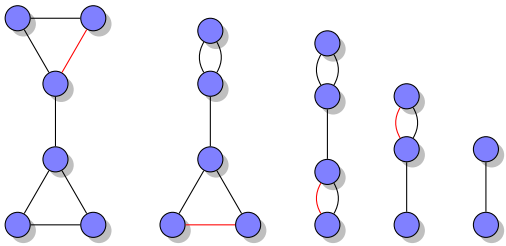
Custo em ambos casos: $O(n)$ aplicações de um algoritmo de fluxo máximo, i.e. $O(mn^2)$ usando o algoritmo de Orlin (ou $O(nm^{1+o(1)})$ com o algoritmo de Chen et al. (2022)).

> Gomory-Hu is simply this. Let a *fat vertex* V represent all vertices V. While there's a fat vertex C, choose two vertices $u, u' \in V$, compute a minimum uu'-cut, and separate V into the parts U and U'. This gives a tree T. Now: the value of each minimum uv-cut equals the value of the minimum uv-cut in T, i.e. the lightest edge in the single uv-path. Furthermore, by removing this lightest edge, we recover the parts.

Solução randomizada para pesos unitários No que segue supomos que os pesos são unitários, i.e. $c_a = 1$ para $a \in A$. Uma abordagem simples é baseada na seguinte observação: se escolhemos uma aresta que não faz parte de um corte mínimo, e contraímo-la (i.e. identificamos os vértices adjacentes), obtemos um grafo menor, que ainda contém o corte mínimo. Se escolhemos uma aresta aleatoriamente, a probabilidade de por acaso escolher uma aresta de um corte mínimo é baixa.

```
1  cmr(G) :=
2    while G possui mais que dois vértices
3      escolhe uma aresta {u,v} aleatoriamente
4      identifica u e v em G
5    end while
6    return o corte definido pelos dois vértices em G
```

Exemplo 4.4
Uma sequencia de contrações (das arestas vermelhas).



◊

Dizemos que uma aresta "sobrevive" uma contração, caso ele não foi contraído.

tsurvival}

Lema 4.5
A probabilidade que os k arestas de um corte mínimo sobrevivem $n - n'$ contrações (de n para n' vértices) é $\Omega((n'/n)^2)$.

Prova. Como o corte mínimo é k, cada vértice possui grau pelo menos k, e portanto o número de arestas após da iteração $0 \leq i < n - n'$ e maior ou igual a $k(n-i)/2$ (com a convenção que a "iteração 0" produz o grafo inicial). Supondo que as k arestas do corte mínimo sobreviveram a iteração i, a probabilidade de não sobreviver a próxima iteração é pelo menos $k/(k(n-i)/2) = 2/(n-i)$. Logo, a probabilidade do corte sobreviver $n - n'$ iterações é pelo menos

$$\prod_{0 \leq i < n-n'} 1 - \frac{2}{n-i} = \prod_{0 \leq i < n-n'} \frac{n-i-2}{n-i}$$
$$= \frac{(n-2)(n-3)\cdots(n'-1)}{n(n-1)\cdots(n'+1)} = \frac{n'(n'-1)}{n(n-1)} = \Omega((n'/n)^2).$$

■

Teorema 4.7
Dado um corte mínimo C de tamanho k, a probabilidade do algoritmo cmr retornar C é $\Omega(n^{-2})$.

Prova. Caso o grafo possui n vértices, o algoritmo termina em $n-2$ iterações: podemos aplicar o lema acima com $n' = 2$. ■

Observação 4.3
O que acontece se repetimos o algoritmo algumas vezes? Seja C_i uma variável que indica se o corte mínimo foi encontrado na repetição i. Temos $\Pr(C_i = 1) \geq 2n^{-2}$ e portanto $\Pr(C_i = 0) \leq 1 - 2n^{-2}$. Para kn^2 repetições, vamos encontrar $C = \sum C_i$ cortes mínimos com probabilidade

$$\Pr(C \geq 1) = 1 - \Pr(C = 0) \geq 1 - (1 - 2n^{-2})^{kn^2} \geq 1 - e^{-2k}.$$

Para $k = \log n$ obtemos $\Pr(C \geq 1) \geq 1 - n^{-2}$. ◊

Since $\exp(x) \geq (1 + x/n)^n$ for all $n > 0$ and x, for $x = -2k$ and $n = kn^2$ we have

$$\exp(-2k) \geq (1 - 2k/kn^2)^{kn^2} = (1 - 2n^{-2})^{kn^2}.$$

Logo, ao repetir o algoritmo $n^2 \log n$ vezes e retornar o menor corte encontrado, achamos o corte mínimo com probabilidade razoável. Se a implementação realiza uma contração em tempo $O(n)$ o algoritmo possui complexidade $O(n^2)$ e com as repetições em total $O(n^4 \log n)$.

Implementação de contrações Para garantir a complexidade acima, uma contração tem que ser implementada em $O(n)$. Isso é possível tanto na representação por uma matriz de adjacência, quanto na representação pela listas de adjacência. A contração de dois vértices adjacentes resulta em um novo vértice, que é adjacente aos vizinhos dos dois. Na contração arestas de um vértice com si mesmo são removidas. Múltiplas arestas entre dois vértices tem que ser mantidas para garantir o Lema 4.5.

Um algoritmo melhor (Karger e Stein, 1996) O problema principal com o algoritmo acima é que nas últimas iterações, a probabilidade de contrair uma aresta do corte mínimo é grande. Para resolver esse problema, executaremos o algoritmo duas vezes para instâncias menores, para aumentar a probabilidade de não contrair o corte mínimo. Define $f(n) = \left\lceil 1 + n/\sqrt{2} \right\rceil$.

```
1  cmr2(G) :=
2    if (G possui menos que 6 vértices)
3      determina o corte mínimo C por exaustão
4      return C
5    else
6      n' := f(n)
7      seja G1 o resultado de n - n' contrações em G
8      seja G2 o resultado de n - n' contrações em G
9      C1 :=cmr2(G1)
10     C2 :=cmr2(G2)
11     return o menor dos dois cortes C1 e C2
12   end if
```

Esse algoritmo possui complexidade de tempo $O(n^2 \log n)$ e encontra um corte mínimo com probabilidade $\Omega(1/\log n)$.

Lema 4.6
A probabilidade de um corte mínimo sobreviver $n - f(n)$ contrações é pelo menos $1/2$.

Prova. Pelo lema 4.5 a probabilidade é pelo menos

$$\frac{f(n)(f(n)-1)}{n(n-1)} \geq \frac{(1+n/\sqrt{2})(n/\sqrt{2})}{n(n-1)} = \frac{\sqrt{2}+n}{2(n-1)} \geq \frac{n}{2n} = \frac{1}{2}.$$

■

Seja $P(n)$ a probabilidade que um corte com k arestas sobrevive caso o grafo

possui n vértices. Temos

$$\begin{aligned}
\Pr(\text{o corte sobrevive em } G_1) &\geq 1/2\, P(f(n)) \\
\Pr(\text{o corte sobrevive em } G_2) &\geq 1/2\, P(f(n)) \\
\Pr(\text{o corte não sobrevive em } G_1 \text{ nem } G_2) &\leq (1 - 1/2P(f(n)))^2 \\
P(n) = \Pr(\text{o corte sobrevive em } G_1 \text{ ou } G_2) &\geq 1 - (1 - 1/2P(f(n)))^2 \\
&= P(f(n)) - 1/4P(f(n))^2
\end{aligned}$$

Para resolver essa recorrência, define $Q(k) = P(\sqrt{2}^k)$ com base $Q(0) = 1$ para obter a recorrência simplificada

$$\begin{aligned}
Q(k+1) = P(\sqrt{2}^{k+1}) = P(\left\lceil 1 + \sqrt{2}^k \right\rceil) - 1/4P(\left\lceil 1 + \sqrt{2}^k \right\rceil)^2 \\
\approx P(\sqrt{2}^k) - P(\sqrt{2}^k)^2/4 = Q(k) - Q(k)^2/4
\end{aligned}$$

e depois $R(k) = 4/Q(k) - 1$ com base $R(0) = 3$ para obter

$$\frac{4}{R(k+1)+1} = \frac{4}{R(k)+1} - \frac{4}{(R(k)+1)^2} \iff R(k+1) = R(k) + 1 + 1/R(k).$$

> The above is, BTW, an example of a recurrence that Akra-Bazzi can't handle.

$R(k)$ satisfaz

$$k < R(k) < k + H_{k-1} + 3$$

Prova. Por indução. Para $k = 1$ temos $1 < R(1) = 13/3 < 1 + H_0 + 3 = 5$. Caso a HI está satisfeito, temos

$$\begin{aligned}
R(k+1) &= R(k) + 1 + 1/R(k) > R(k) + 1 > k + 1 \\
R(k+1) &= R(k) + 1 + 1/R(k) < k + H_{k-1} + 3 + 1 + 1/k = (k+1) + H_k + 3
\end{aligned}$$

■

Logo, $R(k) = k + \Theta(\log k)$, e com isso $Q(k) = \Theta(1/k)$ e finalmente $P(n) = \Theta(1/\log n)$.
Para determinar a complexidade do algoritmo cmr2 observe que temos $O(\log n)$ níveis de recursão e cada contração pode ser feita em tempo $O(n^2)$, portanto

$$T_n = 2T(f(n)) + O(n^2).$$

Aplicando o teorema de Akra-Bazzi obtemos a equação característica $2(1/\sqrt{2})^p = 1$ com solução $p = 2$ e

$$T_n \in \Theta(n^2(1 + \int_1^n \frac{cu^2}{u^3} du)) = \Theta(n^2 \log n).$$

Check and cite Karger,Stein, A New Approach to the Minimum Cut Problem

Generalized Karger-Stein amplification. Consider a randomized algorithm constructing an object as follows. Given input f_0 apply a sequence of k operators to obtain $f_1, f_2, \ldots, f_k$, where f_k is the result (or contains it in some easily extractable form). What makes the algorithm randomized is that each step has a certain probability p_i to destroy the desired object, and we return something sub-optimal. There the probability to produce the desired object is $p = \prod_{i \in [k]} 1 - p_i$.
The amplification uses the fact that $p_1 < p_2 < \cdots < p_k$, i.e. the probability to destroy the object is higher in the later steps. If we repeat the process n times and return the "best" result found – we need to be able to evaluate the quality – we have $C = \sum_i C_i$ successes, where $P[C_i = 1] = p$ and therefore

$$P[C \geq 1] = 1 - p[C = 0] = 1 - (1 - p)^n$$

and setting $n = m/p$ we get

$$P[C \geq 1] = 1 - (1 - p)^{1/p^m} \geq 1 - e^{-m}.$$

We also can assume that for instance size $n \to \infty$ we have $p \to 0$.
Challenge: show how probing more at the lower levels increases the probability more effectively. Is this not very similar to "go with the winners"? Can we apply this to "largest path" by repeatedly contracting edges?

4.4. Teste de primalidade

Um problema importante na criptografia é encontrar números primos grandes (p.ex. RSA). Escolhendo um número n aleatório, qual a probabilidade de n ser primo?

Teorema 4.8 (Hadamard (1896), Vallée Poussin (1896))
(Teorema dos números primos.)
Para $\pi(n) = |\{p \leq n \mid p \text{ primo}\}|$ temos

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

(Em particular $\pi(n) = \Theta(n/\ln n)$.)

Portanto, a probabilidade de um número aleatório no intervalo $[2, n]$ ser primo assintoticamente é somente $1/\ln n$. Então para encontrar um número primo, temos que testar se n é primo mesmo. Observe que isso não é igual a fatoração de n. De fato, temos testes randomizados (e determinísticos) em tempo polinomial, enquanto não sabemos fatorar nesse tempo. Uma abordagem simples é testar todos os divisores:

```
1  Primo1(n) :=
2    for i = 2,3,5,7,..., ⌊√n⌋ do
3      if i|n return ``Não''
4    end for
5    return ``Sim''
```

O tamanho da entrada n é $t = \log n$ bits, portanto o número de iterações é $\Theta(\sqrt{n}) = \Theta(2^{t/2})$ e a complexidade $\Omega(2^{t/2})$ (mesmo contando o teste de divisão com $O(1)$) desse algoritmo é exponencial. Para testar a primalidade mais eficiente, usaremos uma característica particular dos números primos.

Teorema 4.9 (Fermat, Euler)
Para p primo e $a \geq 0$ temos

$$a^p \equiv a \mod p.$$

Prova. Por indução sobre a. Base: evidente. Seja $a^p \equiv a$. Temos

$$(a+1)^p = \sum_{0\leq i\leq p} \binom{p}{i} a^i$$

e para $0 < i < p$

$$p \mid \binom{p}{i} = \frac{p(p-1)\cdots(p-i+1)}{i(i-1)\cdots 1}$$

porque p é primo. Portanto $(a+1)^p \equiv a^p + 1$ e

$$(a+1)^p - (a+1) \equiv a^p + 1 - (a+1) = a^p - a \equiv 0.$$

(A última identidade é a hipótese da indução.) ■

Definição 4.4
Para $a, b \in \mathbb{Z}$ denotamos com (a, b) o maior divisor em comum (MDC) de a e b. No caso $(a, b) = 1$, a e b são números *coprimos*.

{th:

Teorema 4.10 (Divisão modulo p)
Caso p é primo e $(b, p) = 1$

$$ab \equiv cb \mod p \Rightarrow a \equiv c \mod p.$$

(Em palavras: Numa identidade modulo p podemos dividir por números coprimos com p.)

$2 \cdot 3 \equiv 7 \cdot 3 \mod 5$, so $2 \equiv 7 \mod 5$, but $2 \cdot 4 \equiv 4 \cdot 4 \mod 8$, but $2 \not\equiv 4 \mod 8$.

Prova.

$$\begin{aligned} ab \equiv cd &\iff \exists k\, ab + kp = cb \\ &\iff \exists k\, a + kp/b = c \end{aligned}$$

Como $a, c \in \mathbb{Z}$, temos $kp/b \in \mathbb{Z}$ e $b|k$ ou $b|p$. Mas $(b, p) = 1$, então $b|k$. Definindo $k' := k/b$ temos $\exists k'\, a + k'p = c$, i.e. $a \equiv c$. ■

Residual problem: do we need p to be prime here? Probably not, $(b, p) = 1$ is sufficient (otherwise even my example does not make sense). Check the Algebra book.

Logo, para p primo e $(a, p) = 1$ (em particular se $1 \leq a < p$)

eq:fermat}

$$a^{p-1} \equiv 1 \mod p. \tag{4.5}$$

Um teste melhor então é

```
1  Primo2(n) :=
2    seleciona a ∈ [1, n − 1] aleatoriamente
3    if (a, n) ≠ 1 return ``Não''
4    if a^{n−1} ≡ 1 return ``Sim''
5    return ``Não''
```

Complexidade: Uma multiplicação e divisão com $\log n$ dígitos é possível em tempo $O(\log^2 n)$. Portanto, o primeiro teste (o algoritmo de Euclides em $\log n$ passos) pode ser feito em tempo $O(\log^3 n)$ e o segundo teste (exponenciação modular) é possível implementar com $O(\log n)$ multiplicações (exercício!).

Corretude: O caso de uma resposta "Não" é certo, porque n não pode ser primo. Qual a probabilidade de falhar, i.e. do algoritmo responder "Sim", com n composto? O problema é que o algoritmo falha no caso de *números Carmichael*.

Definição 4.5
Um número composto n que satisfaz $a^{n-1} \equiv 1 \mod n$ é um *número pseudo-primo com base* a. Um *número Carmichael* é um número pseudo-primo para qualquer base a com $(a, n) = 1$.

Os primeiros números Carmichael são $561 = 3 \times 11 \times 17$, 1105 e 1729 (veja OEIS A002997). Existe um número infinito deles:

Teorema 4.11 (Alford et al. (1994))
Seja $C(n)$ o número de números Carmichael até n. Assintoticamente temos $C(n) > n^{2/7}$.

Exemplo 4.5
$C(n)$ até 10^{10} (OEIS A055553):

n	1	2	3	4	5	6	7	8	9	10
$C(10^n)$	0	0	1	7	16	43	105	255	646	1547
$\lceil (10^n)^{2/7} \rceil$	2	4	8	14	27	52	100	194	373	720

. ◊

Caso um número n não é primo, nem número de Carmichael, mais que $n/2$ dos $a \in [1, n-1]$ com $(a, n) = 1$ não satisfazem (4.5) ou seja, com probabilidade $> 1/2$ acharemos um testemunha que n é composto. O problema é que no caso de números Carmichael não temos garantia.

{th:modula

Teorema 4.12 (Raiz modular)
Para p primo temos

$$x^2 \equiv 1 \mod p \Rightarrow x \equiv \pm 1 \mod p.$$

O teste de Miller-Rabin usa essa característica para melhorar o teste acima. Podemos escrever $n - 1 = 2^t u$ para um u ímpar. Temos $a^{n-1} = (a^u)^{2^t} \equiv 1$. Portanto, se $a^{n-1} \equiv 1$,

$$\text{Ou } a^u \equiv 1 \mod p \text{ ou existe um menor } i \in [0, t] \text{ tal que } (a^u)^{2^i} \equiv 1$$

Caso p é primo, $\sqrt{(a^u)^{2^i}} = (a^u)^{2^{i-1}} \equiv -1$ pelo teorema (4.12) e a minimalidade de i (que exclui o caso $\equiv 1$). Por isso:

Definição 4.6
Um número n é um *pseudo-primo forte com base* a caso

$$\text{Ou } a^u \equiv 1 \mod p \text{ ou existe um menor } i \in [0, t-1] \text{ tal que } (a^u)^{2^i} \equiv -1 \tag{4.6}$$ {eq:

```
1  Primo3(n) :=
2    seleciona a ∈ [1,n−1] aleatoriamente
3    if (a,n) ≠ 1 return ``Não''
4    seja n−1 = 2^t u
5    if a^u ≡ 1 return ``Sim''
6    if (a^u)^{2^i} ≡ −1 para um i ∈ [0,t−1] return ``Sim''
7    return ``Não''
```

Teorema 4.13 (Monier (1980) e Rabin (1980))
Caso n e composto e ímpar, mais que 3/4 dos $a \in [1, n-1]$ com $(a, n) = 1$ não satisfazem o critério (4.6) acima.

Portanto com k testes, a probabilidade de falhar $\Pr(\text{Sim} \mid n \text{ composto}) \leq (1/4)^k = 2^{-2k}$. De fato a probabilidade é menor:

Teorema 4.14 (Damgård et al., 1993)
A probabilidade de um único teste falhar para um número com k bits é $\leq k^2 4^{2-\sqrt{k}}$.

Exemplo 4.6
Para $n \in [2^{499}, 2^{500} - 1]$ a probabilidade de não detectar um n composto com um único teste é menor que

$$499^2 \times 4^{2-\sqrt{499}} \approx 2^{-22}.$$

$\Diamond$

Teste determinístico O algoritmo pode ser convertido em um algoritmo determinístico, testando pelo menos 1/4 dos a com $(a, n) = 1$. De fato, para o menor testemunho $w(n)$ de um número n ser composto temos

eq:grhdet}

$$\text{Se o HGR é verdade: } w(n) < 2\log^2 n \tag{4.7}$$

com HGR a hipótese generalizada de Riemann (uma conjetura aberta). Supondo HGR, obtemos um algoritmo determinístico com complexidade $O(\log^5 n)$. Em 2002, Agrawal et al. (2004) descobriram um algoritmo determinístico (sem a necessidade da HGR) em tempo $\tilde{O}(\log^{12} n)$ que depois foi melhorado para $\tilde{O}(\log^6 n)$.

4.5. O problema é achar "a agulha no palheiro"

Discutir

Teorema 4.15 (Valiant-Vazirani)
Supõe que temos um algoritmo polinomial que, dado uma fórmula em forma normal conjuntiva que é satisfatível por uma única atribuição, encontra-la. (Para outras entradas o resultado do algoritmo pode ser arbitrário.) Então NP = RP.

4.6. Encontrar a mediana

Fala sobre o algoritmo randomizado de encontrar o k-ésimo elemento de uma sequencia ordenada, que é mais simples que a versão determinística. Ver p.ex. Arora/Barak, 7.2.1.

4.7. Notas

Um applet com uma implementação do teste de Miller e Rabin se encontra aqui.

4.8. Exercícios

Exercício 4.1
Encontre um primo $\mathfrak{p}$ e um valor $\mathfrak{b}$ tal que a identidade do teorema 4.10 não é correta.

Exercício 4.2
Encontre um número $\mathfrak{p}$ não primo tal que a identidade do teorema 4.12 não é correta.

5. Complexidade e algoritmos parametrizados

A complexidade de um problema geralmente é resultado de diversos elementos. Um *algoritmo parametrizado* separa explicitamente os elementos que tornam um problema difícil, dos que são simples de tratar. A análise da *complexidade parametrizada* quantifica essas partes separadamente. Por isso, a complexidade parametrizada é chamada uma "complexidade de duas dimensões".

Exemplo 5.1
O problema de satisfatibilidade (SAT) é NP-completo, i.e. não conhecemos um algoritmo cuja complexidade cresce somente polinomialmente com o tamanho da entrada. Porém, a complexidade deste problema cresce principalmente com o número de variáveis, e não com o tamanho da entrada: com k variáveis e entrada de tamanho n solução trivial resolve o problema em tempo $O(2^k n)$. Em outras palavras, para *parâmetro* k fixo, a complexidade é linear no tamanho da entrada. ◊

Definição 5.1
Um problema que possui um parâmetro $k \in \mathbb{N}$ (que depende da instância) e permite um algoritmo de complexidade $f(k)|x|^{O(1)}$ para entrada x e com f uma função arbitrária, se chama *tratável por parâmetro fixo* (ingl. fixed-parameter tractable, fpt). A classe de complexidade correspondente é FPT.

Um problema tratável por parâmetro fixo se torna tratável na prática, se o nosso interesse são instâncias com parâmetro pequeno. É importante observar que um problema permite diferentes parametrizações. O objetivo de projeto de algoritmos parametrizados consiste em descobrir para quais parâmetros que são pequenos na prática o problema possui um algoritmo parametrizado. Neste sentido, o algoritmo parametrizado para SAT não é interessante, porque o número de variáveis na prática é grande.
A seguir consideramos o problema NP-complete de *cobertura de vértices*. Uma versão parametrizada é

---

k-COBERTURA DE VÉRTICES

Instância Um grafo não-direcionado $G = (V, A)$ e um número k[1].

Solução Uma cobertura C, i.e. um conjunto $C \subseteq V$ tal que $\forall a \in A$:

$a \cap C \neq \emptyset$.

Parâmetro O tamanho k da cobertura.

Objetivo Minimizar $|C|$.

Abordagem com força bruta:

```
1  mvc(G = (V, A)) :=
2    if A = ∅ return ∅
3    seleciona aresta {u, v} ∈ A não coberta
4    C1 := {u} ∪ mvc(G \ {u})
5    C2 := {v} ∪ mvc(G \ {v})
6    return a menor entre as coberturas C1 e C2
```

Supondo que a seleção de uma aresta e a redução dos grafos é possível em $O(n)$, a complexidade deste abordagem é dado pela recorrência

$$T_n = 2T_{n-1} + O(n)$$

com solução $T_n = O(2^n)$. Para achar uma solução com no máximo k vértices, podemos podar a árvore de busca definido pelo algoritmo mvc na profundidade k. Isso resulta em

Teorema 5.1
O problema k-cobertura de vértices é tratável por parâmetro fixo em $O(2^k n)$.

Prova. Até o nível k vamos visitar $O(2^k)$ vértices na árvore de busca, cada um com complexidade $O(n)$. ■

O projeto de algoritmos parametrizados frequentemente consiste em

- achar uma parametrização tal que o parte super-polinomial da complexidade é limitada para um parte do problema que depende de um parâmetro k que é pequeno na prática;
- encontrar o melhor algoritmo possível para o parte super-polinomial.

Exemplo 5.2
Considere o algoritmo direto (via uma árvore de busca, ou backtracking) para SAT.

```
1  BT−SAT(φ, α) :=
2    if α é atribuição completa: return φ(α)
```

[1] Introduzimos k na entrada, porque k mede uma característica da solução. Para evitar complexidades artificias, entende-se que k nestes casos é codificado em *unário*.

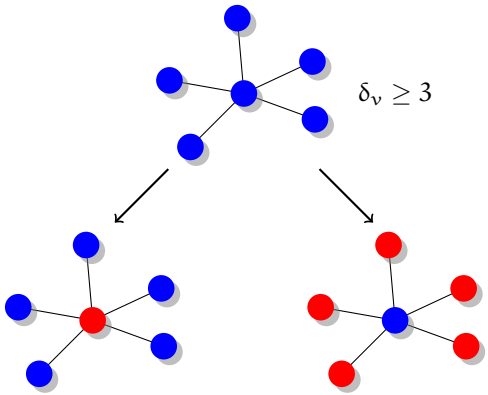


Figura 5.1.: Subproblemas geradas pela decisão da inclusão de um vértice v. Vermelho: vértices selecionadas para a cobertura. {fig:mvc1}

```
3    if alguma cláusula não é satisfeita: return false
4    if BT−SAT(φ, α1) return true
5    return BT-SAT(φ, α0)
```

($\alpha 0$ e $\alpha 1$ denotam extensões de uma atribuição parcial das variáveis.)
Aplicado a 3SAT, das 8 atribuições por cláusula podemos excluir uma que não a satisfaz. Portanto a complexidade de BT-SAT é $O(7^{n/3}) = O(\sqrt[3]{7}^n) = O(1.9129^n)$. (Exagerando – mas não mentindo – podemos dizer que isso é uma aceleração exponencial sobre a abordagem trivial que testa todas 2^n atribuições.)
O melhor algoritmo para 3-SAT possui complexidade $O(1.324^n)$. ◊

Um algoritmo melhor para cobertura de vértices Consequência: O projeto cuidadoso de uma árvore de busca pode melhorar a complexidade. Vamos aplicar isso para o problema de cobertura de vértices.
Um melhor algoritmo para a k-cobertura de vértices pode ser obtido pelas seguintes observações

- Caso o grau máximo Δ de G é 2, o problema pode ser resolvido em tempo $O(n)$, porque G é uma coleção de caminhos simples e ciclos.

- Caso contrário, temos pelo menos um vértice v de grau $\delta_v \geq 3$. Ou esse vértice faz parte da cobertura mínima, ou todos seus vizinhos $N(v)$ (veja figura 5.1).

```
1  mvc'(G) :=
2    if Δ(G) ≤ 2 then
3      determina a cobertura mínima C em tempo O(n)
4      return C
5    end if
6    seleciona um vértice v com grau δ_v ≥ 3
7    C_1 := {v} ∪ mvc'(G \ {v})
8    C_2 := N(v) ∪ mvc'(G \ N(v))
9    return a menor cobertura entre C_1 e C_2
```

O algoritmo resolve o problema de cobertura de vértices mínima de forma exata. Se podamos a árvore de busca após selecionar k vértices obtemos um algoritmo parametrizado para k-cobertura de vértices. O número de vértices nessa árvore é

$$V_i \leq V_{i-1} + V_{i-4} + 1.$$

Lema 5.1
A solução dessa recorrência é $V_i = O(1.3803^i)$.

Teorema 5.2
O problema k-cobertura de vértices é tratável por parâmetro fixo em $O(1.3803^k n)$.

Prova. Considerações acima com trabalho limitado por $O(n)$ por vértice na árvore de busca. ■

Prova. (Do lema acima.) Com o ansatz $V_i \leq c^i$ obtemos uma prova por indução se para um $i \geq i_0$

$$\begin{aligned} & V_i \leq V_{i-1} + V_{i-4} + 1 \leq c^{i-1} + c^{i-4} + 1 \leq c^i \\ \iff\ & c^{i-4}(c^4 - c^3 - 1) \geq 1 \\ \iff\ & c^4 - c^3 - 1 \geq 0 \end{aligned}$$

(O último passo é justificado porque para $c > 1$ e i_0 suficientemente grande o produto vai ser ≥ 1.) $c^4 - c^3 - 1$ possui uma única raiz positiva ≈ 1.32028 e para $c \geq 1.3803$ temos $c^3 - c^2 - 1 \geq 0$. ■

6. Outros algoritmos

6.1. O problema de soma de intervalos

No *problema de soma de intervalos* (ingl. range-sum problem) queremos manter números $a_1, \ldots, a_n$ sobre duas operações: add(i, v) aumenta a_i por v e get(k) retorna $\sum_{i\in[k]} a_i$. Nota que a soma sobre qualquer intervalo $[j, k]$ contíguo, $\sum_{i\in[j,k]} a_i$, é get(k) − get$(j-1)$. Numa implementação direta por um vetor essas operações possuem complexidade $O(1)$ e $O(n)$.
Para uma operação $O : \mathbb{N} \to \mathbb{N}$ seja $Oi = \{i, O(i), O(O(i)), \ldots\} \cap [n]$ o *orbit* de i sobre O.

{th:fenwic

Teorema 6.1
Caso operações O e P satisfazem

$$|Ox \cap Py| = [x \le y] \qquad (\odot)$$ {orbit}

as operações

```
1   add(i, v) := a_j := a_j + v para todo j ∈ Oi
2   get(k) := return Σ_{i∈Pk} a_i
```

resolvem o problema da soma de intervalos.

Prova. Por indução sobre as operações add. Supõe get$(k) = \sum_{i\in[k]} a_i$. Depois de uma operação add(i, v) temos: (i) Caso $i > k$: get$(k) = \sum_{i\in Pk} a'_i = \sum_{i\in Pk} a_i = \sum_{i\in[k]} a_i$ porque $|Oi \cap Pk| = 0$. (ii) Caso $i \le k$: get$(k) = \sum_{i\in Pk} a'_i = v + \sum_{i\in Pk} a_i = v + \sum_{i\in[k]} a_i$ porque $|Ox \cap Py| = 1$. ■

Exemplo 6.1
A solução por um vetor que armazena os a_i diretamente corresponde com $O(i) = i$ e $P(i) = i - 1$. Operações add e get tem complexidade $O(1)$ e $O(n)$, respectivamente. (Critério $(\odot)$ é satisfeito porque $Oi = \{i\}$, $Pi = [i]$.) ◊

Exemplo 6.2
Com $O(i) = i+1$ e $P(i) = i$ obtemos uma solução em que a_i armazena as somas parciais. As operações agora tem complexidade $O(n)$ e $O(1)$. (Critério $(\odot)$ é satisfeito porque $Oi = \{i, i+1, \ldots, n\}$ e $Pi = \{i\}$.) ◊

Exemplo 6.3
Seja $O(i) = i+2^{r(i)}$ e $P(i) = i-2^{r(i)}$ com $r(i)$ o índice do bit menos significativo (LSB) na representação binária de i. Por definição é claro que a órbita de O cresce, i.e. $O(i) > i$, e o do P decresce, i.e. $P(i) < i$.

Proposição 6.1
Critério ($\odot$) é satisfeito.

Prova. Se $x > y$, temos $|Ox \cap Py| = 0$, pois a órbita de O cresce e a de P decresce. Pelo mesmo motivo, se $x = y$, então $|Ox \cap Py| = |\{x, y\}| = 1$ é válido. Agora, suponha que $x < y$. Podemos escrever $x = h + s_x$, $y = h + 2^b + s_y$, onde b é o bit mais significativo diferente de x e y, $h \geq 2^{b+1}$ e $0 \leq s_x, s_y < 2^b$. Considere primeiro $s_x = 0$. Então, $x = h \in Py$, já que P remove repetidamente bit menos significativo (least significant bit, LSB) e, portanto, $x \in Ox \cap Py$. Para qualquer outro $o \in Ox$, $o \neq x$, temos $o \geq x + 2^{r(x)} \geq x + 2^{b+1}$, mas para $p \in Py$, $p \leq h + 2^b + s_y = x + 2^b + s^y < x + 2^b + 2^b = x + 2^{b+1}$. Portanto, $|Ox \cap Py| = |\{x\}| = 1$.
Agora considere $s_x > 0$. Afirmamos que $Ox \cap Py = \{m\}$, onde $m = h + 2^b$. Novamente, é fácil ver que $m \in Py$, pois P remove repetidamente o LSB. Para ver que $m \in Ox$, considere as iterações $s_i = O^i(s_x)$, $i = 0, 1, 2, \ldots$. Se $s_i < 2^b$, então $s_i \leq (2^b - 1) - (2^{r(s_i)} - 1) = 2^b - 2^{r(s_i)}$, já que $r(s_i) < b$ é o LSB. Assim, para o primeiro iterado tal que $s_i \geq 2^b$, temos $s_i = s_{i-1} + 2^{r(s_{i-1})} \leq 2^b$, portanto $s_i = 2^b$ e, assim, $m = h + 2^b \in Ox$.
Agora considere $o \in Ox$ e $p \in Py$ com $o, p < m$. Temos $o \geq x = h + s_x > h$, mas também $p \leq m - 2^{r(m)} = h$, portanto, nenhum outro elemento desse tipo está em $Ox \cap Py$. Por fim, considere $o \in Ox$ e $p \in Py$ com $o, p > m$. Então, $o \geq m + 2^{r(m)} = m + 2^b = h + 2^b + 2^b = h + 2^{b+1}$ e $p \leq y = h + 2^b + s_y < h + 2^b + 2^b = h + 2^{b+1}$. Portanto, novamente, nenhum outro elemento desse tipo está em $Ox \cap Py$. ■

Prova. If $x > y$, we have $|Ox \cap Py| = 0$, since O's orbit increases, and P's decreases. For the same reason, if $x = y$ then $|Ox \cap Py| = |\{x, y\}| = 1$ holds.
Now assume $x < y$. Then we can write $x = h + s_x$, $y = h + 2^b + s_y$, where b is the highest different bit of x and y, $h \geq 2^{b+1}$, and $0 \leq s_x, s_y < 2^b$. Consider first $s_x = 0$. Then $x = h \in Py$, since P repeatedly removes the LSB, and thus $x \in Ox \cap Py$. For any other $o \in Ox$, $o \neq x$, we have $o \geq x + 2^{r(x)} \geq x + 2^{b+1}$ but for $p \in Py$, $p \leq h + 2^b + s_y = x + 2^b + s^y < x + 2^b + 2^b = x + 2^{b+1}$. Therefore $|Ox \cap Py| = |\{x\}| = 1$.
Now consider $s_x > 0$. We claim $Ox \cap Py = \{m\}$, where $m = h + 2^b$. It is again easy to see that $m \in Py$, since P repeatedly removes the LSB. To see that $m \in Ox$ consider iterates $s_i = O^i(s_x)$, $i = 0, 1, 2, \ldots$. If $s_i < 2^b$ then $s_i \leq (2^b - 1) - (2^{r(s_i)} - 1) = 2^b - 2^{r(s_i)}$, since $r(s_i) < b$ is the LSB. Thus, for the first iterate such that $s_i \geq 2^b$, we have $s_i = s_{i-1} + 2^{r(s_{i-1})} \leq 2^b$, so $s_i = 2^b$, and thus $m = h + 2^b \in Ox$.
Now consider $o \in Ox$ and $p \in Py$ with $o, p < m$. We have $o \geq x =$

$h + s_x > h$, but also $p \leq m - 2^{r(m)} = h$, so no other such element is in $Ox \cap Py$. Finally, consider $o \in Ox$ and $p \in Py$ with $o, p > m$. Then $o \geq m + 2^{r(m)} = m + 2^b = h + 2^b + 2^b = h + 2^{b+1}$, and $p \leq y = h + 2^b + s_y < h + 2^b + 2^b = h + 2^{b+1}$. So again, no other such element is in $Ox \cap Py$. ■

Proposição 6.2
As operações `add` and `get` tem complexidade $O(\log n)$.

Prova. Por indução, $O^i(x) \geq x + \sum_{0 \leq j < i} 2^j \geq 2^i$, de modo que a órbita de O tem no máximo $\log_2 n$ elementos. Da mesma forma, $P^i(y) \leq n - \sum_{0 \leq j < i} 2^j = n - 2^i + 1$ e a órbita de P também tem no máximo $\log_2 n$ elementos. As duas operações podem ser implementadas de forma eficiente por $O(i) = (i \mid (i-1)) + 1$ e $P(i) = i\&(i-1)$ em tempo $O(1)$. ■

Prova. By induction $O^i(x) \geq x + \sum_{0 \leq j < i} 2^j \geq 2^i$ so O's orbit has at most $\log_2 n$ elements. Similarly, $P^i(y) \leq n - \sum_{0 \leq j < i} 2^j = n - 2^i + 1$ and P's orbit also has at most $\log_2 n$ elements. As duas operações podem ser implementadas de forma eficiente por $O(i) = (i \mid (i-1)) - 1$ e $P(i) = i\&(i-1)$ em tempo $O(1)$. ■

Even if we don't use the bit operations, numbers x and y have at most $\log_2 n$ bits, and we can go bit over bit to implement O and P. That would increase the complexitty to $O(\log^2 n)$.

◊

That's nice, but unnecessarily complicated. (I also lots the original reference, may this one was it; there is also this.) A segment tree archieves the same simpler, and can be extended to do more.

Exercício 6.1
Mostre que as operações $O(i) = i \mid i + 1$ e $P(i) = (i\&(i+1)) - 1$ satisfazem o critério do teorema 6.1. Qual a interpretação das operações na representação binária? Você conseque dar uma definição aritmética equivalente? Qual a complexidade de `add` e `get` usando essas operações?

6.2. Amostragem discreta

6.2.1. Amostragem sem reposição

{sec

Queremos selecionar k números de $[n]$ sem reposição. Uma forma simples de conseguir isso é definir um vetor $s_i = i$, $i \in [n]$ e para $j \in [k]$ trocar um elemento aleatório em $s_{[j,n]}$ com s_j. No final $s_{[k]}$ contém a amostra desejada. O custo é $O(n)$ tempo e espaço, porque usa um vetor de tamanho n. Uma abordagem melhor usa uma tabela hash mapeando índices para valores, sem armazenar os valores default $i \mapsto i$. Com isso o custo de tempo e espaço é reduzido para $O(k)$ que é essencialmente ótimo.

6.2.2. Distribuições discretas

Queremos amostrar de uma distribuição discreta com probabilidades p_i, $i \in [n]$. Uma abordagem muito simples é *rejection sampling*. Seja $\bar{p} = \max_{i \in [n]} p_i$. Selecionamos um item $i \in [n]$ e um número em $q = [0, \bar{p}]$ uniformemente e rejeitamos se $q > p_i$. A taxa de aceitação é $1/(n\bar{p})$.

Uma ideia melhor é *tower sampling*. Aqui, armazenamos as somas parciais $q_i = \sum_{j \in [i]} p_j$, $i \in [n]$, amostramos um número aleatório uniforme $r \in U[0, 1]$ e, em seguida, fazemos uma busca binária pelo menor i, de modo que $r \geq q_i$. O pré-processamento leva tempo $O(n)$, a amostragem apenas $O(\log n)$.

A solução para o problema de soma de intervalo acima permite atualizar as somas de prefixo no tempo $O(\log n)$. Portanto, podemos aplicar a amostragem de torre dinamicamente com tempo de atualização de $O(\log n)$ e tempo de amostragem de $O(\log^2 n)$, já que temos no máximo $\log n$ consultas, cada uma de custo $O(\log n)$.

Uma ideia ainda melhor é *alias sampling*. Primeiro, subdivida todos os p_i em itens de baixa probabilidade $L = \{i \mid p_i < 1/n\}$, boa probabilidade $G = \{i \mid p_i = 1/n\}$ e alta probabilidade $H = \{i \mid p_i > 1/n\}$. Logo, se $L = H = \emptyset$, podemos fazer uma amostragem uniforme de G. Em seguida, observamos que $L = \emptyset$ sse $H = \emptyset$, pois as probabilidades dos n itens devem somar 1. Portanto, ou somos bons ou temos um par L-H. Para esse par, crie um compartimento "bom" combinando o item L com uma parte adequada do item H. Lembre-se dos compartimentos de origem e realoque a parte restante do item H para L, G ou H. Agora ainda temos n compartimentos, mas um compartimento bom (tipo G) a mais. Repita até que tenhamos apenas compartimentos bons. Isso leva no máximo $O(n)$ tempo, pois podemos ter no máximo n compartimentos bons.

Para amostragem, armazene em $s_1, s_2, \ldots, s_{2n}$ números de itens de modo que o compartimento i represente os itens s_{2i} e s_{2i+1}. (Para compartimentos

puramente bons, $s_{2i} = s_{2i+1}$.) Armazene também a massa de probabilidade do primeiro item s_{2i} em cada compartimento em $q_1, q_2, \ldots, q_n$. (Novamente, para compartimentos puramente bons, $q_i = 1$).
Agora podemos fazer a amostragem da seguinte forma em tempo $O(1)$:

```
1   x = U(0, 1]
2   b = ⌈nx⌉ // localizar o compartimento
3   r = [(nx mod 1) > q_b] // localizar o item no compartimento
4   retornar s_{2b+r}
```

Vamos estudar agora a amostragem de reservatório (ingl. reservoir sampling). Aqui, o problema é escolher um elemento da sequência $1, 2, \ldots, n$ com probabilidades p_i, mas on-line, ou seja, visitando a sequência uma vez. É claro que poderíamos ler toda a sequência e fazer uma amostragem como acima. Portanto, a restrição aqui é que temos $O(1)$ de memória.
Vamos examinar primeiro o caso uniforme que possui uma solução fácil. Mantenha um item selecionado, inicialmente nenhum, e substitua-o pelo item i com probabilidade $1/i$. A correção pode ser facilmente demonstrada por indução. Suponha que, para n itens, tenhamos $p_i = 1/n$. Então, para $n + 1$, escolhemos $n + 1$ com probabilidade $1/(n + 1)$, ou qualquer um dos outros itens com probabilidade $p_i \cdot n/(n + 1) = 1/(n + 1)$, conforme necessário.
Agora generalizamos isso para selecionar itens de $m > 1$ e pesos gerais $w_1, \ldots, w_n$ (ou seja, os pesos não precisam ser normalizados). Isso funciona da seguinte forma. Para cada item, calcule o valor $U[0, 1]^{1/w_i}$ e mantenha os m itens de maior valor. Podemos ver facilmente por que isso é correto no caso especial de amostragem uniforme. Nesse caso, é melhor definir $w_1 = \cdots = w_n = 1$. Então, basta sortear n números aleatórios em $U[0, 1]$ e pegar os m itens de maiores valores.
O algoritmo acima requer n números aleatórios, e o número esperado de atualizações do conjunto escolhido é $O(m \log n/m)$. Há uma versão que precisa de apenas $O(m \log n/m)$ amostras aleatórias. Esses algoritmos também podem ser usados para criar uma amostra aleatória de tamanho k com reposição, executando k instâncias paralelas que selecionam $m = 1$ item cada.

We want to sample from a discrete distribution with probabilities p_i, $i \in [n]$. A very simple approach is via *rejection sampling*. Let $\bar{p} = \max_{i \in [n]} p_i$. We select an item $i \in [n]$ and a number in $q = [0, \bar{p}]$ uniformly, and reject if $q > p_i$. The acceptance rate is $1/(n\bar{p})$.
A better idea is *tower sampling*. Here we store the partial sums $q_i = \sum_{j \in [i]} p_j$, $i \in [n]$, sample a uniform random number $r \in U[0, 1]$, and then binary search for the smallest i, such that $r \geq q_i$. Pre-processing

takes time $O(n)$, sampling only $O(\log n)$. More details can be found in Krauth (2006, Section 1.2.3). The solution to the interval sum problem above allows to update prefix sums in time $O(\log n)$. Therefore we can apply tower sampling dynamically with update time $O(\log n)$ and sample time $O(\log^2 n)$, since we have at most $\log n$ queries, each of which costs $O(\log n)$.

An even better idea is *alias sampling*. First subdivide all p_i into items with low probability $L = \{i \mid p_i < 1/n\}$, good probability $G = \{i \mid p_i = 1/n\}$, and high probability $H = \{i \mid p_i > 1/n\}$. Then, if $L = H = \emptyset$, we can sample uniformly from G. Next, we observe $L = \emptyset$ iff $H = \emptyset$, since the probabilities of the n items must sum to 1. Therefore either we are good, or we have a L-H pair. For such a pair, create a "good" bin by combining the L item, with an adequate part of the H item. Remember the source bins, and reallocate the remaining part of the H item into L, G, or H. Now we still have n bins, but one G bin more. Repeat until we have only G bins. This takes at most $O(n)$ time, since we can have at most n G bins.

For sampling, store in $s_1, s_2, \ldots, s_{2n}$ item numbers such that bin i represents items s_{2i} and s_{2i+1}. (For purely good bins, $s_{2i} = s_{2i+1}$.) Also store the probability mass for the first item s_{2i} in each bin in $q_1, q_2, \ldots, q_n$. (Again, for purely good bins $q_i = 1$.)

Now we can sample as follows in time $O(1)$:

```
1  x = U[0,1]  b = ⌈nx⌉  // find the bin
2  r = [(nx mod 1) > q_b]  // find the item in the bin
3  return s_{2b+r}
```

We can see tower sampling as sampling in a full binary tree (divide items accordingly). Alias sampling can be seen as a tree of depth 2 where the root has n bins as children, and each child either is a leaf, representing a single item, or has two leafs, corresponding to the two possible items. Decisive is that the first level is uniform (or at least regular, such that a child can be sampled in $O(1)$). Does this view bring some insight?

Let us now turn to reservoir sampling. Here, the problem is choosing an element from the sequence $1, 2, \ldots, n$ with probabilities p_i, but online, i.e. by visiting the sequence once. Clearly we could read the entire sequence, and sample as above. So the restriction here is that we have $O(1)$ memory.

We look first at the uniform case. This has an easy solution. Keep a currently selected item, initially none, and replace it by item i with probability $1/i$. Correctness can be easily shown by induction. Assume for n items we have $p_i = 1/n$. Then for $n+1$, we choose $n+1$ with probability $1/(n+1)$, or any of the other items with probability $p_i \cdot n/(n+1) = 1/(n+1)$ as required.
We now generalize this to selecting $m > 1$ items and general weights $w_1, \ldots, w_n$ (i.e. the weights need not be normalized). This works as follows. For each item compute value $U[0,1]^{1/w_i}$ and keep the m items of largest value. We can see easily why this is correct in the special case of uniform sampling. Here it's best to set $w_1 = \cdots = w_n = 1$. Then we just draw n random numbers in $U[0,1]$ and take the m items of largest values.
The above algorithm requires n random numbers, and the expected number of updates of the chosen set is $O(m \log n/m)$. There's a version that needs only $O(m \log n/m)$ random samples. These algorithms can also be used to create a random sample of size k with replacement, by running k parallel instances selecting $m = 1$ item each.

Open:

- Demonstrate the above.
- Does similarity of biased random keys with the sampling above (which in the original paper also uses the notion of "keys") suggest a possibility of generating non-uniform permutations in BRKGAs?
- Weighted reservoir sampling should also be connected to the ideas of (Krauth, 2006); I see similarities.

A good source on sampling is the book of Krauth (2006). Alias sampling is well explained by Pătraşcu (2011). Weighted reservoir sampling is from Efraimidis e Spirakis (2005).

Notas Uma boa fonte sobre amostragem é o livro de Krauth (2006). Para amostragem sem reposição ver Ting (2021) e Bentley e Floyd (1987). Alias sampling é bem explicada por Pătraşcu (2011). A amostragem de reservatório ponderada é de Efraimidis e Spirakis (2005).

6.3. Set covering

(A lecture on demand held Aug 12, 2024. The first part is based on a chapter on set partitioning of Balas, the second part on a paper of Bläsius and others.)

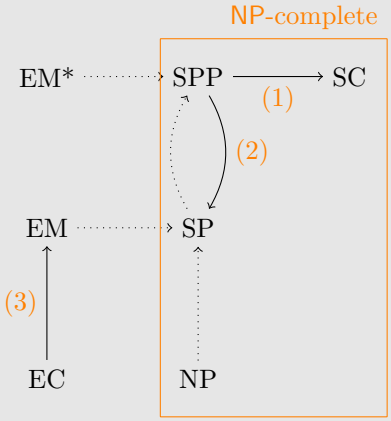
Let $A \in \mathbb{R}^{m \times n}$ be the element-set containment matrix over m elements and n sets, i.e. columns are sets, with 1s at the rows of their elements, rows are elements, with 1s at the column of the sets they are contained in. We have the NP-complete set covering problem

$$\min\{cx \mid Ax \geq \mathbf{1}, x \in \mathbb{B}^n\} \qquad \text{(SC)} \quad \texttt{\{sc\}}$$

and the related problems

$$\min\{cx \mid Ax = \mathbf{1}, x \in \mathbb{B}^n\} \qquad \text{(SPP)} \quad \texttt{\{spp\}}$$
$$\max\{cx \mid Ax \leq \mathbf{1}, x \in \mathbb{B}^n\} \qquad \text{(SP)} \quad \texttt{\{sp\}}$$
$$\max\{\mathbf{1}y \mid A_G y \leq \mathbf{1}, y \in \mathbb{B}^m\} \qquad \text{(EM)} \quad \texttt{\{em\}}$$
$$\min\{\mathbf{1}y \mid A_G y \geq \mathbf{1}, y \in \mathbb{B}^m\} \qquad \text{(EC)} \quad \texttt{\{ec\}}$$
$$\max\{\mathbf{1}x \mid A_G^t x \leq \mathbf{1}, x \in \mathbb{B}^n\} \qquad \text{(NP)} \quad \texttt{\{np\}}$$
$$\min\{\mathbf{1}x \mid A_G^t x \geq \mathbf{1}, x \in \mathbb{B}^n\} \qquad \text{(NC)} \quad \texttt{\{nc\}}$$

namely set partitioning SPP, set packing SP, edge matching EM (and its perfect version EM*), edge cover EC, node packing NP (i.e. maximum independent sets), and node cover NC (i.e. minimum vertex cover). Here $A_G \in \mathbb{R}^{n \times m}$ is the node-edge incidence matrix of graph G with n vertices and m edges (columns represent edges with at most two 1s, rows represent vertices, the 1s are its incident edges).
We have the following relationship, where solid arcs are problem reductions, and dotted arcs go from special cases to more general problems:



The reductions are as follows.

(1) SPP to SC. Penalize excess by writing

$$\min\{cx + \Theta \mathbf{1} y \mid Ax - y = \mathbf{1}, y \geq 0, x \in \mathbb{B}^n\}.$$

Then $y = Ax - \mathbf{1}$ so $\Theta \mathbf{1} y = \Theta \mathbf{1} Ax - m\Theta$ and we can write

$$\min\{-\Theta m + c'x \mid Ax \geq \mathbf{1}, x \in \mathbb{B}^n\},$$

with $c' = \Theta \mathbf{1} A + c$ (we can drop constant $-\Theta m$). For large $\Theta > \mathbf{1}c$ optimal solutions are the same.

(2) SPP to SP Penalize slack by writing

$$\min\{cx + \Theta \mathbf{1} y \mid Ax + y = \mathbf{1}, y \geq 0, x \in \mathbb{B}^n\}.$$

Then $y = \mathbf{1} - Ax$ so $\Theta \mathbf{1} y = \Theta m - \Theta \mathbf{1} Ax$, and letting $c' = \Theta \mathbf{1} A - c$ we have

$$\min\{-c'x + \Theta m \mid Ax \leq \mathbf{1}, x \in \mathbb{B}^n\} = \max\{c'x \mid Ax \leq \mathbf{1}, x \in \mathbb{B}^n\}$$

(3) EC to EM EC must cover all n vertices. This can be done with n edges. However, in a matching M we cover two vertices per edge, thus

the value of EC minimizes $n - |M|$. In other words we maximize $|M|$, i.e. solve an EM, since n is a constant.
(We note that all can be reduced to SC, but there's no simple reduction from SC to SPP.)

6.3.1. Further related problems

Let $G = (V, \mathcal{F})$ be a hypergraph with hyperedges $\mathcal{F} \subseteq V$. The *hitting set problem* is

$$\min_{S \subseteq V}\{|S| \mid F \cap S \neq \emptyset, \forall F \in \mathcal{F}\} \tag{HS}$$ {hs}

$$\min\{\mathbf{1}y \mid \underbrace{y^t A \geq \mathbf{1}^t}_{A^t y \geq \mathbf{1}}, y \in \mathbb{B}^m\} \tag{HS}$$ {hs2}

where in the latter formulation we just have SC with the roles of sets and elements inverted. (In SC sets covers element they contain; changing roles, seeing sets as elements and elements they cover as sets their are contained in, we have a hitting set.)

6.3.2. Solution strategies

In the 2000s ILP was SOTA. We look at Bläsius et al. (2022) which achieve a median speedup of 25 over 929 instances where Gurobi takes ≥ 0.01 seconds. (The total test set has 4256 instances, with 4114 trivial ones, and 6 that take more than 24 hours. This leaves 136 core instances, of which 58 are random and 78 applied.)
The solution technique is branch and bound. It solves subproblems over chosen vertices S, open vertices V, and exluded vertives $\bar{S}$.

```
1  hs(S,V,S̄) :=
2    greedy(S ∪ V) { Compute UB }
3    while reduction or pruning possible
4      computer lower bounds
5      if lb ≥ ub: return
6      apply first applicable reduction
7    end while
8    choose highest degree vertex v ∈ V
9    hs(S ∪ {v}, V \ {v}, S̄) { include }
10   hs(S, V \ {v}, S̄ ∪ {v}) { exclude }
```

We can see that this is almost canonical, with inclusion before exclusion. What counts are lower and upper bounds, reductions, the branching strategy, and efficient data structures. We go over them in turn.

6.3.3. Upper bounds

Repeatedly pick greedily the vertex of highest degree. This is a $\log n$-approximation, and works well in practice. It can be done in time $O(n + m)$ by keeping vertices in buckets, and repeatedly choosing one of highest degree, and then going over the neighbors to relocate them.

6.3.4. Lower bounds

There are several ones of increasing complexity.

Max-degree Each vertex hits at most $d_{\max} = \max_{v \in V} \delta_v$ sets. So $\lceil |\mathcal{F}|/d_{\max} \rceil$ is a lower bound.

Sum-degree Let $\delta_1 \geq \delta_2 \geq \cdots \geq \delta_n$, and choose the smallest i such that the prefix sum $\sum_{j \in [i]} \delta_j \geq |\mathcal{F}|$. Then j is a lower bound, since fewer vertices cannot hit all sets.

Efficiency Take any solution $S \subseteq V$ of cost $|S|$ and write

$$|S| = \sum_{v \in S} 1 = \sum_{v \in S} \underbrace{\sum_{F \in \mathcal{F}(s)} 1/\delta_v}_{\text{push cost to edges}} \stackrel{!}{=} \sum_{F \in \mathcal{F}} \underbrace{\sum_{v \in S \cap F} 1/\delta_v}_{\geq \min_{v \in F} 1/\delta_v} \geq \sum_{F \in \mathcal{F}} \min_{v \in F} 1/\delta_v.$$

In the relaxation to the minimum we observe that at least one vertex of highest degree pays for edge F. The cost is now independent of S, and we have lower bound $\lceil \sum_{F \in \mathcal{F}} \min_{v \in F} 1/\delta_v \rceil$. (The vertex cover viewpoint is: each vertex costs $1/s$ where s is the largest set it is contained in.)

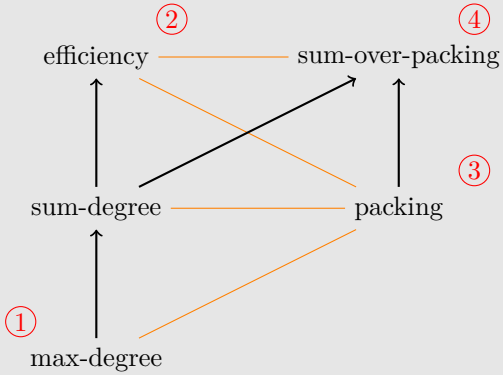
Packing If $P \subseteq \mathcal{F}$ are pairwise disjoint edges, $|P|$ is a lower bound. This is an independent set in the intersection graph of the edges.

Sum-over-packing As above, take a packing P. The vertices that hit P hit at most

$$b_P = \Big(\sum_{F \in P} \max_{v \in F} \delta_v\Big) - |P|$$

edges in the rest $\mathcal{F} \setminus P$. Now assume $b_P < |\mathcal{F} \setminus P|$, then we need to hit $r = |\mathcal{F} \setminus P| - b_P$ more edges. So again let $\delta_1 \geq \delta_2 \geq \cdots \geq \delta_n$, but with degrees in $\mathcal{F} \setminus P$, and with the vertex of highest degree *out*, and choose the smallest i such that $\sum_{j \in [i]} \delta_j \geq r$. Then $|P| + j$ is a lower bound.

The relations are as follows, where arcs indicate domination, and orange edges incomparability. The order these bounds are computed is shown in red.



They cost (1) $O(n)$, (2) $O(D)$, (3) $O(D + m \log m)$, and (4) $O(D)$.

6.3.5. Reduction rules

We have four reduction rules. Again circled numbers give the order and the cost to apply them. Here $D = \sum_{v \in V} \delta_v = \sum_{F \in \mathcal{F}} |F|$ (note that in traditional graphs $D = 2m$ since $|F \models 2$).

Unit edge If an edge has size one, pick its vertex. (5) $O(m)$.

Edge domination If edges $F_1 \subseteq F_2$, remove F_2 (since hitting F_1 implies hitting F_2). (9) $O(mD)$.

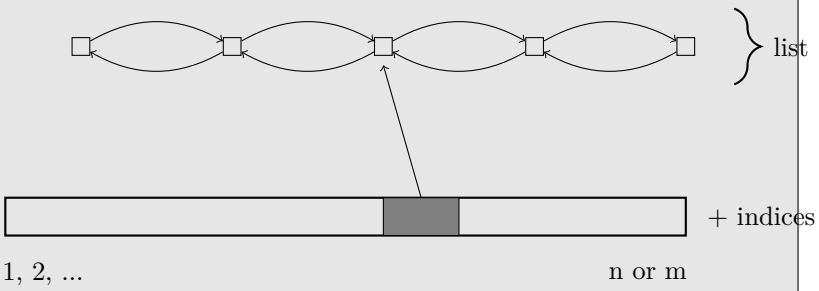
Vertex domination If $\mathcal{F}(v_1) \supseteq \mathcal{F}(v_2)$ delete v_2 (since all v_2 hits v_1 hits, too). (10) $O(nD)$.

Costly discard (a kind of anticipated branching) if removing $v \in V$ makes $ub \geq lb$ select v. (6): update efficiency bound, $O(D)$, (7): update

packing bound, no repacking, $O(D + m \log m)$, (8): repack for the 3 vertices of highest degree $O(D + m \log m)$.

6.3.6. Details

Keeping the graph Vertices have edge lists, edges vertex lists; all are kept sorted in a data structure called an "ordered subset list".



In this way we have the following operations:

init() to $\{s_1, \ldots, s_n\}$ (ordered) in $O(n)$;

del(i) in $O(1)$;

undo last del in $O(1)$;

traverse in either direction in $O(|S|)$.

We also keep an undo stack.

Upper bounds Via bucket heaps, in $O(D)$.

Packing bound Computed heuristically, as follows:

a) *Approximate* the min degree order of edges F in the conflict graph by $\sum_{v \in F} \delta_v$, done in $O(m \log m)$.

b) Visit edges F in order, select F if possible, mark vertices. Each check costs $|F|$ so total cost $D = \sum_{F \in \mathcal{F}} |F|$.

Costly discard with efficiency bound Each $F \in \mathcal{F}$ may lose a vertex, so cost $\min_{v \in F} 1/\delta_v$ goes to the second smallest vertex; it is remembered when computing the efficiency bound for the instance, so cost $O(D)$ once. Then: for each $v \in V$ and for each $F \in \mathcal{F}(v)$, check if v is current minimum, and increase. This costs $\sum_{v \in V} \sum_{F \in \mathcal{F}(v)} 1 = D$.

Costly discard with packing bound In the current packing P all edges $\mathcal{F} \setminus P$ intersect some $F \in P$, and thus are *blocked* by some $v \in \bigcup P$ such that $F \cap \bigcup P = \{v\}$. (Only those blocked by a single vertex are interesting.) Now do the following three steps:

a) initialize: for each $v \in V$ find B_v, the edges blocked by it in $O(D)$, and sort B_v by highest degree of a contained vertex (except v) in $O(m \log m)$

b) check: for each $v \in V$ go over B_v and add edges greedily

c) for the $c = 3$ vertices of highest degree: rebuild the packing from scratch

Edge and vertex domination For vertices going over all pairs in $O(\sum_{i,j} \delta_i + \delta_j) = O(nD)$; for edges ditto in $O(\sum_{F_1,F_2} |F_1| + |F_2|) = O(mD)$. (There are theoretical indications that subquadratic time is not possible.) In practice: *set tries* over $[n]$ store a family of sets over operations (Savnik, 2013)

add(S) $T = T \cup \{S\}$ in $(|S|)$

has subset(S) in $O(|S| + \|T\|)$

has superset(S) in $O(|S| + \|T\|)$

as follows: root $\emptyset$, children always *greater*, each node also has a "last flag" (for prefixes).

Now for edges: go over $|F_1| \leq |F_2| \leq \cdots \leq |F_n|$, for each F_i: if T has a subset discard F_i else insert into T. For vertices: go over $\delta_1 \geq \delta_2 \geq \cdots \geq \delta_n$, and for each v_i: if T has a superset of $\mathcal{F}(v_i)$ discard v_i, otherwise insert $\mathcal{F}(v_i)$ into T.

A. Material auxiliar

Definições

Definição A.1 {def:polim}
Uma relação binária R é *polinomialmente limitada* se

$$\exists p \in \mathsf{poly} : \forall (x, y) \in R : |y| \leq p(|x|)$$

Definição A.2 (Pisos e tetos) {def:pisot}
Para $x \in \mathbb{R}$ o *piso* $\lfloor x \rfloor$ é o maior número inteiro menor que x e o *teto* $\lceil x \rceil$ é o menor número inteiro maior que x. Formalmente

$$\lfloor x \rfloor = \max\{y \in \mathbb{Z} \mid y \leq x\}$$
$$\lceil x \rceil = \min\{y \in \mathbb{Z} \mid y \geq x\}$$

O *parte fracionário* de x é $\{x\} = x - \lfloor x \rfloor$.

Observe que o parte fracionário sempre é positivo, por exemplo $\{-0.3\} = 0.7$.

Proposição A.1 (Regras para pisos e tetos)
Pisos e tetos satisfazem

$$x \leq \lceil x \rceil < x + 1 \tag{A.1}$$ {eq:teto}
$$x - 1 < \lfloor x \rfloor \leq x \tag{A.2}$$ {eq:piso}

Definição A.3
Uma função f é *convexa* se ela satisfaz a desigualdade de Jensen

$$f\big(\Theta x + (1 - \Theta)y\big) \leq \Theta f(x) + (1 - \Theta) f(y). \tag{A.3}$$

Similarmente uma função f é *concava* caso $-f$ é convexo, i.e., ela satisfaz

$$f(\Theta x + (1 - \Theta)y) \geq \Theta f(x) + (1 - \Theta) f(y). \tag{A.4}$$

Exemplo A.1
Exemplos de funções convexas são x^{2k}, $1/x$. Exemplos de funções concavas são $\log x$, $\sqrt{x}$. ◊

{pro

Proposição A.2
Para $\sum_{i\in[n]}\Theta_i = 1$ e pontos x_i, $i \in [n]$ uma função convexa satisfaz

$$f\Big(\sum_{i\in[n]}\Theta_i x_i\Big) \le \sum_{i\in[n]}\Theta_i f(x_i) \tag{A.5}$$ {eq:

e uma função concava

$$f\Big(\sum_{i\in[n]}\Theta_i x_i\Big) \ge \sum_{i\in[n]}\Theta_i f(x_i) \tag{A.6}$$ {eq:

Prova. Provaremos somente o caso convexo por indução, o caso concavo sendo similar. Para $n = 1$ a desigualdade é trivial, para $n = 2$ ela é válida por definição. Para $n > 2$ define $\bar\Theta = \sum_{i\in[2,n]}\Theta_i$ tal que $\Theta + \bar\Theta = 1$. Com isso temos

$$f\Big(\sum_{i\in[n]}\Theta_i x_i\Big) = f\big(\Theta_1 x_1 + \sum_{i\in[2,n]}\Theta_i x_i\big) = f(\Theta_1 x_1 + \bar\Theta y)$$

onde $y = \sum_{j\in[2,n]}(\Theta_j/\bar\Theta)x_j$, logo

$$\begin{aligned} f\Big(\sum_{i\in[n]}\Theta_i x_i\Big) &\le \Theta_1 f(x_1) + \bar\Theta f(y) \\ &= \Theta_1 f(x_1) + \bar\Theta f\Big(\sum_{j\in[2,n]}(\Theta_j/\bar\Theta)x_j\Big) \\ &\le \Theta_1 f(x_1) + \bar\Theta \sum_{j\in[2,n]}(\Theta_j/\bar\Theta) f(x_j) = \sum_{i\in[n]}\Theta_i x_i \end{aligned}$$

■

A.1. Algoritmos

{alg:mmpd}

Soluções do problema da mochila com Programação Dinâmica

Mochila máxima (Knapsack)

- Seja $S^*(k, v)$ a solução de tamanho menor entre todas soluções que
  - usam somente itens $S \subseteq [1, k]$ e

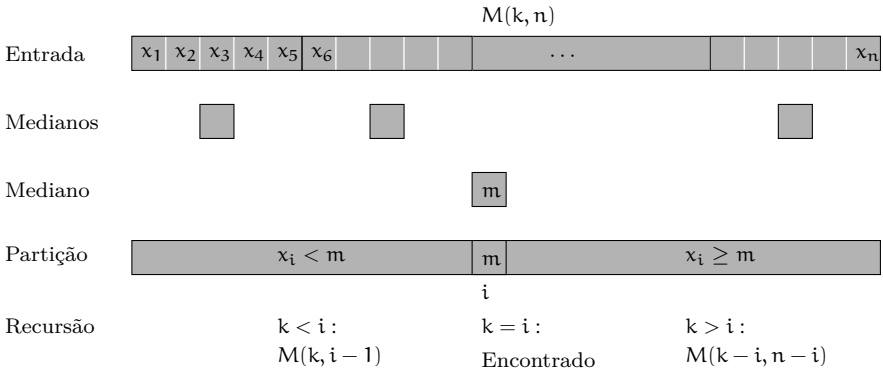


Figura A.1.: Funcionamento do algoritmo recursivo para seleção. {fig:selec

– tem valor exatamente v.

- Temos

$$\begin{aligned} S^*(k,0) &= \emptyset \\ S^*(1,v_1) &= \{1\} \\ S^*(1,v) &= \text{undef} \qquad \text{para } v \neq v_1 \end{aligned}$$

Mochila máxima (Knapsack)

- S^* obedece a recorrência

$$S^*(k,v) = \min_{\text{tamanho}} \begin{cases} S^*(k-1, v-v_k) \cup \{k\}, & \text{se } v_k \leq v \text{ e } S^*(k-1,v-v_k) \text{ definido} \\ S^*(k-1,v) \end{cases}$$

- Menor tamanho entre os dois

$$\sum_{i \in S^*(k-1, v-v_k)} t_i + t_k \leq \sum_{i \in S^*(k-1,v)} t_i.$$

- Melhor valor: Escolhe $S^*(n,v)$ com o valor máximo de v definido.

- Tempo e espaço: $O(n \sum_{i \in [n]} v_i)$.

{sec

Seleção Dado um conjunto de números, o problema da seleção consiste em encontrar o k-ésimo maior elemento. Com ordenação o problema possui solução em tempo $O(n \log n)$. Mas existe um outro algoritmo mais eficiente. Podemos determinar o mediano de grupos de cinco elementos, e depois o recursivamente o mediano m desses medianos. Com isso, o algoritmo particiona o conjunto de números em um conjunto L de números menores que m e um conjunto R de números maiores que m. O mediano m é na posição $i := |L| + 1$ desta sequência. Logo, caso $i = k$ m é o k-ésimo elemento. Caso $i > k$ temos que procurar o k-ésimo elemento em L, caso $i < k$ temos que procurar o $k - i$-ésimo elemento em R (ver figura A.1).
O algoritmo é eficiente, porque a seleção do elemento particionador m garante que o subproblema resolvido na segunda recursão é no máximo um fator $7/10$ do problema original. Mais preciso, o número de medianos é maior que $n/5$, logo o número de medianos antes de m é maior que $n/10 - 1$, o número de elementos antes de m é maior que $3n/10 - 3$ e com isso o número de elementos depois de m é menor que $7n/10 + 3$. Por um argumento similar, o número de elementos antes de m é também menor que $7n/10 + 3$. Portanto temos um custo no caso pessimista de

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 5 \\ T(\lceil n/5 \rceil) + \Theta(7n/10 + 3) + \Theta(n) & \text{caso contrário} \end{cases}$$

e com $5^{-p} + (7/10)^p = 1$ temos $p = \log_2 7 \approx 0.84$ e

$$\begin{aligned} T(n) &= \Theta\left(n^p \left(1 + \int_1^n u^{-p} du\right)\right) \\ &= \Theta(n^p(1 + (n^{1-p}/(1-p) - 1/(1-p)))) \\ &= \Theta(c_1 n^p + c_2 n) = \Theta(n). \end{aligned}$$

g:selection}

Algoritmo A.1 (Seleção)
Entrada Números $x_1, \ldots, x_n$, posição k.

Saída O k-ésimo maior número.

```
1  S(k,{x_1,...,x_n}) :=
2    if n ≤ 5
3      calcula e retorne o k-ésimo elemento
4    end if
5    m_i := median(x_{5i+1},...,x_{min(5i+5,n)}) para 0 ≤ i < ⌈n/5⌉.
```

```
6     m := S(⌈⌈n/5⌉/2⌉, m_1, ..., m_{⌈n/5⌉−1})
7     L := {x_i | x_i < m, 1 ≤ i ≤ n}
8     R := {x_i | x_i ≥ m, 1 ≤ i ≤ n}
9     i := |L| + 1
10    if i = k then
11      return m
12    else if i > k then
13      return S(k, L)
14    else
15      return S(k − i, R)
16    end if
```

B. Técnicas para a análise de algoritmos

Análise de recorrências

{th:mak}

Teorema B.1 (Akra-Bazzi e Leighton)
Dado a recorrência

$$T(x) = \begin{cases} \Theta(1), & \text{se } x \leq x_0, \\ \sum_{1 \leq i \leq k} a_i T(b_i x + h_i(x)) + g(x), & \text{caso contrário,} \end{cases}$$

com constantes $a_i > 0$, $0 < b_i < 1$ e funções g, h, tal que

$$|g'(x)| \in O(x^c); \qquad |h_i(x)| \leq x/\log^{1+\epsilon} x$$

para um $\epsilon > 0$ e a constante x_0 e suficientemente grande

$$T(x) \in \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$$

com p tal que $\sum_{1 \leq i \leq k} a_i b_i^p = 1$.

Teorema B.2 (Graham et al. (1988))
Dado a recorrência

$$T(n) = \begin{cases} \Theta(1), & n \leq \max_{1 \leq i \leq k} d_i, \\ \sum_i \alpha_i T(n - d_i), & \text{caso contrário,} \end{cases}$$

seja α a raiz com a maior valor absoluto com multiplicidade l do *polinômio característico*

$$z^d - \alpha_1 z^{d-d_1} - \cdots - \alpha_k z^{d-d_k}$$

com $d = \max_k d_k$. Então

$$T(n) = \Theta(n^l \alpha^n) = \Theta^*(\alpha^n).$$

Bibliografia

[1] Manindra Agrawal, Neeraj Kayal e Nitin Saxena. "PRIMES is in P". Em: *Annals of Mathematics* 160.2 (2004), pp. 781–793.

[2] W. R. Alford, A. Granville e C. Pomerance. "There are infinitely many Carmichael numbers". Em: *Annals Math.* 140 (1994).

[3] *Algorithm Engineering*. http://www.algorithm-engineering.de. Deutsche Forschungsgemeinschaft.

[4] H. Alt et al. "Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m \log n})$". Em: *Information Processing Letters* 37 (1991), pp. 237–240.

[5] June Andrews e J. A. Sethian. "Fast marching methods for the continuous traveling salesman problem". Em: *Proc. Natl. Acad. Sci. USA* 104.4 (2007). DOI: 10.1073/pnas.0609910104.

[6] Sanjeev Arora e Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[7] G. Ausiello et al. *Complexity and approximation – Combinatorial Optimization Problems and their Approximability Properties*. Springer-Verlag, 1999. URL: http://www.nada.kth.se/~viggo/approxbook.

[8] Brenda S. Baker. "A new proof for the first fit decreasing bin packing algorithm". Em: *J. Alg.* 6 (1985), pp. 49–70. DOI: 10.1016/0196-6774(85)90018-5.

[9] Jon Bentley e Bob Floyd. "Programming pearls: a sample of brilliance". Em: *Communications of the ACM* 30.9 (set. de 1987), pp. 754–757. ISSN: 1557-7317. DOI: 10.1145/30401.315746.

[10] Claude Berge. "Two theorems in graph theory". Em: *Proc. National Acad. Science* 43 (1957), pp. 842–844.

[11] John R. Black Jr. e Charles U. Martel. *Designing Fast Graph Data Structures: An Experimental Approach*. Rel. téc. Department of Computer Science, University of California, Davis, 1998.

[12] Thomas Bläsius et al. "An Efficient Branch-and-Bound Solver for Hitting Set". Em: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial e Applied Mathematics, jan. de 2022, pp. 209–220. ISBN: 9781611977042. DOI: 10.1137/1.9781611977042.17.

[13] G. S. Brodal, R. Fagerberg e R. Jacob. *Cache Oblivious Search Trees via Binary Trees of Small Height*. Rel. téc. RS-01-36. BRICS, 2001.

[14] Andrei Broder e Michael Mitzenmacher. "Network applications of Bloom filter: A survey". Em: *Internet Mathematics* 1.4 (2003), pp. 485–509.

[15] G. Cattaneo et al. "Maintaining dynamic minimum spanning trees: An experimental study". Em: *Discrete Applied Mathematics* 158.5 (mar. de 2010), pp. 404–425. ISSN: 0166-218X. DOI: 10.1016/j.dam.2009.10.005.

[16] Bernhard Chazelle. "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity". Em: *Journal ACM* 47 (2000), pp. 1028–1047.

[17] Li Chen et al. "Maximum Flow and Minimum-Cost Flow in Almost-Linear Time". Em: *tbd* (2022). DOI: 10.48550/arxiv.2203.00671. arXiv: 2203.00671 [cs.DS].

[18] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.

[19] Ivan Damgård, Peter Landrock e Carl Pomerance. "Average case error estimates for the strong probable prime test". Em: *Mathematics of computation* 61.203 (1993), pp. 177–194.

[20] Brian C. Dean, Michel X. Goemans e Nicole Immorlica. "Finite termination of "augmenting path"algorithms in the presence of irrational problem data". Em: *ESA'06: Proceedings of the 14th conference on Annual European Symposium*. Zurich, Switzerland: Springer-Verlag, 2006, pp. 268–279. DOI: 10.1007/11841036_26.

[21] R. Dementiev et al. "Engineering a Sorted List Data Structure for 32 Bit Keys". Em: *Workshop on Algorithm Engineering & Experiments*. 2004, pp. 142–151.

[22] Yefim Dinitz. "Algorithm for solution of a problem of maximum flow in a network with power estimation". Em: *Doklady Akademii Nauk SSSR* 11 (1970), pp. 1277–1280.

[23] Yefim Dinitz. “Dinitz’ Algorithm: The Original Version and Even’s Version”. Em: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. por Oded Goldreich, Arnold L. Rosenberg e Alan L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 218–240. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_10.

[24] Yann Disser e Martin Skutella. “The Simplex Algorithm is NP-mighty”. Em: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. USA: Society for Industrial e Applied Mathematics, 2015, pp. 858–872. DOI: 10.1137/1.9781611973730.59.

[25] Ran Duan, Seth Pettie e Hsin-Hao Su. “Scaling algorithms for approximate and exact maximum weight matching”. Em: *CoRR* abs/1112.0790 (2011).

[26] Ran Duan, Seth Pettie e Hsin-Hao Su. “Scaling Algorithms for Weighted Matching in General Graphs”. Em: *ACM Trans. Algorithms* 14.1 (2018), pp. 225–231. DOI: 10.1145/3155301.

[27] J. Edmonds. “Paths, Trees, and Flowers”. Em: *Canad. J. Math* 17 (1965), pp. 449–467.

[28] J. Edmonds e R. Karp. “Theoretical improvements in algorithmic efficiency for network flow problems”. Em: *JACM* 19.2 (1972), pp. 248–264.

[29] Pavlos S. Efraimidis e Paul G. Spirakis. “Weighted Random Sampling”. Em: *Encyclopedia of Algorithms*. 2005.

[30] Jenő Egerváry. “Matrixok kombinatorius tulajdonságairól (On combinatorial properties of matrices)”. Em: *Matematikai és Fizikai Lapok* 38 (1931), pp. 16–28.

[31] T. Feder e R. Motwani. “Clique Partitions, Graph Compression and Speeding-Up Algorithms”. Em: *Journal of Computer and System Sciences* 51.2 (out. de 1995), pp. 261–272. ISSN: 0022-0000. DOI: 10.1006/jcss.1995.1065.

[32] T. Feder e R. Motwani. “Clique partitions, graph compression and speeding-up algorithms”. Em: *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing (23rd STOC)*. 1991, pp. 123–133.

[33] L. R. Ford e D. R. Fulkerson. “Maximal flow through a network”. Em: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[34] András Frank. *On Kuhn’s Hungarian Method – A tribute from Hungary*. Rel. téc. Egerváry Research Group on Combinatorial Optimization, 2004.

[35] C. Fremuth-Paeger e D. Jungnickel. "Balanced network flows VIII: a revised theory of phase-ordered algorithms and the $O(\sqrt{n}m \log(n^2/m)/\log n$ bound for the nonbipartite cardinality matching problem". Em: *Networks* 41 (2003), pp. 137–142.

[36] Martin Fürer e Balaji Raghavachari. "Approximating the minimu-degree steiner tree to within one of optimal". Em: *Journal of Algorithms* (1994).

[37] H. N. Gabow. "Data structures for weighted matching and nearest common ancestors with linking". Em: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 434–443.

[38] Harold N. Gabow, Zvi Galil e Thomas H. Spencer. "Efficient implementation of graph algorithms using contraction". Em: *Journal of the ACM* 36.3 (jul. de 1989), pp. 540–572. ISSN: 1557-735X. DOI: 10.1145/65950.65954.

[39] Giorgio Gallo e Stefano Pallottino. "Shortest path algorithms". Em: *Annals of Operations Research* 13.1 (dez. de 1988), pp. 1–79. ISSN: 1572-9338. DOI: 10.1007/bf02288320.

[40] Ashish Goel, Michael Kapralov e Sanjeev Khanna. "Perfect Matchings in $O(n \log n)$ Time in Regular Bipartite Graphs". Em: *STOC 2010*. 2010.

[41] A. V. Goldberg e A. V. Karzanov. "Maximum skew-symmetric flows and matchings". Em: *Mathematical Programming A* 100 (2004), pp. 537–568.

[42] Olivier Goldschmidt e Dorit S. Hochbaum. "Polynomial Algorithm for the k-Cut Problem". Em: *Proc. 29th FOCS*. 1988, pp. 444–451.

[43] Michel Gondran e Michel Minoux. *Graphs and Algorithms*. Wiley, 1984.

[44] Ronald Lewis Graham, Donald Ervin Knuth e Oren Patashnik. *Concrete Mathematics: a foundation for computer science*. Addison-Wesley, 1988.

[45] J. Hadamard. "Sur la distribution des zéros de la fonction zeta(s) et ses conséquences arithmétiques". Em: *Bull. Soc. math. France* 24 (1896), pp. 199–220.

[46] Bernhard Haeupler, Siddharta Sen e Robert E. Tarjan. "Heaps simplified". Em: *(Preprint)* (2009). arXiv:0903.0116.

[47] Carl Hierholzer. "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren". Em: *Mathematische Annalen* 6 (1873), pp. 30–32. DOI: 10.1007/bf01442866.

[48] J. E. Hopcroft e R. Karp. "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs". Em: *SIAM J. Comput.* 2 (1973), pp. 225–231.

[49] Juraj Hromkovič. *Algorithmics for hard problems*. Springer, 2001.

[50] David S. Johnson. “A theoretician’s guide to the experimental analysis of algorithms”. Em: *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. 2002.

[51] David S. Johnson. “Near-optimal bin packing algorithms”. Tese de doutoramento. Massachusetts Institute of Technology. Dept. of Mathematics, 1973. URL: http://hdl.handle.net/1721.1/57819.

[52] David S. Johnson e Michael R. Garey. “A 71/60 theorem for bin packing”. Em: *J. Complex.* 1.1 (1985), pp. 65–106. DOI: 10.1016/0885-064X(85)90022-6.

[53] Michael J. Jones e James M. Rehg. *Statistical Color Models with Application to Skin Detection*. Rel. téc. CRL 98/11. Cambridge Research Laboratory, 1998.

[54] Haim Kaplan e Uri Zwick. “A simpler implementation and analysis of Chazelle’s soft heaps”. Em: *SODA ’09: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. New York, New York: Society for Industrial e Applied Mathematics, 2009, pp. 477–485.

[55] Adam Karczmarz e Jakub Łącki. “Simple Label-Correcting Algorithms for Partially Dynamic Approximate Shortest Paths in Directed Graphs”. Em: *2020 Symposium on Simplicity in Algorithms (SOSA)*. 2005, pp. 106–120. DOI: 10.1137/1.9781611976014.15.

[56] David R. Karger e Clifford Stein. “A new approach to the minimum cut problem”. Em: *Journal of the ACM* 43.4 (1996), pp. 601–640. DOI: 10.1145/234533.234534.

[57] Erica Klarreich. “Researchers Achieve ’Absurdly Fast’ Algorithm for Network Flow”. Em: *Quanta Magazine* (jun. de 2022). URL: https://www.quantamagazine.org/researchers-achieve-absurdly-fast-algorithm-for-network-flow-20220608.

[58] Jon Kleinberg e Éva Tardos. *Algorithm design*. Addison-Wesley, 2005.

[59] Bernhard Korte e Jens Vygen. *Combinatorial optimization – Theory and Algorithms*. 4th. Springer, 2008.

[60] Werner Krauth. *Statistical Mechanics: Algorithms and Computation*. OUP, 2006.

[61] H. W. Kuhn. “The Hungarian Method for the assignment problem”. Em: *Naval Re4search Logistic Quarterly* 2 (1955), pp. 83–97.

[62] Jerry Li e John Peebles. “Replacing Mark Bits with Randomness in Fibonacci Heaps”. Em: *Int. Coloq. Automata, Languages, and Progr.* Ed. por Magnús Halldórsson et al. Vol. 9134. LNCS. 2015, pp. 886–897.

[63] S. Micali e Vijay V. Vazirani. "An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs". Em: *Proc. 21th FOCS*. 1980, pp. 17–27.

[64] Edward Minieka. *Optimization Algorithms for Networks and Graphs*. Dekker, 1978.

[65] L. Monier. "Evaluation and comparison of two efficient probabilistic primality testing algorithms". Em: *Theoret. Comp. Sci.* 12 (1980), pp. 97–108.

[66] J. Munkres. "Algorithms for the assignment and transporation problems". Em: *J. Soc. Indust. Appl. Math* 5.1 (1957), pp. 32–38.

[67] K. Noshita. "A theorem on the expected complexity of Dijkstra's shortest path algorithm". Em: *Journal of Algorithms* 6 (1985), pp. 400–408.

[68] Christos H. Papadimitriou e Kenneth Steiglitz. *Combinatorial optimization: Algorithms and complexity*. Dover. Prentice-Hall, 1982.

[69] Joon-Sang Park, Michael Penner e Viktor K. Prasanna. "Optimizing Graph Algorithms for Improved Cache Performance". Em: *IEEE Trans. Par. Distr. Syst.* 15.9 (2004), pp. 769–782.

[70] Mihai Pătraşcu. *Follow-up: Sampling a discrete distribution*. 19 de set. de 2011. URL: http://infoweekly.blogspot.com/2011/09/follow-up-sampling-discrete.html.

[71] Michael O. Rabin. "Probabilistic algorithm for primality testing". Em: *J. Number Theory* 12 (1980), pp. 128–138.

[72] Emma Roach e Vivien Pieper. "Die Welt in Zahlen". Em: *Brand eins* 3 (2007).

[73] J.R. Sack e J. Urrutia, eds. *Handbook of computational geometry*. Elsevier, 2000.

[74] Iztok Savnik. "Index Data Structure for Fast Subset and Superset Queries". Em: *Availability, Reliability, and Security in Information Systems and HCI*. Springer Berlin Heidelberg, 2013, pp. 134–148. ISBN: 9783642405112. DOI: 10.1007/978-3-642-40511-2_10.

[75] Alexander Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*. Springer, 1997.

[76] Alexander Schrijver. *Combinatorial optimization. Polyhedra and efficiency*. Vol. A. Springer, 2003.

[77] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Science*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 1999.

[78] Mohit Singh e Lap Chi Lau. "Approximating minimum bounded degree spanning trees to within one of the optimal". Em: *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007.

[79] Robert Endre Tarjan. "Finding Optimum Branchings". Em: *Netw.* 7 (1977), pp. 25–35. DOI: 10.1002/net.3230070103.

[80] Terrazon. *Soft Errors in Electronic Memory – A White Paper*. Rel. téc. Terrazon Semiconductor, 2004.

[81] Daniel Ting. "Simple, Optimal Algorithms for Random Sampling Without Replacement". Em: (abr. de 2021). DOI: 10.48550/ARXIV.2104.05091. arXiv: 2104.05091 [cs.DS].

[82] C.-J. de la Vallée Poussin. "Recherches analytiques la théorie des nombres premiers". Em: *Ann. Soc. scient. Bruxelles* 20 (1896), pp. 183–256.

[83] Norman Zadeh. "Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows". Em: *J. ACM* 19.1 (1972), pp. 184–192.

[84] Uri Zwick. "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate". Em: *Theoretical Computer Science* 148.1 (1995), pp. 165–170. DOI: DOI:10.1016/0304-3975(95)00022-O.

Índice