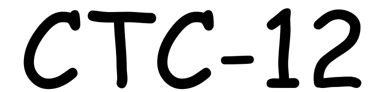
CTC-12

Projeto e Análise de Algoritmos

Carlos Alberto Alonso Sanches



11) Algoritmos probabilísticos e aproximativos

Monte Carlo e *Las Vegas*, fatores de aproximação

Algoritmos probabilísticos

- Características:
  - Fazem escolhas aleatórias durante sua execução.
  - Por esse motivo, seu comportamento pode mudar mesmo quando os dados de entrada são mantidos.
  - Estes algoritmos também são chamados de *aleatorizados* ou *randômicos* (*randomized algorithms*).
- Vantagens:
  - Um algoritmo probabilístico tem boa chance de alcançar resultados equivalentes aos obtidos por um algoritmo convencional semelhante, mas de uma maneira mais eficiente.
  - Geralmente, são de simples elaboração.

Monte Carlo x *Las Vegas*

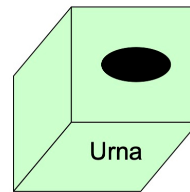
- Algoritmo *Monte Carlo*
  - Garante bom tempo de execução, geralmente polinomial.
  - Sua "aposta" está na corretude: pode fornecer respostas erradas com uma determinada probabilidade.
- Algoritmo *Las Vegas*
  - Garante resposta correta.
  - Sua "aposta" está no tempo de execução: pode não ser polinomial.
- Um algoritmo *Las Vegas*, que fosse forçado a terminar após um limite de tempo, se tornaria um algoritmo *Monte Carlo*.
- A iteração de um algoritmo *Monte Carlo*, combinado com um mecanismo de verificação do resultado, pode gerar um algoritmo *Las Vegas*, desde que haja garantia de término...

Exemplos abordados

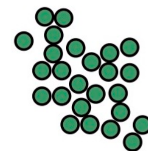
- Algoritmos *Monte Carlo*
  - Problema da urna
  - Valor superior à mediana
  - Verificação do produto de matrizes
  - Detecção de números primos
- Algoritmos *Las Vegas*
  - *QuickSort* aleatório
  - Seleção do k-ésimo elemento

Problema da urna

- Considere uma urna com uma quantidade indeterminada de bolas.



- Há somente duas possibilidades para esta urna:
  1) Contém somente bolas verdes.



  2) Contém uma quantidade **idêntica** de bolas verdes e brancas.

- Dada uma urna, como identificar eficientemente em qual dos dois casos acima ela se enquadra?

Um algoritmo *Monte Carlo*

```
UrnaMC (U,k) {
    while (k > 0) {
        escolha ao acaso uma bola da urna U;
        if (essa bola for branca) return "Caso 2";
               else reponha essa bola na urna U;
        k--;
    }
    return "Provavelmente Caso 1";
}
```

- Probabilidade de erro associada às respostas:
  - Caso 1: $1/2^k$
  - Caso 2: zero

Valor superior à mediana

- Suponha que, dentre n números não ordenados, deseja-se encontrar um valor que seja maior do que a mediana.
- O melhor algoritmo convencional gastará tempo $\Theta(n)$ para encontrar uma resposta.
- E se n for muito grande? Por exemplo, $n = 10^{10}$?
- Uma maneira alternativa: selecionar 100 números aleatoriamente e encontrar o máximo dentre eles.
- Essa solução pode ser calculada em tempo constante, mas estará sujeita a erro com probabilidade $1/2^{100}$.
- Esta probabilidade é muito menor do que a chance de falha de *hardware*, ou mesmo de ocorrer um terremoto aqui...

Verificação do produto de matrizes

- Dadas as matrizes A, B e C de ordem n, deseja-se saber se A.B = C.
- A solução imediata gasta tempo de pior caso $\Theta(n^3)$. Caso se utilize um algoritmo especial (Strassen, por exemplo), este tempo cai para $\Theta(n^k)$, onde $2<k<3$.
- Exemplo com n=3:

$$A = \begin{vmatrix} 12 & 14{,}1 & 18 \\ 81{,}5 & 10{,}11 & 6{,}86 \\ 1 & 2 & 1 \end{vmatrix} \quad B = \begin{vmatrix} 1 & 4{,}11 & 9 \\ 1{,}12 & 1{,}11 & 16{,}86 \\ 11 & 212 & 9 \end{vmatrix} \quad C = \begin{vmatrix} 225{,}792 & 3870{,}971 & 507{,}726 \\ 168{,}2842 & 1800{,}5071 & 965{,}6946 \\ 14{,}24 & 218{,}33 & 51{,}72 \end{vmatrix}$$

$$81{,}5 \;.\; 1 + 10{,}11 \;.\; 1{,}12 + 6{,}86 \;.\; 11 = 168{,}2832$$

Um algoritmo *Monte Carlo*

```
VerifMultMatrMC (A,B,C,k) {
    while (k > 0) {
        gere aleatoriamente um vetor coluna binário r;
        if (A.(B.r) ≠ C.r) return "A.B ≠ C";
        k--;
    }
    return "Provavelmente A.B = C";
}
```

- Considerando o exemplo anterior:

r	A.(B.r)	C.r
[0 0 1]	[507,726 965,6946 51,72]	[507,726 965,6946 51,72]
[0 1 0]	[3880,971 1800,5071 218,33]	[3880,971 1800,5071 218,33]
[0 1 1]	[4388,6970 2766,2017 270,05]	[4388,6970 2766,2017 270,05]
[1 0 0]	[225,792 168,2832 14,24]	[225,792 168,2842 14,24]
[1 0 1]	[733,5179 1133,9778 65,9599]	[733,5179 1133,9788 65,9599]
[1 1 0]	[4106,7629 1968,7903 232,57]	[4106,7629 1968,7913 232,57]
[1 1 1]	[4614,489 2934,4849 284,29]	[4614,489 2934,4859 284,29]

Análise deste algoritmo *Monte Carlo*

- As multiplicações entre matriz e vetor gastam tempo $\Theta(n^2)$.
- Pior caso: quando houver um único valor errado em C.
  - Metade dos vetores r detectarão esse erro (basta terem 1 na posição correspondente).
  - Quando houver um número maior de erros em C, a probabilidade de que algum seja identificado é ainda maior. Pode ocorrer o caso em que eles se ocultem mutuamente pela própria adição ("efeito paridade"), mas isso é muito improvável...
- Após k iterações, podemos concluir que a probabilidade de que este algoritmo apresente uma resposta equivocada é no máximo $1/2^k$.
  - Neste caso, afirmaria que A.B = C, quando na verdade haveria algum erro em C...

Detecção de números primos

- A segurança dos atuais sistemas criptográficos de chave pública baseia-se em um simples fato: é extremamente difícil fatorar números grandes.
- Essa dificuldade torna-se ainda maior se o número a ser fatorado for o produto de dois números primos. Portanto, é conveniente dispor de um eficiente método de geração de números primos grandes.
- O algoritmo convencional para a detecção da primalidade de um número n gasta tempo de pior caso $\Theta(n^{1/2})$.
- Há algoritmos probabilísticos de detecção de números primos mais eficientes que se baseiam em resultados da *Teoria de Números.*

Teste de primalidade de Fermat

- *Pequeno Teorema de Fermat*: se n for primo, então a^{n-1} mod n = 1, onde $a \in N$ e $1<a<n$.
- Esse teorema sugere um algoritmo *Monte Carlo*:

```
PrimFermatMC (n,k) {
    while (k > 0) {
        gere aleatoriamente a∈N tal que 1<a<n;
        if (a^(n-1) mod n ≠ 1) return "n não é primo";
        k--;
    }
    return "Provavelmente n é primo";
}
```

- No entanto, também há números não primos que satisfazem a condição do teorema: são os chamados *números de Carmichael*.
  - Exemplos: 561 (3.11.17), 1729 (7.13.19), 2821 (7.13.31), 6601 (7.23.41), 10585 (5.29.73), 15841 (7.31.73), 29341 (13.37.61), 115921 (13.37.241), etc.
- Embora sejam poucos, não se conhece a quantidade de números de Carmichael. Por isso, não é possível calcular a probabilidade de erro deste algoritmo...

Teorema de Levin

- Sejam n um número inteiro positivo ímpar e $a \in N$, com $0<a<n$.
  - Se n for primo, então:
    - $a^{(n-1)/2}$ mod n = 1 para metade dos valores de a
    - **e** $a^{(n-1)/2}$ mod n = -1 para a outra metade dos valores de a
  - Se n não for primo, então:
    - $a^{(n-1)/2}$ mod n = 1 para todos os valores de a
    - **ou** $a^{(n-1)/2}$ mod n $\neq \pm 1$ para pelo menos metade dos valores de a
- Exemplos:

n = 11

n é primo

a	$a^{(n-1)/2}$ mod n
1	+1
2	-1
3	+1
4	+1
5	+1
6	-1
7	-1
8	-1
9	+1
10	-1

n = 9

n não é primo

a	$a^{(n-1)/2}$ mod n
1	1
2	7
3	0
4	4
5	4
6	0
7	7
8	1

Um algoritmo *Monte Carlo*

```
PrimLeviMC (n,k) {
    while (k > 0) {
        gere aleatoriamente a∈N tal que 0<a<n;
        if (a^(n-1) mod n ≠ ±1) return "n não é primo";
        k--;
    }
    if (todos os restos forem 1) return "Provavelmente n não é primo";
    return "Provavelmente n é primo";
}
```

- Complexidade de tempo: $O(k.\log n) = O(\log n)$.
- Casos em que o algoritmo dá resposta errada:
  - quando n é primo e todos os k restos forem 1;
  - quando n não é primo e todos os k restos forem 1 ou -1.
- Em ambos os casos, a probabilidade de erro é $1/2^k$.

QuickSort aleatório

- No algoritmo *QuickSort*, a escolha do pivô pode ser feita de forma aleatória, dando origem a um algoritmo *Las Vegas*:

```
QuickSortLV(min, max) {
    if (min < max) {
       p = PartitionLV(min, max);
       QuicksortLV(min, p-1);
       QuicksortLV(p+1, max);
    }
}
```

```
int PartitionLV(left, right) {
    x = random(left, right);
    v[left] ↔ v[x];
    pivot = v[left];
    l = left + 1;
    r = right;
    while (true) {
        while (l < right && v[l] < pivot) l++;
        while (r > left && v[r] >= pivot) r--;
        if (l >= r) break;
        v[l] ↔ v[r];
    }
    v[left] = v[r];
    v[r] = pivot;
    return r;
}
```

- Veremos a seguir que este algoritmo gasta tempo O(n.log n).

Análise do *QuickSort* aleatório

- Seja T(n) o número esperado de comparações realizadas pelo *QuickSort* aleatório em um vetor com n elementos. Sabemos que o número esperado de trocas entre elementos desse vetor será no máximo igual a T(n).
- Considerando cada escolha do pivô com mesma probabilidade, temos:

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1))$$

$$T(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

- De maneira análoga ao caso médio do *QuickSort*, podemos provar por indução em n que $T(n) \leq 2n.\ln n$.
- Portanto, T(n) = O(n.log n).

Seleção do k-ésimo elemento

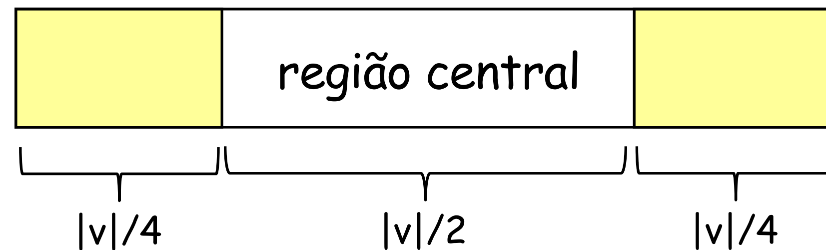
- Dado um vetor v *não ordenado* com n elementos, deseja-se encontrar seu k-ésimo menor elemento, $1 \le k \le n$.
- Um algoritmo *Las Vegas* :

```
SelectLV (v,k) {
    escolha aleatoriamente x ∈ v;
    S⁻ ← ∅;
    S⁺ ← ∅;
    for each aᵢ ∈ v
        if (aᵢ < x) S⁻ ← S⁻ ∪ {aᵢ};
                else S⁺ ← S⁺ ∪ {aᵢ};
    if (|S⁻| = k-1) return x;
    if (|S⁻| ≥ k) return SelectLV (S⁻,k);
    return SelectLV (S⁺,k-|S⁻|);
}
```

- O valor de x afeta a complexidade de tempo:
  - Uma boa escolha: x é a mediana $\Rightarrow T(n) \le T(n/2) + c.n \Rightarrow T(n) = O(n)$
  - Uma má escolha: x é o menor elemento $\Rightarrow T(n) \le T(n-1) + c.n \Rightarrow T(n) = O(n^2)$

Escolhas aleatórias

- Ideia chave: em cada recursão, o tamanho do conjunto escolhido decresce segundo uma determinada taxa.
- Considere a seguinte representação do vetor v *ordenado* :



- Se x estiver na região central de v, no pior caso haverá recursão com um vetor de tamanho 3n/4.
- A probabilidade de que isso ocorra é 1/2, pois metade dos elementos de v estão na região central.
- Esta argumentação pode ser generalizada, isto é, para regiões centrais com outros tamanhos.

Análise deste algoritmo *Las Vegas*

- Seja X a variável aleatória que corresponde ao número de passos do algoritmo `selectLV (n,k)`.
- Consideremos a chamada inicial desse algoritmo como sua fase 0, e cada eventual recursão como uma nova fase de sua execução.
- Se X_i é o número de passos na i-ésima fase, então $X = X_0 + X_1 + X_2 + \ldots$
- Portanto, bastam em média duas tentativas de escolha de x em cada fase para que ou $|S^+| \geq n/4$ ou $|S^-| \geq n/4$...
- Deste modo, o tempo de execução será provavelmente O(n).

Algoritmos aproximativos

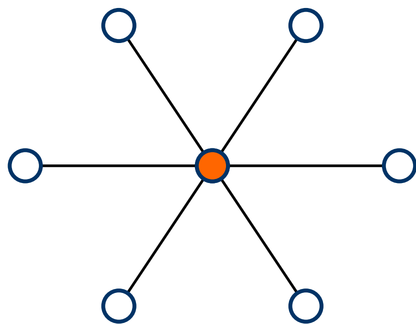
- O que se pode fazer na resolução de um problema intratável?
- Algumas possibilidades:
  - Quando a entrada for pequena, um algoritmo de tempo exponencial talvez seja satisfatório.
  - Identificar alguns casos especiais e relevantes que sejam solucionáveis em tempo polinomial.
  - Encontrar soluções *quase ótimas* em tempo polinomial:
    - através de *heurísticas* (não serão estudadas neste curso);
    - com *algoritmos aproximativos* (ou *aproximados*), cuja otimalidade da solução é determinada com precisão matemática.

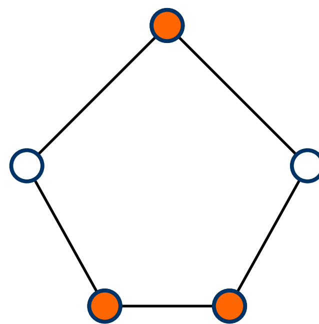
Fator de aproximação

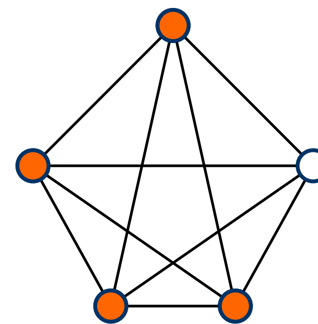
- Em problemas de otimização, cada solução tem um custo associado:
  - a solução ótima tem custo C^*;
  - um algoritmo aproximativo proporciona uma solução quase ótima de custo C.
- Comparação entre C^* e C:
  - problemas de minimização: $C/C^* > 1$;
  - problemas de maximização: $C^*/C > 1$.
- Se um algoritmo aproximativo encontra soluções onde máx $\{C/C^*, C^*/C\} \leq \rho(n)$, para qualquer entrada de tamanho n, então dizemos que tem *fator de aproximação* $\rho(n)$, ou seja, é $\rho(n)$-aproximado.
- Veremos alguns exemplos.

Cobertura de vértices

- Dado um grafo $G=(V,E)$, uma cobertura de vértices é um subconjunto $V' \subseteq V$ tal que, se $(u,v) \in E$, então ou $v \in V'$ ou $u \in V'$ (ou ambos).
- O *Problema da Cobertura de Vértices* consiste em encontrar uma cobertura mínima de G.
- Exemplos:



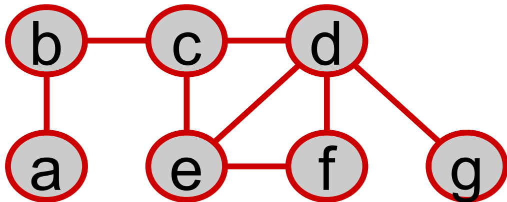


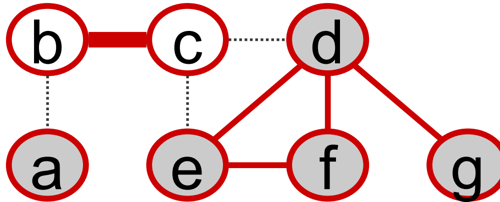


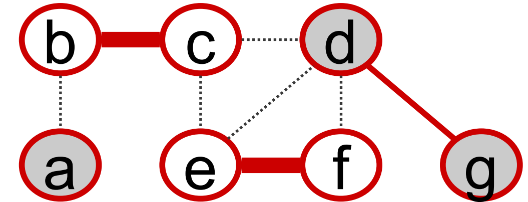
Uma solução aproximada

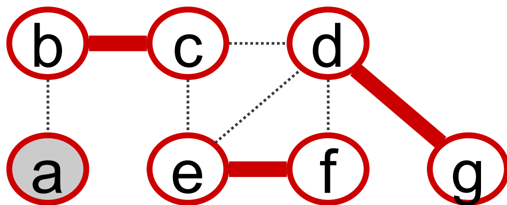
- Ideia:
  - Escolher uma aresta qualquer e incluir seus vértices na solução.
  - Remover de G todas as arestas incidentes nesses dois vértices.
  - Repetir o processo até que não haja mais arestas em G.

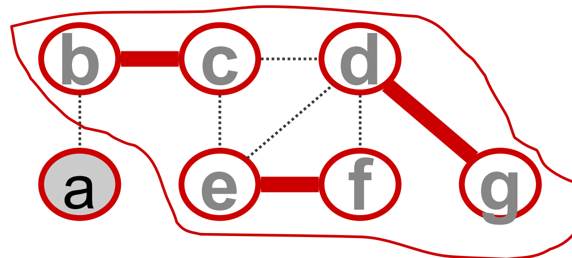
Exemplo



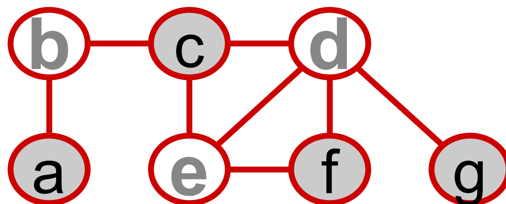








Solução quase ótima
(C = 6 vértices)



Solução ótima
(C^* = 3 vértices)

Algoritmo

```
ApproxVertexCover (V,E) {
    X ← ∅;
    S ← E;
    while (S ≠ ∅) {
        Seja <u,v> ∈ S;
        X ← X ∪ {u} ∪ {v};
        Remover de S toda aresta incidente em u ou v;
    }
    return X;
}
```

Tempo: $O(|V| + |E|)$

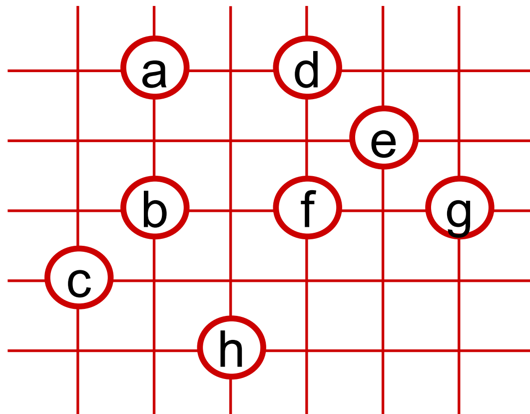
Cálculo do fator de aproximação

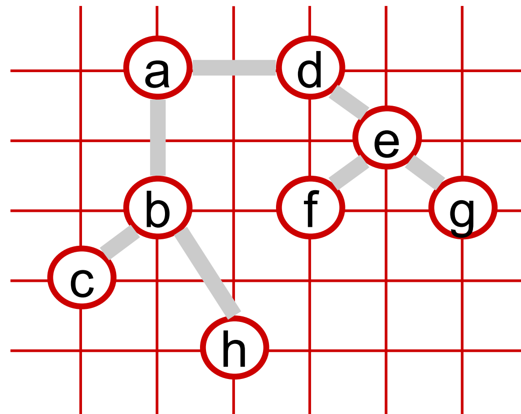
- Seja A o *conjunto de arestas* escolhidas pelo algoritmo anterior.
- É fácil observar que a cobertura encontrada tem custo $C = 2.|A|$, ou seja, $|A| = C/2$.
- Por outro lado, a cobertura ótima deverá incluir ao menos um vértice de cada aresta de A. Portanto, $C^* \geq |A|$.
- Logo, $C^* \geq C/2$, ou seja, $C/C^* \leq 2$.
- Portanto, esse algoritmo é 2-aproximado.

Caixeiro viajante

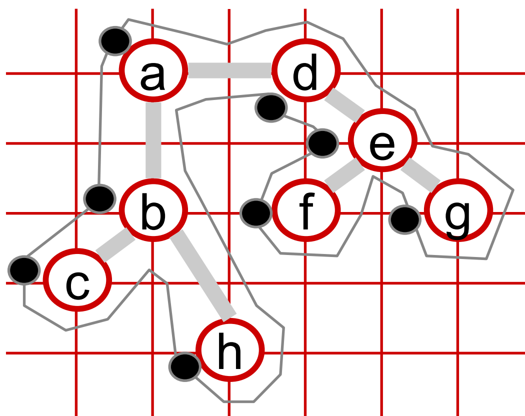
- Dado um grafo ponderado G, deseja-se encontrar um ciclo de custo mínimo que passe uma única vez em cada vértice.
- É conhecido um algoritmo aproximativo para um caso particular deste problema: quando o grafo é *completo* (todo par de vértices é conectado) e os custos das arestas são *distâncias euclidianas*.
- Ideia desse algoritmo:
  - Encontrar a árvore T de espalhamento de custo mínimo.
  - Fazer o percurso pré-ordem em T.
  - Transformar esse percurso num ciclo.

Exemplo

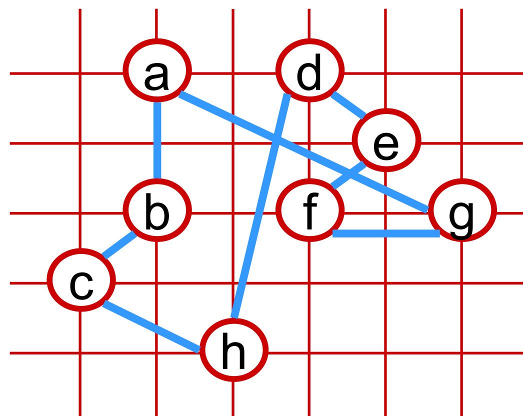




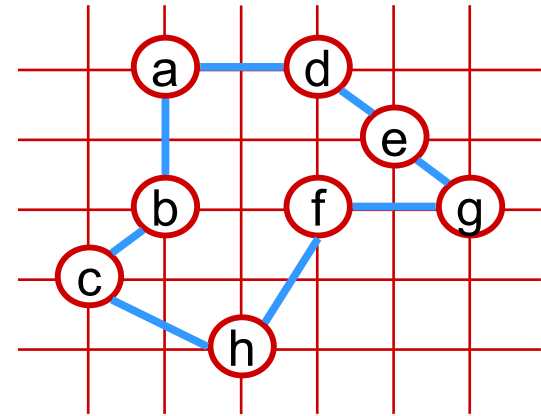
Árvore T: espalhamento de custo mínimo



Ciclo C′ (com repetição de arestas)



Solução C quase ótima



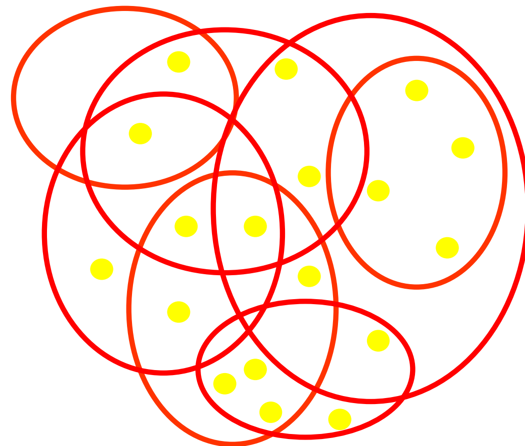
Solução C* ótima

Comparação com a solução ótima

- Lembrando:
  - T: árvore de espalhamento de custo mínimo
  - C': ciclo decorrente do percurso pré-ordem em T (com repetição de arestas)
  - C: ciclo baseado em C' (eliminando as arestas repetidas)
  - C^*: ciclo de custo mínimo
- Seja $c(G)$ o custo associado a um grafo G, ou seja, a somatória dos custos das suas arestas.
- Se removermos uma aresta do ciclo mínimo C^*, obteremos uma árvore de espalhamento. Portanto, $c(T) \leq c(C^*)$.
- No ciclo C', cada aresta de T ocorre exatamente 2 vezes. Portanto, $c(C) \leq c(C') = 2.c(T)$.
- Logo, $c(C) \leq 2.c(C^*)$, ou seja, $c(C)/c(C^*) \leq 2$.
- Este algoritmo também é 2-aproximado.

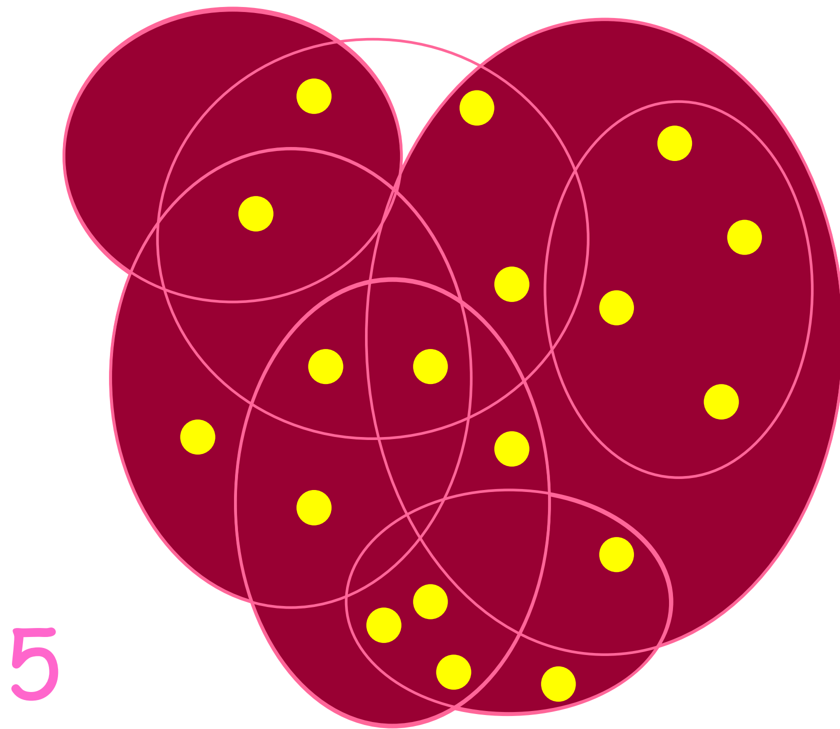
Cobertura de conjuntos

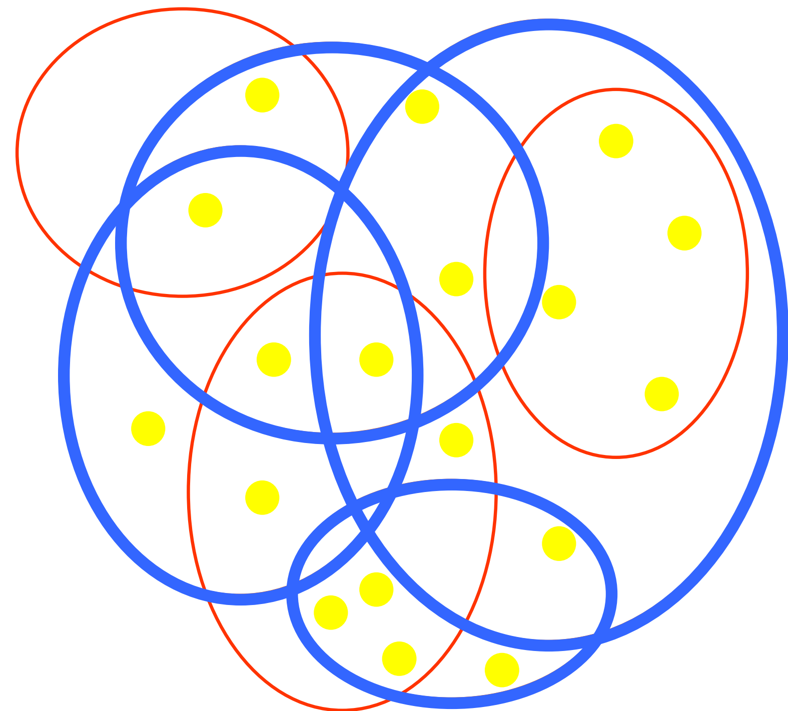
- Dados:
  - um conjunto X com n elementos;
  - uma família F de subconjuntos de X, onde cada elemento de X pertence a pelo menos um desses subconjuntos.
- Deseja-se encontrar o menor subconjunto de F que contenha todos os elementos de X.
- Exemplo:



Uma solução aproximada

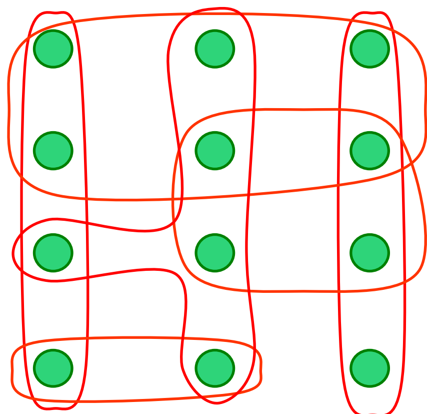
- Um algoritmo guloso: em cada passo, escolhe o subconjunto com o maior número de elementos ainda não cobertos.
- Exemplo:



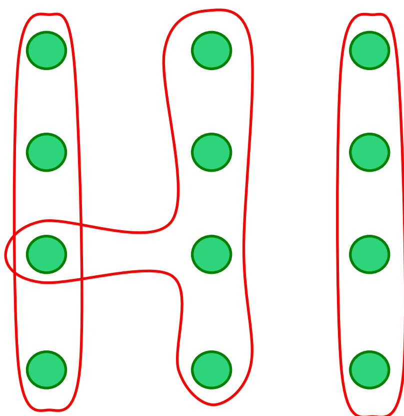


Solução ótima: 4

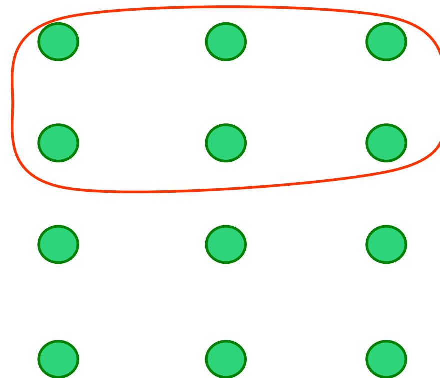
Outro exemplo



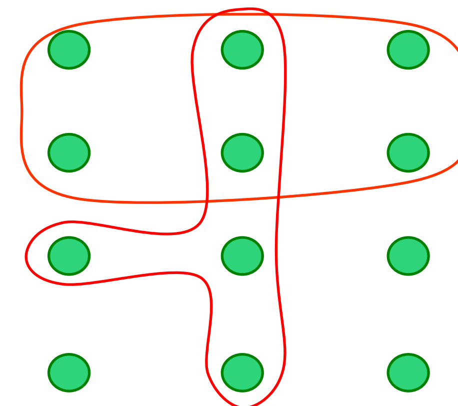
Subconjuntos



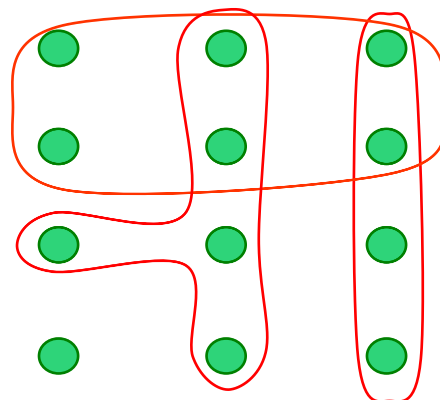
Solução ótima



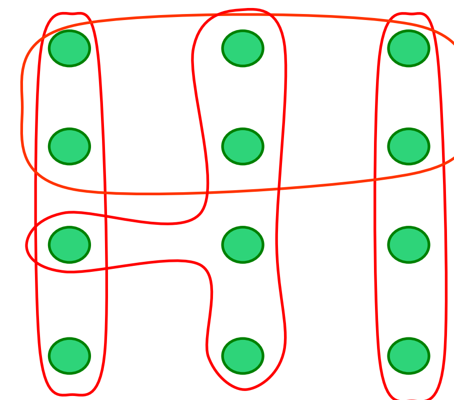
1ª escolha



2ª escolha



3ª escolha



4ª escolha

Algoritmo guloso de Chvátal (1979)

```
GreedySetCover (X,F) {
    R ← X;    // R contém os elementos ainda não cobertos
    C ← ∅;
    while (D ≠ ∅) {
          selecionar S ∈ F que maximiza |S ∩ R|;   } O(n.|F|)
          R ← R - S;
          C ← C ∪ {S};
    }
    return C;
}
```

mín (n, |F|) iterações

Tempo: $O(n.|F|.\text{mín}(n,|F|))$

Cálculo do fator de aproximação

- Seja S_i o i-ésimo subconjunto selecionado pelo algoritmo guloso. Quando S_i for incluído na cobertura C, diremos que incorre em um custo 1. Desse modo, o custo total da solução gulosa será o número de subconjuntos incluídos em C.
- Para cada subconjunto incluído em C, dividiremos o seu custo entre os elementos de X que foram cobertos pela primeira vez.
- Portanto, se $x \in X$ é coberto pela primeira vez por S_i, então $c_x = 1/|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|$.
- Depois que o algoritmo guloso selecionou S_i, seja $S \in F$ um subconjunto restante.
- Seja $u_i = |S - (S_1 \cup S_2 \cup \ldots \cup S_i)|$ o total de elementos de S ainda não cobertos.
- Por definição, $u_0 = |S|$.
- Sabemos também que $u_{i-1} > u_i$, pois $u_{i-1} - u_i$ é quantidade de elementos cobertos por S_i pela primeira vez.
- Seja k o índice mínimo tal que $u_k = 0$, ou seja, $S = S_1 \cup S_2 \cup \ldots \cup S_k$ e $S \neq S_1 \cup S_2 \cup \ldots \cup S_{k-1}$.
- Vamos calcular a somatória dos custos associados aos elementos de S:

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} \frac{(u_{i-1} - u_i)}{|Si - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

Cálculo do fator de aproximação

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} \frac{(u_{i-1} - u_i)}{|Si - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} \frac{(u_{i-1} - u_i)}{u_{i-1}}$$, pois $|Si - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| \geq u_{i-1}$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}$$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j}$$, pois $j \leq u_{i-1}$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)$$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))$$

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Sabemos que $H(n) \leq 1 + \ln n$

$$\sum_{x \in S} c_x \leq H(u_0) - H(u_k)$$

$$\sum_{x \in S} c_x \leq H(u_0) - H(0)$$

$$\sum_{x \in S} c_x \leq H(u_0)$$

$$\sum_{x \in S} c_x \leq H(|S|)$$

Cálculo do fator de aproximação

Sabemos que $|C| = \sum_{x \in X} c_x$

Como x pode estar em mais de um subconjunto: $\sum_{s \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$

Portanto: $|C| \leq \sum_{s \in C^*} \sum_{x \in S} c_x$

$$|C| \leq \sum_{s \in C^*} H(|S|)$$

$$|C| \leq |C^*|. \sum_{s \in C^*} H(\text{máx}|S|)$$

$$|C| \leq |C^*|. (1 + \ln n)$$