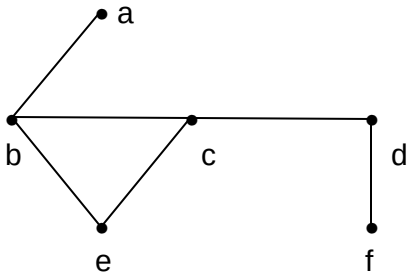
Teoria dos Grafos

José de Oliveira Guimarães
Departamento de Computação - UFSCar
jose@dc.ufscar.br

1 Introdução

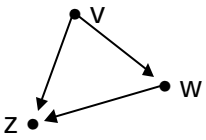
1.1 Conceitos Básicos

Um grafo G = (V, E) é um conjunto V de vértices e um conjunto E de arestas (edges em Inglês) onde cada aresta é um par de vértices (Ex.: (v, w)). Um grafo é representado graficamente usando bolinhas para vértices e retas ou curvas para arestas. Exemplo:



Este grafo possui V = {a, b, c, d, e, f} e E = {(a, b), (b, c), (b, e), (c, e), (c, d), (d, f)} onde (a, b) é uma aresta entre vértice a e b. Normalmente, arestas do tipo (a, a) não são permitidas.

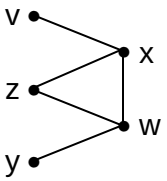
Um grafo pode ser dirigido ou não dirigido. Em um grafo dirigido, a ordem entre os vértices de uma aresta (v, w) é importante. Esta aresta é diferente da aresta (w, v) e é representada com uma flecha de v para w:



V = {v, w, z} E = {(v, w), (v, z), (w, z)}

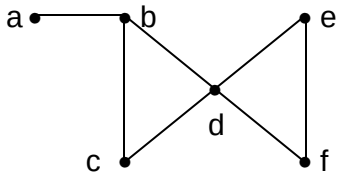
Em um grafo não dirigido, (v, w) = (w, v).

Um *caminho* (path) é uma seqüência de vértices $v_1, v_2, \ldots, v_n$ conectados por arestas $(v_1, v_2), (v_2, v_3), \ldots (v_{n-1}, v_n)$. As arestas são também consideradas como parte do caminho. Ex.:



caminhos: v, x, w, y
v, x
y, w, z

Um *circuito* é um caminho onde $v_1 = v_n$, como b, c, d, e, f, d, b.



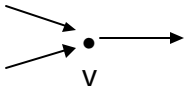
Um circuito será simples se nenhum vértice aparecer mais de uma vez, exceto o primeiro e o último. Um circuito simples é chamado de *ciclo*.

Definição: Dado um grafo, dizemos que vértice v é adjacente a vértice w (ou aresta E) se existe aresta (v, w) no grafo (ou e = (v, w)).

Definição: Um grafo é conectado se existe um caminho entre dois vértices quaisquer do grafo.

Definição: Dígrafo é um grafo dirigido.

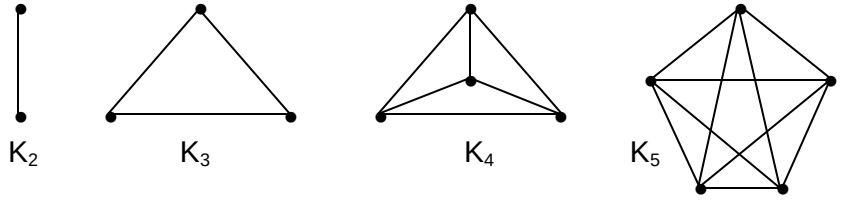
Definição: O grau de um vértice é o número de arestas adjacentes a ele. Em um grafo dirigido, o grau de entrada de um vértice v é o número de arestas (w, v) e o grau de saída é o número de arestas (v, w). Exemplo:



v possui grau de entrada 2 e grau de saída 1.

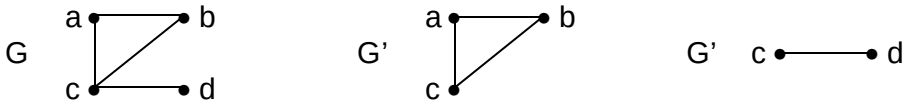
Definição: Uma fonte é um vértice com grau de entrada 0 e grau de saída ≥ 1. Um sumidouro é um vértice com grau de saída 0 e grau de entrada ≥ 1.

Definição: Um grafo é completo quando existe uma aresta entre dois vértices quaisquer do grafo. O grafo completo de n vértices é denotado por K_n. Exemplos:

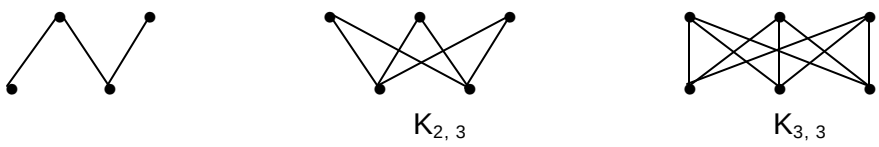


(a) (b) (c) (d)

Definição: Um subgrafo G' = (V', E') de um grafo G = (V, E) é um grafo tal que V' ⊆ V e E' ⊆ E. Exemplos:

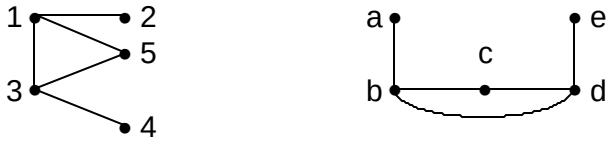


Definição: Um grafo G = (V, E) é bipartido se V pode ser dividido em dois conjuntos V_1 e V_2 tal que toda aresta de G une um vértices de V_1 a outro de V_2. Exemplos:



Um grafo bipartido completo (GBC) possui uma aresta ligando cada vértice de V_1 a cada vértice de V_2. Se $n_1 = |V_1|$ e $n_2 = |V_2|$, o GBC é denotado por $K_{n1, n2}$.

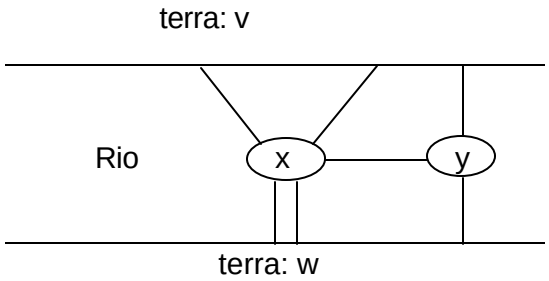
Definição: Dados dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, dizemos que G_1 é isomorfo a G_2 se e somente se existe uma função f: $V_1 \rightarrow V_2$ tal que $(v, w) \in E_1$ se $(f(v), f(w)) \in E_2$, para todo $v, w \in V_1$. Exemplo:



f = {(1, b), (5, c), (3, d), (2, a), (4, e)}

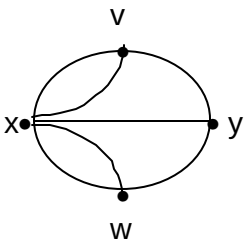
1.2 História

O primeiro problema de teoria dos grafos foi o das pontes de Könisberg. Como no desenho:



Esta cidade possuía um rio com duas ilhas conectadas por sete pontes como mostra o desenho acima. O problema é saber se é possível caminhar de um ponto qualquer da cidade e retornar a este ponto passando por cada ponte exatamente um vez.

Euler resolveu este problema criando um grafo em que terra firme é vértice e ponte é aresta:



Quando caminhamos por um vértice, nós temos que entrar e sair dele (ou vice-versa, no caso do ponto inicial), o que significa que usamos um número par de arestas cada vez que passamos por um vértice.

Como o grafo acima possui vértices com número ímpar de arestas, a resposta para o problema é NÃO.

1.3 Aplicações de Teoria dos Grafos

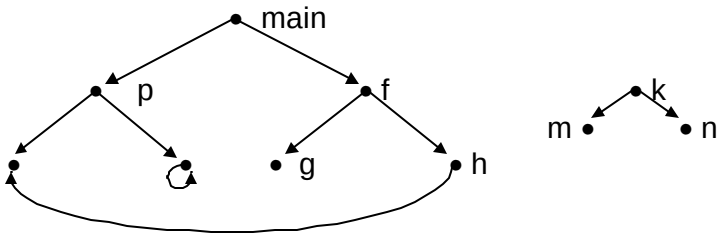
Nos itens abaixo são detalhados alguns problemas que podem ser resolvidos utilizando teoria dos grafos.

❖ Existem funções inúteis no programa?

Neste exemplo utilizaremos a linguagem C. Considere que funções são vértices e existe aresta de f para g se existe chamada a g no corpo de f:

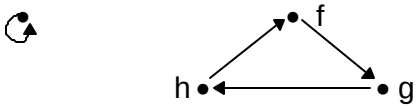
```
void f (int n)
{ if   (n > 5)
     g ( );
  …
}
```

Monta-se um grafo de todo o programa:



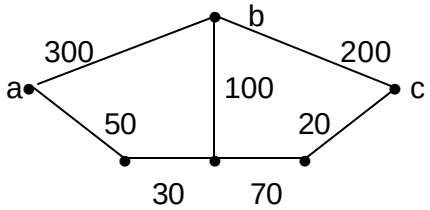
A execução do programa começa na função main que pode chamar as funções p e f. A função f pode chamar g e h. Claramente, funções k, m e n nunca serão chamadas e podem ser removidas na ligação do programa.

❖ Usando a mesma representação, podemos descobrir se um programa possui recursão direta ou indireta. Pode existir recursão se existe ciclo no grafo:



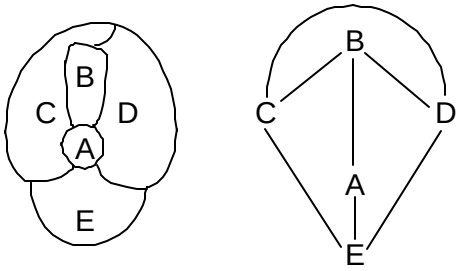
❖ Um vendedor deve passar por várias cidades e retornar ao ponto inicial. Qual o trajeto de menor distância possível ?

❖ Qual a menor distância entre duas cidades a e c ?



❖ Como ir da cidade "a" a "c" passando pelo número mínimo de cidades?

❖ Quantas cores são necessárias para colorir um mapa no plano?

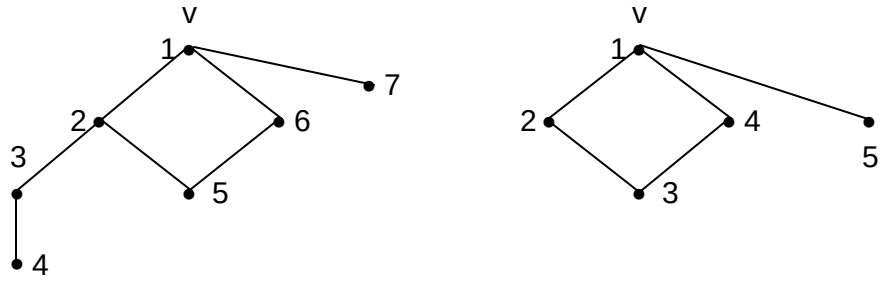


❖ Outros:
- Labirinto
- Eliminação de código morto
- make do Unix

1.4 Algoritmos de Busca

- Busca em profundidade (Depth-First Search DFS)

Uma DFS começa em um vértice v chamado raiz e caminha por todos os vértices que podem ser alcançados a partir de v. Observe que v é um parâmetro passado ao algoritmo DFS. Qualquer vértice pode ser a raiz. Diremos que v é conectado a w se existir a aresta (v, w). Os desenhos abaixo mostram a ordem de visita aos vértices de acordo com buscas em profundidade começando em v. Vértice numerado n é visitado antes de n+1.



Observe que, se executado manualmente por duas pessoas, o algoritmo DFS pode produzir duas numerações diferentes. Por exemplo, no grafo da esquerda acima as numerações 1 2 3 4 5 6 7 e 1 6 5 2 3 4 7 estão ambas corretas para DFS.

O algoritmo para busca em profundidade marca v (raiz) como visitado, pega um vértice w conectado a v (a aresta (v, w) existe) e continua a DFS em w. Depois que a busca em w termina, o algoritmo toma outro vértice z conectado a v ainda não visitado e faz a busca em z. O algoritmo termina quando todos os vértices ligados a v já foram marcados (visitados). O algoritmo para DFS é mostrado a seguir e incorpora código para fazer preWork e postWork, que são códigos executados ao marcar um vértice e após visitar uma aresta. PreWork e postWork dependem da finalidade para a qual DFS está sendo utilizada.

```
Algorithm DFS(G, v)
    Entrada: G = (V, E) e um vértice v ∈ V.
begin
marque v
faça preWork sobre v
for each aresta (v, w) ∈ E do
  if w não foi marcado
  then
    DFS(G, w);
    faça postWork para (v, w)
  endif
end
```

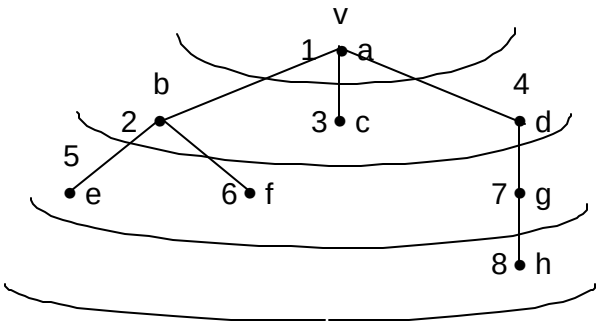
O algoritmo DFS pode ser usado para percorrer todos os vértices de um grafo e para descobrir se um grafo é conectado ou não. Se, após uma DFS, todos os vértices forem visitados, então o grafo é conectado.

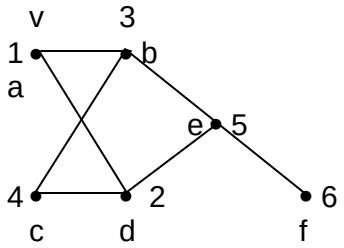
- Busca em largura (Breadth-First Search)

Uma BFS é feita em um grafo G = (V, E) começando em um vértice v. Primeiro o algoritmo visita v e todos os vértices conectados a v, chamados filhos de v. Isto é, o algoritmo visita vértices w tal que (v, w) ∈ E.

No segundo passo, o algoritmo visita todos os netos de v. Isto é, os vértices que não estão conectados diretamente a v mas estão conectados a algum vértice que está conectado a v. O algoritmo prossegue deste modo até que todos os vértices alcançáveis por v sejam visitados.

O primeiro passo do BFS visita todos os vértices que estão a uma aresta de distância de v, o segundo passo visita vértices que estão a duas arestas de distância de v e assim por diante. Os desenhos a seguir ilustram esta interpretação do BFS.





O algoritmo é dado abaixo.

```
Algorithm BFS (G, V)

  Entrada: Grafo G = (V, E) e vértice v.

begin
```

```
marque v
coloque v no fim da fila F
while F não é vazia do
  begin
  remova o primeiro vértice w de F
  faça preWork sobre v
  for each aresta (w, z), tal que Z não é marcado, do
    begin
    marque z
    insira z no fim da fila F
    end
  end
end
```

Não há postWork para busca em largura.

1.5 Exercícios

Algumas recomendações para fazer esta lista e para a prova:

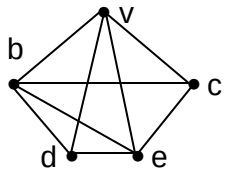
- ✓ todas as questões devem ser justificadas ou provadas;
- ✓ você pode fazer quaisquer algoritmos pedidos nos exercícios em linguagem de alto nível, a mesma empregada nos algoritmos DFS e BFS;
- ✓ se você não sabe fazer uma prova, experimente trabalhar com exemplos de grafos até ter uma idéia intuitiva de como deve ser a prova;
- ✓ use e abuse de indução finita, a ser ainda estudada;
- ✓ o contrário de "todos os elementos do conjunto satisfazem P" é "existe um elemento do conjunto que não satisfaz P" e vice-versa. Esta informação é utilizada em provas por absurdo onde é necessário negar a proposição que se quer provar;
- ✓ quando provar algo por absurdo, escreva o contrário da hipótese para deixar claro o que você está fazendo;
- ✓ os teoremas do tipo "A é válido se e somente se B é válido" exigem duas provas: "A implica B" e "B implica A";
- ✓ uma proposição do tipo "Se A ocorrer, então B *pode* acontecer" necessita somente de um exemplo para ser provado;
- ✓ O número entre parênteses após o número do exercício diz a importância relativa deste exercício. Números maiores são mais importantes.

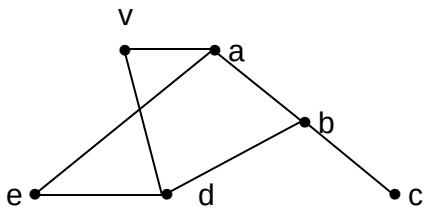
1. Faça um grafo com três vértices de grau par e dois de grau ímpar. Tente ficar com apenas um de grau ímpar. Acrescente uma aresta no grafo de tal forma que existam quatro vértices de grau par. Acrescente um vértice no grafo original e qualquer número de arestas de tal forma que o número de vértices de grau ímpar fique igual a três. Pode ser que algumas das operações anteriores não seja possível.

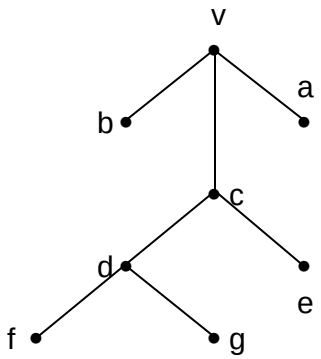
2. Prove: em um grafo qualquer, há um número par de vértices de grau impar. O grau de um vértice é o número de arestas adjacentes a ele. Dica: este teorema pode ser provado apenas com informações locais a cada vértice. Utilize as regras da aritmética na prova.

3. Considere uma representação por grafos de uma entrega de presentes de amigo invisível onde as pessoas são os vértices e existe aresta de v para w se v irá presentear w. Diga qual é a forma *geral* do grafo assim formado.

4. Faça a busca em profundidade nos grafos abaixo, começando em v.





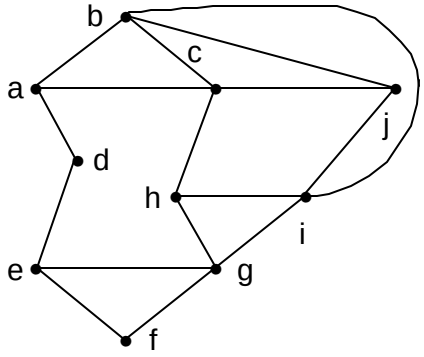


5. Em orientação a objetos, os métodos de uma classe B são os métodos definidos em B e aqueles herdados das superclasses de B, o que pode ser representado por um grafo. Sendo met (B) o conjunto dos métodos definidos em uma classe (sem contar os métodos herdados) e super (B) o conjunto das superclasses de B, faça um algoritmo que retorne todos os métodos de uma dada classe usando a representação em grafos. Considere que todas as classes definam métodos com nomes diferentes entre si.

6. Explique como BFS pode ser usado para descobrir a distância mínima, em número de arestas, entre dois vértices de um grafo.

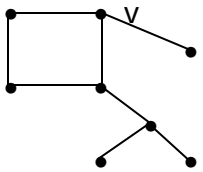
7. Faça um algoritmo que elimina de um programa todas as subrotinas que nunca poderão ser chamadas em execução.

8. Encontre um caminho entre a e j no grafo abaixo que passe por todas as arestas exatamente uma vez.



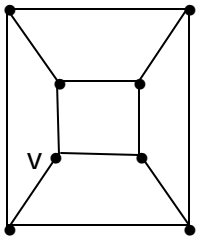
9. Faça um algoritmo que particione um grafo G em seus componentes conexos G_1, G_2, … G_n.

10. Faça DFS e BFS nos grafos abaixo, começando em v.



(a)

(b)



(c)

11. Encontre, se houver, um circuito em cada um dos grafos acima.

12. Faça um algoritmo para coleta de lixo em uma representação do programa em forma de grafos e outras estruturas de sua escolha.

13. Um sistema de computação (banco de dados, sistema operacional) possui uma senha de acesso para cada usuário que dá direito ao uso do sistema. Um usuário pode dar a outro o direito de usar sua área (ou conta). Se x puder usar a área de y e y puder usar a de z, então x poderá usar a área de z.

Usando uma entrada de dados de sua escolha, faça um algoritmo que retorna true se um usuário A pode usar a área de B.

14. Dado um grafo G = (V, E), construa um grafo G' = (V, E') tal que (v, w) $\in$ E' se existe caminho de v para w em G. Usando esta idéia, resolva o algoritmo anterior em tempo constante, após pré-processar os dados de entrada.

15. Prove que, dadas seis pessoas, uma de duas coisas acontece: ou três delas se conhecem ou três não se conhecem.

16. Considere o seguinte algoritmo para construir uma árvore de DFS para um grafo G:

```
Algoritm Build_DFS_Tree(G, v)
  Entrada: G = (V, E) e v um vértice de V
  Saída: T, a árvore DFS de T, inicialmente vazia.
          para cada vértice, v.DFS é inicializado com um número
de
         busca

begin
Inicialmente, DFS_number = 1;

use DFS com o seguinte preWork:
  v.DFS = DFS_number;
  DFS_number++;
e o seguinte postWork:
```

```
  if w não estava marcada, adicione a aresta (v, w) em T
end
```

Um vértice v é chamado de ancestral de w em T com raiz r se v está no caminho único entre r e w. Então w é um descendente de v.

Prove que cada aresta de um grafo G = (V, E) ou pertence à árvore T construída por `Build_DFS_Tree` ou conecta dois vértice de G, um dos quais é um ancestral de outro em T.

17. Prove: seja G = (V, E) um grafo dirigido e seja T = (V, F) uma árvore DFS de G. Se (v, w) é uma aresta de E tal que v.DFS < w.DFS, então w é um descendente de v na árvore T.

18. A afirmação da questão anterior vale também para grafos não dirigidos ?

19. Faça um grafo dirigido de tal forma que,

a) se DFS começar em um dos vértices, todos os demais são atingidos. Se começar em qualquer dos outros vértices, nenhum será atingido;
b) se DFS começar no vértice v, quatro dos oitos vértices serão atingidos. Se a DFS começar em w e z, três vértices serão atingidos. Se começar em t, u e z, dois vértices serão atingidos. Se começar em k e m, apenas o vértice s será atingido, que é um sumidouro.

20. Verifique se uma árvore binária é balanceada. Use apenas preWork e postWork de DFS.

21. Um subgrafo induzido de G = (V, E) é um grafo H = (U, F) tal que $U \subseteq V$ e F inclui todas as arestas (v, w) em E tal que v e w estão em U. Uma árvore de espalhamento de G é um subgrafo conectado que contém todos os vértices e nenhum ciclo.

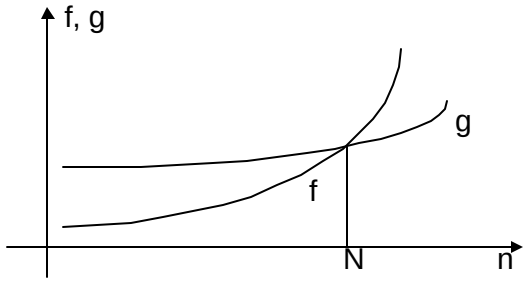
Seja G = (V, E) um grafo não dirigido conectado e seja T uma árvore DFS de G com raiz em v. Prove:

a) Seja H um arbitrário subgrafo induzido de G. Mostre que a interseção de T e H não é necessariamente a árvore de espalhamento de H;
b) Seja R uma subárvore de T e seja S um subgrafo de G induzido pelos vértices em R. Prove que R pode ser uma árvore de DFS de S.

22. Você está organizando uma conferência de cientistas de diferentes disciplinas e você tem uma lista de pessoas que poderia chamar. Assuma que qualquer pessoa na lista viria para a conferência desde que tivesse um certo número de pessoas da área para trocar idéias. Para cada cientista, há uma lista de outros cientistas com quem ele poderia conversar (trocar idéias). Faça um algoritmo para convidar o número máximo de cientistas possível assumindo que qualquer um deles viria se tivesse >= 3 pessoas da mesma área para conversar.

2 Complexidade de Algoritmos

Dizemos que uma função g (n) é O (f (n)) se existem constantes c e N tal que, para $n \geq N$, $g(n) \leq c\, f(n)$.



Por exemplo, $2n + 1 = O(n)$ pois $2n + 1 \leq 3n$ para $n \geq 1$. Ou $2n^2 + 5n + 10 = O(n^2)$ pois $2n^2 + 5n + 10 \leq 50\, n^2$ para $n \geq 1$. A igualdade g (n) = O (f (n)) diz que cf (n) supera g (n) para todos os números maiores que um certo N. Então:

$O(\log_2 n) = O(\log_{10} n) = O(\log n)$
$O(3n) = O(5n + 7) = O(n)$
$O(a_r n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0) = O(n^k)$
$n = O(n^2)$

A notação O é usada para estimar o número de passos executados por dado algoritmo considerando que sua entrada possui n bits. Como n bits implica n/8 bytes ou n/32 (ou n/16) palavras, podemos associar n com número de bits, bytes ou palavras, já que a diferença entre eles é de uma constante (8, 32 ou 16).

Por exemplo, vamos analisar o algoritmo que, dado um vetor de tamanho n, encontra o seu maior elemento. Os passos em que estamos interessados são as comparações (<, <=, >, >=, ==, !=). Isto é, queremos saber quantas comparações teremos.

Para encontrar o maior elemento, usamos todos os elementos do vetor uma vez e fazemos n - 1 comparações. Então dizemos que a complexidade deste algoritmo é O(n).

Um dos algoritmos para ordenar um vetor toma o maior elemento e o coloca na última posição. Então o algoritmo chama a si mesmo para ordenar os primeiros n - 1 elementos do vetor. Então, o número de comparações feitas é:

n - 1 para encontrar o maior elemento
n - 2 para encontrar o segundo maior elemento
n - 3 para encontrar o terceiro maior elemento
...
1 para encontrar o maior dentre os dois últimos elementos

Assim, o total de comparações é $(n-1) + (n-2) + \ldots + 1 = (1 + (n-1))(n-1)/2 = (n^2 - n)/2 = O(n^2)$

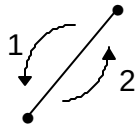
Em geral, não se especifica o significado da complexidade (número de comparações, número de testes, visitas a vértices, etc) porque este significado pode ser deduzido pelo contexto. Por exemplo, quando dizemos que um algoritmo para ordenação é $O(n \log n)$, estamos nos referindo a número de comparações.

Se um algoritmo para grafos possuir complexidade $O(|E|)$, (considerando um grafo G (V, E) como entrada), então cada aresta será visitada um número constante de vezes.

Não interessa quantas vezes (um, dois, …), desde que seja um número que independe do número de vértices e arestas do grafo. As barras em $|E|$ indicam a cardinalidade do conjunto E, que é o seu número de elementos (número de arestas).

Vejamos a complexidade dos algoritmos vistos até agora.

DFS: Cada aresta é visitada duas vezes, uma de cada vértice a que ela é adjacente:



Então, o número de visitas é $2|E|$. O grafo pode ser desconectado e o número de subgrafos é no máximo igual ao número de arestas, $|V|$. Então a complexidade do algoritmo é $O(|E| + |V|)$.

BFS: O mesmo raciocínio do DFS. Complexidade $O(|E| + |V|)$.

Os algoritmos cuja complexidade são da forma $O(n^k)$, k constante, são chamados de polinomiais e são considerados eficientes. Algoritmos com complexidade $O(k^n)$, k constante, são chamados exponenciais. Exemplo:

$$O(1), O(n^2), O(n^3), O(|V|^3 + |E|^3), O(n) \Rightarrow \text{polinomiais}$$
$$O(2^n), O(3^n) \Rightarrow \text{exponenciais}$$

Em geral, algoritmos exponenciais são muito lentos para serem usados na prática. Veja a tabela abaixo que mostra o tempo em segundos para executar diversos algoritmos (cujas complexidades são dadas) considerando-se n = 1000 e que o computador executa 1000 passos do algoritmo por segundo.

Complexidade	$\log_2 n$	n	$n \log_2 n$	$n^{1.5}$	n^2	n^3	$1 . 1^n$
tempo (seg)	0 . 01	1	10	32	1000	1000000	10^{39}

Algoritmos Probabilísticos

Problema: Dado um conjunto de números $x_1, x_2, \ldots x_n$, selecione um elemento que é maior ou igual a n/2 elementos do conjunto.

Uma opção para resolver este problema é ordenar o conjunto e aí tomar o máximo elemento, o que pode ser feito em O (n log n). Outra possibilidade mais barata seria ir selecionando o máximo elemento do conjunto até termos usado n/2 elementos. Não é difícil de ver que nenhum outro algoritmo é melhor que este, já que obrigatoriamente n/2 elementos devem ser usados (pela definição).

Contudo, existe um algoritmo melhor se não exigirmos que a solução esteja 100% correta. Tomemos dois elementos x e y quaisquer do conjunto. Assuma $x \geq y$. A probabilidade de que x seja maior ou igual a n/2 elementos é $\geq ½$ A probabilidade de que x seja menor do que n/2 elementos é $< ½$[1] Então, a probabilidade de que x e y sejam menores

[1] Se há muitos elementos iguais à mediana, a probabilidade é > 1/2. Um número é a mediana de um conjunto se metade dos outros é menor do que ele e metade é maior.

que n/2 elementos é < ¼ Ou seja, a probabilidade de que ou x ou y seja maior do que n/2 elementos é ≥ 3/4. Se y for, x também será, já que x ≥ y. Então, a probabilidade de x ser ≥ n/2 elementos é ≥ ¾

Tomando k números e selecionando o máximo dentre eles, digamos w, a probabilidade de que w seja maior ou igual a n/2 elementos é $1 - 1/2^k$. Se usarmos k = 100, a probabilidade de w não estar na metade superior do conjunto será quase 0. Observe que precisamos fazer k - 1 comparações para achar w, o que independe de n.

Este tipo de algoritmo é chamado algoritmo de *Monte Carlo*. Ele pode dar o resultado errado mas a probabilidade é mínima. Um outro tipo de algoritmo é chamado de *Las Vegas*. Este tipo sempre retorna um resultado correto e possui um tempo de execução (Complexidade) esperado (leia-se *médio*) baixo. Contudo, ele pode demorar um tempo arbitrariamente longo para executar.

2.1 Exercícios

23. (1) Qual a complexidade da função abaixo em função de n, no pior caso?

```
int f (int n, int v[], int a)
// n > 0, 1 ≤ a ≤ n, n é o tamanho do vetor
{
  int b = n/a;
  int i = 1, s = 0;

  while ( --a > 0 && --b > 0) {
      i++;
      s += v[a];
  }
  return s;
}
```

24. Qual a complexidade do algoritmo abaixo, em função de n ? Observe que a função `f` do exercício anterior é utilizada. A chamada de função `binSearch(a, v, k)` faz uma busca binária por a no vetor `v` de k elementos.

```
int g( int a, int n, int v[] )
  // v possui n elementos
{
  int k = f(n, a);
  return binSearch( a, v, k );
}
```

25. Faça um algoritmo cuja complexidade seja $O(\sqrt{\log n})$

26. Faça um algoritmo cuja complexidade seja $O(n\sqrt{n})$

27. (4) Calcule as complexidades dos algoritmos abaixo.

a)

```
  // assuma que a entrada seja o vetor
  i = 1;
  while i <= n and v[i] < m do
    i = i + 1;
```

b)

```
// assuma que a entrada seja duas matrizes nxn, A e B. Cuidado!
for i = 1 to n do
    for j = 1 to n do
      begin
      soma = 0;
      for k = 1 to n do
        soma = soma + A [i, k] * B [k, j];
      C[i, j] = soma;
      end
```

c) um algoritmo que soma os k primeiros elementos de uma lista, k = mínimo(3, número de elementos da lista).

d) o algoritmo de Monte Carlo visto anteriormente que encontra um elemento x em uma lista tal que x é maior do que metade dos elementos da lista.

28. (4) Simplifique:

(a) $O(n) + O(n^2) + O(n^3)$

(b) $O(n^2) + O(n \log n)$

(c) $O(1) + O(n)$

(d) $O(n^{2 \cdot 81}) + O(2^n)$

29. (1) Simplifique:

(a) $O(2^{2n}) + O(n^3) + O(2^n)$

(b) $O(n!) + O(2^n)$

3 Indução Finita

Indução finita é uma técnica de provar teoremas também usada no projeto de algoritmos. Suponha que queiramos provar um teorema T que possua um parâmetro n. Para provar que T é válido para qualquer valor de n, n >= 1, provamos que:

1 - T é válido quando n = 1;
2 - Se T for válido para n - 1, então T será válido para n.

1 é chamado de base. A prova de que esta técnica funciona é óbvia: T é válido para 1 pela regra 1, para 2 pela regra 2, para 3 pela regra 2, ... A suposição de que T é válido para n - 1 é chamada "Hipótese de Indução" (HI). Vejamos alguns exemplos:

Hipótese: Sendo $S_n = 1 + 2 + 3 + \ldots + n$, então
$S_n = n (n + 1)/2$

Prova: Para n = 1, $S_1 = 1$ e $S_1 = 1 (1+ 1)/2 = 1$, o que prova a hipótese.

Suponha que a hipótese é válida para n - 1, isto é,
$S_{n-1} = (n - 1) (n - 1 + 1)/2 = (n - 1) n/2$.

Provaremos que ela é válida para S_n. Sendo
$S_n = S_{n-1} + n$
então
$S_n = (n - 1)n/2 + n = (n^2 - n + 2n)/2 =$
$= n (n + 1)/2$
o que prova a hipótese.
qed.

Hipótese: $x^n - y^n$ é divisível por x - y para todos os números naturais x, y, n.

Para n = 1, $x^1 - y^1$ é trivialmente divisível por x - y. Suponha que a hipótese seja válida para n - 1, isto é, x - y divide $x^{n-1} - y^{n-1}$ $\forall$ x, y. Reescrevendo $x^n - y^n$, temos:

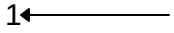
$$x^n - y^n = \underbrace{(x^{n-1} - y^{n-1})}_{\text{divisível por } x - y} (x + y) - xy \underbrace{(x^{n-2} - y^{n-2})}_{\text{divisível por } x - y}$$

Como o lado direito é divisível por x - y por hipótese, o lado esquerdo também o é.
qed.

Até agora utilizamos o seguinte princípio de indução: se uma hipótese T, com um parâmetro n, é verdadeiro para n = 1 e para cada $n > 1$ a verdade de T para n - 1 implica que T é verdadeiro para n, então T é verdadeiro para todo $n \in \mathbb{N}$.

De fato, existem vários tipos de indução diferentes deste:

1. pode-se considerar que T é verdade para valores k entre 1 e n - 1 e então provar a validade para n;
2. pode-se considerar um caso base quando n = m, m uma constante, assumir que T é válido para n e então provar a validade para n - 1.



1 . n = m
2 . n ⇒ n - 1

3. pode-se entender o caso base para mais de um valor como n = 1 e n = 2. Assumindo que T é verdade para n - 1 e n - 2, prova-se que T é verdade para n;
4. pode-se assumir que T é válido apenas para um subconjunto de n, como os números pares (ou primos, ou divisíveis por 5). O caso base seria n = 2 e a hipótese de indução seria "T é válido para n - 2, n par".

Outro caso seria "T é válido para potências de 2", que são 1, 2, 4, 8,… A hipótese de indução seria "T é válido para $n/2 = 2^{k-1}$" e então tentaríamos provar T para $n = 2^k$. Neste caso poderíamos aplicar outra indução para provar que T é válido entre 2^{k-1} e 2^k.

3.1 Projeto de Algoritmos por Indução

Construiremos um algoritmo para calcular o valor de um polinômio num dado ponto. Isto é, calcular $P_n(x)$, dado x, assumindo $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$.

Usaremos a técnica indutiva que é admitir que sabemos como resolver um problema menor e então usaremos este resultado para resolver o problema completo. A hipótese de indução (HI) para o problema de calcular $P_n(x)$ é:

HI: Sabemos como calcular $P_{n-1}(x)$ sendo

$$P_{n-1}(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a, x + a_0.$$

O caso base é n = 0 que resulta em $P_0(x) = a_0$ que é trivial. Para calcular $P_n(x)$ usando $P_{n-1}(x)$ temos apenas que adicionar $a_n x^n$.

$$P_n(x) = P_{n-1}(x) + a_n x^n$$

Esta fórmula resulta no algoritmo abaixo

```
Algorithm Pol(a', n, x)

     Entrada: a' = (a0, a1, …, an), n e x
     Saída:  an x^n + … + a1x + a0

begin
S = a0
for i = 1 to n do
    S  = S + ai * x^i
return S;
end
```

ou na versão recursiva:

```
Algorithm Pol(a', n, x)

     Entrada: a' = (a0, a1, …, an), n e x
     Saída: an x^n + … + a1x + a0
```

```
begin
if a' == a0
then
  return a0
else
   return Pol (a' - {an}, n - 1, x) + anx^n
endif
end
```

Este algoritmo não é eficiente pois ele faz

$$n + (n - 1) + (n - 2) + \ldots + 2 + 1 = n (n + 1)/2$$

multiplicações e n adições. Usaremos uma HI diferente para conseguir um resultado melhor. Para isto definimos

$$Q_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_1$$

HI: Sabemos como calcular $Q_{n-1}(x)$

O caso base é n = 0 que é o caso trivial. Para calcular P_n usando Q_{n-1} usamos a seguinte fórmula:

$$P_n(x) = x\, Q_{n-1}(x) + a_0$$

O número total de multiplicações é n e o número de somas é n. O algoritmo final é:

```
Algorithm Pol (a', n, x)

     Entrada: a = (a0, a1, … an), n e x
     Saída: an x^n + an - 1 x^n - 1 + … + a1 x + a0

begin
if a' == a0
then
   return a0
else
   return x*Pol ((a1, a2, …, an), n - 1, x) + a0
endif
end
```

3.2 Problema da Celebridade

Um problema interessante é o chamado “problema da celebridade”. Há um conjunto de n pessoas e queremos descobrir uma celebridade entre elas. Uma celebridade é uma pessoa que não conhece as outras n - 1 pessoas e é conhecida por todas elas. Nós só podemos perguntar a cada pessoa se ela conhece alguma outra. Assim, o número total de perguntas que poderemos fazer é n (n - 1).

Podemos representar este problema usando um grafo *dirigido* onde existirá uma aresta de v para w se v conhecer w. O objetivo seria encontrar um vértice com n - 1 arestas de entrada e 0 de saída.

Para resolver este problema, tentaremos reduzir o tamanho do problema para n - 1 eliminando uma pessoa que não é celebridade. Tomando duas pessoas A e B quaisquer e perguntando se A conhece B, podemos eliminar:

- A se a resposta for sim, pois uma celebridade não conhece ninguém.
- B se a resposta for não, pois todo mundo conhece a celebridade.

Deste modo podemos tomar duas pessoas A e B quaisquer, eliminar uma delas (Digamos A, que não é celebridade) e resolver o problema para as restantes n - 1 pessoas. Neste caso há duas possibilidades:

1. A celebridade está entre as restantes n - 1 pessoas.
2. Não há celebridade.

Se não há celebridade entre as restantes n - 1 pessoas, não há celebridade entre as n pessoas porque A não é celebridade.

Se há celebridade X entre as n - 1 pessoas, temos que verificar se:

- A conhece X.
- X não conhece A.

Se a resposta para estas duas questões é sim, X é uma celebridade também entre as n pessoas. O caso base para este problema acontece quando n = 2, que é trivial. A HI é:

HI: Sabemos como encontrar uma celebridade entre n - 1 pessoas.

A resolução do problema para n pessoas usando a solução para n - 1 já foi dada acima.

Em cada um dos n - 1 passos do algoritmo, há:

- Um teste para descobrir quem não é celebridade entre A e B.
- Dois testes para descobrir se a celebridade encontrada para n - 1 também é celebridade para n.

Ao todo temos 3 (n - 1) teste. Ou seja, para resolver este problema usamos apenas uma pequena parte de sua entrada, que é n (n - 1).

3.3 Exercícios

30. Prove por indução:

- $2^n < n!$ para $n \geq 4$;

- $(1 + x)^n \geq 1 + nx$
- $\frac{1}{n+1} + \frac{1}{n+2} + \ldots + 1 > \frac{13}{24}$

31. É possível colorir as regiões formadas por qualquer número de linhas no plano com somente duas cores. Duas regiões vizinhas devem ter cores diferentes. Três ou mais linhas não se interceptam no mesmo ponto.

32. Prove: se n+1 bolas são colocadas dentro de n caixas, pelo menos uma caixa irá conter mais de uma bola.

33. Faça algoritmos de ordenação utilizando indução finita. Utilize as seguintes HI:
    a) HI: sei como ordenar n - 1 elementos;
    b) HI: sei como ordenar n/2 elementos;

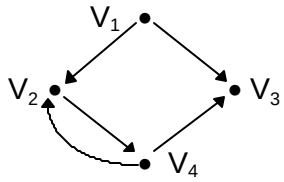
34. Prove que existe no máximo um vértice com n - 1 arestas de entrada e 0 de saída em um grafo de n vértices.

4 Estruturas de Dados para Grafos

Um grafo G = (V, E) é usualmente representado por uma matriz ou lista de adjacências.

1. Matriz de adjacência.

Sendo n o número de vértices de G, uma matriz de adjacência para G é uma matriz A = (a_{ij}) $n \times n$ tal que $a_{ij} = 1$ se $(v_i, v_j) \in E$.

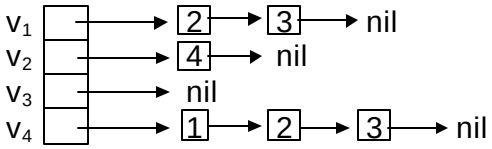


$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

A desvantagem desta representação é que ela ocupa muito espaço se há poucas arestas. Neste caso, a maior parte da matriz é inútil. A vantagem é que podemos saber se uma aresta existe ou não em tempo constante.

2. Lista de Adjacências.

Há um vetor de n posições cada uma apontando para uma lista. A posição i do vetor aponta para uma lista contendo números j tal que $(v_i, v_j) \in E$. Para o grafo anterior temos:



4.1 Exercícios

35. Represente um grafo não dirigido de n vértices usando apenas metade de uma matriz n $\times$ n.

36. Seja G(V, E) um grafo. Defina a matriz S = (s_{ij}), $|E| \times |E|$, como:

$s_{ij} = 1$ se as arestas c_i e c_j são incidentes

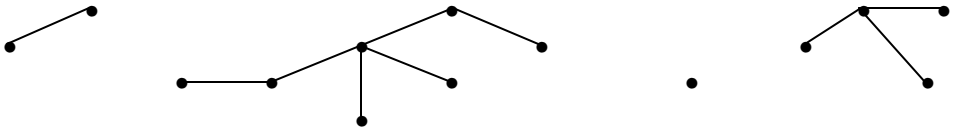
$s_{ij} = 0$ caso contrário.

Então é S uma representação de G ? Isto é, S contém todas as informações do grafo ? Provar ou dar contra-exemplo.

37. Utilizando uma matriz de adjacências, faça um algoritmo que resolva o problema da celebridade.

5 Árvores

Definição: uma árvore é um grafo conectado que não contém um ciclo. Exemplos:



Definição: uma floresta é um conjunto de árvores.

Árvores possuem várias propriedades interessantes. Elas são dadas abaixo e admitimos que o número de vértices é maior ou igual a 2.

Proposição 1: Há um único caminho ligando dois vértices quaisquer de um grafo.

Prova: Suponha que haja dois caminhos distintos ligando vértices v e w no grafo. Caminhos distintos possuem pelo menos um vértice não comum a ambos.

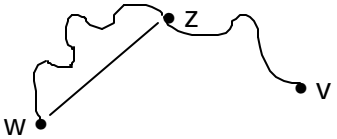
Se isto acontecer, haverá um par de vértices v_1 e w_1 que é ligado por dois caminhos que não possuem vértices em comum (exceto v_1 e w_1) e portanto haverá um ciclo, o que contradiz a hipótese de que o grafo é uma árvore.

Proposição 2: Existe pelo menos um vértice que é ligado a apenas uma aresta.

Prova: Provaremos por redução ao absurdo. Tentaremos provar que a negação da proposição é verdadeira, chegando a um absurdo. A negação é "Todos os vértices são ligados a mais de uma aresta".

Tomando um vértice v qualquer, construímos um caminho começando em v. Entramos em um dado vértice e saímos por outro. Isto é sempre possível já que cada vértice é ligado a pelo menos duas arestas.

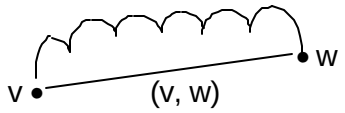
Como o número de vértices é finito, em algum momento estaremos em um vértice w e a única opção será ir para um vértice z que já está no caminho. Neste ponto temos um ciclo formado pelo caminho de z a w (subconjunto do caminho de v a w) e pela aresta (w, z). Como árvores não possuem ciclos, a hipótese está errada, o que prova a proposição.



Proposição 3: A remoção de qualquer aresta da árvore cria exatamente duas novas árvores.

Prova: Tentaremos provar a proposição contrária, que é "Pode-se remover uma aresta de uma árvore e ela continuar conectada". Note que a remoção de uma aresta poderia criar no máximo duas novas árvores em um grafo qualquer, nunca três. Assumiremos isto como óbvio.

Suponha que podemos remover aresta (v, w) da árvore e ela continue conectada. Então existe um caminho entre v e w que não envolve aresta (v, w), o que significa que existe um ciclo no grafo original formado por este caminho e (v, w).

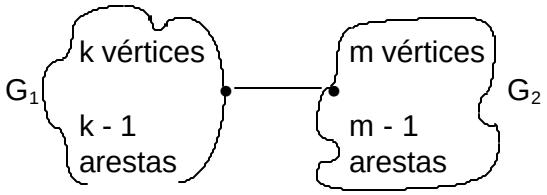


Assim, quando uma aresta (v, w) for removida, o grafo se desconectará formando uma árvore que contém v e outra que contém w.

Proposição 4: Uma árvore com n vértices possui n - 1 arestas.

Prova: Usaremos a própria proposição como HI. O caso base é n = 2 e é trivial.

Removendo uma aresta da árvore temos duas árvores G_1 e G_2 de acordo com a proposição 3. Assumindo que G_1 possui k vértices e G_2 m vértices, pela HI temos que G_1 e G_2 possuem k - 1 e m - 1 arestas, respectivamente. Logo, o grafo original possui (k - 1) + (m - 1) + 1 = (k + m) - 1 arestas. Como o número de vértices da árvore original é k + m, a hipótese está provada.



Proposição 5: A adição de uma aresta na árvore cria exatamente um ciclo.

Prova: A adição de uma aresta (v, w) cria pelo menos um ciclo que é aquele formado pelo caminho ligando v e w na árvore (Proposição 1) e pela aresta (v, w).

Agora provaremos que a adição de uma aresta cria um e só um ciclo. Suponha que mais de um ciclo é criado pela adição de (v, w):

Neste caso existem dois caminhos diferentes ligando v a w que não possuem aresta (v, w), o que contradiz a proposição 1.

5.1 Exercícios

38. Prove: Em uma árvore, a remoção de um vértice desconecta o grafo.

39. A versão não dirigida de um grafo dirigido G' é uma árvore. G' possui uma raiz com grau de entrada 0 (nenhuma aresta incidente) e todos os vértices possuem grau de saída 0 ou 2. Calcule o número de vértices de G' com grau 0 em função do número n de vértices com grau 2. Prove por indução.

40. Qual o número mínimo de bits em que pode ser colocada a *forma* de uma árvore binária de n vértices ?

41. Mostre com um exemplo: a remoção de um vértice e suas arestas adjacentes de uma árvore pode resultar em duas ou mais árvores.

42. Seja G uma floresta com n vértices e k árvores. Quantas arestas possui G ?

43. Faça um algoritmo que diga se um grafo é uma árvore ou não.

44. Prove que um grafo conexo com um número mínimo de arestas é uma árvore.

45. Os números $d_1, d_2, \ldots d_n$ são tais que

$$d_1 + d_2 + \ldots + d_n = 2n - 2$$

Prove que existe uma árvore com n vértices cujos graus são $d_1, d_2, \ldots d_n$.

46. Dada uma árvore T e k subárvores de T tal que cada par de subárvores possui pelo menos um vértice em comum, prove que há pelo menos um vértice em comum a todas as subárvores.

47. Uma árvore binária completa é definida como se segue. Uma árvore de tamanho 0 consiste de 1 nó que é a raiz. uma árvore binária completa de altura h + 1 consiste de duas árvores binárias completas de tamanho h cujas raízes são conectadas a uma nova raiz. Se T uma árvore binária completa de altura h. A altura de um nó é h menos a distância do nó da raiz (a raiz tem altura h e as folhas, altura 0). Prove que a soma das alturas de todos os nós em T é $2^{h+1} - h - 2$.

48. Seja G = (V, E) uma árvore binária não dirigida com n vértices. Nós podemos construir uma matriz quadrada de ordem n tal que a entrada ij seja igual à distância entre v_i e v_j. Projete um algoritmo O(n2) para construir esta matriz para uma árvore dada em forma de lista de adjacências.

49. Seja G = (V, E) uma árvore binária. A distância entre dois vértices em G é o tamanho do caminho conectando estes dois vértices (vizinhos tem distância 1). O diâmetro de G é a distância máxima sobre todos os pares de vértices. Projete um algoritmo linear para encontrar o diâmetro de uma certa árvore.

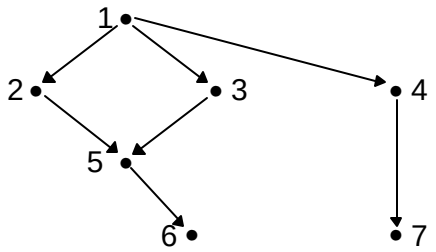
6 Ordenação Topológica

Suponha que existam um conjunto de tarefas que precisam ser executadas, uma por vez. Algumas tarefas dependem de outras e não podem ser começadas antes que estas outras sejam completadas. Todas as dependências são conhecidas e nós queremos fazer um esquema de execução das tarefas que respeite a dependência entre elas. Este problema é chamado ordenação topológica.

Para resolvê-lo, usamos um grafo dirigido onde os vértices são as tarefas e existe uma aresta de v para w se w só pode ser executado após a execução de v. O grafo evidentemente deve ser acíclico (por que?).

Problema: Dado um grafo acíclico dirigido (Direct Acyclic Graph - DAG) G = (V, E) com n vértices, numere os vértices de 1 até n tal que, se v possui número K, então todos os vértices que podem ser atingidos de v por um caminho dirigido possuem números > K.

Deste modo a execução das tarefas pode ser feita na ordem 1, 2, 3, … n. No grafo abaixo, os números nos vértices indicam uma ordenação topológica.



A hipótese de indução é:

> HI: Nós sabemos como numerar um grafo com < n vértices de acordo com as restrições acima.

Utilizaremos indução finita de uma maneira diferente: tomaremos um grafo com n vértices e removeremos um vértice com alguma característica especial. Então teremos um grafo com n - 1 arestas e aplicaremos a HI. Depois adicionamos novamente o vértice removido (com suas arestas).

O caso base é n = 1 que é trivial. Considerando o caso geral, grafo com n vértices, removemos um vértice e aplicamos a hipótese de indução. O vértice a ser removido é aquele que não possui dependentes, isto é, nenhuma aresta "chega" a ele. Este vértice possui grau de entrada 0. De acordo com lema 1 (provado adiante) este vértice sempre existe.

Após encontrar este vértice, o numeramos com 1 e o removemos do grafo juntamente com as arestas adjacentes. Então aplicamos a HI para os n - 1 vértices restantes usando números de 2 a n.

Observe que estes vértices podem formar mais de um grafo — não importa. A HI pode ser aplicada em todos estes grafos porque eles satisfazem a hipótese inicial — são acíclicos dirigidos.

Complexidade: cada vértice e cada aresta é usada um número constante de vezes. Então, a complexidade é O ($|E| + |V|$).

Lema 1: Um grafo acíclico dirigido sempre contém um vértice com grau de entrada 0, isto é, um vértice sem dependências.

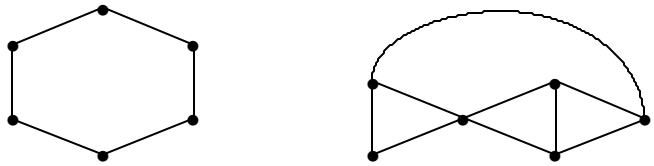
Prova: Suponha o contrário, isto é, todos os vértices possuem grau de entrada > 0 o que significa que existe pelo menos uma aresta que leva a cada vértice do grafo.

Neste caso, tomando um vértice qualquer, poderíamos caminhar no sentido oposto à direção das arestas sem nunca ter que parar, já que sempre existe aresta que chega a cada vértice. Como o número de vértices é finito, em algum momento chegaremos a um vértice em que já passamos antes. Neste caso teremos um ciclo, o que é impossível já que o grafo é acíclico.

7 Conectividade, Caminhos e Ciclos

Definição: Um grafo não dirigido é bi-conectado se existe pelo menos dois caminhos disjuntos em vértices ligando dois vértices quaisquer do grafo.

Exemplos:



Se vértices são computadores (ou processadores) e as arestas são ligações entre eles, um computador pode falhar e ainda sim os outros serão capazes de conversar entre si.

Como caminhos disjuntos em vértices implica em caminhos disjuntos em arestas, uma ligação pode ser interrompida e ainda assim todos os computadores serão capazes de se comunicar entre si.

Grafos bi-conectado possuem um alto grau de conectividade. Situação oposta acontece com árvores. Elas são conectadas mas a remoção de qualquer vértice que não seja folha (ou mesmo uma aresta) as desconecta.

Então, se quisermos ligar vários computadores utilizando o menor número de ligações possível, mas de tal forma que todos possam conversar entre si, devemos usar uma árvore.

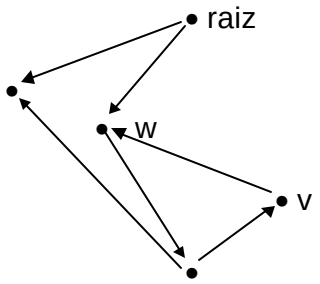
7.1 Ciclos

Problema: Faça um algoritmo que retorna true se um grafo não dirigido G (V, E) possui um ciclo, false caso contrário.

```
Algorithm HaCiclo (G (V, E), v): boolean
  { Faz uma busca em profundidade no grafo }
  Entrada: um grafo G e um vértice v de G

begin
marca v
Seja S o conjunto de arestas (v, w) tal que w não foi marcado
for each aresta (v, w) de S do
  if w não foi marcado
  then
    if HaCiclo (G, w)
    then
      retorne true;
    endif
  else
      { achou um ciclo - não é necessário continuar
        a busca em vértices adjacentes a v }
    return true;
  endif
return false;
```

```
end
```

O algoritmo `HaCiclo` para grafos dirigidos, `HaCicloDirig`, faz uma busca em profundidade e retorna true (há ciclo) se há aresta do vértice corrente (v) para um outro vértice w que está na pilha construída pela busca em profundidade. O ciclo encontrado é formado pelo caminho de w até v e a aresta (v, w):



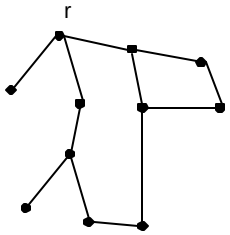

```
Algorithm HaCicloDirig( G(V, E), v, S) : boolean

    Entrada: um grafo G(V, E), um vértice v de G e uma pilha S
    contendo os vértices sendo visitados.
    Saída: retorna true se houver ciclo cujos vértices são
    alcançáveis a partir de v.

begin
marque v
empilhe v na pilha S
for each aresta (v, w) ∈ V do
  if w não foi marcado
  then
    if HaCicloDirig( G, w, S)
    then
      return true;
    endif
  else
    if w está na pilha S
    then
      return true;
    endif
  endif
desempilhe v de S
return false;
end
```

Este algoritmo só funcionará se o ciclo do grafo (se existir) é alcançável a partir do vértice v. Ao chamar este algoritmo, passa-se uma pilha vazia como parâmetro S.

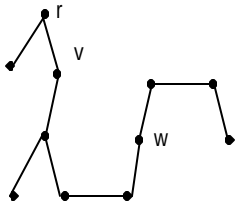
Note que é claro que se os algoritmos retornam true há ciclo no grafo (embora nós não provamos isto). Também é verdade que, se há ciclo no grafo, os algoritmos retornam true, embora o raciocínio para chegar a esta conclusão não seja tão simples.

7.2 Uma Propriedade da Busca em Profundidade

O Algoritmo de busca em profundidade pode construir uma árvore com as arestas que ele percorre na busca. Esta árvore é chamada árvore de DFS.



Grafo exemplo



Árvore de DFS do grafo acima

Vértice (v) será um ancestral de w em uma árvore t com raiz r se v estiver no único caminho entre w e r em t.

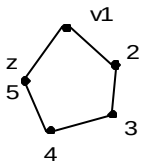
Lema: seja T uma árvore de DFS de um grafo G. Então cada aresta de G:

1. pertence a T ou;
2. conecta dois vértices em G, um dos quais é ancestral do outro em T.

Prova: seja (v,z) uma aresta de G. Suponha que v seja visitado por DFS antes de z. Então uma de duas coisas ocorre:

1. a partir de v, visita-se z. Então (v,z) pertence àarvore de DFS.

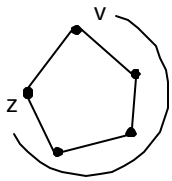
2. z não é visitado pois, a partir de v, visitamos outros vértices que visitaram z:



Então v é ancestral de z.

Prova de que o algoritmo HaCiclo funciona:

Se, partindo de v, encontramos um vértice z já marcado, então v é ancestral de z na árvore de DFS. Isto implica que existe um caminho entre v e z que, juntamente com a aresta (v, z), formam um ciclo:

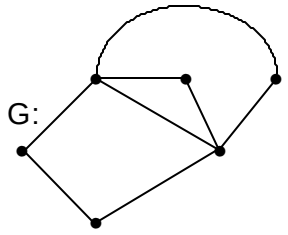


7.3 Caminhos

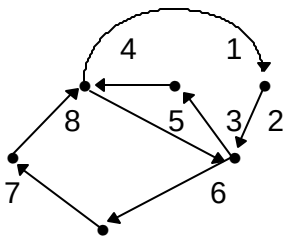
Definição: Um grafo G não dirigido é Eureliano se existe um caminho fechado (circuito) que inclui cada aresta de G. Este caminho é chamado de "caminho Eureliano".

Arestas em um circuito não se repetem. Assim, um caminho Eureliano contém cada aresta do grafo exatamente uma vez.

Exemplo:



Caminho Eureliano:



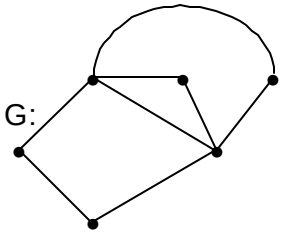
Teorema: Um grafo conectado G é Eureliano se e somente se o grau de cada vértice de G é par.

A prova $\Rightarrow$ é feita por contradição: admita que um vértice possui grau ímpar. O caminho Eureliano passará por ele um certo número de vezes e, a cada vez, utilizará duas

arestas (uma para entrar e outra para sair). Então sobrará uma única aresta que não poderá ser utilizada no caminho: contradição, o que prova que a hipótese $\Rightarrow$ está correta. A prova $\Leftarrow$ é dada pelo algoritmo para encontrar um caminho Eureliano, que é dado a seguir.

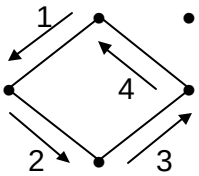
A partir de um vértice v qualquer, comece a percorrer o grafo sem passar por nenhuma aresta duas vezes. Como o grau de cada vértice é par, podemos sempre entrar em um novo vértice por uma aresta e sair por outra:

Como o número de vértices de G é finito, em algum momento retornaremos a v. Como nenhuma aresta foi usada duas vezes, temos um circuito P. Note que este circuito pode não conter todos os vértices do grafo:



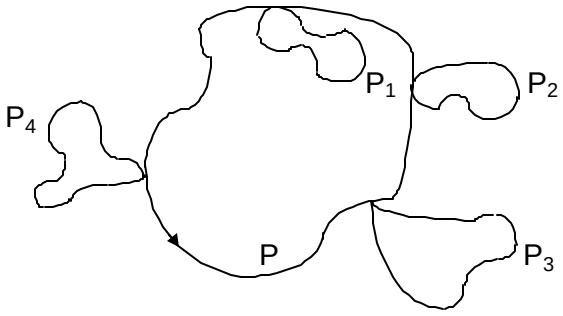
Circuito:

P

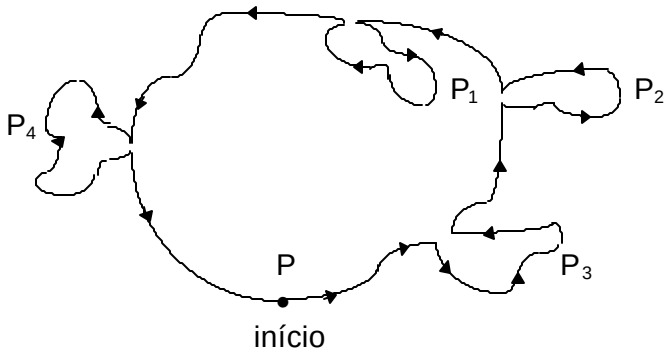


Agora construímos G' retirando de G as arestas do circuito encontrado acima. O grau de cada vértice em G' é par pois o número de arestas removidas de cada vértice é par. G' pode não ser conectado. Sejam G'_1, G'_2, … G'_k os componentes conexos de G'. Cada grafo G'_i é conectado e o grau de cada vértice é par (pois o grau de cada vértice de G' é par). Sejam P_1, P_2, … P_k os circuitos Eurelianos encontrados pela aplicação recursiva deste algoritmo a G'_1, … G'_k.

Para encontrar um Caminho Eureliano para G, começamos a percorrer o circuito P a partir de um vértice qualquer. Quando encontramos um vértice v que pertence a um caminho P_i, percorremos P_i, retornamos a v e continuamos a percorrer P novamente. Deste modo, quanto chegarmos ao vértice inicial de P, teremos percorrido P, P_1, P_2, … P_k. Isto é, teremos um circuito Eureliano. Veja a ilustração abaixo.

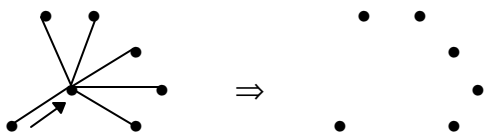


O desenho acima mostra o circuito P (grande) e os circuitos P_1, P_2, P_3 e P_4. obtidos após a remoção de P do grafo. O desenho abaixo mostra como todos estes circuitos podem ser conectados para formar um caminho Eureliano para o grafo.

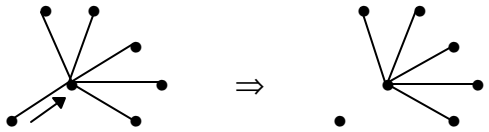


Definição: Um circuito Hamiltoniano em um grafo conectado G é um circuito que inclui cada vértice de G exatamente uma vez.

Encontrar um circuito Hamiltoniano (CH) é claramente mais difícil que encontrar um caminho Eureliano (CE), pois cada vez que atingimos (num percurso) um vértice v a partir de uma aresta "e", nunca mais podemos usar v e quaisquer de suas arestas. Em um CE, não poderíamos usar "e", mas poderíamos usar v e suas outras arestas.



CH



CE

O problema “Encontre um CH para um grafo G” é NP-completo (Veja Seção 16) e portanto nenhum algoritmo será estudado.

O Problema do Caixeiro Viajante (The Traveling Salesman Problem)

Um vendedor quer visitar um grupo de cidades começando e terminando em uma mesma cidade de tal forma que o custo da viagem (distância percorrida) seja o menor possível e que cada cidade seja visitada apenas uma vez. Ou seja, o problema é encontrar um circuito Hamiltoniano de custo mínimo. Naturalmente, este problema é NP-completo.

7.4 Exercícios

50. Faça um grafo que contenha um circuito que não é ciclo.

51. No Unix é possível associar um outro nome a um arquivo ou diretório já existente através do utilitário `ln`:

```
ln NovoNome Arq
```

Após a execução do comando acima, `NovoNome` será um aliás para `Arq`. `ln` pode criar situações perigosas, como fazer uma associação recursiva:

```
$cd   /A/B
$ln  C  /A
```

`$` é o prompt do Unix. Neste caso estamos colocando o diretório `A`, com nome `C`, dentro do próprio `A`. Crie um novo comando `safeln`, em linguagem de alto nível, que usa `ln` e não permite este tipo de situação.

52. Faça um algoritmo que retorna true se um programa pode ser recursivo em execução. Assuma que a entrada do algoritmo é dada em forma de um grafo usando uma representação de sua escolha.

53. Em um sistema operacional, um processo terá sua execução suspendida se ele aguardar um recurso (impressora, disco, …) que está sendo usado por outro processo. Esta situação é chamada de *deadlock*.

Faça um algoritmo que detecta se há *deadlock* entre os processos. Por exemplo, processo A espera B liberar recurso R_1 e B espera A liberar recurso R_2.

54. Prove: se, após a remoção de uma aresta de um grafo, temos um ciclo, o grafo original possui um ou mais ciclos.

55. Se A é a matriz de adjacências de um grafo, $a_{ij} = 1$ se existe uma aresta entre vértices i e j; isto é, se existe um caminho de uma aresta de comprimento 1 (um) entre i e j.

Prove que $(a_{ij})^2$, que é o elemento a_{ij} de A^2, é o número de caminhos entre i e j de tamanho 2. Estenda este resultado para A^k.

56. Dois ciclos em um grafo G' possuem uma e só uma aresta em comum. Prove que o grafo obtido pela remoção desta aresta possui pelo menos um ciclo.

57. Um istmo é uma aresta tal que a sua remoção desconecta o grafo. Mostre que uma aresta é um istmo se e somente se ela não está contida em nenhum circuito.

58. Um *plotter* deve desenhar um certo conjunto de pontos isolados no papel de tal forma que a cabeça de impressão mova o menos possível. Mapeie este problema para grafos.

59. Prove: Se G é um grafo onde o grau de cada vértice é pelo menos dois, então G contém um ciclo.

60. Prove: Dois vértices quaisquer de um grafo bi-conectado estão contidos em um ciclo.

61. Justifique: dois caminhos podem possuir vértices em comum sem possuir arestas em comum.

62. Seja G = (V, E) um grafo não dirigido conectado e seja F o conjunto de vértices com grau ímpar (o grau de um vértice é o número de arestas ligadas a ele). Então podemos dividir F em pares e encontrar caminhos disjuntos em arestas conectando cada par.

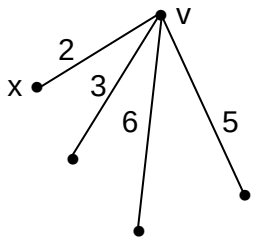
63. Seja G um grafo não dirigido tal que cada vértice possui um grau par. Projete um algoritmo de tempo linear para direcionar as arestas de G de tal forma que, para cada vértice, o grau de saída é igual ao de entrada.

8 Menor Caminho entre Vértices

Definição: Um grafo com comprimentos (ou pesos) associa a cada aresta um comprimento que é um número real positivo. O comprimento de um caminho é a soma dos comprimentos de suas arestas.

Problema: Calcular o menor caminho de um vértice v a todos os outros de um grafo dirigido.

A idéia é considerar os vértices do grafo na ordem dos comprimentos de seus menores caminhos de v. Primeiro, tomamos todas as arestas saindo de v. Seja (v, x) a aresta de menor comprimento entre elas. Então o menor caminho entre v e x é a aresta (v, x) por que qualquer outro caminho envolveria outra aresta (v, w), $w \neq x$ que sozinha já possui comprimento maior que (v, x).



Se quisermos encontrar o vértice y que é o segundo mais perto de v, devemos considerar apenas as arestas (v, y) e os caminhos (v, x), (x, y).

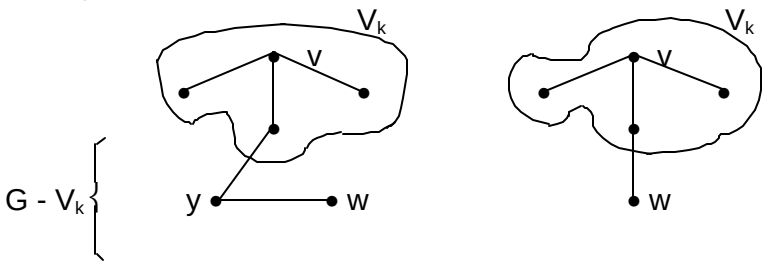
Para resolver este problema por indução, o caso base é encontrar x dado acima, isto é, encontrar o vértice mais próximo de v. A hipótese de indução é:

> HI: Dado um grafo G = (V, E), sabemos como encontrar os K vértices que estão mais próximos de v e os comprimentos dos caminhos de v a estes vértices.

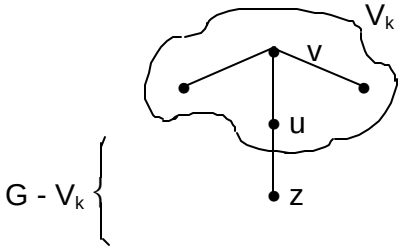
O nosso problema agora é como estender a HI para mais de um vértice. Isto é, admitindo a HI para K, que vértice devemos acrescentar para provar a hipótese para K + 1.

Seja V_k o conjunto contendo v e os K vértices mais próximos de v no grafo. Queremos encontrar um vértice w que é o mais próximo de v entre os vértices $G - V_k$ e encontrar o menor caminho de v a w.

O menor caminho de v a w só pode passar por vértices de V_k. Ele não pode incluir um vértice y que não está em V_k por que então o caminho (v, y) seria menor que (v, w) e y seria mais próximo de v que w.



Então w é ligado a um elemento de V_k por uma única aresta. Considere agora todas as arestas (u, z) tal que $u \in V_k$ e $z \in G - V_k$.



Cada aresta (u, z) define um caminho v … u, z. Um dos vértices z é w, o vértice mais próximo de v dentre todos aqueles fora de V_k. Nós tomaremos todos estes caminhos e escolheremos o menor dentre eles. Deste modo podemos acrescentar um vértice a V_k estendendo a HI de K para K + 1. O vértice escolhido é o w.

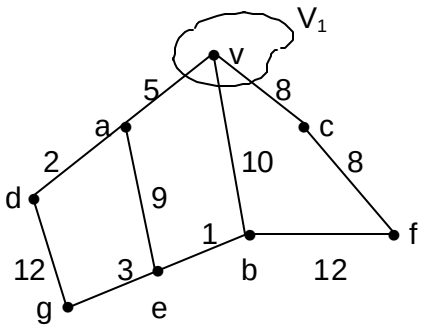
Segue do raciocínio acima que o vértice w adicionado a V_k é o K + 1 mais próximo de V. O conjunto resultante é V_{k+1} que realmente contém os (K + 1) vértices mais próximos de V.

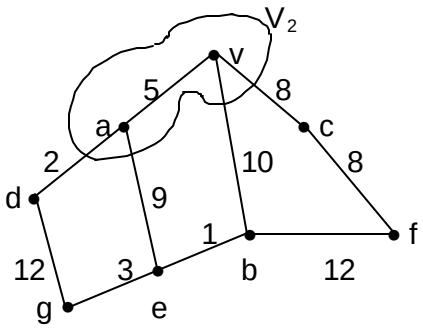
O algoritmo começa com $V_1 = \{v\}$ e a cada passo adiciona a V_k o vértice w tal que f(w) é o menor valor dentre f (y), $y \in G - V_k$. A função f é dada por:

f (y) = distância (v, u) + comprimento (u, y)

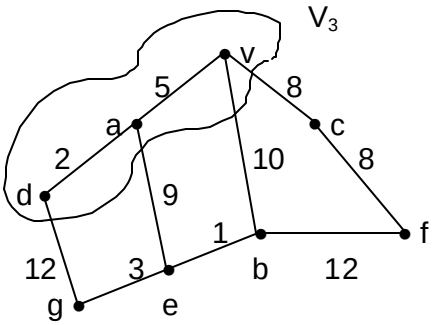
onde u e y são vértices da fronteira entre V_k e $G - V_k$ sendo que $u \in V_k$ e $y \in G - V_k$.

Exemplo:

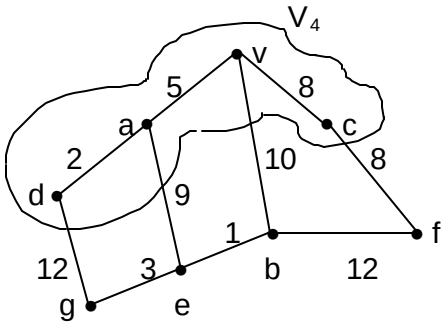




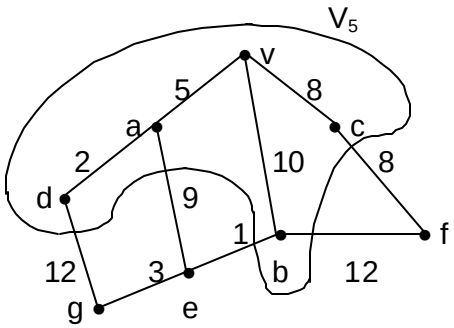
f(d) = dist (d, a, v) = 7
f(e) = dist (e, a, v) = 14
f(c) = dist (c, v) = 8
f(b) = dist (b, v) = 10

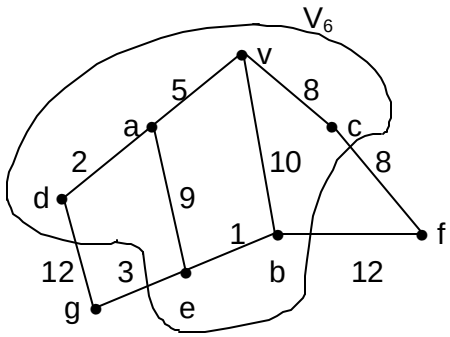


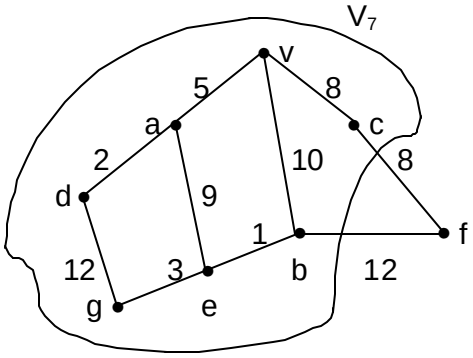
f(g) = dist (g, d, a, v) = 19
f(e) = dist (e, a, v) = 14
f(c) = dist (c, v) = 8
f(b) = dist (b, v) = 10

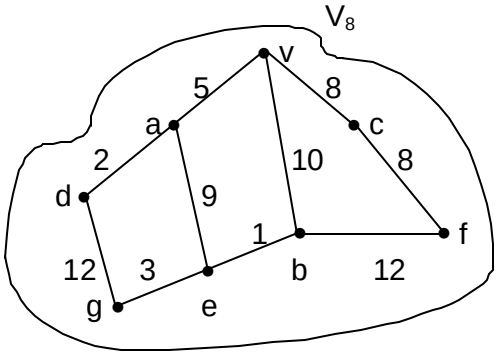


f(g) = dist (g, d, a, v) = 19
f(e) = dist (e, a, v) = 14
f(b) = dist (b, v) = 10
f(f) = dist (f, c, v) = 16



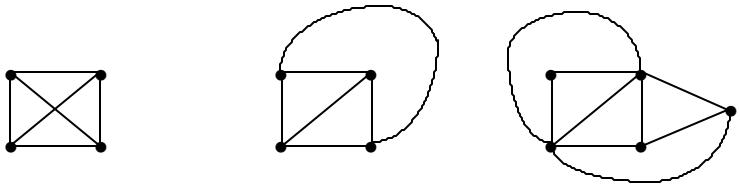






9 Planaridade

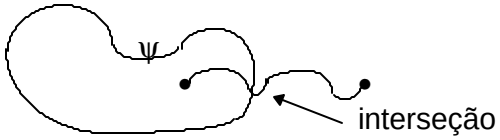
Um grafo planar é aquele que pode ser desenhado no plano de tal forma que duas arestas quaisquer não se interceptam. Exemplo:



(a) (b) (c)

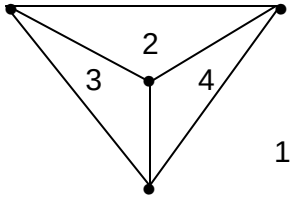
Observe que, apesar de duas arestas de (a) cruzarem, este grafo é planar porque ele pode ser transformado no desenho (b).

Teorema de Jordan: Dada uma curva fechada ψ no plano e dois pontos, um interior e outro exterior a ela, qualquer curva ligando os dois pontos intercepta ψ.



Teorema: k_5 e $k_{3,3}$ não são planares.

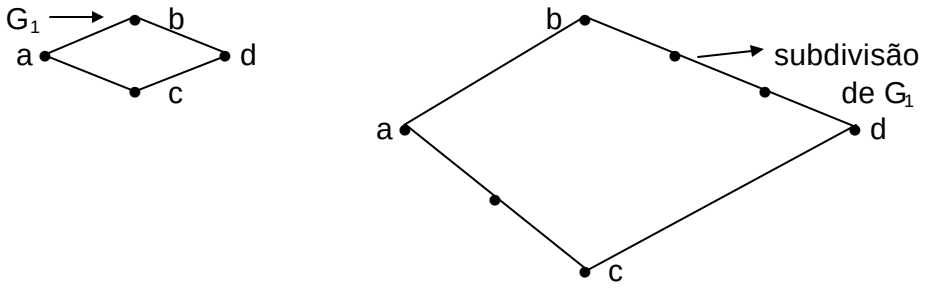
Prova: Provaremos apenas que k_5 não é planar. Usando a representação



o plano fica dividido em quatro regiões. Em qualquer região que coloquemos o quinto vértice, uma aresta que o liga a algum outro vértice cruzará outra aresta (pelo teorema de Jordan).

Definição: A subdivisão de uma aresta é uma operação que transforma a aresta (v, w) em um caminho v, z_1, z_2, … z_k, w sendo $k \geq 0$, onde os z_i são vértices de grau 2 adicionados ao grafo.

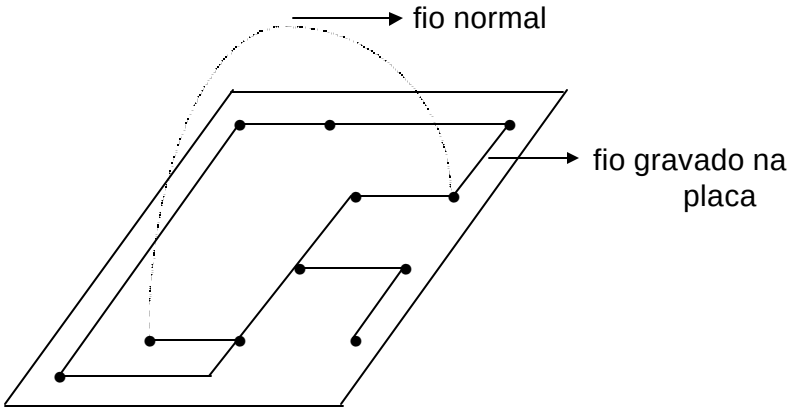
Um grafo G_2 será uma subdivisão do grafo G_1 quando G_2 puder ser obtido de G_1 através de uma seqüência de arestas de G_1. Exemplo:



Teorema de Kuratowski: Um grafo é planar se e somente se ele não contém nenhum subgrafo que é uma subdivisão de K_5 ou $K_{3,3}$.

A prova deste teorema está além do alcance deste curso.

Um circuito eletrônico pode ser considerado um grafo onde as junções são vértices e as arestas são os fios ligando as junções. Se o grafo correspondente ao circuito é planar, todos os fios podem ser gravados na própria placa. Se o grafo não é planar por causa de apenas uma aresta, esta é um fio normal que deve passar por cima da placa. Isto equivale a colocar esta aresta acima do plano contendo o restante do grafo:

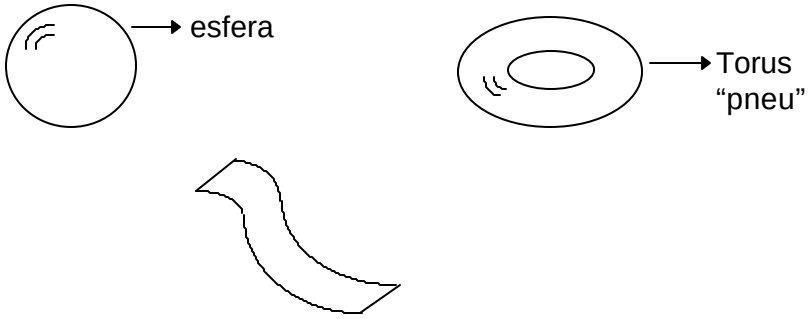


Teorema: Todo grafo planar admite uma representação plana em que todas as linhas são retas.

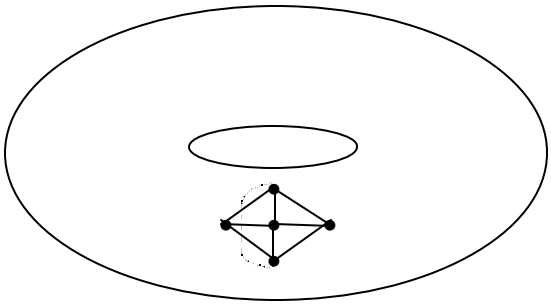
Definição: Um grafo pode ser embebido em uma superfície S se ele pode ser colocado em S de tal forma que quaisquer duas de suas arestas não se cruzam.

Teorema: Para cada superfície S, existe um grafo que não pode ser embebido em S.

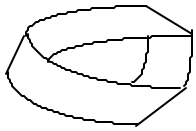
Definição: Uma superfície é uma curva descrita por 2 dimensões, não necessariamente no plano. Ex.:



Nota: K_5 pode ser embebido no Torus:



$K_{3,3}$ pode ser embebido na fita de Möbius



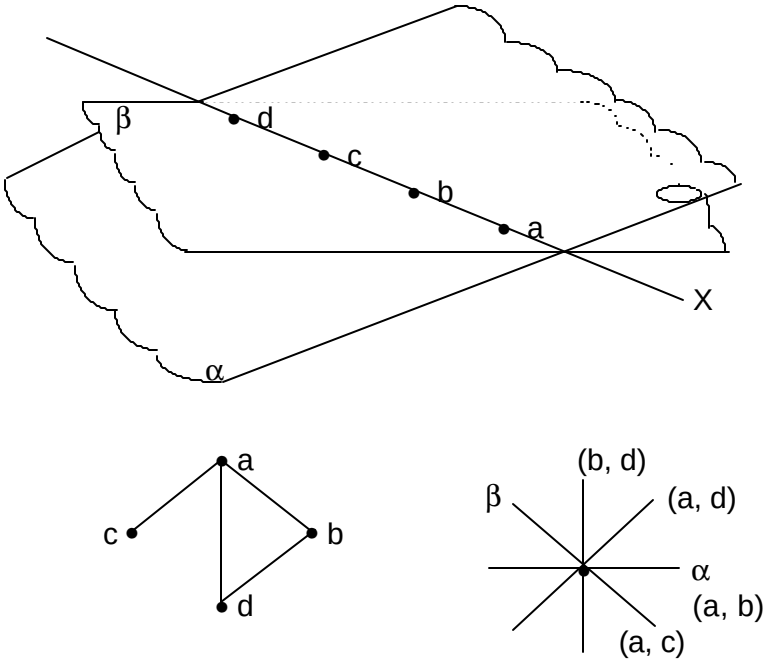
Teorema: Um grafo pode ser embebido na esfera S se ele pode ser embebido no plano.

Existe um algoritmo O (n) para determinar se um grafo é planar ou não, feito por Hopcroft e Tarjan.

Teorema: Qualquer grafo pode ser colocado em um espaço de três dimensões.

Prova: Coloque os vértices do grafo em uma reta X. Então, para cada aresta, faça um plano que contém X. Arestas distintas devem corresponder a planos distintos.

Para cada aresta desenharemos um semicírculo ligando os dois vértices. As arestas não se interceptarão porque elas estarão em planos diferentes.



9.1 Exercícios

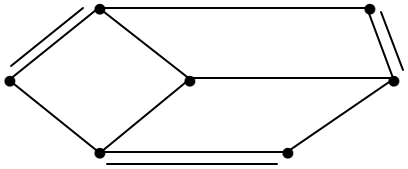
64. Prove: Se G é planar, qualquer subgrafo de G é planar.

65. Prove que $K_{3,3}$ não é planar.

66. Desenhe um grafo não planar com oito vértices com um número mínimo de arestas.

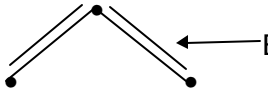
10 Emparelhamento

Dado um grafo, um emparelhamento é um conjunto de arestas t tal que duas delas não possuem vértice em comum. Ex.:



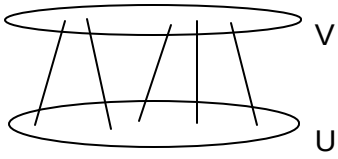
 significa uma aresta do emparelhamento

Pela definição, um vértice não é incidente a mais de uma aresta do emparelhamento:



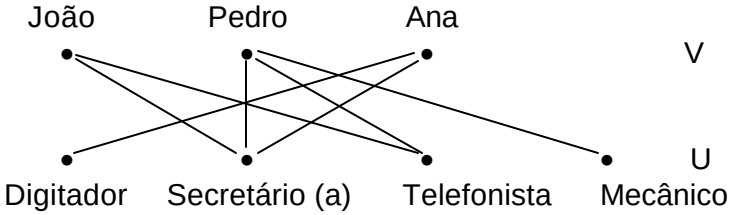
Emparelhamento Bipartido

Seja G = (V, E, U) um grafo bipartido tal que V e U são os conjuntos de vértices disjuntos.

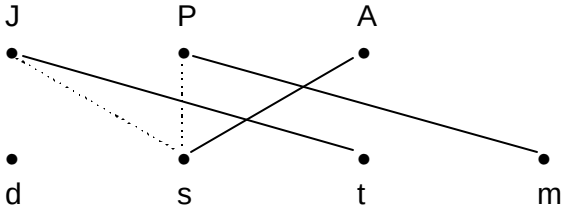


Problema: Encontre um emparelhamento de cardinalidade máxima em G.

V pode representar um conjunto de trabalhadores e U um conjunto de habilidades ou profissões (eletricista, secretária, telefonista, digitador, mecânico). Uma aresta liga um trabalhador a todas as profissões a que ele está habilitado.

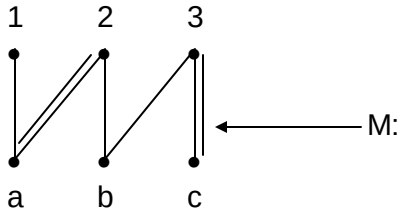


O problema então é dar empregos ao máximo número de pessoas respeitando suas habilidades. Para o grafo acima, uma solução seria:

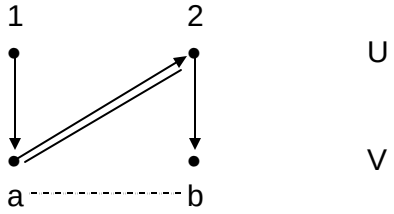


emparelhamento = M = {(J, t), (P, m), (A, s)}

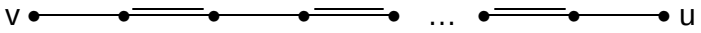
Um *caminho alternante* P para um emparelhamento M é um caminho de um vértice u em U para v em V, ambos não emparelhados em M, tal que as arestas de P estão alternativamente em E - M e M. Ex.:



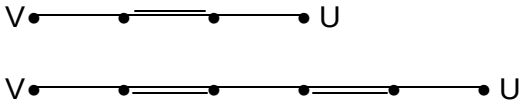
1 a 2 b é um caminho alternante para M:



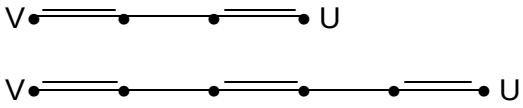
Um caminho alternante sempre será da forma



A primeira e a última aresta não pertencem a M. Se o caminho começasse em V e terminasse em V, ele teria um número par de arestas. Como ele começa em V e termina em U, ele possui um número ímpar. Ex.:



Um caminho alternante terá sempre uma aresta que não pertence a M a mais que o número de arestas em M. Então, podemos inverter as arestas pertencentes a M / não pertencentes criando um emparelhamento M' que possui uma aresta a mais que M:



Sempre que existir um caminho alternante, o emparelhamento M não será o máximo em G. O contrário também é verdadeiro, resultando em

Teorema: Um emparelhamento M em G é um emparelhamento máximo se e somente se G não contém nenhum caminho alternante para M.

Construiremos agora um algoritmo para encontrar o emparelhamento máximo em G baseado neste teorema. Utilizaremos o fato de que qualquer emparelhamento que não é máximo possui um caminho alternante e este pode estender o emparelhamento. Grande

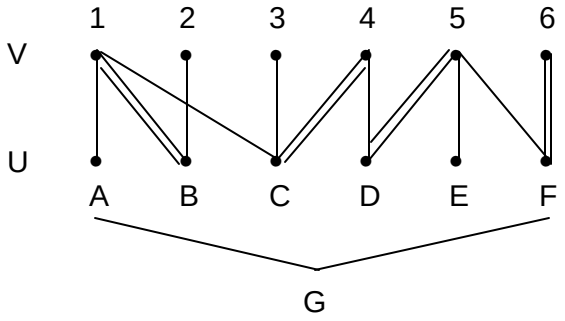
parte dos algoritmos de teoria dos grafos são feitos desta forma: toma-se um teorema e constrói-se um algoritmo baseado nele.

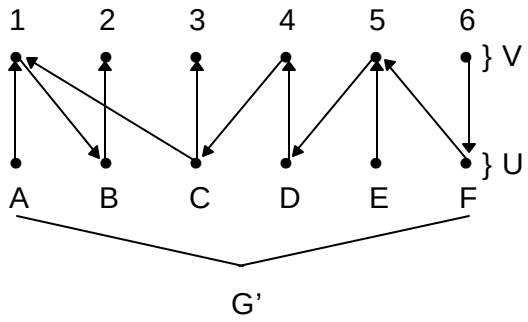
O algoritmo faz uma primeira aproximação escolhendo aleatoriamente arestas para o emparelhamento. Então ele procura caminhos alternantes, modificando o emparelhamento apropriadamente até que não existam mais caminhos alternantes. Então, pelo teorema, o emparelhamento resultante é máximo.

Para encontrar os caminhos alternantes, criamos um grafo dirigido G' a partir de G = (V, E, U) tal que G' possui os mesmos vértices e arestas que G. Cada aresta de G' possui uma direção (afinal, G' é dirigido) que depende da aresta correspondente de G:

- se a aresta de G pertence ao emparelhamento, a direção da aresta de G' é de V para U;
- se não pertence, a aresta de G' aponta de U para V.

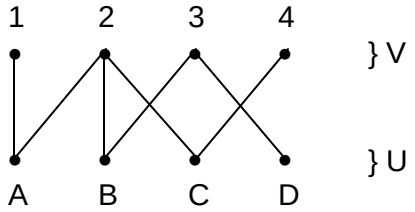
No exemplo abaixo são mostrados G e G':



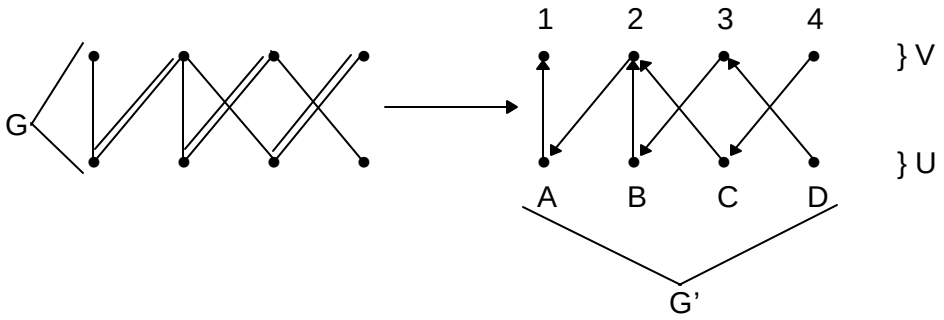


Em G', um caminho qualquer começando em um vértice não acasalado em U e terminando em um vértice não acasalado em V corresponde exatamente a um caminho alternante em G.

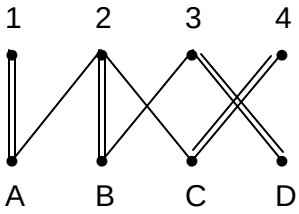
Como exemplo, encontraremos um emparelhamento máximo para o grafo abaixo.



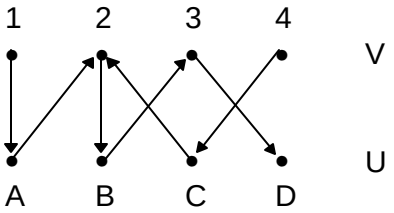
Primeiro, fazemos uma primeira tentativa para M, colocando tantas arestas quanto possível:



Um caminho em G' conforme descrito anteriormente é D3B2A1. Invertendo as arestas de M em G, temos



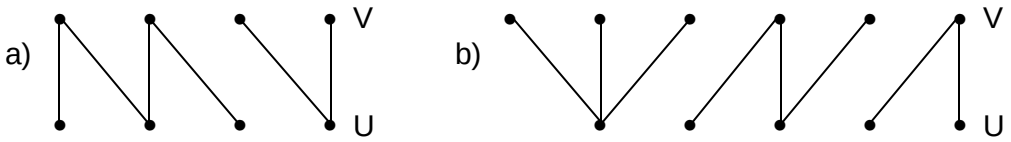
G' resulta em



que não possui mais nenhum vértice não acasalado por onde começar um novo caminho alternante. Portanto o emparelhamento resultante é máximo.

10.1 Exercícios

67. Encontre um emparelhamento máximo para os grafos abaixo.



Utilize o algoritmo dos caminhos alternantes. Mostre a execução de cada passo do algoritmo.

68. Explique porque um caminho alternante possui um número par de arestas do emparelhamento M e um número ímpar de arestas do conjunto E - M.

11 Fluxo em Redes

Seja G = (V, E) um grafo dirigido com um vértice s chamado *fonte* com grau de entrada 0 e um vértice t chamado *sumidouro* com grau de saída 0. A cada aresta e é associada uma capacidade c (e) $>$ 0. A capacidade da aresta e mede a quantidade de fluxo que pode passar através dela.

Um fluxo é uma função f nas arestas de G tal que:

1. $0 \leq f(e) \leq c(e)$. O fluxo através de uma aresta não excede sua capacidade;
2. Para todo $v \in V - \{s, t\}$

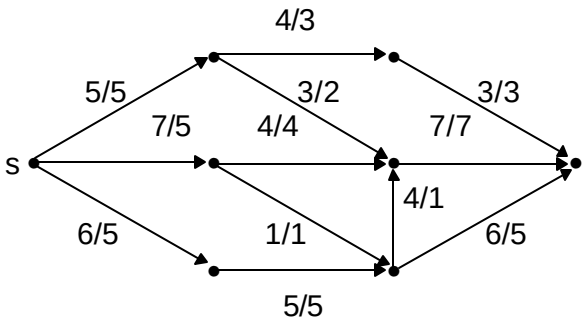
$$\sum_u f(u, v) = \sum_w f(v, w)$$

Isto é, o total de fluxo que entra em v é o mesmo que sai.

Podemos imaginar os vértices como junções e as arestas como canos d'água. Cada cano possui uma capacidade dada em m^3/s além da qual ele arrebenta.

As arestas também podem ser consideradas estradas, os vértices cruzamentos e o fluxo a capacidade de automóveis por minuto da estrada.

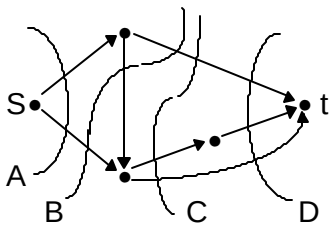
Um exemplo de um grafo com os fluxos e capacidades é mostrado abaixo. 4/3 significa c (e)/f (e); isto é, capacidade 4 e fluxo 3.



Claramente, há um fluxo máximo que pode se originar em s e chegar em t respeitando-se as capacidades de cada aresta.

Definição: Um corte é um conjunto de arestas dirigidas ligando um vértice de um conjunto A de vértices a vértice de um conjunto B. A contém s e B = V - A.

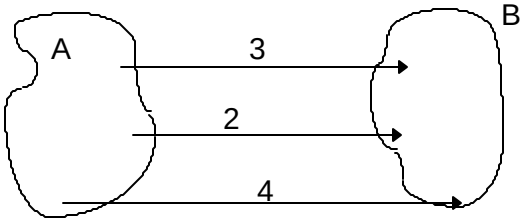
Um corte é um conjunto de arestas que separa um conjunto contendo s de um conjunto contendo t. Ex.:



Os cortes são as linhas

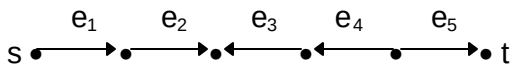
Teorema: O valor do fluxo máximo em uma rede é igual à capacidade do corte de menor capacidade.

A capacidade de um corte é a soma das capacidades de suas arestas. Claramente o fluxo máximo não pode ser maior que a capacidade de algum corte da rede:



De outra forma, o fluxo excederia a capacidade das arestas do corte. A prova de que o fluxo máximo é igual à mínima capacidade de um corte utiliza caminhos alternantes e não será feita aqui.

Definição: Uma seqüência aumentante (SA) em G dado um fluxo f é uma seqüência de arestas que ligariam s a t se as arestas não fossem dirigidas:

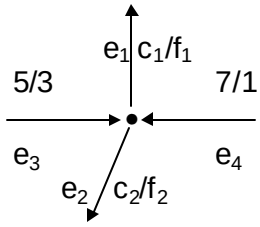


Cada uma das arestas (a, b) da SA deve satisfazer uma das condições abaixo:

1. se (a, b) for dirigida de s para t, então f (a, b) < c (a, b). Isto é, as arestas que apontam" na direção de t podem ter o seu fluxo aumentado. Veja as arestas e_1, e_2 e e_5.

2. Se (a, b) for dirigida de t para s, então f (a, b) > 0. Isto é, as arestas que "apontam" de t para s podem ter o seu fluxo diminuído. Veja e_3 e e_4.

Considere o vértice:

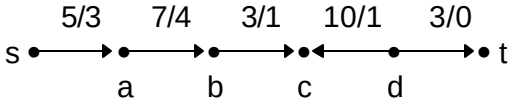


onde 5/3 significa fluxo 3 e capacidade 5. Como todo o fluxo que entra sai, $3 + 1 = f_1 + f_2 = 4$. Claramente podemos aumentar em 1 o fluxo em e_3 e diminuir de 1 em e_4:

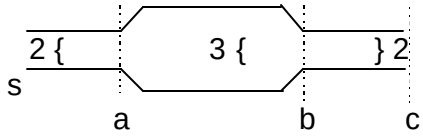
$$(3 + 1) + (1 - 1) = f_1 + f_2 = 4$$

Estamos admitindo que os fluxos do restante do grafo podem ser ajustados para que isto aconteça.

Considere agora a S A



de um grafo G (não representado acima). Então podemos aumentar o fluxo no caminho s - a - b - c de 2, pois as arestas possuem capacidade para tanto. Isto é, a menor diferença c (e) - f (e) no caminho é 2:



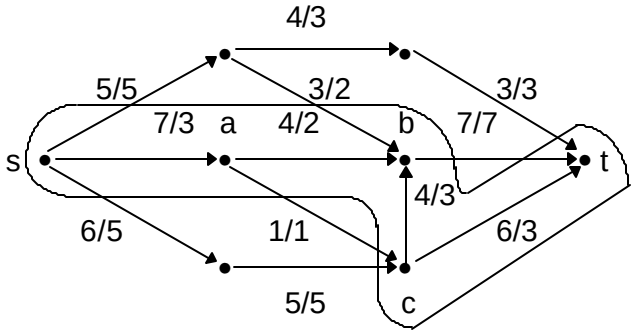
Em c, temos que reduzir o fluxo de d para c de 2, já que o fluxo b → c foi aumentado de 2. Como f (d, c) = 1, isto não é possível.

Então voltamos à SA e aumentamos o fluxo em s - a - b - c de 1. Agora f (d, c) diminui de 1 e consequentemente f (d, t) aumenta de 1.

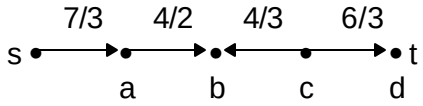
No caso geral, o fluxo em uma SA pode ser aumentado do menor entre os valores:

1. min (c (e) - f (e)) entre as arestas que "apontam" para t.
2. min f (e) entre as arestas que "apontam" para s.

Este valor é maior que zero já que, por definição, c (e) - f (e) > 0 no caso 1 e f (e) > 0 no caso 2. Um exemplo real é:

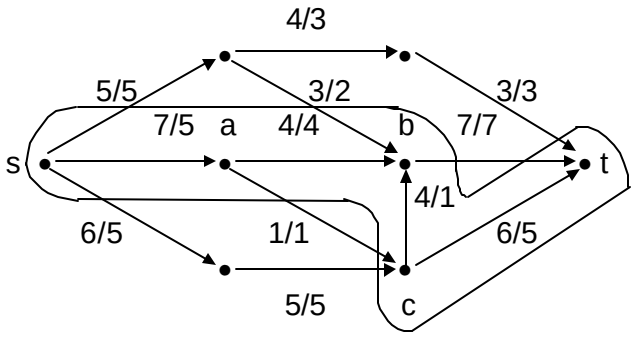


onde existe uma seqüência aumentante s - a - b - c - t:



O mínimo das arestas → t é min (4, 2, 3) = 2
O mínimo das arestas s ← é 3.

Então o fluxo pode ser aumentado de 2 nesta SA:

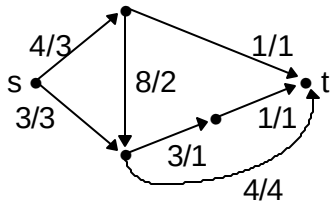


Observe que nenhum fluxo fora da SA é alterado.

Teorema: Um fluxo f é máximo se e somente se ele não admite nenhuma seqüência aumentante.

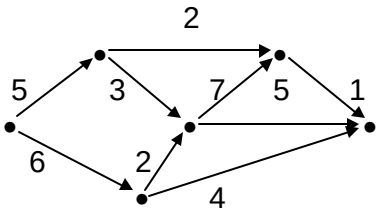
11.1 Exercícios:

69. Maximize o fluxo do grafo abaixo.



fluxo máximo: 6

70. Encontre o fluxo máximo do grafo abaixo utilizando o teorema do corte.



71. Em uma rede, prove que a quantidade de fluxo que sai de s é a mesma que chega em t.

72. Utilize o algoritmo de encontrar o fluxo máximo em uma rede para resolver o problema de encontrar um emparelhamento máximo. Dica: acrescente dois vértices, fonte e sumidouro.

12 Mais Aplicações Práticas de Teoria dos Grafos

12.1 Data-Flow

Em linguagens data-flow, qualquer instrução pode ser executada desde que os valores que ela usa estejam disponíveis. Por exemplo,

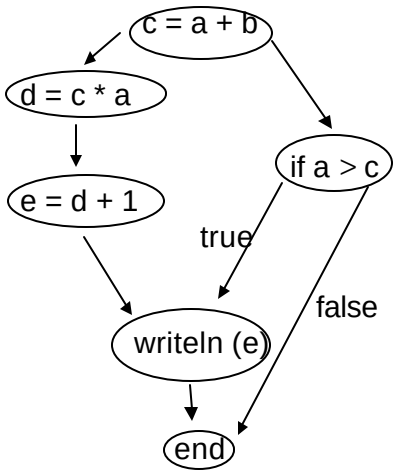
a) `a = 2*b + c;`
pode ser executado tão logo b e c sejam inicializados.

b) `if a > b then` ...
pode ser executado tão logo a e b sejam inicializados.

Dado um procedimento

```
procedure P (a, b : integer)
  var c, d, e : integer;
begin
d = c*a;
e = d + 1;
c = a + b;
if a > c
then
  writeln (e);
endif
end
```

podemos montar um grafo de dependências entre as instruções:



Uma instrução pode ser executada se e somente se todas as que apontam para ela já o foram. Observe que podemos colocar as instruções no procedimento em qualquer ordem.

Os caminhos do grafo que são independentes entre si podem ser executados em paralelo, como

```
d = c*a
e = d + 1          e              if a > c
```

Esta propriedade pode ser usada para compilar programas escritos em uma linguagem seqüencial (ex.: FORTRAN) para um computador paralelo. Os caminhos independentes seriam executados por diferentes processadores.

As ordenações topológicas do grafo descrito acima fornecem as possíveis execuções seqüenciais do programa. Lembre-se de que podem existir (em geral existem) diversas ordenações topológicas (OT) para um mesmo grafo.

Dentre estas OT, o compilador pode selecionar uma que seja mais fácil de otimizar. Por exemplo, suponha que diversas instruções que utilizam a variável i estejam espalhadas por um procedimento:

```
{Ik}   i : = 1;
       …
{Ie}   a : = 2 * i + j;
       …
{Im}   b : = sqrt (i) + sqr (i);
       …
```

Poderia haver uma OT tal que as instruções I_k, I_e e I_m aparecessem juntas:

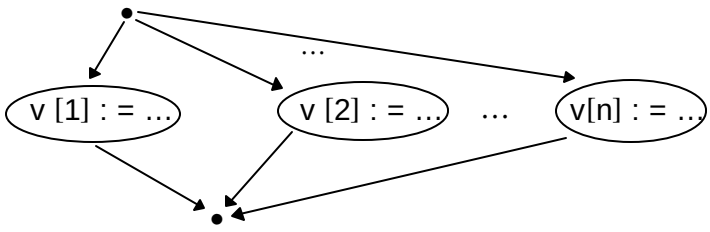
… I_k I_e I_m …

Deste modo o compilador poderia colocar i em um registrador, o que aumentaria a velocidade de execução do programa. Se as instruções I_k, I_e, I_m estiverem espalhadas pelo procedimento (ou OT) e um registrador for associado a i, então as instruções entre I_k, I_e, I_m não poderão utilizar este registrador.

No código

```
for i = 1 to n do
  v [i] = w [i] * w [i] + 1;
```

há n caminhos independentes no grafo:



O que significa que todos eles podem ser executados ao mesmo tempo. E em

```
for i = 1 to n do
```

```
v [i + 1] = v [i] + 1;
```

não há caminhos independentes, significando que paralelismo não é possível.

12.2 Make

O programa make do Unix toma como entrada um arquivo texto contendo comandos da forma

Nome : d_1 d_2 … d_n
    Comandos

Nome é o nome de um arquivo que depende dos arquivos d_1 d_2 … d_n.

Make lê este texto e executa Comandos se a data de última atualização de algum arquivo d_i for mais nova que de Nome. Ou seja, se algum d_i for mais novo que Nome, ele executa Comandos, que são comandos para o Unix que possivelmente deverão atualizar arquivo Nome. Ex.:

```
prog :       a.obj  b.obj   c.obj
     ln -o   prog   a.obj   b.obj  c.obj
a.obj : a.c  prog.h
     cc -c   a.c
b.obj : b.c  prog.h
     cc -c   b.c
c.obj : c.c
     cc -c  c .c
```

ln é o linker e cc o compilador. Se, por exemplo, `b.c` for modificado, make irá compilá-lo novamente (Comando "`cc -c b.c`"), porque ele será mais novo que "`b.obj`". Então "`b.obj`" será mais novo que `prog` que será linkado por "`ln -o prog a.obj b.obj c.obj`".

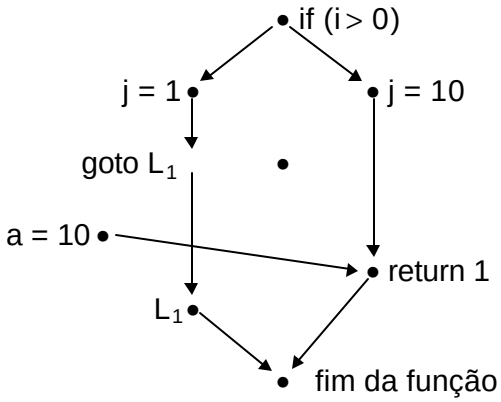
As relações de dependências do make pode ser colocada em forma de um grafo.

12.3 Eliminação de Código Morto

Podemos representar um procedimento como um grafo onde as instruções são vértices e existe aresta dirigida de v para w se w é executado após v (ou pode ser executado, no caso de if's e while's). A função

```
void f(
{
  if (i > 0)
  {
    j = 1;
    goto L1;
    a = 10;
  }
  else
    j = 10;
  return 1;
  L1;
```

}
seria transformada no grafo



Para descobrir o código morto, fazemos uma busca a partir da primeira instrução do procedimento marcando todos os vértices visitados. As instruções correspondentes aos vértices não visitados nunca serão executados e podem ser removidos pelo compilador.

Note que com este mesmo grafo pode-se descobrir se a instrução return será executada por todos os caminhos que ligam o vértice inicial ao vértice "fim da função". A resposta para o grafo acima é : não.

13 Compressão de Dados ¾ Algoritmo de Huffman

Problema: Temos um texto usando caracteres ASCII e queremos comprimi-lo para uma seqüência menor de caracteres.

A técnica que utilizaremos é representar cada caráter por uma seqüência de bits. Então tomaremos o texto original substituindo cada caráter pela seqüência de bits correspondente.

Como existe 128 caracteres ASCII, precisamos de 7 bits para cada caráter se representarmos todos eles usando o mesmo número de bits.

O algoritmo de Huffman representa os caracteres que aparecem mais no texto por uma seqüência menor e os que aparecem mais por uma seqüência maior. Assim ele comprime o texto. Ex.:

Comprimir DABAABCAA assumindo que só usamos caracteres A, B, C e D.

$A \rightarrow 1$

$B \rightarrow 00$

$C \rightarrow 010$

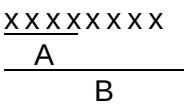
$D \rightarrow 011$

$DABAABCAA \rightarrow 011100110001000 \rightarrow 15$ bits

Se representássemos cada caráter com 2 bits (A = 00, B = 01, C = 10, D = 11) usaríamos $9 \times 2 = 18$ bits.

Poderia haver ambigüidades na representação. Por exemplo, se A = 1 e B = 10, não saberíamos se 1010 é AC ou BB.

Em geral, o código de um caráter não pode ser prefixo de outro.



Esta restrição implica que colocar menos bits para alguns caracteres significa mais bits para outros.

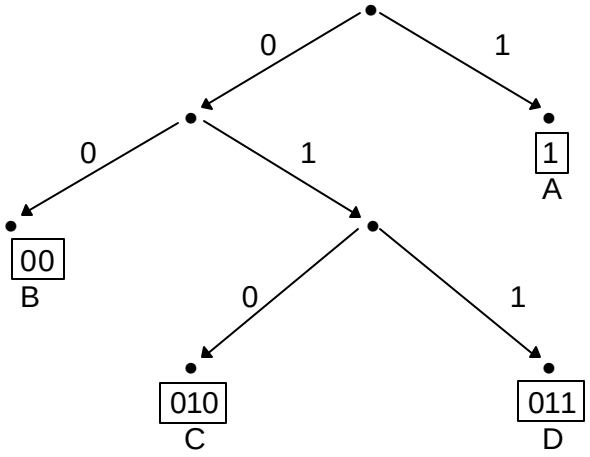
A compressão de dados utilizando esta técnica implica em encontrar uma relação caracteres/bits que satisfaça a restrição de prefixos e que minimize o número total de bits para codificar o texto.

Para saber que caracteres aparecem mais ou menos no texto, calculamos inicialmente a freqüência de cada caráter. Assuma que, em um texto F, os caracteres $c_1, c_2, \ldots c_n$ aparecem com freqüências $f_1, f_2, \ldots f_n$. Uma codificação E associa cada c_i a uma *string* S_i de bits cujo tamanho é s_i. O nosso objetivo é encontrar codificação E que satisfaça a restrição de prefixo e minimize

$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i$$

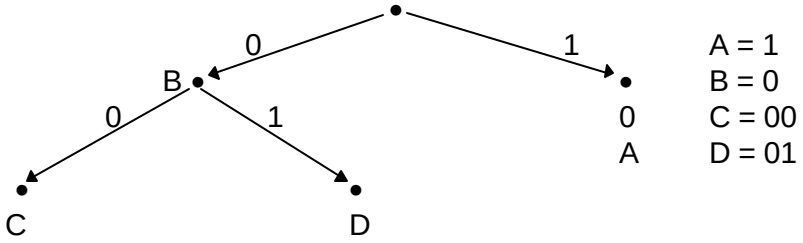
que é o tamanho do texto comprimido. No exemplo anterior, E = { (A, 1), (B, 00), (C, 010), (D, 011) } e $s_1 = 1, s_2 = 2, s_3 = 3, s_4 = 3$ e L (E, F) = 5 . 1 + 2 . 2 + 1 . 3 + 1 . 3 = 15

Considere uma árvore binária dirigida em que cada vértice possui grau de saída 0 ou 2 e as arestas da esquerda e direita estão associados os números 0 e 1, respectivamente:



Associamos as folhas aos caracteres e a seqüência de bits da raiz até a folha como a codificação do caráter. Para decodificar um texto codificado, percorremos a árvore até encontrar o 1.º caráter. Depois fazemos outra busca para o 2.º caráter e assim por diante. Observe que, como os caracteres são folhas, a restrição de prefixo é obedecida.

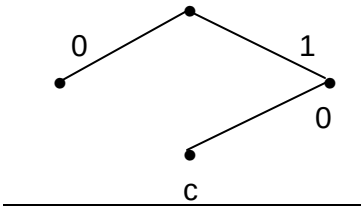
Uma seqüência de bits só poderia ser prefixo de outra se o caráter correspondente estivesse no meio da árvore:



Agora temos que construir uma árvore que minimize L (E, F). Usamos indução finita para reduzir o problema de n para n - 1 caracteres. O caso base é n = 2 e é trivial.

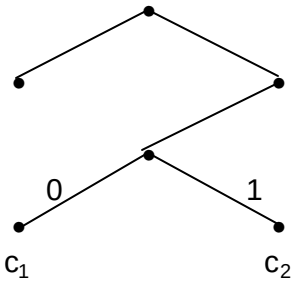
HI: Sabemos como construir a árvore descrita acima para n - 1 caracteres.

Para resolver o problema, tomaremos n caracteres e combinaremos 2 deles em um nó, resultado em n - 1 caracteres. Aplicamos a HI para n - 1 obtendo uma árvore na qual os dois caracteres combinados são introduzidos:



obtido p / n - 1
c = caracteres combinados

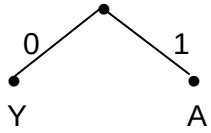
reintroduzindo os 2 caracteres



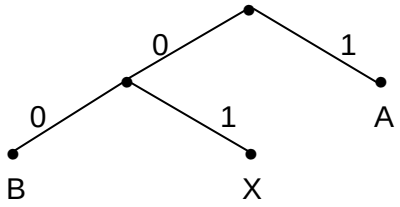
Os caracteres a serem combinados (c_1 e c_2) são os nós com menor freqüência. Nós podemos assumir que c_1 e c_2 são filhos de um mesmo vértice porque cada vértice possui um ou dois filhos.

Obs.: Não provamos que a árvore obtida desta forma minimiza L (E,F).

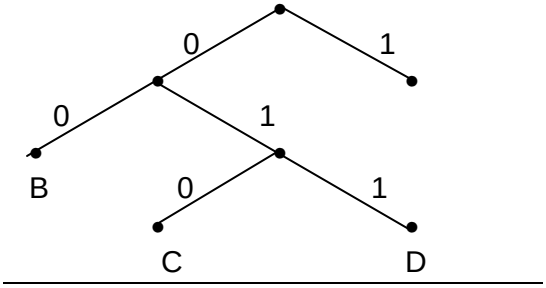
Como exemplo, codificaremos DABAABCAA, onde as seqüências são A = 5, B = 2, C = 1, D = 1.
Combinando C e D, obtemos X com freqüência 2 {A = 5 / B = 2 / X = 2 }. Combinando B e X, obtemos Y com freqüência 4 {A = 5 / Y = 4 }. Este é o caso base, que resulta na árvore



Desdobrando Y, temos



Desdobrando X, temos



Então, A = 1, B = 00, C = 010, D = 011

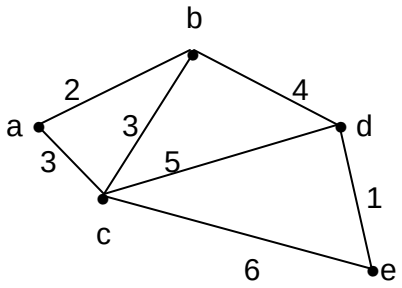
DABAABCAA = 011100110001011

Observe que este algoritmo não é um algoritmo de grafos, embora o uso de grafos facilite a sua compreensão.

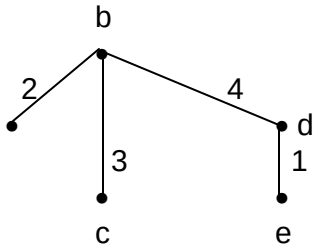
14 Árvore de Espalhamento de Custo Mínimo

Uma árvore de espalhamento de um grafo G é uma árvore contendo todos os vértices de G. Para um dado grafo, podem existir várias árvores de espalhamento diferentes. Se associarmos um peso a cada aresta, uma destas árvores terá um custo menor do que as outras.[2] Esta seção apresenta um algoritmo que encontra a árvore de espalhamento de custo mínimo (AECM) de um grafo. Note que, dentre todos os subgrafos conectados que possuem todos os vértices de G, o que possui custo mínimo é uma árvore. Se tivesse um ciclo, poderíamos retirar a aresta de maior custo e continuaríamos a ter um subgrafo conectado, que ainda teria um custo menor.

G:



Para o grafo acima, teríamos a AECM é

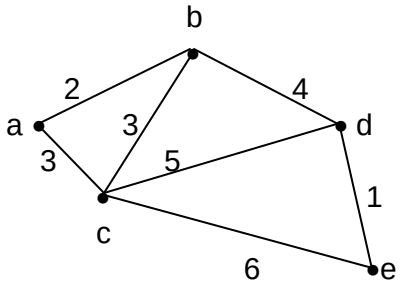


Para encontrar a AECM de um grafo G, usaremos a seguinte HI:

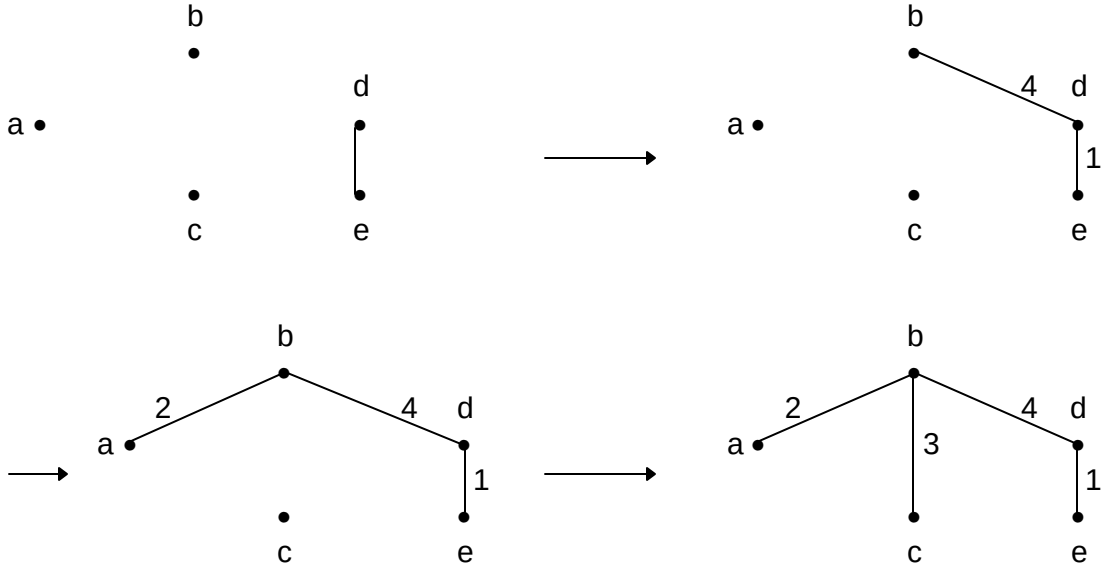
HI: Dado um grafo dirigido G = (V, E), nós sabemos como encontrar um subgrafo T de G com K arestas tal que T é uma árvore que é um subgrafo da AECM de G.

Analisando a HI de uma perspectiva dinâmica, ela começa com uma árvore T vazia e vai acrescentando arestas nesta árvore até obter uma árvore que contém todos os vértices de G. Considerando o grafo

[2] Naturalmente, o custo de uma árvore é a soma dos custos de suas arestas.



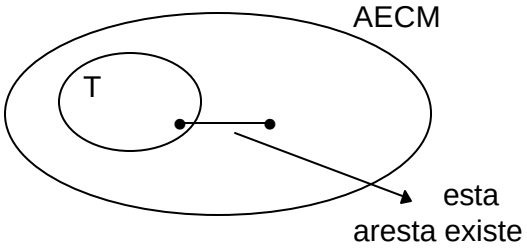
os passos da execução do algoritmo correspondente àHI seriam:



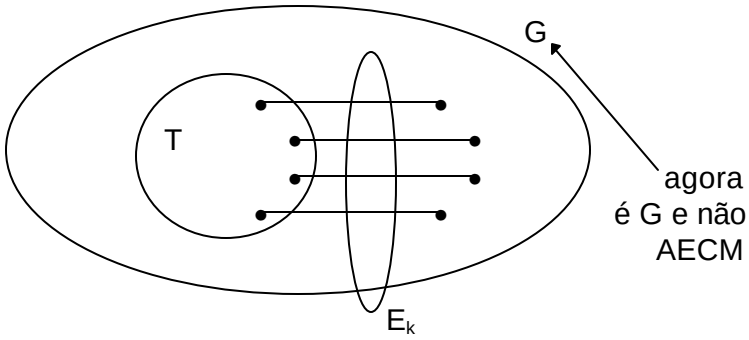
O caso base para a HI é K = 1. Então precisamos escolher a primeira aresta a ser colocada em T. Afirmamos que esta aresta é a de custo mínimo dentre todas as arestas do grafo. Para provar este ponto, suponha que tenhamos a AECM e ela não possui a aresta de custo mínimo. Então podemos acrescentar esta aresta à AECM produzindo um ciclo. Retirando uma outra aresta qualquer do ciclo, obtemos uma árvore que possui um custo menor do que a AECM, o que contradiz a hipótese de que a AECM possui custo mínimo. Logo, a suposição inicial de que a AECM não possui a aresta de custo mínimo está errada.

No caso geral, já encontramos uma árvore T que é subgrafo da AECM e queremos estender T de uma aresta. A união de T com a nova aresta deve ser uma árvore e ser subgrafo da AECM. O problema é qual aresta escolher.

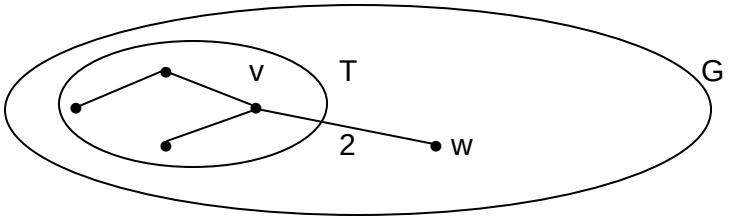
Sendo T um subgrafo da AECM, deve haver pelo menos uma aresta da AECM ligando T a vértice não em T:



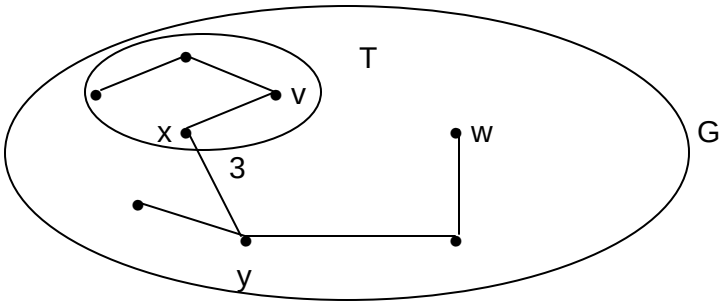
No caso geral, existe mais de uma aresta nesta situação. Seja E_k o conjunto de todas as arestas ligando vértices em T a vértices fora de T. Afirmamos que a aresta de menor custo de E_k pertence àAECM.



Para provar este ponto, assuma que (v, w) é a aresta de menor custo em E_k tal que $v \in T$ e $w \notin T$.



Suponha que (v, w) $\notin$ AECM, temos a configuração



onde apenas as arestas da AECM são mostradas. Como a AECM contém todos os vértices do grafo, existe uma aresta ligando um vértice de T a outro não em T. Suponha que esta aresta seja (x, y), que é diferente de (v, w) já que estamos admitindo que (v, w) $\notin$ AECM.

Existe um caminho entre v e w na AECM que contém (x, y), já que esta aresta conecta T com G - T, $v \in T$, $w \in G - T$. Acrescentando (v, w) na AECM, criamos um ciclo formado pelo caminho entre v e w e a aresta (v, w). Removendo a aresta (x, y) deste ciclo, a AECM continua sendo uma árvore e possui custo menor que a anterior, pois o custo de (v, w) é menor do que (x, y), pois estamos admitindo que o custo de (v, w) é o menor dentre todos os custos de arestas ligando T a G - T (menor custo em E_k).

Acima nós admitimos que a aresta de menor custo em E_k não está na AECM e encontramos uma contradição que é a AECM não ter custo mínimo. Isto é, acrescentando (v, w) e retirando (x, y), encontramos uma árvore em um custo menor ainda.

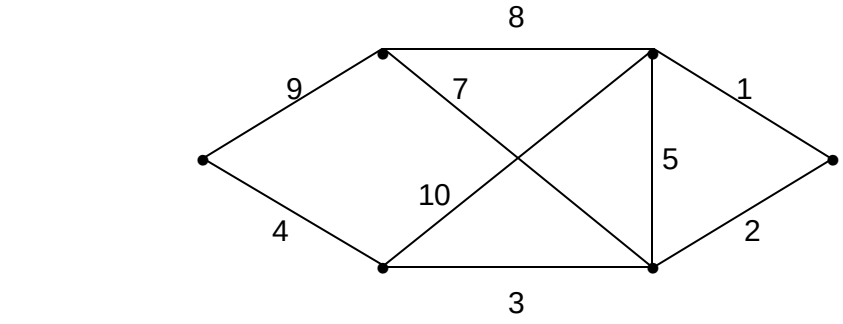
Então a suposição inicial de que (v, w), aresta de menor custo em E_k, não está na AECM, está errada. Isto é ,

$(v, w) \in$ AECM

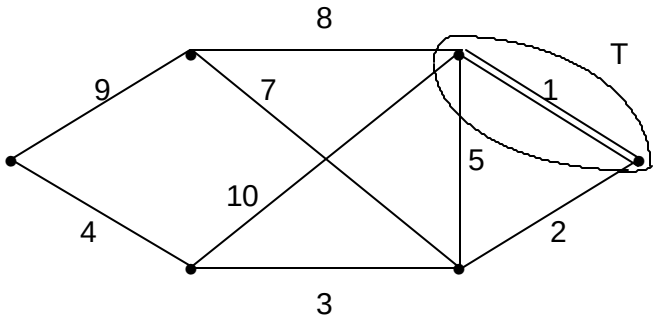
O algoritmo correspondente à HI começa com T = { aresta de menor custo } e vai acrescentando arestas a T até que T possua todos os vértices do grafo. As arestas acrescentadas são as de menor custo que ligam T a G - T.

```
Algorithm AECM( G (V, E) )

begin
T = aresta de G com menor custo
while |T| < |V| do
  T = T U { aresta (v, w) de menor custo tal que v ∈ T e
    w ∈ G - T }
return T;
end
```

Um exemplo completo é mostrado a seguir.

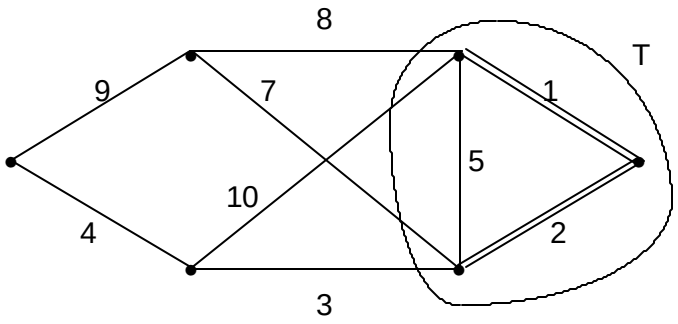


⇒



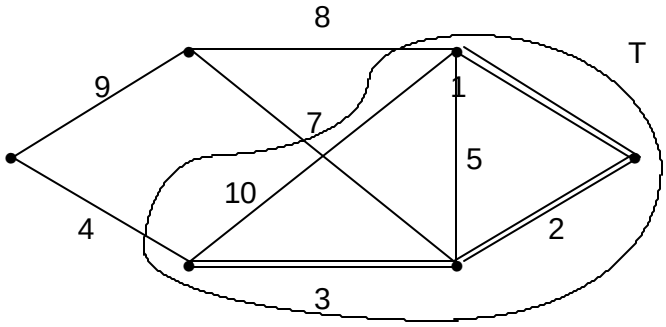
acrescenta min (8, 10, 5, 2)

⇒



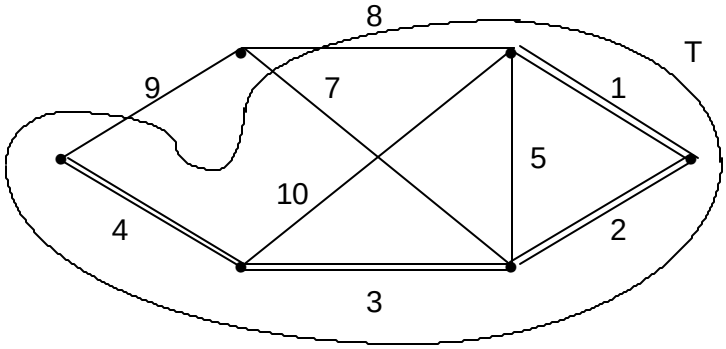
acrescenta min (8, 10, 7, 3)

⇒



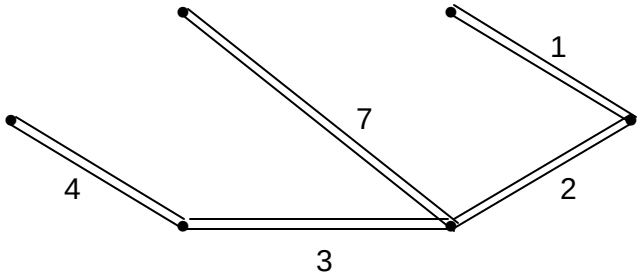
acrescenta min (8, 7, 4)

⇒



acrescenta min (9, 7, 8)

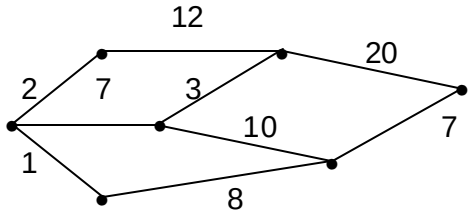
a árvore resultante é



14.1 Exercícios

73. (4) Prove ou mostre um contra-exemplo: A Árvore de Espalhamento de Custo Mínimo de um grafo G (V, E) possui as $|V| - 1$ arestas de menor custo de G.

74. (4) Encontre a AECM do grafo abaixo.



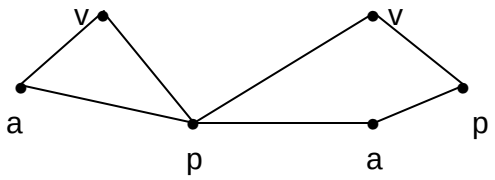
Mostre as árvores obtidas em cada passo do algoritmo.

75. (3) Prove que a aresta de menor custo em um grafo G pertence àAECM de G.

15 Coloração

Seja G (V, E) um grafo e C = $\{C_i \mid 1 \leq i \leq n\}$ um conjunto de cores. Uma coloração de G é a atribuição de cores de C para todos os vértices de G de tal forma que vértices adjacentes tenham cores diferentes. Ex.:

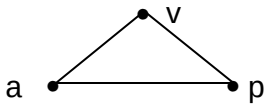
C = {v (ermelho), a (zul), p (reto)}



Uma K-coloração é uma coloração que utiliza K cores.

Definição: O número cromático de um grafo G, indicado por X (G), é o menor número de cores K para o qual existe uma K- coloração de G.

No grafo acima, X (G) = 3, pois o "triângulo"

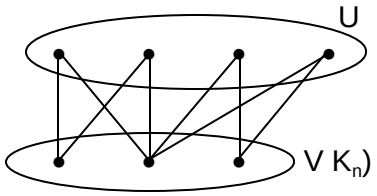


impede que X (G) seja 2.

Um grafo completo de n vértices, também conhecido por K_n necessita de n cores, já que cada vértice está ligado a todos os outros:

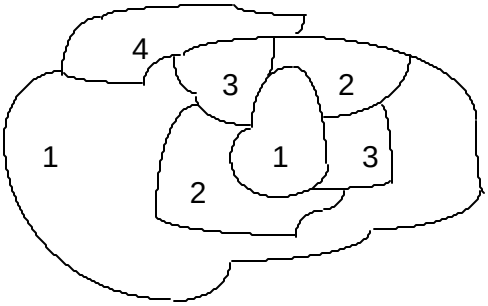
Então, X (K_n) = n. Obviamente, se um grafo G possuir K_n como subgrafo, então X (G) ≥ X (K_n) ou X (G) ≥ n.

Um grafo bipartido pode ser dividido em dois conjuntos U e V tal que não existam arestas dentro de cada conjunto:



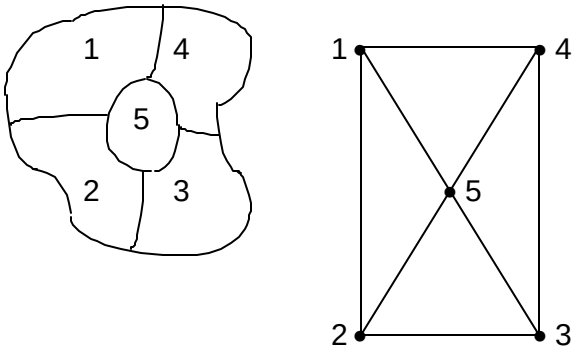
Então, todo grafo bipartido pode ser colorido com apenas duas cores.

O Problema das Quatro Cores: Dado um mapa qualquer (no plano), podemos colori-lo com apenas quatro cores? Por colorir queremos dizer que regiões adjacentes são coloridas com cores diferentes:



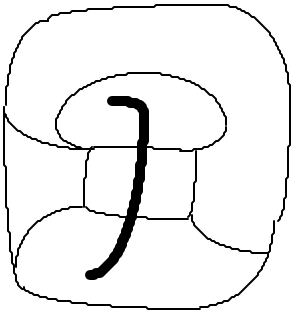
De fato, quatro cores podem colorir qualquer mapa no plano, o que é conhecido por cartógrafos a séculos. Contudo, este teorema só foi provado em 1976 usando teoria dos grafos e um computador.

Este problema pode ser transformado em um problema de grafos associando-se cada região a um vértice. Existe uma aresta entre dois vértices se as duas regiões correspondentes forem adjacentes (fazem fronteira) no mapa. Por exemplo, o mapa da esquerda é transformado no grafo da direita.

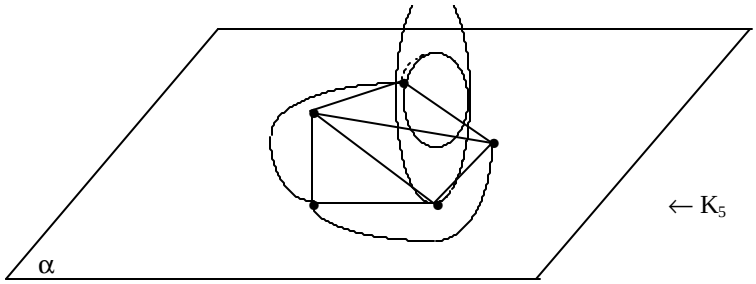


O problema agora é provar que qualquer grafo construído desta forma pode ser colorido com no máximo quatro cores.

Todo grafo obtido de um mapa é planar. Se não fosse, haveria fronteira entre duas regiões que não fazem fronteira no plano — seria uma fronteira no espaço. Veja figura abaixo, onde a ligação preta indica uma ponte acima do papel.



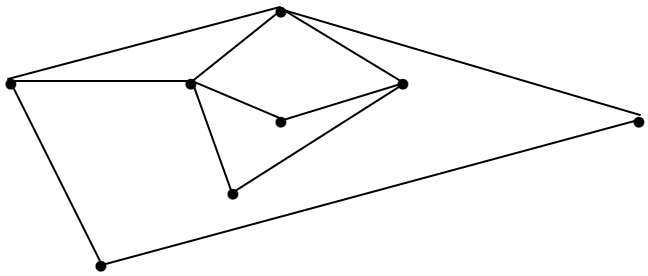
Lembre-se de que um grafo não planar pode ser colocado em um plano desde que algumas arestas liguem alguns vértices no espaço:



a aresta pontilhada é uma aresta no espaço.

15.1 Exercícios

76. O grafo da figura abaixo mostra o mapa rodoviário de um país. Os vértices representam cidades e os arcos, estradas. Em cada cidade, o governo espera construir uma das seguintes obras: um teatro, um centro de esportes, uma piscina. Isto é, apenas uma e sempre uma obra em cada cidade. Decidiu-se que duas cidades ligadas por uma estrada não devem possuir obras semelhantes. É possível atribuir obras a cidades de acordo com essa restrição ? Que problema em teoria dos grafos é equivalente a este ?



77. (4) Mapeie o problema abaixo para grafos, especificando precisamente o que é vértice, aresta e qual seria o problema a ser resolvido. Não se esqueça de dizer os parâmetros do problema. Por exemplo, uma resposta poderia ser “Encontre um clique com k vértices”.

- Uma companhia manufatura os produtos químicos C_1, C_2, … C_n. Alguns destes produtos podem explodir se colocados em contato com outros. Como precaução contra acidentes, a companhia quer construir k armazéns para armazenar os produtos químicos de tal forma que produtos incompatíveis fiquem em armazéns diferentes. Qual é o menor número k de armazéns que devem ser construídos?

16 Redução de Algoritmos

Um problema (não algoritmo) P pode ser reduzido a um problema Q se, conhecido um algoritmo para Q, então podemos encontrar um algoritmo para P. Indicaremos que P pode ser reduzido a Q por

$$P \longleftarrow Q$$

a direção da flecha indica para onde a solução vai.

O problema "encontre o maior elemento de um vetor" pode ser reduzido a "ordene um vetor". Após ordenar o vetor, podemos simplesmente pegar o último elemento, que é o maior.

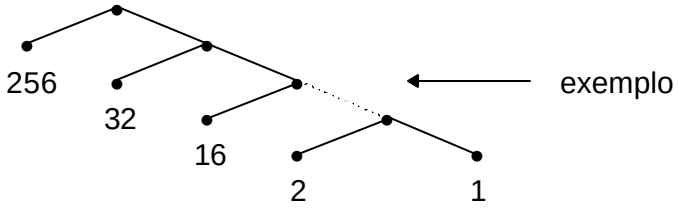
Note que a transformação do resultado de "ordene um vetor" para "encontre o maior…" é feito em tempo O (1).

O problema de multiplicar duas matrizes A e B pode ser reduzido ao problema de elevar uma matriz ao quadrado usando a seguinte fórmula:

$$\begin{pmatrix} O & B \\ A & O \end{pmatrix}^2 = \begin{pmatrix} O & B \\ A & O \end{pmatrix}\begin{pmatrix} O & B \\ A & O \end{pmatrix} = \begin{pmatrix} BA & O \\ O & AB \end{pmatrix}$$

Tendo A e B, construímos a matriz $\begin{pmatrix} O & B \\ A & O \end{pmatrix}$ e invocamos o algoritmo SQR (X) que retorna X^2. Do resultado tomamos BA e AB. Conclusão: A complexidade da multiplicação de matrizes não é maior do que a complexidade do problema "Eleve matriz X ao quadrado". Isto é, tendo disponível um algoritmo para elevar uma matriz ao quadrado, podemos obter um algoritmo para multiplicar matrizes com a mesma complexidade.

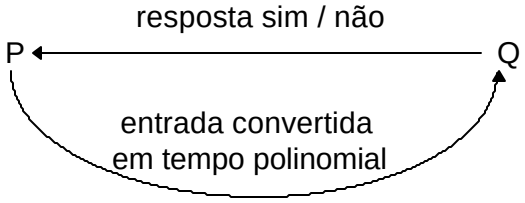
O problema da ordenação dos números X_1, X_2, … X_n pode ser reduzido à compressão de dados pelo método de Huffman da seguinte forma:
Para isto, construa a árvore de Huffman para 2^{X1}, 2^{X2} … 2^{Xn}. Pelo algoritmo, a árvore terá a forma



Os números ordenados podem ser obtidos em tempo linear (O (n)) percorrendo-se a árvore.

Deste ponto em diante nós estudaremos apenas os problemas que retornam apenas "sim" ou "não". A maioria dos problemas pode ser facilmente convertida para este tipo de problema.

Definição: Um problema P é polinomialmente redutível a Q se é possível transformar a entrada de P em entrada para Q em tempo polinomial e Q retorna sim (ou não) se e somente se P retorna sim (ou não) com a mesma entrada. Isto é, a resposta dada por Q pode ser usada como a resposta para P.



Se a conversão da entrada de P para Q tomar tempo O (f (n)) e o algoritmo para Q possuir complexidade O(g(n)), então o algoritmo para P dado pela redução possui complexidade

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

Em particular, se g (n) for exponencial, O(f(n) + g (n)) será O (g (n)). Se g(n) for polinomial, o algoritmo para P dado pela redução também será polinomial. Exemplos:

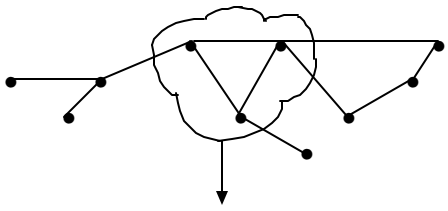
$O(n^3 + 2^n) = O(2^n)$
$O(n^2 + n) = O(n^2)$

P e Q são polinomialmente equivalentes se cada um pode ser reduzido ao outro em tempo polinomial. Todos os problemas que podem ser resolvidos em tempo polinomial são equivalentes polinomialmente entre si (PROVE !).

A classe de todos os **Problemas** que podem ser resolvidos em tempo polinomial é chamada de P. A classe dos problemas para os quais uma dada solução pode ser conferida em tempo polinomial é chamada de NP.

Por exemplo, o problema “ordene um vetor” pertence a NP por que, dado um vetor (supostamente ordenado), podemos conferir se ele está mesmo ordenado em tempo O (n).

O problema “encontre um subgrafo completo em um grafo G” pertence a NP.



Dado o subgrafo encontrado, podemos facilmente descobrir se ele é completo. Aparentemente, a classe NP possui problemas muito mais difíceis que a P, já que para um problema pertencer a NP temos apenas que conferir uma dada solução em tempo polinomial, enquanto que em P precisamos resolver o problema em tempo polinomial. Voltaremos a esta questão adiante.

16.1 Problema da Satisfabilidade

Uma expressão booleana S estará na forma normal conjuntiva (FNC) se ela for o produto de sub-expressões que são somas.

Produto é "e" lógico e soma é "ou" lógico. Exemplo:

$$S = (a + \bar{b} + c)\,(\bar{a} + b + c)\,(\bar{a} + \bar{b} + \bar{c})$$

Qualquer expressão booleana pode ser transformada em FNC. Uma expressão booleana S é satisfazível se suas variáveis podem receber valores 0 ou 1 de tal forma que S seja 1. No exemplo acima, S é satisfazível: se a = 1, b = 1 e c = 0, então S = 1.

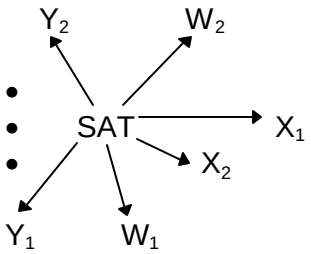
O problema SAT é determinar se dada expressão S é satisfazível ou não. Encontrar os valores da variáveis para os quais S = 1 *não* faz parte do problema SAT. Este problema está claramente em NP porque, dados valores para as variáveis, podemos facilmente descobrir o valor de S. Se S for 1, será satisfazível.

Definição: Um problema X é chamado de NP-completo se

1. X pertence a NP.
2. Cada problema em NP é polinomialmente redutível a X. Isto é, com a solução para X temos a solução para qualquer outro problema.

Teorema de Cook (1971): SAT é NP - completo

Este é um dos resultantes mais importantes da computação teórica, senão o mais importante. Ele implica que, se o problema SAT pode ser resolvido em tempo polinomial, todos os outros problemas NP-completos o serão. Como exemplo, considere que X_1 é NP-completo, como mostra a figura:



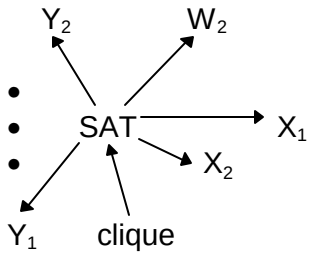
Então a complexidade de X_1 é igual à complexidade do polinômio que transforma a entrada de X_1 para SAT mais a complexidade do SAT, que estamos assumindo ser polinomial. Como O(polinômio) + O(polinômio) = O(polinômio), X_1 também é polinomial. Então, se SAT puder ser resolvido em tempo polinomial, todos os problemas em NP também poderão, implicando P = NP. Infelizmente, ninguém conseguiu ainda provar que SAT pode ou não ser resolvido em tempo polinomial. Isto é, ninguém sabe se P = NP ou $P \neq NP$, embora ninguém acredite que P = NP.

Em 1972, Richard Karp descobriu outros 24 problemas NP-completo. A lista hoje contém 4000 problemas. Se uma solução polinomial for encontrada para um deles, automaticamente teremos a solução polinomial para todos os demais. Se alguém provar que um desses problemas não pode ser resolvido em tempo polinomial, nenhum deles poderá.

Todos os problemas NP-completo são polinomialmente equivalentes entre si, o que indica que eles têm uma estrutura em comum.

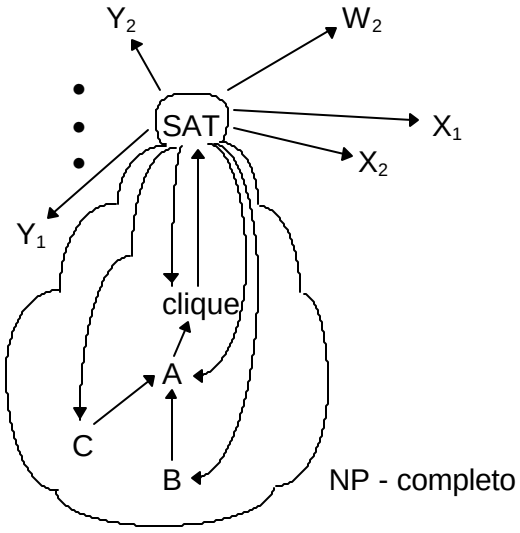
Definição: Clique de um grafo é um subgrafo completo deste grafo.

O problema do clique é: determine se o grafo contém um clique de tamanho ≥ K. Clique é NP - completo, pois SAT é redutível ao problema do clique:



Dado um problema qualquer em NP, digamos X_1, podemos reduzi-lo a SAT e então reduzir SAT ao problema do clique. Resolvendo clique, resolvemos X_1. Então, qualquer problema em NP é redutível ao clique.

Em uma figura, como a acima, representando relações de redução, um problema X será NP-completo se existir um caminho de X para SAT. Como X também pertence a NP, existirá seta de SAT para X, pois qualquer problema em NP pode ser reduzido a SAT. Em geral, não se prova que SAT é redutível a um problema X, mas sim que algum problema NP-completo é redutível a X. No desenho abaixo mostrando as relações de redução, suponha que foi provado que clique é redutível a A. Como o problema do clique é NP-completo, A também o é.



Observe que todos os problemas NP - completo são polinomialmente equivalentes, pois um pode ser reduzido a outro em tempo polinomial. Dados dois problemas NP-completo X e Y, existe sempre um caminho de X para Y e outro de Y para X, pelo menos um deles passando por SAT. Por exemplo, podemos usar a solução de A para resolver C usando o caminho A $\rightarrow$ clique $\rightarrow$ SAT $\rightarrow$ C.

16.2 Exercícios

78. (3) Dados os problemas

(a) Dado um vetor de elementos $x_1, x_2, \ldots x_n$, encontre um elemento x dentre eles tal que x é maior ou igual que n/2 elementos.

(b) Ordene o vetor descrito em (a).

(c) Encontre o menor e o maior elementos do vetor.

Reduza:

- (a) a (b).
- (a) a (c).
- (c) a (b).

79. (5) Assuma que um problema P é polinomialmente redutível a Q e:

(a) A Entrada de P pode ser transformada em entrada para Q em tempo O (n log n).

(b) Um algoritmo AQ que resolve o problema Q pode ser Executado em tempo O (n^2).

Explique como podemos fazer um algoritmo para P usando AQ. Qual a complexidade deste algoritmo ?

80. (5) Um problema P toma um vetor de n inteiros qualquer como entrada e não conhecemos nenhum algoritmo para resolvê-lo. Contudo, descobrimos que P pode ser reduzido a um problema Q, que tem como entrada um vetor ordenado de inteiros. Admitindo que exista um procedimento

procedure ProcQ (vet, n) : boolean;

que resolva o problema Q, faça um procedimento ProcP para resolver o problema P. Admita a existência de um procedimento Ordene (vet, n) que ordena vetor vet de n inteiros.

Qual a complexidade de ProcP se a de ProcQ é:

- O (n^2) ?
- O (n) ?
- O (2^n) ?

81. (5) Assumindo que N e NP são conjuntos, qual a relação entre eles ($\subset$, =, $\neq$, etc) ?

82. (1) A expressão

$$S = (a + b)\ (\bar{a} + b + c)\ (\bar{a} + \bar{b} + \bar{c})\ \bar{c}$$

é satisfazível?

83. (2) Quais dos problemas abaixo pertencem a NP ?

(a) Encontrar um elemento em um vetor que é maior do que todos os outros.

(b) Associar cores aos vértices de um grafo de tal forma que dois vértices adjacentes possuam cores diferentes.

(c) Encontrar a AECM de um grafo.

(d) Encontrar um caminho hamiltoniano de um grafo.

84. (5) Sejam A e B dois problemas NP-completos e P(n) o polinômio que é a complexidade da conversão da entrada de A para B. Se alguém encontrar um algoritmo de complexidade f (n) para resolver B, então, aplicando a redução, teremos um algoritmo de complexidade f (n) + P (n) para A. Isto é:

Complex (Alg. A) = P (n) + f (n)

Suponha que:

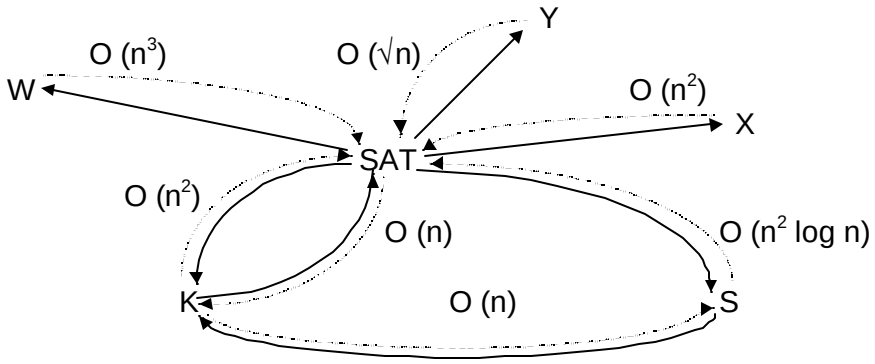
(a) f (n) é polinomial. Podemos afirmar que existe algoritmo para A com complexidade polinomial ? Se existe, mostre-o!

(b) f (n) é exponencial. Podemos afirmar que existe algoritmo para A com complexidade exponencial ?

(c) f (n) é exponencial. Podemos afirmar que não exise algoritmo para A com complexidade polinomial ?

85. (3) Suponha que SAT não seja NP-completo. Como poderíamos provar que o problema do clique é NP-completo ?

86.. (50) Suponha que todos os problemas NP existentes estejam representados na Figura abaixo.



Uma seta de Q para P indica que P pode ser reduzido a Q A seta em linhas pontilhadas mostra a complexidade da transformação da entrada de P para a entrada de Q. Baseado nesta Figura, responda:

(a) Se for encontrado um algoritmo de complexidade $O(n^2)$ para resolver SAT, qual a complexidade dos algoritmos para resolver cada um dos problemas acima se estes algoritmos forem obtidos pela redução dos problemas a SAT ?

(b) Se for encontrado um algoritmo de complexidade $O(n^2)$ para resolver R, qual a complexidade dos algoritmos para resolver cada um dos problemas acima se estes algoritmos forem obtidos pela redução dos problemas a R ?

(c) Se um algoritmo para S tiver complexidade $O(n^2)$, qual seria a complexidade de um algoritmo para R obtido pela redução a S ?

(d) Explique porque R e S são NP-completos.